Red Hat JBoss Data Virtualization 6.3
Development Guide Volume 2: Governance

This guide is intended for developers

Red Hat Customer Content Services

Red Hat JBoss Data Virtualization 6.3 Development Guide Volume 2: Governance

This guide is intended for developers

Red Hat Customer Content Services

Legal Notice

Abstract

This document provides information on Design-Time Governance and the hierarchical database.

Table of Contents

Chapter 1. Read Me

1.1. Governance is Deprecated

Warning

Governance is deprecated. New users should not adopt the features discussed in this book.

1.2. Back Up Your Data

Warning

Red Hat recommends that you back up your system settings and data before undertaking any of the configuration tasks mentioned in this book.

1.3. Variable Name: EAP_HOME

`EAP_HOME` refers to the root directory of the Red Hat JBoss Enterprise Application Platform installation on which JBoss Data Virtualization has been deployed.

1.4. Variable Name: MODE

`MODE` will either be **`standalone`** or **`domain`** depending on whether JBoss Data Virtualization is running in standalone or domain mode. Substitute one of these whenever you see **`MODE`** in a file path in this documentation. (You need to set this variable yourself, based on where the product has been installed in your directory structure.)

1.5. Red Hat Documentation Site

Red Hat's official documentation site is available at https://access.redhat.com/site/documentation/. There you will find the latest version of every book, including this one.

Chapter 2. Governance Overview

2.1. Governance in JBoss Data Virtualization

Governance in JBoss Data Virtualization is supported by this component:

Hierarchical Database

The hierarchical database is often not so much used for governance of artifacts, but more for storing additional data and metadata related to JBoss Data Virtualization. For example, if a user needs to incorporate information in their JBoss Data Virtualization solution such as images, emails or other semi-structured content, they can store that in the hierarchical database and then access it through JBoss Data Virtualization. Or they can put additional descriptive information about their data in the hierarchical database and combine it with other data.

Example 2.1.

Let us say a customer has a database with customer contact information in it. That database has the customer data but does not necessarily contain information about that customer data such as:

- the owner of the data
- whether the data is considered authoritative
- whether this is this the only source of the customer data
- any other attributes that an organization might want to "tag" that data with

If this additional information is stored in the hierarchical database, then it can be accessed either through REST API or via JBoss Data Virtualization so the organization can augment or enrich their data.

Note

The hierarchical database can provide extra value for more advanced scenarios and solutions.

Part I. The Hierarchical Database

Chapter 3. The Hierarchical Database

3.1. The Hierarchical Database

The hierarchical database is a distributed, hierarchical, transactional, and consistent data store. It provides support for queries, full-text search, events, versioning, references, and flexible and dynamic schemas. Clients use the JSR-283 standard Java API for content repositories (also known as JCR) or the hierarchical database's REST API, and can query content through JDBC and SQL.

You can use the hierarchical database for data that is organized in a tree-like hierarchical structure where related data is stored close together and navigation to related content is frequently required.

The hierarchical database automatically extracts the structured information within the files enabling clients to navigate or use queries to find files satisfying complex, structurally-oriented criteria. You can use the hierarchical database for all kinds of applications, including repositories, content management systems, historical data services, provisioning and governance systems, and metadata management systems.

The hierarchical database supports all JCR 2.0 required features:

- Repository acquisition
- Authentication
- Reading/Navigating
- Query
- Export
- Node type discovery
- Permissions and capability checking

The hierarchical database supports most of the JCR 2.0 optional features:

- Writing
- Import
- Observation
- Workspace management
- Access control management
- Versioning
- Locking
- Node type management
- Same-name siblings
- Orderable child nodes
- Shareable nodes
- mix:etag, mix:created and mix:lastModified mixins with automatically created properties

The hierarchical database supports the following query languages:

- The JCR-SQL2 and JCR-JQOM query languages defined in JCR 2.0 (JSR-283)
- The XPath and JCR-SQL query languages defined in JCR 1.0 (JSR-170)
- a full-text search-engine-like language

3.2. Federation

Previously, the hierarchical database owned all of its own data. It stored all of the information about all nodes within an Infinispan cache, and the repository had a single binary store used to persist all **`BINARY`** values.

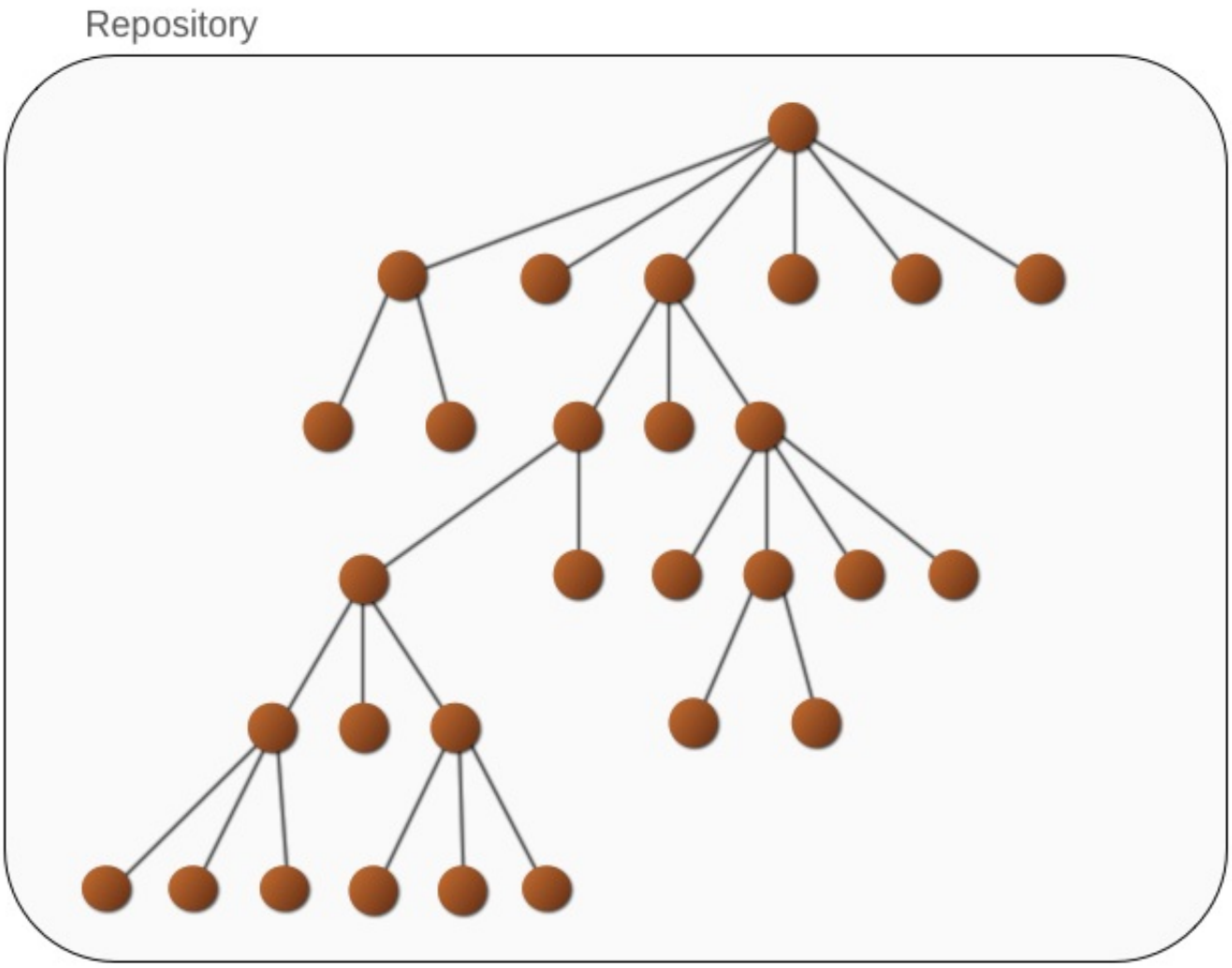


Figure 3.1. Conventional Repository

However, in this release, the hierarchical database provides the ability for clients to access data stored both internally and externally using the single JCR API.

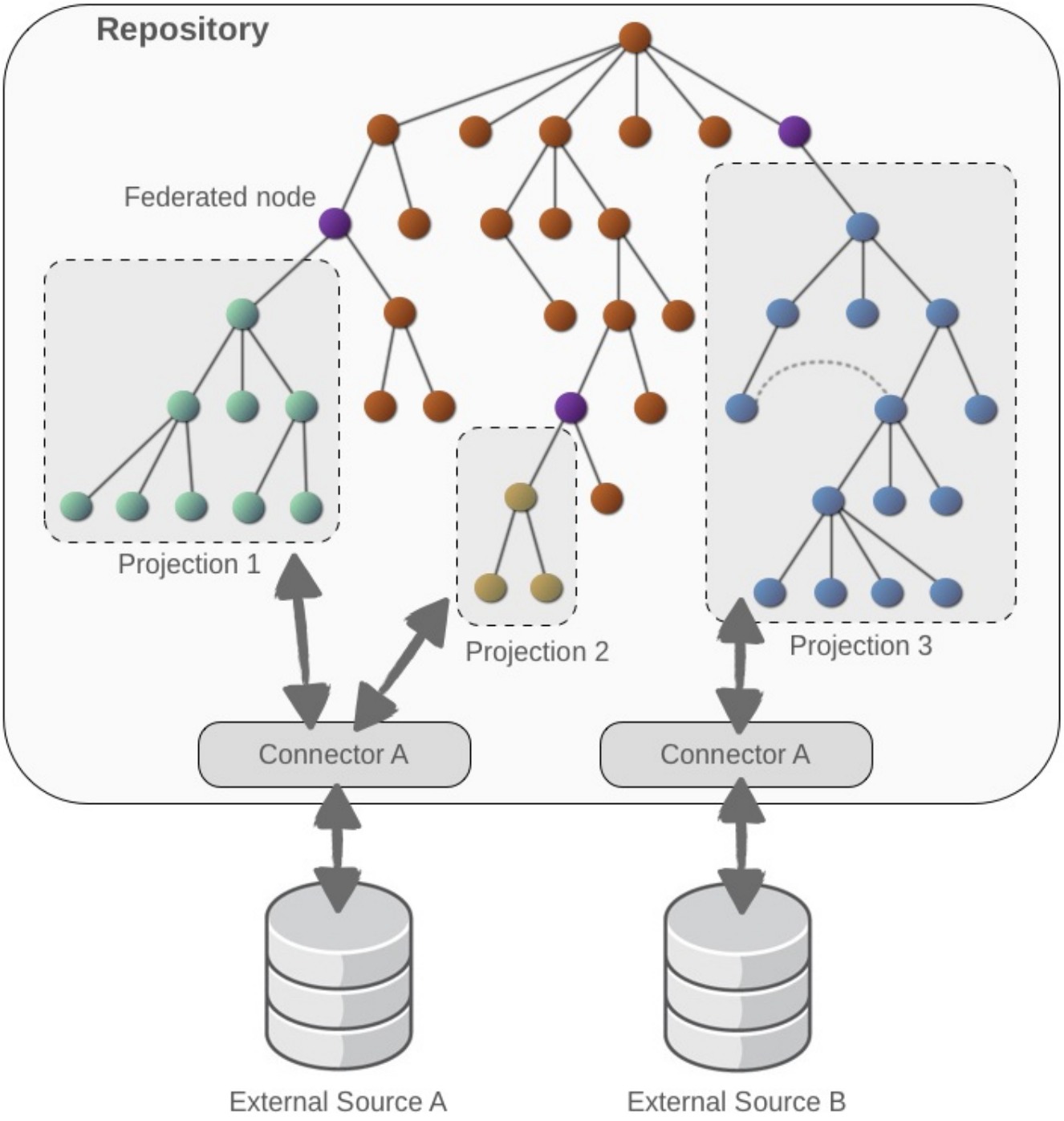


Figure 3.2. Federated Repository

An external system is a system outside of the hierarchical database that owns its own data and that the hierarchical database interacts with to access (and optionally update) that data. The external system might be a data store, or it might be a service that dynamically produces data. Examples of external systems are Oracle 11i, Cassandra, MongoDB, Git, SVN, SAP, file systems, CMIS, RPM repositories, and JCR repositories.

Whereas an external system is a kind of software system, we use the term external source to describe an addressable instance or installation of the external system. For example, external sources might include a particular database instance, a particular Git repository, a particular file system on a specific machine, or a particular instance of a CMIS repository.

In the diagram above, two external sources are shown and labeled "External Source A" and "External Source B". (But the diagram does not define what kind of system they are.)

A hierarchical database connector is the software used to interact with a specific kind of external system. A connector consists of compiled Java classes and resources, and is usually packaged as a JAR with dependencies on 3rd party libraries. The hierarchical database defines a connector SPI (or Service Provider Interface) which the connector must implement. Generally connectors can read and update data in the external system, although a connector implementation may support only read operations.

To be useful, however, a connector must be instantiated and that instance configured to talk to a specific external source . Then that connector instance's job is to create a single, virtual tree of nodes that represents the data in the external source. Note that the connector does not create the entire tree up front; instead, the connector creates the nodes in that virtual tree only when the hierarchical database asks for them. Thus, the potential tree of nodes for a given source might be massive, but only the nodes being used will be materialized.

The diagram of the federated repository shown above includes two connector instances , each of which is configured to talk to one of the external sources.

An internal node is any node within a hierarchical database repository that is owned by the database and stored within the Infinispan cache. In a regular repository (without federation), all nodes are internal nodes.

An external node is any node within a federated hierarchical database repository that is not owned by the database but instead is dynamically generated to represent some portion of data in an external source. Clients view internal and external nodes in exactly the same way, but internally they are handled in different ways.

A federated node is an internal node that contains some children that are external nodes. In other words, only federated nodes can have internal nodes and external nodes as children, whereas internal nodes can only have other internal nodes as children and external nodes can only have other external nodes as children.

A projection is a portion of the repository (really a subgraph) whose nodes are all external nodes that are representations of some of the data in an external source. The nodes are dynamically generated (by the connector's logic) as needed, and can optionally be cached for a configurable amount of time.

The federated repository diagram above shows three projections, labeled "Projection 1", "Projection 2", and "Projection 3". Strictly speaking, projections do not have a name, so the labels are merely for discussion purposes. Note how projections 1 and 2 both project external nodes from "External Source A", whereas projection 3 only projects the external nodes from "External Source B". We often will talk about an external source as having one or more projections; thus "External Source A" has two projections ("Projection 1" and "Projection 2"), while "External Source B" has only one projection ("Projection 3").

Each projection maps a specific subtree of the virtual tree (created by a connector talking to an external source) underneath a specific federated node. A simple path is used to identify the subtree of external nodes, and a simple path is used to identify the federated node. The hierarchical database uses a projection expression that is a string with these two paths:

```
<workspace-name>':' <path-to-federated-node> '=>' <path-in-external-source-
of-node>
```

where

- **<workspace-name>** is the name of the workspace where the projection is to be placed
- **<path-to-federated-node>** is a regular absolute path to the existing internal node under which the external nodes are to appear
- **<path-in-external-source-of-node** is a regular absolute path in the virtual tree created by the connector of the node whose children are to appear as children of the federated node.

Projections can be defined within a repository's configuration (making them available immediately upon startup of the repository) or programmatically added or removed by client applications using the **`FederationManager`** interface.

The hierarchical database public API includes the **`org.modeshape.jcr.api.federation.FederationManager`** interface that defines several methods for programmatically creating and removing projections. Note that at this time it is not possible to programmatically create, modify, or remove external sources, so these must be defined within the repository configuration.

As clients navigate the nodes in the repository, they typically ask for one (or multiple) children of a particular node. Clients repeat this process until they access the node(s) they're looking for.

The hierarchical database performs these operations differently depending upon the kind of node:

- If the parent is an internal node , then all children will also be internal. Therefore, to find a particular child by name, the hierarchical database obtains the parent's child reference to obtain the child's node key, and then looks up the node with that key in the Infinispan cache. This is the "conventional" behavior, and this incurs no overhead even when the repository is configured to use federation.
- If the parent is a federated node , then the process is very similar to internal nodes, except that the interna and external child references are managed separately. The hierarchical database then looks at the child's node key to determine (from the key itself) if the child exists in the Infinispan cache or in an external source. If in an external source, the hierarchical database then calls to the connector to ask for the representation of the requested node.
- If the parent is an external node , then the hierarchical database obtains the parent's child reference and looks up the node with that key in the same connector. The connector then generates a representation of the requested node.

All nodes (both internal and external) can be accessed by **`Session.getNodeByIdentifier(String`** , where the identifier is the same string returned by calling the **`getIdentifier()`** method on the node. The hierarchical database can tell from the identifier whether it is for an external node, and if so it will look up the node in the connector.

Note

Per the JCR specification, clients should treat these identifiers as opaque. In fact, hierarchical database identifiers follow a fairly complex pattern that will likely be difficult to reverse engineer, and which may change at any time.

The hierarchical database actually uses an in-memory LIRS cache of the nodes. So, although the navigation and lookup steps mentioned above do not discuss using the LIRS cache, the hierarchical database always consults this cache when it needs to find a node with a particular node key. If found in the cache, the node will be used. If the cache does not contain the node, then it will consult the Infinispan cache or the connector to obtain (and cache) the node.

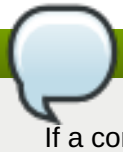
Normally, nodes in the LIRS cache are evicted after a certain (but configurable) time. However, external nodes can have an additional internal property that specifies the maximum time that the node can be in the cache. Or, an external source can be configured with a global time to live value. Either way, the LIRS cache ensures that the nodes are evicted at the appropriate time.

Of course, a node is also evicted from the cache if the node has been changed and persisted (e.g., via **`Session.save()`** or user transaction commit), even if that change was made on a different process in the cluster.

A connector decides which external nodes are to be indexes.

- The connector instance can be configured with a **`queryable`** boolean parameter that states whether any of the content is to be queryable. This defaults to **`true`** .
- The connector can mark any or all nodes as not queryable .

Thus, even though a connector implementation may be written such that some or all of the external nodes can be queried, a repository configuration can configure an instance of that connector and override the behavior so that no nodes are queryable.



Note

If a connector is implemented by marking all nodes as not queryable , then configuring an instance of that connector with **`queryable=true`** has no effect.

Any nodes that are queryable will be included in the index, as long as the hierarchical database is notified of new nodes. By default, external nodes are not automatically indexed. To index them, use the public API for reindexing.

Once indexed, the nodes can be queried like any other nodes.

A connector works by creating a node representation of external data, and that node contains the references to the node's children. These references are relatively small (the ID and name of the child), and for many connectors this is sufficient and fast enough. However, when the number of children under a node starts to increase, building the list of child references for a parent node can become noticeable and even burdensome, especially when few (if any) of the child references may ultimately be resolved into nodes because no client actually uses those references.

A pageable connector is one that will expose the children of nodes in a "page by page" fashion, where the parent node only contains the first page of child references and subsequent pages are loaded only if needed. This turns out to be quite effective, since when clients navigate a specific path (or ask for a specific child of a parent by its name) the hierarchical database does not need to use the child references in a node's document and can instead have the connector resolve such (relative or absolute external) paths into an identifier and then ask for the document with that ID.

Therefore, the only time the child references are needed are when clients iterate over the children of a node. A pageable connector will only be asked for as many pages as needed to handle the client's iteration, making it very efficient for exposing a node structure that can contain nodes with numerous children.

3.3. Architecture

3.3.1. The Hierarchical Database Engine

Perhaps the most important component in the hierarchical database is the engine, which is responsible for managing and making available all of the configured repositories. When the database is embedded into an application, the application is better of manually instantiating the **`org.modeshape.jcr.ModeShapeEngine`** class and explicitly invoking the **`start()`** , **`deployRepository(...)`** and **`shutdown()`** methods in appropriate places within the application's own lifecycle. Note that repository configurations can be updated even when the repository is running and in use. The hierarchical database can also be deployed to a server (e.g., JBoss EAP, Tomcat, etc.) so that the server manages the lifecycle of the engine.

Every repository in a **`ModeShapeEngine`** instance has a unique name, and applications can easily use the engine to get a particular repository by name. If used within an environment that has JNDI, the hierarchical database will also register each repository into JNDI so that applications can easily look it up. See the documentation for all the ways to find a repository.

3.3.2. Repository Configuration

Each repository is configured separately with a file that conforms to the JSON format. (Note that when installed into JBoss EAP, configuring the hierarchical database is done through EAP's configuration system.) The configuration files can be read with the **`org.modeshape.jcr.RepositoryConfiguration`** class, and the resulting **`RepositoryConfiguration`** instances can be passed to the **`ModeShapeEngine.deployRepository(...)`** and **`ModeShapeEngine.updateRepository(...)`** methods.

3.3.3. Clustering

The hierarchical database can be clustered at the repository level. This means that a repository with the same name is deployed to multiple engines (typically in separate processes), and those repository instances are aware of each other so that events that originate in one repository instance will be forwarded to all other repository instances in the cluster. Additionally, the Infinispan cache(s) used in each repository should also be clustered, so that Infinispan can coordinate changes to the data stored in the cache(s).

There are two other important aspects of clustering: storage and indexing.

3.3.4. Clustering: Storage

If the Infinispan caches use cache stores to persist content to the filesystem, a database, cloud storage and so forth then this storage must be compatible with clustering. For example, if the cache store content on the file system, then the cache used by each repository instance must have its own non-shared directory in which the cache can persist information. (Infinispan clustering will use network messaging to ensure that multiple instances that "own" a particular piece of data are all kept in sync.) Some of Red Hat JBoss Data Grid's cache stores are sharable , which means that multiple instances can all share a single store.

3.3.5. Clustering: Indexing

Each repository instance uses indexes to help answer queries. When clustering a repository, the repository has to know whether it owns the indexes (in which case the repository will update the indexes to reflect all changes that originate from the local or remote repository instances) or whether indexes are shared (in which case the repository will update the indexes only when changes that are made with that repository instance). Note that even in the shared case, the index files might be local copies that are periodically cloned from a master set.

Local indexes are much easier to configure, but the disadvantage is that every repository is hereby updating its own indexes for every change (so there is duplicate work). This might cause a write-heavy system to become inundated with changes.

Shareable indexes are more difficult to configure (they require the use and proper configuration of JMS and/or JGroups), but are generally more capable of handling large amounts of updates.

3.3.6. Public APIs

- **`javax.jcr`** - This is the standard JCR 2.0 API, and it actually is not in our codebase but is available in Maven. It has no dependencies.
- **`modeshape-jcr-api`** - the hierarchical database's small extension to the standard JCR 2.0 API. This

public API was meant to be used by client applications that already use the JCR API, but it is entirely optional. Many of the interfaces extend the functionality offered by standard interfaces, so most of the time clients can cast standard JCR instances to these interfaces only when they need a method specific to the hierarchical database. A few interfaces are new concepts that clients might need to access. It only depends on the JCR API JAR. Note that the public API will only ever be modified in a backward-compatible fashion: while some methods might be deprecated at any time (though we do not anticipate doing so), changes that are not backward compatible (e.g., removal of deprecated methods) will only occur on major releases. This module also defines the Sequencer SPI, since sequencer implementations only need the JCR API and this public API.

3.3.7. Sequencers

All of the sequencer artifacts are named in a similar way: **`modeshape-sequencer-name`** . For example, the DDL sequencer is in the **`modeshape-sequencer-ddl`** module, while the WSDL sequencer is in the **`modeshape-sequencer-wsdl`** module.

The use of sequencers in a repository is entirely optional. And because nearly all of the sequencers depend upon third-party libraries, we've put each sequencer into a separate artifact so that only the required dependencies are included.

3.3.8. Core Modules

- **`modeshape-common`** - A simple set of domain-independent utilities and classes that are available for use in any other module. Some of these might be similar to those available in other third-party libraries, but were create and are maintained here to help minimize third-party dependencies (especially when small fractions of the third party libraries would be used). This includes the hierarchical database's framework for internationalization (I18n) and the logging framework that is a slight facade on top of several other logging systems, including SLF4J, Log4J, Logback, JDK logging. Sure, SLF4J is already a logging abstraction framework, but using our own abstraction makes it easier for developers to hook up the hierarchical database logging to their preferred framework (include the appropriate logging JAR on the classpath, or fallback to JDK logging) and it also allows the hierarchical database to enforce using only internationalized logging messages (except for debug and trace, which take string messages). Therefore, this module has no required dependencies, but will use one of the logging frameworks if they are available on the classpath.
- **`modeshape-schematic`** - A library for working with JSON and BSON documents, for storing them inside Infinispan, and for editing them in a way that allows for the changes to be recorded as a set of changes to the documents and atomically apply them. (The latter is what distinguishes this library from other JSON or BSON libraries.) Supports reading a document from JSON and/or BSON, and writing a document to JSON and/or BSON. The hierarchical database stores each node as a document inside Infinispan, and this library encapsulates all of the domain-independent logic for doing this. The module depends on several Infinispan artifacts.
- **`modeshape-jcr`** - The primary module that contains the hierarchical database engine and implementations of the standard JCR API and the hierarchical database's public API. It also defines several SPIs, including the Connector SPI (for federation) and the BinaryStore SPI (for storing binary values). It contains the file system connector and CND sequencer (since neither is dependent upon any other libraries and thus are too simple to be distinct artifacts).

3.3.9. Connectors

All of the connector artifacts are named in a similar way: **`modeshape-connector-name`** . For example, the Git connector is in the **`modeshape-connector-git`** module, while the CMIS connector is in the **`modeshape-connector-cmis`** module.

The use of federation (and thus connectors) in a repository is entirely optional. And because nearly all of the connectors depend upon third-party libraries, we've put each connector into a separate artifact so that only the required dependencies are included.

3.3.10. Web APIs

The hierarchical database has a number of web-based APIs that may optionally be used by remote clients to interact with one or more repositories.

- **REST Service** - a RESTful service that enables navigating, searching, modifying and deleting nearly any content in the repositories (see the detailed API documentation in the REST Service 3.x section). All representations are in JSON, XML or text form. Each operation creates a new session, fulfills the request, and then closes the session; sessions longer than a single request are not possible. Versioned content can be manipulated: if it is changed, it is checked out, modified, saved, and checked back in. However, the rest of the JCR functionality is not available. The WAR file is named **`modeshape-web-jcr-rest-war-<version>.war`** .
- **WebDAV Service** - exposes content via WebDAV, enabling WebDAV clients and operating systems to mount the repositories as network disk drives. This service exposes a small amount of the hierarchical database's functionality, and allows clients to basically navigate, download, and upload files and folders. The WAR file is named **`modeshape-web-jcr-webdav-war-<version>.war`** .
- **CMIS Service** - exposes an API that conforms to CMIS . The CMIS functionality exposes the ability to navigate, download, and upload folders and CMIS documents. The WAR file is named **`modeshape-web-jcr-cmis-war-<version>.war`** .

Each of these services can be independently deployed to a web or application server and in which the hierarchical database must be running. Each service talks to a single (local) **`ModeShapeEngine`** instance (typically found via JNDI) and will work with all of the repositories deployed to that engine.

3.3.11. JDBC Driver

The hierarchical database supports several query languages to allow client applications to find content independent of its hierarchical location. The JCR-SQL2 language is by far the most powerful, and the hierarchical database provides a JDBC driver that applications can use to query a repository (running in the same process or in a remote process where the REST service is available). The driver JAR is self-contained, making it pretty easy to incorporate into existing JDBC-aware applications.

3.4. Clustering

You can create a hierarchical database repository that stands alone and is self-contained, or you can create a cluster of repositories that all work together to ensure all content is accessible to each of the repositories.

When you create a cluster, a client talking to any of the processes in the cluster will see exactly the same content and the same events. In fact, from a client perspective, there is no difference between talking to a repository that is clustered versus one that is not.

The hierarchical database can be clustered in a variety of ways, but the biggest decision will be to determine where to store the content. Much of this flexibility comes from the power and flexibility of Infinispan, which can use a variety of topologies.

3.4.1. Local Caching

In a local mode, the hierarchical database is not clustered at all. This is the default, so if you do not tell both the database and Infinispan to cluster, each process will happily operate without communicating or sharing any content. Updates on one process will not be visible to any of the other processes.

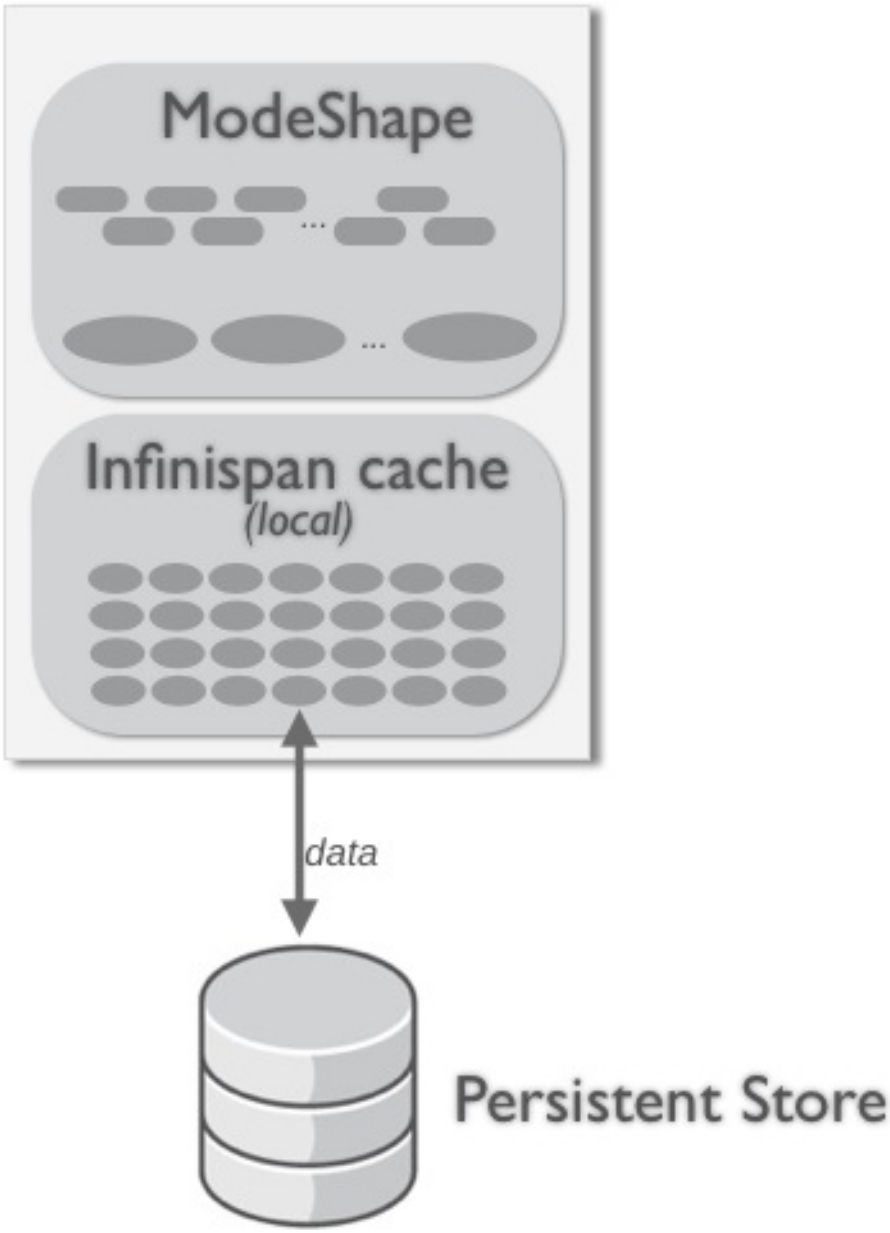


Figure 3.3. Local topology

Note that in the local, non-clustered topology data must be persisted to disk or some other system. Otherwise, if the hierarchical database process terminates, all data will be lost.

3.4.2. Replicated Clustering

The simplest clustering topology is to have each replicate all content across each member of the cluster. This means that each cluster member has its own storage for content, binaries, and indexes - nothing is shared. However, hierarchical database (and Infinispan) processes in the cluster communicate to ensure that locks are acquired as necessary and that committed changes in each member are replicated to all other members of the cluster.

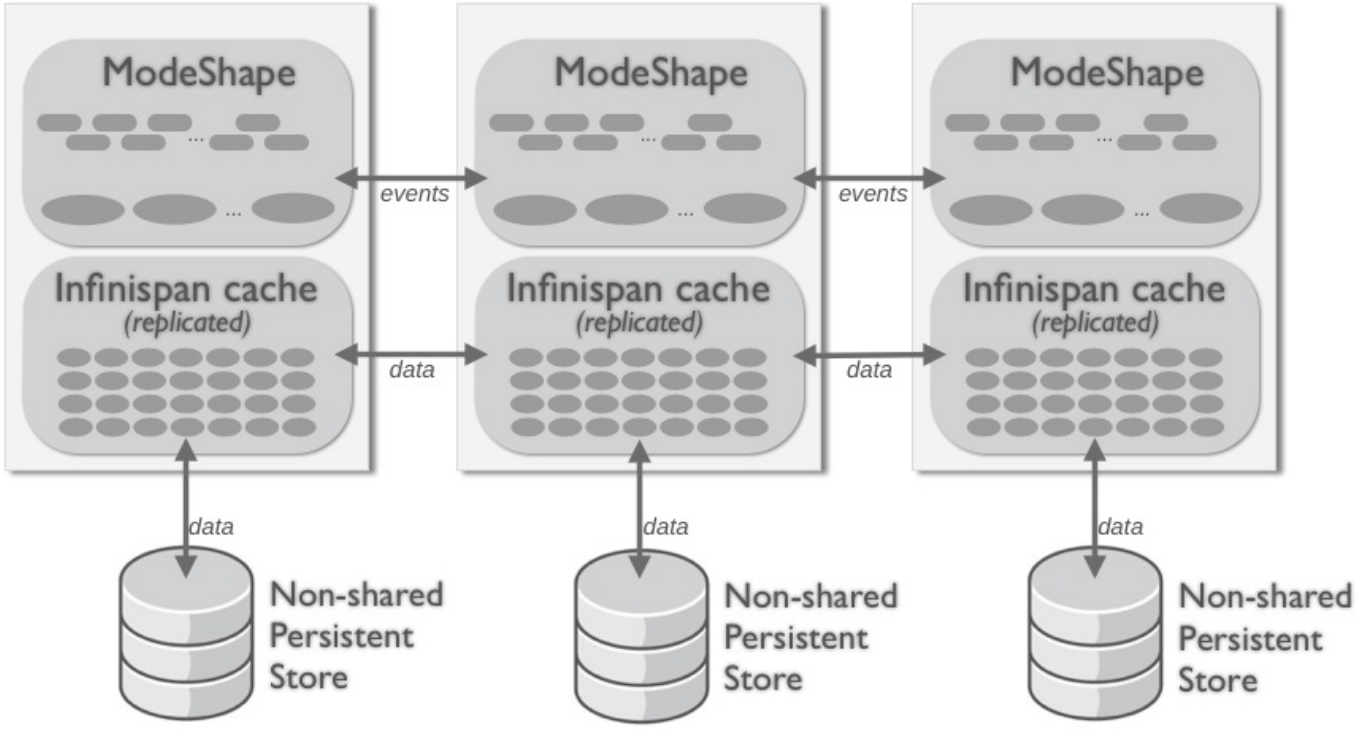


Figure 3.4. Replicated cluster topology with non-shared storage

The advantage of this topology is that each member of the cluster has a complete set of content, so all reads can be satisfied with locally-held data. This works great for small to medium-sized repositories. Additionally, because repositories share nothing, it is simple to add or remove cluster instances.

Replication works well for repositories with fairly large amounts of content, and with relatively few members of the cluster. Typically replication is used when you want clustering for fault-tolerance purpose, to handle larger workloads of clients, or when the hardware is not terribly powerful.

Note that the diagram above shows that each process has its own non-shared persistent store. Persistently storing the content is recommended, typically because all of the cluster members will likely be in a single data center and thus share some risk of common failure.

However, it is also possible to avoid persistent storage altogether, since the data is copied to multiple locations. But it is also possible for all of the members to share a persistent store, as long as that persistent store is transactional and capable of coordinating multiple concurrent operations. (An example of this is a relational database.)

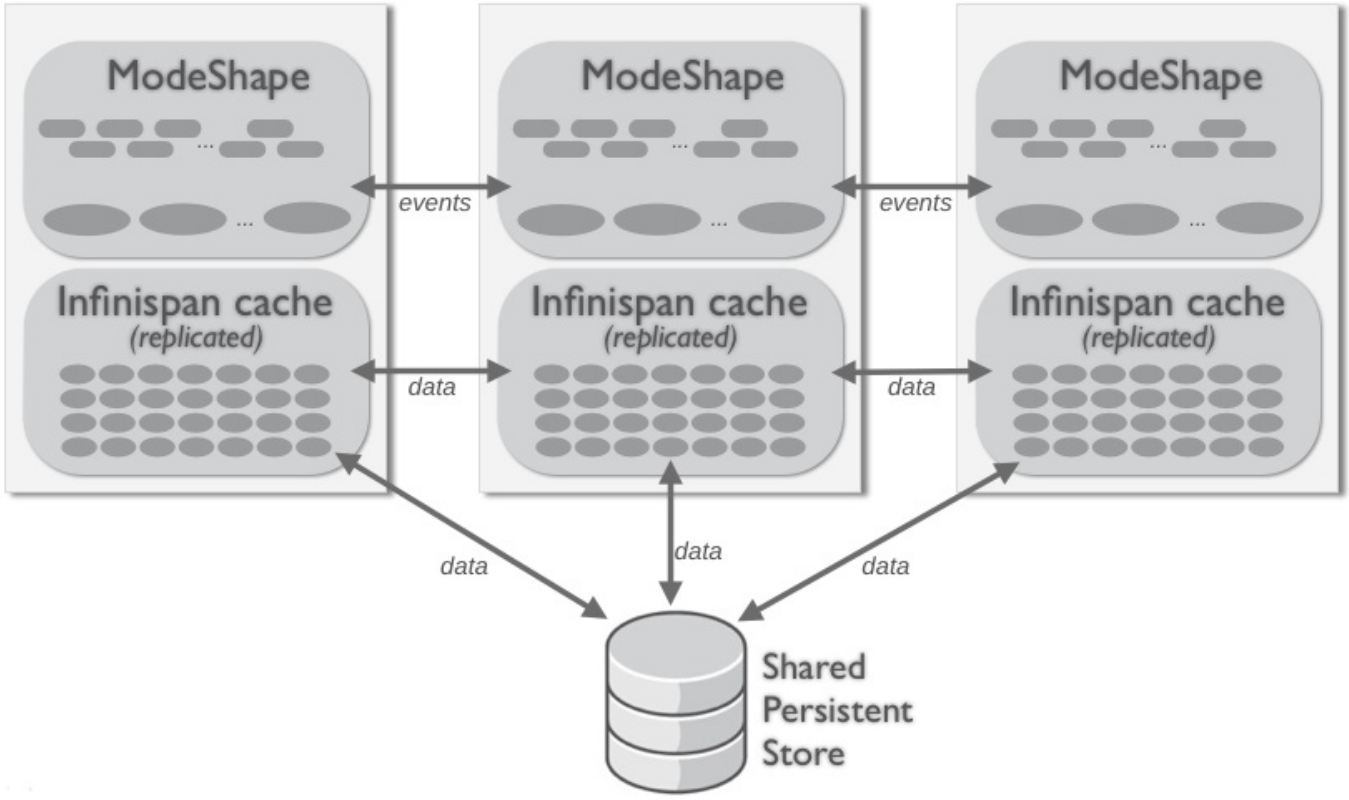


Figure 3.5. Replicated cluster topology with shared storage

3.4.3. Distributed Clustering

With larger cluster sizes, however, it is not as efficient for every member in the cluster to have a complete copy of all of the data. Additionally, the overhead of coordination of locks and inter-process communication starts to grow. This is when the distributed cluster topology becomes very advantageous.

In a distributed cluster, each piece of data is owned/managed by more than two members but fewer than the total size of the cluster. In other words, each bit of data is distributed across enough members so that no data will be lost if members catastrophically fail. And because of this, you can choose to not use persistent storage but to instead rely upon the multiple copies of the in-memory data, especially if the cluster is hosted in multiple data centers (or sites). In fact, a distributed cluster can have a very large number of members.

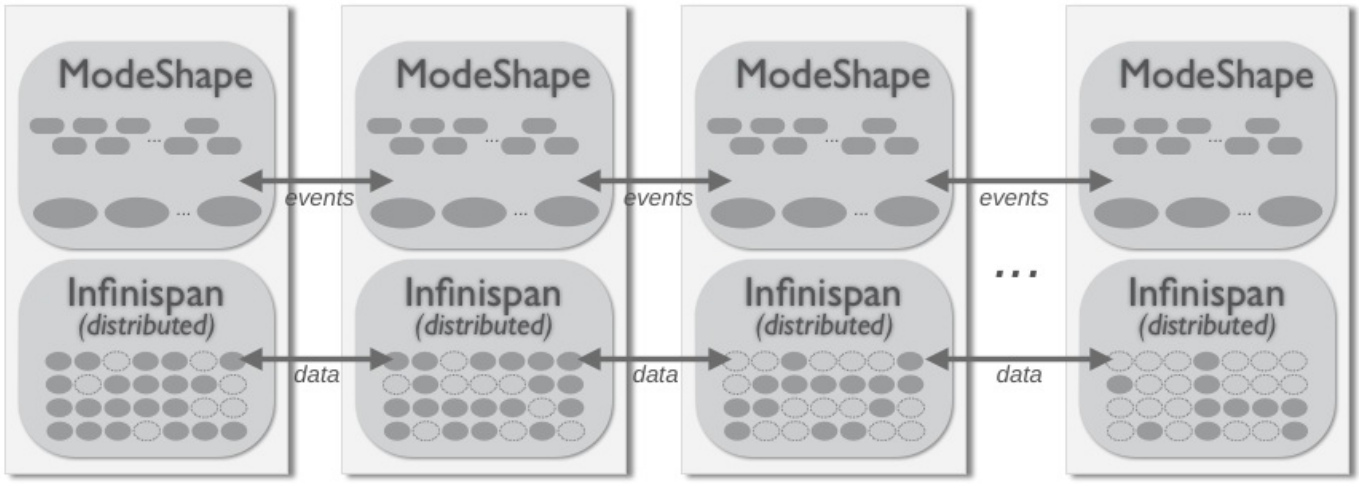


Figure 3.6. Distributed cluster topology

In this scenario, when a client requests some node or binary value, the hierarchical database (via Infinispan)

looks to see which member owns the node and forwards the request to that node. (Each repository instance maintains a cache of nodes, so subsequent reads of the same node will be very quick.)

3.4.4. Remote Clustering

The final topology is to cluster the hierarchical database as normal but to configure Infinispan to use a remote data grid. The benefit here is that the data grid is a self contained and separately managed system, and all of the specifics of the Infinispan configuration can be hidden by the data grid. Additionally, the data grid could itself be replicated or distributed across one or multiple physical sites.

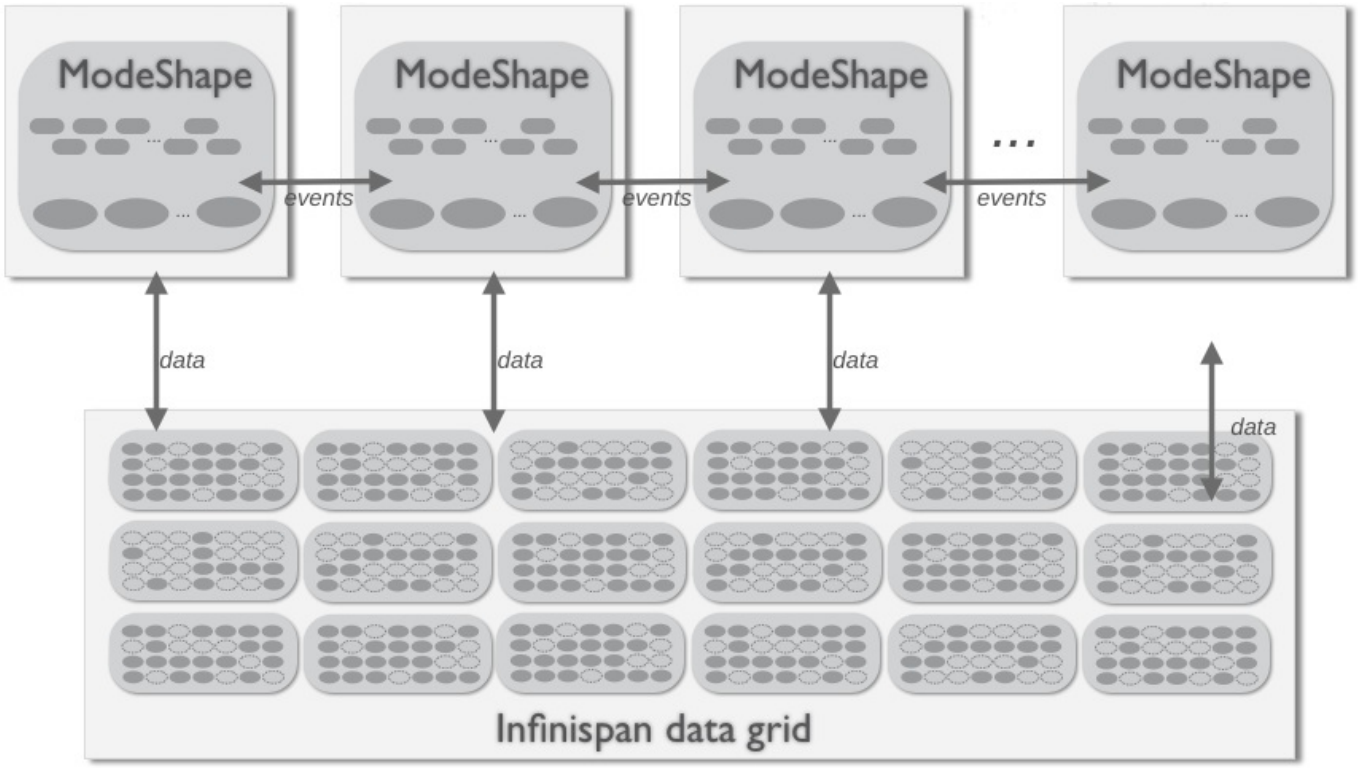


Figure 3.7. Cluster topology with remote (data grid) storage

Because of differences in the remote and local Infinispan interfaces, the only way to get this to work is to use a local cache with a remote cache store.

3.5. Sequencing

Many repositories are used (at least in part) to manage files and other artifacts, including service definitions, policy files, images, media, documents, presentations, application components, reusable libraries, configuration files, application installations, databases schemas, management scripts, and so on. Most JCR repository implementations will store those files and maybe index them for searching.

The hierarchical database sequencers can automatically unlock the structured information buried within all of those files, and this useful content derived from your files is then stored back in the repository where your client applications can search, access, and analyze it using the JCR API. Sequencing is performed in the background, so the client application does not have to wait for (or even know about) the sequencing operations.

The following diagram shows conceptually how these automatic sequencers do this.

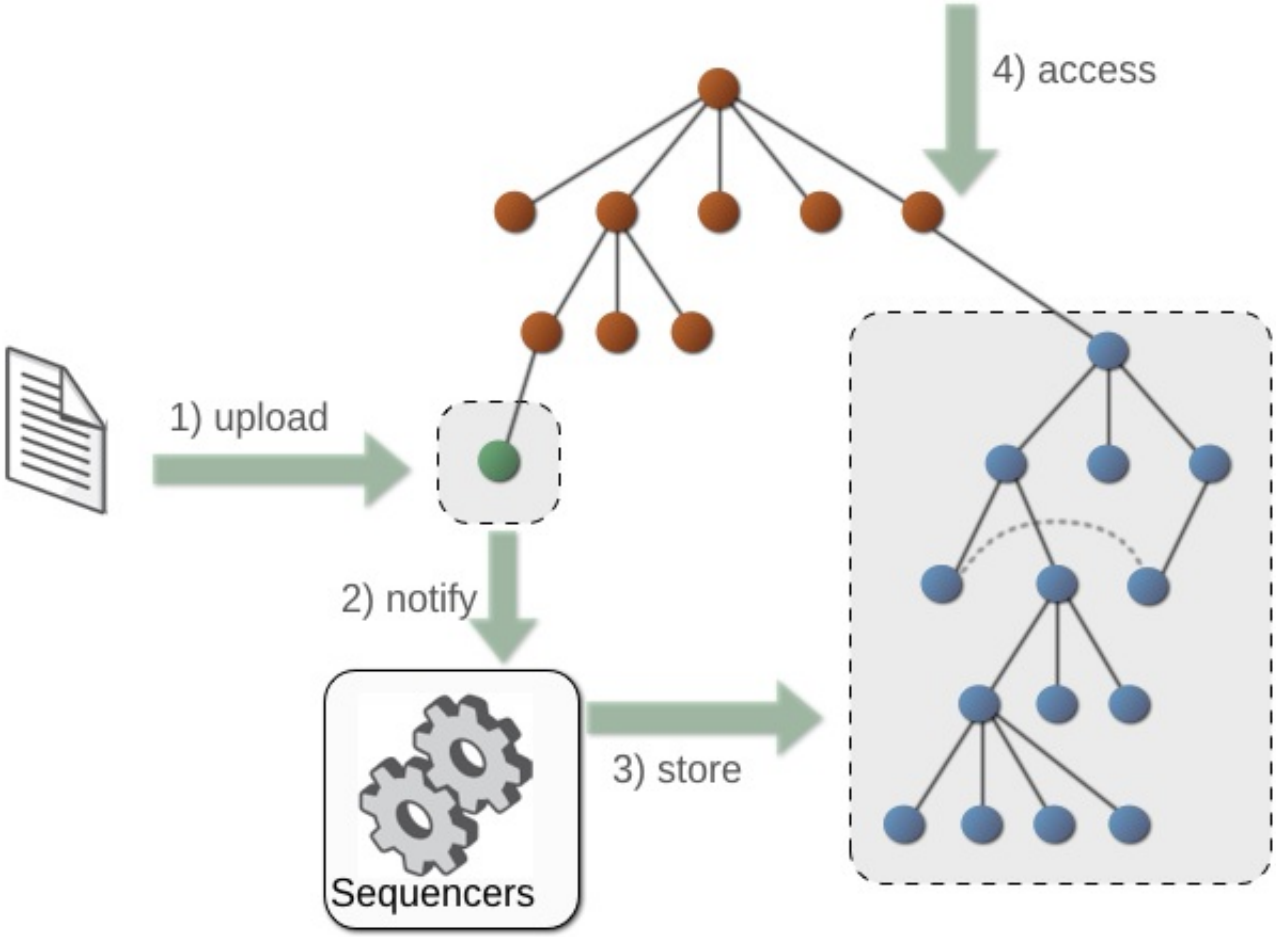


Figure 3.8. Sequencing Workflow

As of this release, your applications can use a session to explicitly invoke a sequencer on a specified property. We call these manual sequencers . Any generated output is included in the session's transient state, so nothing is persisted until the application calls **`session.save()`** .

Note

Prior to this release, the hierarchical database only had support for automatic sequencers.

3.5.1. Sequencers

Sequencers are POJOs that implement a specific interface, and when they are called they process the supplied input, extract meaningful information, and produce an output structure of nodes that somehow represents that meaningful information. This derived information can take almost any form, and it typically varies for each sequencer. For example, the hierarchical database comes with an image sequencer that extracts the simple metadata from different kinds of image files (e.g., JPEG, GIF, PNG, etc.). Another example is the Compact Node Definition (CND) sequencer that processes the CND files to extract and produce a structured representation of the node type definitions, property definitions, and child node definitions contained within the file. A third example is a sequencer that works on XML Schema Documents might parse the XSD content and generate nodes that mirror the various elements, and attributes, and types defined within the schema document.

Sequencers allow a repository to help you extract more meaning from the artifacts you already are managing, and makes it much easier for applications to find and use all that valuable information. All without your applications doing anything extra.

Each repository can be configured with any number of sequencers. Each one includes a name, the POJO class name, an optional classpath (for environments with multiple named classloaders), and any number of POJO-specific fields. Upon startup, the hierarchical database creates each sequencer by instantiating the POJO and setting all of the fields, then initializing the sequencer so it can register any namespaces or node type definitions.

There are two kinds of sequencers, automatic and manual .

3.5.2. Automatic Sequencers

An automatic sequencer has a path expression that dictates which content in the repository the sequencer is to operate upon. These path expressions are really patterns and look somewhat like simple regular expressions. When persisted content in the repository changes, the hierarchical database automatically looks to see which (if any) sequencers might be able to run on the changed content. If any of the sequencers do match, the hierarchical database automatically calls them by supplying the changed content. At that point, the sequencer then processes the supplied content and generates the output, and the hierarchical database then saves that generated output to the repository.

To use an automatic sequencer, add or change content in the repository that matches the sequencer's path expression. For example, if an XSD sequencer is configured for nodes with paths like **`/files//*.xsd`**, then upload a file into that location and save it. The hierarchical database will detect that the XSD sequencer should be called, and will do the rest. The generated content will magically appear in the repository.

3.5.3. Manual Sequencers

A manual sequencer is a sequencer that is configured without path expressions. Because no path expressions are provided, the hierarchical database cannot determine when/where these sequencers should be applied. Instead, manual sequencers are intended to be called by client applications.

For example, consider that a session uploaded a file at **`/files/schemas/Customers.xsd`**, and this node has a primary type of **`nt:file`**. (This means the file's content is stored in the **`jcr:data`** property the **`jcr:content`** child node.) The session has not yet saved any of this information, so it is still in the session's transient state. The following code shows how an XSD sequencer configured with name "XSD Sequencer" is manually invoked to place the generated content directly under the **`/files/schemas/Customers.xsd`** node (and adjacent to the **`jcr:content`** node):

```
Node fileNode = session.getNode("/files/schemas/Customers.xsd");
    Property content = fileNode.getProperty("jcr:content/jcr:data");
    Node output = fileNode; // could be anywhere!

    boolean success = session.sequence("XSD Sequencer", content, output);
```

The **`sequence(...)`** method returns **`true`** if the sequencer generated output, or **`false`** if the sequencer could not use the input and instead did nothing. Remember that when the **`sequence(...)`** does return, any generated output is only in the session's transient state and **`session.save()`** must be called to persist this state.

3.5.4. Built-in Sequencers

The hierarchical database comes with sequencer implementations for a variety of file types:

Input files	Derives
XML Documents	A node is created for each XML element, properties are created for each XML attribute, and each declared namespace is registered in the workspace.
XML Schema Documents (XSDs)	A node structure that represents the structure and semantics of the XSD, including the attribute declarations, element declarations, simple type definitions, complex type definitions, import statements, include statements, attribute group declarations, annotations, other components, and even attributes with a non-schema namespace.
WSDL 1.1 files	A node structure that represents the WSDL file's messages, port types, bindings, services, types (including embedded XML Schemas), documentation, and extension elements (including HTTP, SOAP and MIME bindings).
ZIP files	Extracts the files and folders contained in the archive file, representing them as **`nt:file`** and **`nt:folder`** nodes. The resulting files will be candidates for further sequencing.
Delimited and fixed-width text files	A simple node structure reflecting the rows of data fields.
DDL files	A node structure that represents the parsed data definition statements from SQL-92, Oracle, Derby, and PostgreSQL. The resulting structure is largely the same for all dialects, though some dialects have non-standard additions to their grammar that result in dialect-specific additions to the graph structure.
Red Hat JBoss Data Virtualization relational models	A rich node structure containing all the objects defined in the models, including the catalogs/schemas, tables, views, columns, primary keys, foreign keys, indexes, procedures, procedure results, extension properties, and data source information. The structure will also contain the select, update, insert and delete transformations in the case of virtual models.
Red Hat JBoss Data Virtualization virtual databases	A node structure that mirrors the relational model files, XSDs, and additional metadata. The resulting relational model files will be candidates for further sequencing.
Compact Node Definition files	

3.5.5. Configuring an Automatic Sequencer

Each sequencer must be configured to describe the areas or types of content that the sequencer is capable of handling. This is done by specifying these patterns using path expressions that identify the nodes (or node patterns) that should be sequenced and where to store the output generated by the sequencer.

A path expression consists of two parts: a selection criteria (or an input path) and an output path:

```
inputPath => outputPath
```

3.5.5.1. Input Path

The inputPath part defines an expression for the path of a node that is to be sequenced. Input paths consist o '/' separated segments, where each segment represents a pattern for a single node's name (including the same-name-sibling indexes) and '@' signifies a property name.

Example 3.1. Input Path Samples

Input Path	Description
/a/b	Match node "b" that is a child of the top level node "a". Neither node may have any same-name-siblings.
/a/*	Match any child node of the top level node "a".
/a/*.txt	Match any child node of the top level node "a" that also has a name ending in ".txt".
/a/b/@c	Match the property "c" of node "/a/b".
/a/b[2]	The second child named "b" below the top level node "a".
/a/b[2,3,4]	The second, third or fourth child named "b" below the top level node "a".
/a/b[*]	Any (and every) child named "b" below the top level node "a".
//a/b	Any node named "b" that exists below a node named "a", regardless of where node "a" occurs. Again, neither node may have any same-name-siblings.

With these simple examples, you can probably discern the most important rules. First, the '`*`' is a wildcard character that matches any character or sequence of characters in a node's name (or index if appearing in between square brackets), and can be used in conjunction with other characters (e.g., `*.txt`).

Second, square brackets (i.e., '`[`' and '`]`') are used to match a node's same-name-sibling index. You can put a single non-negative number or a comma-separated list of non-negative numbers. Use '0' to match a node that has no same-name-siblings, or any positive number to match the specific same-name-sibling.

Third, combining two delimiters (e.g., '`//`') matches any sequence of nodes, regardless of what their names are or how many nodes. Often used with other patterns to identify nodes at any level matching other patterns. Three or more sequential slash characters are treated as two.

Many input paths can be created using these simple rules. However, input paths can be more complicated. Here are some more examples:

Input Path	Description
/a/(b|c|d)	Match children of the top level node "a" that are named "b", "c" or "d". None of the nodes may have same-name-sibling indexes.
/a/b[c/d]	Match node "b" child of the top level node "a", when node "b" has a child named "c", and "c" has a child named "d". Node "b" is the selected node, while nodes "c" and "d" are used as criteria but are not selected.
/a(/(b|c|d|)/e)[f/g/@something]	Match node "/a/b/e", "/a/c/e", "/a/d/e", or "/a/e" when they also have a child "f" that itself has a child "g" with property "something". None of the nodes may have same-name-sibling indexes.

These examples show a few more advanced rules. Parentheses (i.e., '**(**' and '**)**') can be used to define a set of options for names, as shown in the first and third rules. Whatever part of the selected node's path appears between the parentheses is captured for use within the output path, similar to regular expressions. Thus, the first input path in the previous table would match node **/a/b**, and **b** would be captured and could be used within the output path using **$1**, where the number used in the output path identifies the parentheses. Here are some examples of what's captured by the parenthesis and available for use in the output path:

Input Path	$1	$2	$3
/a/(b|c|d)	"b" or "c" or "d"	n/a	n/a
/a/b[c/d]	n/a	n/a	n/a
/a(/(b|c|d|)/e)[f/g/@something]	"/b/e" or "/c/e" or "/d/e" or "/e"	"b" or "c" or "d" or ""	n/a

Square brackets can also be used to specify criteria on a node's properties or children. Whatever appears in between the square brackets does not appear in the selected node. This distinction between the **selected path** and the **changed path** becomes important when writing custom sequencers .

3.5.5.2. Output Paths

The outputPath part of a path expression defines where the content derived by the sequencer should be stored. Typically, this points to a location in a different part of the repository, but it can actually be left off if the sequenced output is to be placed directly under the selected node. The output path can also use any of the capture groups used in the input path.

3.5.5.3. Workspaces in Input and Output Paths

So far, we've talked about how input paths and output paths are independent of the workspace. However, there are times when it is desirable to configure sequencers to only work against content in a specific workspace. In these cases, it is possible to specify the workspace names before the path. For example:

Input Path	Description
:default:/a/(b|c|d)	Match nodes in the "default" workspace within any source that are children of the top level node "a" and named "b", "c" or "d". None of the nodes may have same-name-sibling indexes.
:/a/(b|c|d)	Match nodes in any within any source that are children of the top level node "a" and named "b", "c" or "d". None of the nodes may have same-name-sibling indexes. (This is equivalent to the path **/a/(b|c|d)**.)

Again, the rules are pretty straightforward. You can leave off the workspace name, or you can prepend the path with **workspaceNamePattern:**, where **workspaceNamePattern** is a regular-expression pattern used to match the applicable workspace names. A blank pattern implies any match, and is a shorthand notation for the '**.***' regular expression. Note that the repository names may not include forward slashes (e.g., '**/**') or colons (e.g., '**:**').

3.5.5.4. Example Path Expression

Let's look at an example sequencer path expression:

```
default://(\*.(jpg\|jpeg\|gif\|bmp\|pcx\|png)\[*])\[/jcr:content@jcr:data]
=> meta:/images/\$1
```

This matches a changed **jcr:data** property on a node named **jcr:content[1]** that is a child of a node whose name ends with **.jpg**, **.jpeg**, **.gif**, **.bmp**, **.pcx**, or **.png** (that may have any same-name-sibling index) appearing at any level in the **default** workspace. Note how the selected path capture the filename (the segment containing the file extension), including any same-name-sibling index. This filename is then used in the output path, which is where the sequenced content is placed under the **/images** node in the **meta** workspace.

So, consider a PNG image file is stored in the **default** workspace in a repository configured with an image sequencer and the aforementioned path expression, and the file is stored at **/jsmith/photos/2011/08/09/reunion.png** using the standard **nt:file** pattern. This means that an **nt:file** node named **reunion.png** is created at the designated path, and a child node named **jcr:content** will be created with primary type of **nt:resource** and a **jcr:data** binary property (at which the image file's content is store).

When the session is saved with these changes, the hierarchical database discovers that the

```
{{/jsmith/photos/2011/08/09/reunion.png/jcr:content/jcr:data}}
```

property satisfies the criteria of the sequencer, and calls the sequencer's **execute(...)** method with the selected node, input node, input property and output node of **/images** in the **meta** workspace. When the **execute()** method completes successfully, the session with the change in the **meta** workspace are saved and the content is immediately available to all other sessions using that workspace.

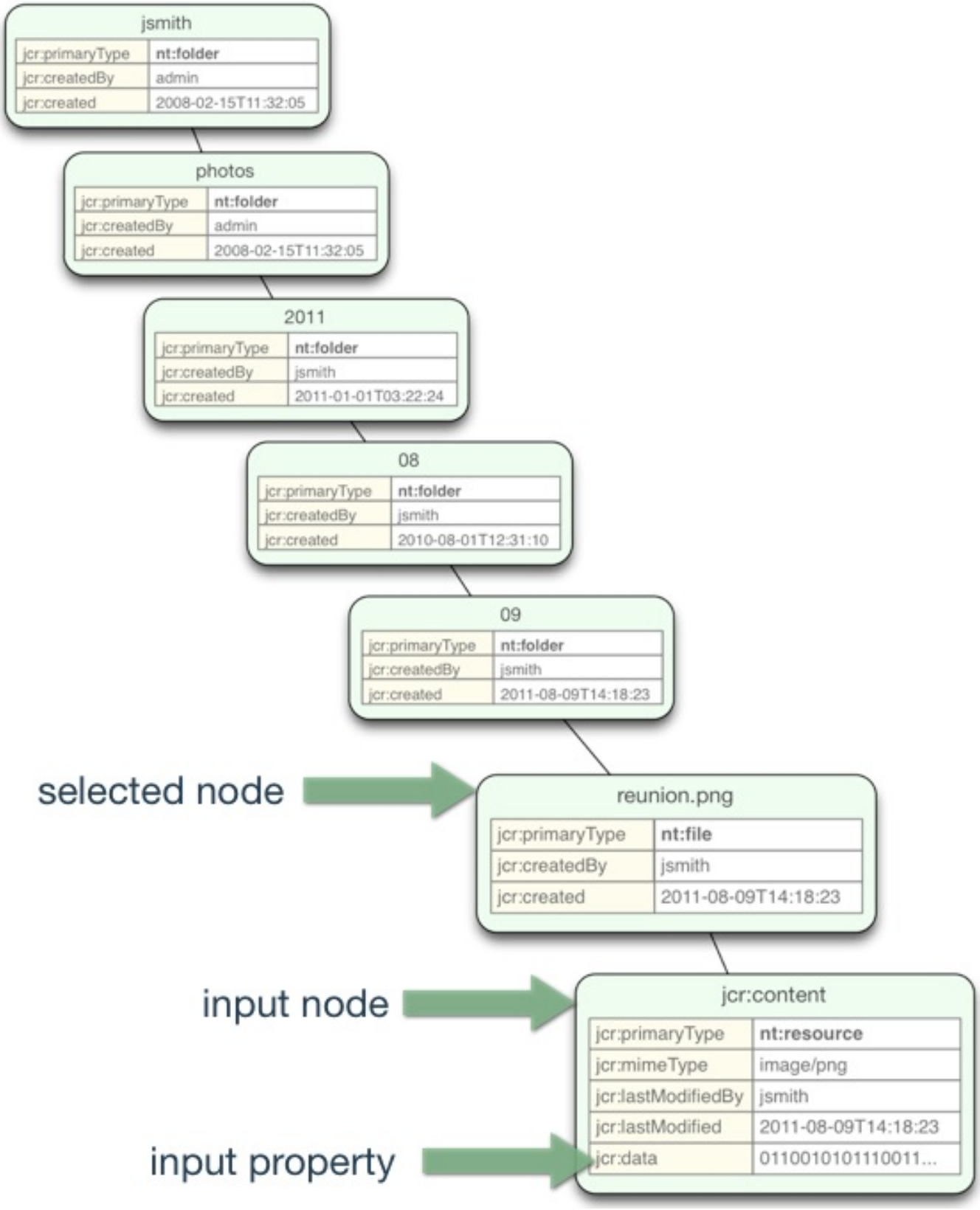


Figure 3.9. Sequencing an Uploaded File

3.5.5.5. Observing Automatic Sequencing

When your application creates or uploads content that will kick off a sequencing operation, the sequencing is actually done asynchronously. If you want to be notified when the sequencing is complete, you can use the observation feature to register a listener for the sequencing event.

The first step is to create a class that implements **`javax.jcr.observation.EventListener`**. Normally

this is pretty easy, but in our case we want to block until the listener is notified via a separate thread. An easy way to do this is to use a **`java.util.concurrent.CountDownLatch`** , and to count down the latch as soon as we get our event. (If we carefully register the listener using criteria for only the sequencing output we're interested in, we'll know we'll only receive one event.)

Here's our implementation that captures from the first event whether the sequencing was successful and the path of the output node, and then counts down the latch:

```
public class SequencingListener implements
javax.jcr.observation.EventListener {
    private final CountDownLatch latch;
    private volatile String sequencedNodePath;
    private volatile boolean successfulSequencing;
    public SequencingListener( CountDownLatch latch ) {
        this.latch = latch;
    }

    @Override
    public void onEvent( javax.jcr.observation.EventIterator events ) {
        if ( sequencedNodePath != null ) return;
        try {
            javax.jcr.observation.Event event =
(javax.jcr.observation.Event)events.nextEvent();
            this.sequencedNodePath = event.getPath();
            this.successfulSequencing = event.getType() ==
org.modeshape.jcr.observation.Event.Sequencing.NODE_SEQUENCED;
            latch.countDown();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    public boolean isSequencingSuccessful() {
        return this.successfulSequencing;
    }

    public String getSequencedNodePath() {
        return sequencedNodePath;
    }
}
```

We could then register this using the public API:

```
Session session = ...
ObservationManager observationManager =
session.getWorkspace().getObservationManager();

String outputPath = .. // the path at or below which the output is to be
placed
// Listen for sequencing completion or failure events, via the ALL type ...
int eventTypes = org.modeshape.jcr.api.observation.Event.Sequencing.ALL;
boolean isDeep = true; // if outputPath is ancestor of the sequencer output,
false if identical
String[] uuids = null; // Don't care about UUIDs of nodes for sequencing
events
String[] nodeTypes = null; // Don't care about node types of output nodes
```

```
for sequencing events
boolean noLocal = false; // We do want events for sequencing happen locally
(as well as remotely)

// Now create a listener implementation that will be called when the event is
here ...
CountDownLatch latch = new CountDownLatch(1);
SequencingListener listener = new SequencingListener(latch);
observationManager.addEventListener(listener,eventTypes,outputPath,isDeep,
                                    uuids, nodeTypes, noLocal);

// Now, block until the latch is decremented (by the listener) or when our
max wait time is exceeded
latch.await(15, TimeUnit.SECONDS);

if ( listener.isSequencingSuccessful() ) {
   // Grab the output produced by the sequencer ...
} else {
   // Handle the failure ...
}
```

Chapter 4. Using the Hierarchical Database with Red Hat JBoss EAP

4.1. Configuring the Hierarchical Database

4.1.1. Hierarchical Database Configuration

Although JBoss Data Virtualization comes with a hierarchical database already configured, this topic describes the steps required to configure another if so desired.

Procedure 4.1. Task

1. **Start the JBoss EAP server**

   Start JBoss EAP in standalone mode with the configuration of your choice. For example, the following starts with the **`standalone.xml`** configuration file:

   ```
   $ bin/standalone.sh -c=standalone.xml
   ```

2. **Start the JBoss EAP Management CLI**

   You can use the JBoss EAP command line interface (CLI) tool to directly manipulate the configuration of the running server. If the server is running in domain mode, the CLI immediately propagates the changes to all the servers. Start the CLI and connect to your server as shown below:

   ```
   $ ./bin/jboss-cli.sh
   You are disconnected at the moment. Type 'connect' to connect to the
   server or 'help' for the list of supported commands.
   [disconnected /] connect
   [standalone@localhost:9999 /]
   ```

3. **Add a hierarchical database subsystem**

   Add the subsystem to the current configuration as shown below:

   ```
   [standalone@localhost:9999 /] /extension=org.modeshape:add()
   {"outcome" => "success"}
   [standalone@localhost:9999 /] ./subsystem=modeshape:add
   {"outcome" => "success"}
   ```

   This updates the configuration's XML file (in this case **`standalone.xml`**) immediately.

4. **Add a hierarchical database repository**

   Before adding a repository, add or configure the JBoss EAP resources for the repository to use.

   a. **Add an Infinispan cache**

      Each hierarchical database repository stores its content in an Infinispan cache. The following steps show how to put this cache in a new cache container called **`modeshape`**, which you can use for other repositories:

```
[standalone@localhost:9999 /] /subsystem=infinispan/cache-
container=modeshape:add
{"outcome" => "success"}
```

Once you have your container, here is how you can define a local cache named **`sample`** that uses non-XA transactions and persists all content immediately to the **`modeshape/store/sample`** directory under the **`standalone/data`** directory:

```
      [standalone@localhost:9999 /] /subsystem=infinispan/cache-
container=modeshape/local-cache=sample:add
{"outcome" => "success"}
[standalone@localhost:9999 /] /subsystem=infinispan/cache-
container=modeshape/local-
cache=sample/transaction=TRANSACTION:add(mode=NON_XA)
{
    "outcome" => "success",
    "response-headers" => {
        "operation-requires-reload" => true,
        "process-state" => "reload-required"
    }
}
[standalone@localhost:9999 /] /subsystem=infinispan/cache-
container=modeshape/local-cache=sample/file-
store=FILE_STORE:add(path="modeshape/store/sample",relative-
to="jboss.server.data.dir",passivation=false,purge=false)
{
    "outcome" => "success",
    "response-headers" => {"process-state" => "reload-required"}
}
```

These commands run successfully, however, the last few may require a reload of the Infinispan service. This means that your changes are saved to the configuration, but these few may not take effect until the next restart or until you explicitly perform the reload:

```
[standalone@localhost:9999 /] :reload
{
    "outcome" => "success",
    "response-headers" => {"process-state" => "reload-required"}
}
[standalone@localhost:9999 /] :reload
{"outcome" => "success"}
```

b. **Add the repository**

After defining the services for the repository to use, here is how you can define a repository called **`sample`**:

```
[standalone@localhost:9999 /]
./subsystem=modeshape/repository=sample:add(security-
domain="modeshape-security",cache-name="sample",cache-
container="modeshape")
{"outcome" => "success"}
```

This command configures the **sample** repository to use the sample Infinispan cache in the **modeshape** cache container, and to use the **modeshape-security** security domain created earlier. Restart is not required after defining a repository. Most of the administrative operations take effect immediately even when applications are actively using the repository.

You need not define the ***security-domain="modeshape-security"*** attribute because the repository uses a security domain with that name by default. Also, by default the repository tries to use an Infinispan cache with the name "modeshape" which is same as the repository in the cache container. You can specify these, but any attributes that match the default value will not be serialized to the XML configuration file.

4.1.2. Advanced Repository Configuration

Procedure 4.2. Task

1. You can view the complete definition of a repository at any point by running the following command in the Management CLI:

```
[standalone@localhost:9999 /]
/subsystem=modeshape/repository=sample:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {
        "allow-workspace-creation" => true,
        "anonymous-roles" => undefined,
        "anonymous-username" => "<anonymous>",
        "binary-storage" => undefined,
        "cache-container" => "modeshape",
        "cluster-name" => undefined,
        "cluster-stack" => undefined,
        "default-workspace" => "default",
        "enable-monitoring" => true,
        "index-storage" => undefined,
        "indexing-analyzer-classname" =>
"org.apache.lucene.analysis.standard.StandardAnalyzer",
        "indexing-analyzer-module" => undefined,
        "indexing-async-max-queue-size" => 0,
        "indexing-async-thread-pool-size" => 1,
        "indexing-batch-size" => -1,
        "indexing-mode" => "SYNC",
        "indexing-reader-strategy" => "SHARED",
        "indexing-thread-pool" => "modeshape-workers",
        "jndi-name" => undefined,
        "minimum-binary-size" => 4096,
        "predefined-workspace-names" => undefined,
        "rebuild-indexes-upon-startup" => "IF_MISSING",
        "security-domain" => "modeshape-security",
        "sequencer" => undefined,
        "use-anonymous-upon-failed-authentication" => false
    }
}
```

   This shows all the attributes including the ones that are not set, or set to their default values.

2. To view details about each attribute and child, use the **:read-resource-description()**

command. For example:

```
[standalone@localhost:9999 /]
/subsystem=modeshape/repository=sample:read-resource-
description(recursive=true)
{
    "outcome" => "success",
    "result" => {
        "description" => "ModeShape repository",
        "attributes" => {
            "cache-name" => {
                "type" => STRING,
                "description" => "The name of the cache that is to be
used for storing this repository's content",
                "expressions-allowed" => false,
                "nillable" => true,
                "min-length" => 1L,
                "max-length" => 2147483647L,
                "access-type" => "read-write",
                "storage" => "configuration",
                "restart-required" => "resource-services"
            },
            "cache-container" => {
                "type" => STRING,
                "description" => "The name of the cache container
that contains the cache to be used for storing this repository's
content",
                "expressions-allowed" => false,
                "nillable" => true,
                "min-length" => 1L,
                "max-length" => 2147483647L,
                "access-type" => "read-write",
                "storage" => "configuration",
                "restart-required" => "resource-services"
            },
            "jndi-name" => {
                "type" => STRING,
                "description" => "The optional alias in JNDI where
this repository is to be registered, in addition to
'jcr/{repositoryName}",
                "expressions-allowed" => false,
                "nillable" => true,
                "min-length" => 1L,
                "max-length" => 2147483647L,
                "access-type" => "read-write",
                "storage" => "configuration",
                "restart-required" => "resource-services"
            },
...
```

Here, the output shows the description of each attribute, the criteria for valid values, whether expressions such as system variables are allowed or not, and whether a restart is required or not before changes take effect.

Most of the attributes have defaults, but the descriptions do not list some of the defaults because the defaults are functions of other attributes. For example, every repository is registered in JNDI under **`jcr/repositoryName`**, and also under the JNDI name explicitly set with the **`jndi-name`** attribute.

4.1.3. Repository Attributes

The following table contains the list of all the attributes for a hierarchical database repository:

Table 4.1. Repository Attributes

Attribute Name	Description
allow-workspace-creation	Specifies whether authenticated and authorized JCR users can create additional workspaces beyond the predefined, system, and default workspaces. The default value is 'true'. Set this to 'false' when you need to fix the workspaces.
anonymous-roles	The list of names (of type String) of the roles for all anonymous users. An empty String in the role name results in disabling the logins. By default, anonymous users are given all roles: 'connect', 'readonly', 'readwrite', and 'admin'.
anonymous-username	The username for all anonymous users. The username <anonymous> is used by default.
cache-container	The name of the Infinispan cache container containing the cache. If not provided, the "modeshape" cache container is used.
cache-name	The name of the Infinispan cache where repository content is stored. If not provided, the repository name is used for the cache name.
cluster-name	Defines the name of the communication channel used to share events amongst all repository instances in the cluster. By default there is no value. This means that the repository is not participating in a cluster.
cluster-stack	Specifies the name of the JGroups stack used by the repository to create a channel for events when the repository is clustered. By default there is no value. This means that the repository is not participating in a cluster.
default-workspace	The name of the workspace to be used when sessions are created without specifying an explicit workspace name. By default, the "default" workspace name is used.
enable-monitoring	Specifies whether the repository is to maintain the metrics that can be used to monitor the performance and activities. The default value is 'true', which means that the monitoring is enabled.
indexing-analyzer-classname	The fully-qualified name of the Lucene analyzer implementation class. The default value is **`org.apache.lucene.analysis.standard.StandardAnalyzer`**.

Attribute Name	Description
indexing-analyzer-module	The name of the module that contains the specified analyzer class. No value is specified by default, which means that the class is visible to the hierarchical database engine.
indexing-async-max-queue-size	The maximum size of the queue used for asynchronous indexing. By default the value is '0'. The value is ignored if synchronous indexing is enabled.
indexing-async-thread-pool-size	The size of the thread pool used for asynchronous indexing. By default the value is '1'. The value is ignored if synchronous indexing is enabled.
indexing-batch-size	The size of the indexing batches. The default value is '-1', which means the batch sizes are unlimited.
indexing-mode	The concurrency mode for indexing. The valid values are 'SYNC' and 'ASYNC'.
indexing-reader-strategy	The strategy for sharing (or not sharing) index readers. The valid values are 'SHARED' and 'NOT_SHARED'.
indexing-thread-pool	The name of the thread pool that the repository indexing system should use. The default value is 'modeshape-workers'.
jndi-name	The repository is always bound in JNDI to the name 'jcr/{repositoryName}', however you can use this attribute to specify an additional location in JNDI where the repository is to be registered.
minimum-binary-size	The size threshold that dictates whether String and binary values should be stored in the binary store. String and binary values smaller than this value are stored with the node, whereas String and binary values with a size equal to or greater than this limit are stored separately from the node in the binary store, keyed by the SHA-1 hash of the value. This is a space and performance optimization that stores each unique large value only once. The default value is '4096' bytes, or 4 kilobytes.
predefined-workspace-names	The names of the workspaces that the repository ensures exist (or create if necessary) when the repository starts up.
rebuild-indexes-upon-startup	Specifies whether the indexes need to be rebuilt immediately when each process starts up. Valid values are 'IF_MISSING' , 'ALWAYS' or 'NEVER. By default the value is 'IF_MISSING'.
rebuild-upon-startup-mode	Specifies whether index rebuilding at startup should be synchronous or asynchronous. Valid values are 'SYNC' and 'ASYNC'. The default value is 'SYNC'.
rebuild-upon-startup-include-system-content	Specifies whether the system content area (the nodes below /jcr:system) should be indexed or not when rebuilding indexes at startup. The default value is 'FALSE'
security-domain	The name of the security domain that should be used for JAAS authentication. The default value is 'modeshape-security'

Attribute Name	Description
use-anonymous-upon-failed-authentication	Indicates that the failed authentication attempts will not result in a **`javax.jcr.LoginException`**, but will instead fall back to anonymous access. If anonymous access is not enabled, then failed login attempts throw a **`LoginException`**. The default value is 'false'.
default-initial-content	The file which should be treated as the default initial content imported into all workspace.
workspaces-initial-content	A set of (workspaceName, initial content file) pairs, which defines the custom initial content files for each workspace.
node-types	A sequence of node-type elements, where the value of each element represents a path to a CND file. This file should be imported at repository startup.
external-sources	A sequence of source elements, where each element contains the definition of an external source

4.1.4. Sequencers

Sequencers are POJOs that implement a specific interface. Sequencers allow a hierarchical database repository to help you extract more meaning from the artifacts you already are managing, and makes it much easier for applications to find and use all that valuable information. You can configure a repository with any number of sequencers and each one applies to content in the repository matching specific patterns. When content in the repository changes, it is automatically checked to see which sequencers might be able to run on the changed content. If any of the sequencers match, the hierarchical database automatically calls them by supplying the changed content. At that point, the sequencer's job is to process the supplied input, extract meaningful information, and write that derived information back into the repository where it can be accessed, searched and used by your client applications.

The derived information can take almost any form, and it typically varies for each sequencer. For example, an image sequencer is provided that extracts the simple metadata from different kinds of image files (such as JPEG, GIF, and PNG).

Another example is the Compact Node Definition (CND) sequencer that processes the CND files to extract and produce a structured representation of the node type definitions, property definitions, and child node definitions contained within the file.

4.1.5. Adding and Removing Sequencers

You can use the CLI to dynamically add and remove sequencers.

Procedure 4.3. Task

1. Add a sequencer to the sample repository that operates against comma-separated value (CSV) files uploaded under the ***`/files`*** node as shown below:

```
[standalone@localhost:9999 /]
/subsystem=modeshape/repository=sample/sequencer=delimited-text-
sequencer:add(
classname="org.modeshape.sequencer.text.DelimitedTextSequencer",
module="org.modeshape.sequencer.text", path-expressions=
["/files(//*.csv[*])/jcr:content[@jcr:data] =>
/derived/text/delimited/$1"], properties=[{ "splitPattern"=>"," }])
```

```
{"outcome" => "success"}
[standalone@localhost:9999 /]
/subsystem=modeshape/repository=sample/sequencer=delimited-text-
sequencer:read-resource()
{
    "outcome" => "success",
    "result" => {
        "classname" =>
"org.modeshape.sequencer.text.DelimitedTextSequencer",
        "module" => "org.modeshape.sequencer.text",
        "path-expressions" =>
["/files(//*.csv[*])/jcr:content[@jcr:data] =>
/derived/text/delimited/$1"],
        "properties" => [{"splitPattern" => ","}]
    }
}
```

This sequencer has an additional splitPattern property that specifies the delimiter.

2. To remove a sequencer, invoke the remove operation on the appropriate item as shown below:

```
[standalone@localhost:9999 /]
/subsystem=modeshape/repository=sample/sequencer=delimited-text-
sequencer:remove()
{"outcome" => "success"}
```

4.1.6. Specify Index Storage

Procedure 4.4. Task

1. To specify where indexes are stored, add the index storage resource to your configuration as shown below:

```
[standalone@localhost:9999 /]
/subsystem=modeshape/repository=sample/configuration=index-
storage:add()
{"outcome" => "success"}
```

2. Once you add the index storage node, add the storage type with required optional parameters as shown below:

```
[standalone@localhost:9999 /]
/subsystem=modeshape/repository=sample/configuration=index-
storage/storage-type=master-file-index-storage:add(connection-factory-
jndi-name=conn-name,queue-jndi-name=queue-name, path=/somepath,
source-path=/someotherpath)
{"outcome" => "success"}
```

4.1.7. Specify Binary Storage

Procedure 4.5. Task

1. To specify where large binary values are stored, you need to first add the binary storage resource to your configuration as shown below:

```
[standalone@localhost:9999 /]
/subsystem=modeshape/repository=sample/configuration=binary-
storage:add()
{"outcome" => "success"}
```

2. Once you add the binary storage node, add the storage type with the required optional parameters as shown below:

```
[standalone@localhost:9999 /]
/subsystem=modeshape/repository=sample/configuration=binary-
storage/storage-type=file-binary-storage:add(path=/somepath)
{"outcome" => "success"}
```

4.1.8. Configure Composite Binary Stores

Composite binary stores are different from the rest of the standard binary stores, as they can aggregate any number of standard binary stores.

Procedure 4.6. Task

1. Configure composite binary stores via CLI as shown below:

```
[standalone@localhost:9999 /]
/subsystem=modeshape/repository=sample/configuration=binary-
storage/storage-type=composite-binary-storage:add()
```

Ensure that each nested store has a **`store-name`** property that is unique within the composite store and that the appropriate **`resource-container`** is used when adding the store. Corresponding to each of the standard binary stores, the following **`resource-containers`** are available:

- nested-storage-type-file - for file system binary stores
- nested-storage-type-cache - for cache binary stores
- nested-storage-type-db - for database binary stores
- nested-storage-type-custom - for custom (user defined) binary stores

2. Add a file system binary store to a composite store as shown below:

```
[standalone@localhost:9999 /]
/subsystem=modeshape/repository=sample/configuration=binary-
storage/storage-type=composite-binary-storage/nested-storage-type-
file=filesystem1:add(store-name=filesystem1, path="/somepath")
```

You can remove a file system binary store from a composite store as shown below:

```
[standalone@localhost:9999 /]
/subsystem=modeshape/repository=sample/configuration=binary-
storage/storage-type=composite-binary-storage/nested-storage-type-
file=filesystem1:remove()
```

4.1.9. Add and Remove Authentication and Authorization Providers

You can use the CLI to dynamically add and remove custom authentication and authorization providers.

Procedure 4.7. Task

1. If your **`org.modeshape.jcr.security.AuthorizationProvider`** implementation is named **`org.example.MyAuthProvider`** and is added to a new **`org.example.auth`** module, then use the following command to add this provider to the "sample" repository:

```
[standalone@localhost:9999 /]
/subsystem=modeshape/repository=sample/authenticator=custom:add(classn
ame="org.example.MyAuthProvider", module="org.example.auth")
{"outcome" => "success"}
[standalone@localhost:9999 /]
/subsystem=modeshape/repository=sample/authenticator=jaas:read-
resource()
{
    "outcome" => "success",
    "result" => {
        "classname" => "org.modeshape.jcr.security.JaasProvider",
        "module" => "org.modeshape",
        "properties" => undefined
    }
}
```

2. To remove an authentication provider, invoke the "remove" operation on the appropriate item as shown below:

```
[standalone@localhost:9999 /]
/subsystem=modeshape/repository=sample/authenticator=custom:remove()
{"outcome" => "success"}
```

4.1.10. Set Instance-Level Fields on Provider Instances

Instance-level fields can be set on the provider instances.

Procedure 4.8. Task

- You can set the ***`auth-domain`*** field on the MyAuthProvider instance to the String value "global". To do this, add them via the ***`properties`*** parameter, which is a list of documents that each contain a single name-value pair:

```
[standalone@localhost:9999 /]
/subsystem=modeshape/repository=sample/authenticator=custom:add(classname=
"org.example.MyAuthProvider", module="org.example.auth", properties=[
{"foo"=>"bar"}, {"baz"=>"bam"} ] )
{"outcome" => "success"}
/subsystem=modeshape/repository=sample/authenticator=custom:read-
resource()
{
    "outcome" => "success",
    "result" => {
        "classname" => "org.example.MyAuthProvider",
        "module" => "org.example.auth",
        "properties" => [
```

```
            {"foo" => "bar"},
            {"baz" => "bam"}
        ]
    }
}
```

4.1.11. Add JDBC Data Source

Prerequisities

- Before adding a data source, add the driver as shown below:

```
[standalone@localhost:9999 /] /subsystem=datasources/jdbc-
driver=modeshape-driver:add(driver-name="modeshape-driver", driver-module-
name="org.modeshape.jdbc", driver-class-
name="org.modeshape.jdbc.LocalJcrDriver")
{"outcome" => "success"}
```

Procedure 4.9. Task

- Add the JDBC Data Source as shown below:

```
[standalone@localhost:9999 /] /subsystem=datasources/data-
source="java:/datasources/ModeShapeDS":add(jndi-
name="java:/datasources/ModeShapeDS",driver-name="modeshape-
driver",connection-url="jdbc:jcr:jndi:jcr?repositoryName=artifacts",user-
name="admin",password="admin")
{"outcome" => "success"}
```

Note

A preconfigured datasource **`java:/datasources/ModeShapeDS`** already exists by default.

4.1.12. Add and Remove External Sources

Procedure 4.10. Task

1. You can add one or more external sources to an existing repository to enable federation. Here is an example on how you can link an external file system source(via the FileSytemConnector) to the sample repository using the CLI:

```
[standalone@localhost:9999 /]
/subsystem=modeshape/repository=sample/source=fsSource:add(classname="
org.modeshape.connector.filesystem.FileSystemConnector",properties=
[{"directoryPath"=>"."}], readonly="true",
 projections=["default:/projection1 => /"], cacheTtlSeconds="1")
{"outcome" => "success"}

[standalone@localhost:9999 /]
/subsystem=modeshape/repository=sample/source=fsSource:read-resource()
{
```

```
    "outcome" => "success",
    "result" => {
        "cacheTtlSeconds" => "1",
        "classname" =>
"org.modeshape.connector.filesystem.FileSystemConnector",
        "module" => undefined,
        "projections" => ["default:/projection1 => /"],
        "properties" => [{"directoryPath" => "."}],
        "queryable" => undefined,
        "readonly" => "true"
    }
}
```

2. Specify the following attributes when adding an external source:
   - classname (mandatory) - The fully qualified name of the **`Connector`** class that allows content to be retrieved and written to that external source.
   - module (optional) - The name of the JBoss EAP module where the above class is found.
   - projections (optional) - A list of projection expressions representing predefined projection paths for the source. Projections can either be defined here or programmatically using the **`FederationManager.createProjection(...)`** method.
   - queryable (optional) - A flag indicating if the content exposed from the external source should be indexed by the repository or not. By default, this flag is set.
   - readonly (optional) - A flag indicating if only read or both read and write is possible on the source.
   - cacheTtlSeconds (optional) - The number of seconds any given node is to be held in the cache of the corresponding workspace from the external source.
   - properties (optional) - An array of key-value pairs that allow any custom attributes to be passed down on the Connector implementation class.
3. To remove an external source, invoke the remove method on the source as shown below:

```
[standalone@localhost:9999 /]
/subsystem=modeshape/repository=sample/source=fsSource:remove()
{"outcome" => "success"}
```

4.1.13. Working with Batch Mode

You can combine all commands except for the initial **`/extension=org.modeshape:add()`** command, into a batch operation:

```
[standalone@localhost:9999 /] /extension=org.modeshape:add()
{"outcome" => "success"}
[standalone@localhost:9999 /] batch
[standalone@localhost:9999 / #] (paste the commands here)
[standalone@localhost:9999 / #] run-batch
The batch executed successfully.
```

You can edit the batches before runnning them and paste multiple commands into a batch.

4.1.14. Clustering Configuration

Before configuring the hierarchical database to run in a cluster, ensure the JGroups subsystem is present in the JBoss EAP configuration.

Then the following parts need to be configured.

1. Replicated Infinispan caches for the repository store and the binary store:

```
/subsystem=infinispan/cache-
container=modeshape:add(module="org.modeshape")
/subsystem=infinispan/cache-
container=modeshape/transport=TRANSPORT:add(lock-timeout="60000")
/subsystem=infinispan/cache-container=modeshape/replicated-
cache=sample:add(mode="SYNC", batching="true")
/subsystem=infinispan/cache-container=modeshape/replicated-
cache=sample/transaction=TRANSACTION:add(mode=NON_XA)
/subsystem=infinispan/cache-container=modeshape/replicated-
cache=sample/file-store=FILE_STORE:add(path="modeshape/store/sample-
${jboss.node.name}",relative-
to="jboss.server.data.dir",passivation=false,purge=false)
/subsystem=infinispan/cache-container=modeshape-binary-
store:add(module="org.modeshape")
/subsystem=infinispan/cache-container=modeshape-binary-
store/transport=TRANSPORT:add(lock-timeout="60000")
/subsystem=infinispan/cache-container=modeshape-binary-
store/replicated-cache=sample-binary-data:add(mode="SYNC",
batching="true")
/subsystem=infinispan/cache-container=modeshape-binary-
store/replicated-cache=sample-binary-
data/transaction=TRANSACTION:add(mode=NON_XA)
/subsystem=infinispan/cache-container=modeshape-binary-
store/replicated-cache=sample-binary-data/file-
store=FILE_STORE:add(path="modeshape/binary-store/sample-data-
${jboss.node.name}",relative-
to="jboss.server.data.dir",passivation=false,purge=false)
/subsystem=infinispan/cache-container=modeshape-binary-
store/replicated-cache=sample-binary-metadata:add(mode="SYNC",
batching="true")
/subsystem=infinispan/cache-container=modeshape-binary-
store/replicated-cache=sample-binary-
metadata/transaction=TRANSACTION:add(mode=NON_XA)
/subsystem=infinispan/cache-container=modeshape-binary-
store/replicated-cache=sample-binary-metadata/file-
store=FILE_STORE:add(path="modeshape/binary-store/sample-metadata-
${jboss.node.name}",relative-
to="jboss.server.data.dir",passivation=false,purge=false)
```

2. The main repository:

```
/subsystem=modeshape/repository=sample:add(cache-
container="modeshape",cache-name="sample",cluster-name="modeshape-
sample",cluster-stack="tcp",security-domain="modeshape-security")
```

3. Indexing:

```
/subsystem=modeshape/repository=sample/configuration=index-
storage:add()
```

```
/subsystem=modeshape/repository=sample/configuration=index-
storage/storage-type=local-file-index-
storage:add(path="modeshape/indexes/sample-indexes-
${jboss.node.name}")
```

4. Binary Storage:

```
/subsystem=modeshape/repository=sample/configuration=binary-
storage:add()
/subsystem=modeshape/repository=sample/configuration=binary-
storage/storage-type=cache-binary-storage:add(data-cache-name="sample-
binary-data", metadata-cache-name="sample-binary-metadata", cache-
container="modeshape-binary-store")
```

4.2. Using Repositories with JCR API

4.2.1. JCR API

The JCR API is a powerful and easy way to access or manipulate repository content from within a deployed web application or service. The hierarchical database makes using the JCR API very easy. You can get a **`javax.jcr.Repository`** object that represents one of the repositories running within the hierarchical database subsystem and start using the API.

4.2.2. Find the JCR Repository

You can use any of the following ways to find repository instances within JBoss EAP:

- Use Java EE resource injection
- Look up a Repository in JNDI
- Look up the hierarchical database's **`Repositories`** instance and use it to find the **`Repository`** by name
- Use JCR's **`javax.jcr.RepositoryFactory`** and the Service Loader

These methods rely upon JNDI and the fact that the hierarchical database registers itself and each of the **`Repository`** instances into JNDI. The hierarchical database engine, which implements the **`org.modeshape.jcr.api.Repositories`** interface, is registered at **`jcr`**, while each repository is registered at **`jcr/{repositoryName}`**. You can optionally specify an additional JNDI location in the repository configuration. It is useful when you deploy an application that is already looking up a **`Repository`** instance at a specific JNDI name that can not be easily changed. For example, for a repository named "sample", the hierarchical database engine automatically registers it into JNDI (in the global context) at **`jcr/sample`**, although **`java:jcr/sample`** also works in JBoss EAP.

4.2.3. Use Java EE Resource Injection

JBoss EAP is a Java EE compliant application server, which means that your code can use Java EE resource injection to automatically get a reference to the **`Repository`** instance. Here is a snippet from a **`ManagedBean`** example that has the "sample" repository injected automatically:

```
@ManagedBean
public class MyBean {
```

```
  @Resource(mappedName="java:/jcr/sample")
  private javax.jcr.Repository repository;

  ...
}
```

When you deploy your application, JBoss EAP automatically starts the "sample" repository. When you undeploy your application, JBoss EAP automatically stops the "sample" repository unless there are other applications or subsystems using it. This works because the JBoss EAP deployer encounters the @Resource annotation and automatically adds a dependency for the application on the JNDI binding service associated with the specified JNDI name, which depends upon relevant **`Repository`** instance.

4.2.4. Get a Repository Instance from JNDI

You can get a repository by directly looking up the repository in JNDI as shown below:

```
InitialContext context = new InitialContext();
javax.jcr.Repository repository = (javax.jcr.Repository)
context.lookup("jcr/sample");
```

Consider using this approach if you deploy your application to multiple containers including some non-EE containers.

4.2.5. Use RepositoryFactory of JCR

Not all deployment environments have JNDI support. The JCR 2.0 specification defines a pattern that uses the Java SE Service Loader facility to find **`javax.jcr.RepositoryFactory`** instances and use them to get your repository instance. This mechanism also works with for JBoss EAP. If your components that use JCR, are deployed or reused in other applications that are deployed to environments having no JNDI or Java EE support, you can consider this way to look up JCR repositories:

```

String configUrl = "jndi:jcr/sample";
Map<String, String> parameters =
java.util.Collections.singletonMap("org.modeshape.jcr.URL", configUrl);
javax.jcr.Repository repository = null;
for (RepositoryFactory factory :
java.util.ServiceLoader.load(RepositoryFactory.class)) {
    repository = factory.getRepository(parameters);
    if (repository != null) break;
}
```

The RepositoryFactory implementations look for a single ***`org.modeshape.jcr.URL`*** parameter that should be a URL of the form "jndi:jndiName". As your "sample" repository is registered into JNDI at **`jcr/sample`**, you can use **`jndi:jcr/sample`** for the URL.

4.2.6. Use a Repositories Container

Sometimes your applications may need to do more than look up repository instances. For example, your application may need to know which repositories exist. The hierarchical database provides an implementation of the **`org.modeshape.jcr.api.Repositories`** interface that defines several useful methods:

```
public interface Repositories {

    /**
     * Get the names of the available repositories.
     *
     * @return the immutable set of repository names provided by this
server; never null
     */
    Set<String> getRepositoryNames();

    /**
     * Return the JCR Repository with the supplied name.
     *
     * @param repositoryName the name of the repository to return; may not
be null
     * @return the repository with the given name; never null
     * @throws javax.jcr.RepositoryException if no repository exists with
the given name or there is an error communicating with
     *         the repository
     */
    javax.jcr.Repository getRepository( String repositoryName ) throws
javax.jcr.RepositoryException;
}
```

The **`getRepositoryNames()`** method returns an immutable set of names of all existing repositories, while the **`getRepository(String)`** method obtains the JCR repository with the specified name.

The hierarchical database always registers the implementation of this interface in JNDI at the "jcr" (or "java:jcr") name. The following code shows how to directly look up a repository named "sample" using this interface:

```
InitialContext context = new InitialContext();
Repositories repositories = (Repositories) context.lookup("jcr");
javax.jcr.Repository repository = repositories.get("sample");
```

You can also use the repositories object with the RepositoryFactory-style mechanism. In this case, the URL should contain **`jndi:jcr?repositoryName=`*`repositoryName`***. Here is how you can find the "sample" repository using this technique:

```
String configUrl = "jndi:jcr/sample";
Map<String, String> params = new HashMap<String, String>();
params.put(org.modeshape.jcr.api.RepositoryFactory.URL, "jndi:jcr?
repositoryName=sample");
javax.jcr.Repository repository = null;
for (RepositoryFactory factory :
java.util.ServiceLoader.load(RepositoryFactory.class)) {
    repository = factory.getRepository(parameters);
    if (repository != null) break;
}
```

Here is how you can do the same, separating the URL and repository name:

```
String configUrl = "jndi:jcr/sample";
Map<String, String> params = new HashMap<String, String>();
params.put(org.modeshape.jcr.api.RepositoryFactory.URL, "jndi:jcr");
params.put(org.modeshape.jcr.api.RepositoryFactory.REPOSITORY_NAME,
"sample");
javax.jcr.Repository repository = null;
for (RepositoryFactory factory :
java.util.ServiceLoader.load(RepositoryFactory.class)) {
    repository = factory.getRepository(parameters);
    if (repository != null) break;
}
```

Here is how you can use resource-injection:

```
@ManagedBean
public class MyBean {
  @Resource(mappedName="java:/jcr")
  private org.modeshape.jcr.api.Repositories repositories;

  ...
}
```

4.2.7. Deploy JCR Web Applications

The modular classloading system in JBoss EAP enables your application to only see those Java APIs that your applications use. As the JCR API is not one of the standard JEE APIs, your application needs to explicitly state that it needs the JCR API and optionally the hierarchical database API. You can manually specify the modules that your application uses in any of the following ways:

- Specify dependencies in your **`MANIFEST.MF`** file.
- Override dependencies with the **`jboss-deployment-structure.xml`** file.

4.2.8. Specify Dependencies with MANIFEST.MF

You can specify dependencies in your application's **`META-INF/MANIFEST.MF`** file by adding the following line:

```
Dependencies: javax.jcr, org.modeshape.jcr.api export services,
org.modeshape export services
```

Adding this line gives your application visibility to the standard JCR API and to the hierarchical database public API. It also ensures that the hierarchical database service is running by the time your application needs it. It also exports any services such as RepositoryFactory implementations, so that the ServiceLoader can find them.

If you modify the**`MANIFEST.MF`** file, ensure that you include a newline character at the end of the file.

4.2.9. Override Dependencies with jboss-deployment-structure.xml

The **`jboss-deployment-structure.xml`** file is a JBoss specific deployment descriptor. You can use it to control class loading in a fine grained manner. Like the **`MANIFEST.MF`** file, you can use this file to add

dependencies. This file can also prevent automatic dependencies from being added, define additional modules, change an EAR deployment's isolated class loading behavior, and add additional resource roots to a module. Here is a snippet of the **`jboss-deployment-structure.xml`** file:

```
<jboss-deployment-structure>
  ...
  <deployment>
    ...
    <dependencies>
      ...
      <!-- These are equivalent to the "Dependencies: javax.jcr ..." line in
the MANIFEST.MF -->
      <module name="javax.jcr" />
      <module name="org.modeshape.jcr.api" services="import" />
      <module name="org.modeshape" services="import" />
      ...
    </dependencies>
    ...
  </deployment>
  ...
</jboss-deployment-structure>
```

4.2.10. Build an Application with Maven

As the hierarchical database and JBoss EAP are built with Maven, we recommend you to use Maven to build and test your application.

Procedure 4.11. Task

1. To build an application with Maven, include the hierarchical database as a provided dependency in your application's POM file. You can do this for each of the artifacts you need, however it is easier to use the hierarchical database's BOM in your <dependencyManagement> section. The example below shows how the POM file specifies the BOM in the dependencyManagement section and how you can specify the Java EE 6 APIs:

   ```
   <project ...>
       <!-- ... -->
       <dependencyManagement>
           <dependencies>
               <!-- Define the version of JBoss' Java EE 6 APIs we want
   to import.
                    Any dependencies from org.jboss.spec will have
   their version defined by this
                    BOM -->
               <!-- JBoss distributes a complete set of Java EE 6 APIs
   including
                    a Bill of Materials (BOM). A BOM specifies the
   versions of a "stack" (or
                    a collection) of artifacts. We use this here so
   that we always get the correct
                    versions of artifacts. Here we use the jboss-
   javaee-6.0-with-tools stack
                    (you can read this as the JBoss stack of the Java
   ```

```
EE 6 APIs, with some extras
                  tools for your project, such as Arquillian for
testing) -->
            <dependency>
                <groupId>org.jboss.bom</groupId>
                <artifactId>jboss-javaee-6.0-with-tools</artifactId>
                <version>1.0.0.M11</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
            <!-- Import the ModeShape BOM for embedded usage. This
adds to the "dependenciesManagement" section
                 defaults for all of the modules we might need, but
we still have to include in the
                 "dependencies" section the modules we DO need. The
benefit is that we don't have to
                 specify the versions of any of those modules.-->
            <dependency>
                <groupId>org.modeshape.bom</groupId>
                <artifactId>modeshape-bom-jbosseap</artifactId>
                <version>3.2.0.Final</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
    <!-- ... -->
</project>
```

2. Specify the version that you want to use in the following line:

```
<version>3.2.0.Final</version>
```

   As the modeshape-bom-jbossas is a BOM, it includes default "version" and "scope" values for all of the hierarchical database artifacts and (transitive) dependencies.

3. The BOMs add default values for several Java EE6 and hierarchical database artifacts and dependencies, respectively. To make them available in your application, add dependencies for all the artifacts that you directly use. In the case of the hierarchical database, the following is the JCR API and the database's public API:

```
<dependencies>
    ...
    <!-- Directly depend on the JCR 2.0 API -->
    <dependency>
      <groupId>javax.jcr</groupId>
      <artifactId>jcr</artifactId>
    </dependency>
    <!-- Directly depend on ModeShape's public API -->
    <dependency>
      <groupId>org.modeshape</groupId>
```

```
      <artifactId>modeshape-jcr-api</artifactId>
    </dependency>
    ...
  </dependencies>
```

Here you do not have to specify any versions or scope of these artifacts because they are specified in the BOMs used in the <dependencyManagement> section. In the case of these two artifacts, the default scope is "provided", which means that Maven makes them available for compilation. However, since they are provided by JBoss EAP, the hierarchical database runtime environment does not include them in any produced artifacts like WAR files.

Warning

Your deployable web applications and services need not contain any of the JARs from the hierarchical database, JCR API, Infinispan, Hibernate Search, Lucene, Joda Time, Tika, or any of the other libraries that the database uses. Doing so will result in (convoluted) deployment errors regarding class incompatibilities or class cast exceptions. If your code directly uses these libraries, add them as a dependency in your **`MANIFEST.mf`** file or **`jboss-deployment-structure.xml file`**.

4.2.11. Build an Application with Non-Maven Tools

If you are using a non-Maven tool to build your application, ensure that the resulting deployments such as WAR and EAR files do not contain any of the JARs from the hierarchical database, JCR API, Infinispan, Hibernate Search, Lucene, Joda Time, or any of the other libraries that the database uses. These are provided by the subsystem in JBoss EAP. If your code uses any of these, add them as a dependency in your **`MANIFEST.mf`** file or **`jboss-deployment-structure.xml`** file.

4.3. Using Repositories with REST in EAP

4.3.1. RESTful API

The RESTful API is a simple JAX-RS web application that is packaged as a WAR file, and the kit automatically deploys this WAR file. HTTP clients use the hierarchical database's RESTful API. Hence it is easy to write a simple client application to read and write repository content using the RESTful API. However, since your web browser is a simple HTTP client, you can use it to directly interact with the RESTful API. The RESTful API automatically installs when you install the hierarchical database into a JBoss EAP installation.

4.3.2. Using RESTful API to Check the Availability of the Repositories

Procedure 4.12. Task

1. To use the RESTful API to check the health and availability of the repositories, point your browser to http://localhost:8080/modeshape-rest/. This results in a JSON response that is similar to the following:

```
{
    "sample": {
        "repository": {
            "name": "sample",
            "resources": {
```

```
                "workspaces": "/modeshape-rest/sample"
            },
            "metadata": {
                "option.retention.supported": "false",
                ...
            }
        }
    }
}
```

The response document lists the named repositories that are available. In this case, there is only one "sample" repository, and its nested document provides the name, resources and metadata for the repository. The "resources" nested document contains the usable (relative) link.

2. To use the link to get more information about the repository, issue a GET to the resource at http://localhost:8080/modeshape-rest/sample, which you can do by pointing your browser to this URL. When you do this, the RESTful service returns a JSON response document describing the "sample" repository as shown below:

```
{
    "default": {
        "workspace": {
            "name": "default",
            "resources": {
                "query": "/modeshape-rest/sample/default/query",
                "items": "/modeshape-rest/sample/default/items"
            }
        }
    }
}
```

This document describes the repository and lists the named workspaces. In this case, there is a single "default" workspace, and the following resources available for use:

- http://localhost:8080/modeshape-rest/sample/default/items exposes the repository's nodes via RESTful methods.
- http://localhost:8080/modeshape-rest/sample/default/query allows RESTful clients to POST queries and receive responses containing the results.

3. Continue to navigate the content of the "default" workspace in the "sample" repository. You can not issue a POST with your web browser without some HTML/JavaScript content on the page. For example, if you point your browser to http://localhost:8080/modeshape-rest/sample/default/items, you will get a response that describes the root node of that workspace:

```
{
    "properties": {
        "jcr:primaryType": "mode:root",
        "jcr:uuid": "81513257505d64/"
    },
    "children": ["jcr:system"]
}
```

Here, the root node has two properties, jcr:primaryType and jcr:uuid (since the node is also mix:referenceable), and one child node jcr:system.

4. You can append the child name to your URL (for example, http://localhost:8080/modeshape-rest/sample/default/items/jcr:system) to get the information about the "/jcr:system" node:

```
{
    "properties": {
        "jcr:primaryType": "mode:system"
    },
    "children": ["jcr:nodeTypes", "jcr:versionStorage",
"mode:namespaces", "mode:locks"]
}
```

Here, the "/jcr:system" node has only one property but has four children.

5. You can look at the mode:namespaces child node by pointing your browser to http://localhost:8080/modeshape-rest/sample/default/items/jcr:system/mode:namespaces to get its JSON representation:

```
{
    "properties": {
        "jcr:primaryType": "mode:namespaces"
    },
    "children": ["jcr", "nt", "mix", "sv", "mode", "xml", "xmlns",
"xs", "xsi"]
}
```

Here, you can see only one property, while there are 9 children, one for each registered namespace, where the node name is the namespace prefix.

6. You can get the JSON representation of the "jcr" namespace by pointing your browser to http://localhost:8080/modeshape-rest/sample/default/items/jcr:system/mode:namespaces/jcr:

```
{
    "properties": {
        "jcr:primaryType": "mode:namespace",
        "mode:generated": "false",
        "mode:uri": "http:\/\/www.jcp.org\/jcr\/1.0"
    }
}
```

Here, the "/jcr:system/mode:namespaces/jcr" node has three properties and no children.

4.4. Using Repositories with WebDAV in EAP

4.4.1. WebDAV in EAP

The hierarchical database includes a WebDAV interface that clients can use to access, create, update, and delete update nt:file and nt:folder nodes in the repositories, treating these nodes as if they are files and folders on a network file system. Many applications and operating systems are WebDAV clients that you can

use with the hierarchical database. For example, you can mount a repository (or parts of it) as a network drive on most operating systems, and then upload or download files and folders using standard OS operations and graphical tools. You can access all repositories and authenticate them using the 'connect' role. The WebDAV service is packaged as a WAR file and is automatically deployed. You can undeploy it if it is not needed.

4.4.2. Connecting to the Repository with WebDAV

Procedure 4.13. Task

- Connect to the WebDAV service available on your EAP instance using the URL:

  http://localhost:8080/modeshape-webdav/repositoryName/workspaceName/pathInWorkspace

  here,

  - ***repositoryName*** is the name of the repository you want to connect to.
  - ***workspaceName*** is the name of the workspace to be accessed.
  - ***pathInWorkspace*** is the JCR path to the top-level nt:folder (or nt:file) node to be accessed. This is optional.

4.4.3. WebDAV Server Configuration

The WebDAV server is deployed as a WAR and configured mostly through its web configuration files located within the deployment at standalone**`/deployments/modeshape-webdav.war`**. The **`WEB-INF/web.xml`** defines the following parameters:

Table 4.2. WEB-INF/web.xml Parameters

Parameter Name	Description	Value
`org.modeshape.web.jcr.REPOSITORY_PROVIDER`	The fully-qualified name of the class that implements the **`org.modeshape.web.jcr.spi.RepositoryProvider`** interface. This remains the same, unless you are using the WebDAV server to connect to a different JCR implementation.	**`org.modeshape.web.jcr.spi.FactoryRepositoryProvider`**
`org.modeshape.jcr.URL`	This parameter is specific to the **`FactoryRepositoryProvider`** implementation and specifies the JNDI URL of the Repositories implementation.	***`jndi:jcr`***

Parameter Name	Description	Value
org.modeshape.web.jcr.webdav.CONTENT_MAPPER_CLASS_NAME	The fully-qualified name of the class that implements the **org.modeshape.web.jcr.webdav.ContentMapper** interface that is responsible for mapping content nodes to WebDAV responses. The **DefaultContentMapper** implementation maps nodes with type ***nt:folder*** and ***nt:file*** to WebDAV folders and files, respectively. You can provide your own implementation to map WebDAV content to other node content or structures.	***org.modeshape.web.jcr.webdav.DefaultContentMapper***
org.modeshape.web.jcr.webdav.NEW_FOLDER_PRIMARY_TYPE_NAME	Each folder created through the WebDAV servlet is created as a node with this primary node type.	***nt:folder***
org.modeshape.web.jcr.webdav.NEW_RESOURCE_PRIMARY_TYPE_NAME	This primary node type creates each resource (such as a file) through the WebDAV servlet.	***nt:file***
org.modeshape.web.jcr.webdav.NEW_CONTENT_PRIMARY_TYPE_NAME	This primary node type creates content through the WebDAV servlet.	***nt:resource***
org.modeshape.web.jcr.webdav.RESOURCE_PRIMARY_TYPE_NAMES	Nodes with any of the primary node types in this comma-delimited list is exposed to WebDAV clients as file nodes.	***nt:file***
org.modeshape.web.jcr.webdav.CONTENT_PRIMARY_TYPE_NAMES	Nodes with any of the primary node types in this comma-delimited list is exposed to WebDAV clients as content nodes (that is, nodes that have the content of the files).	***nt:resource, mode:resource***

4.4.4. Authentication and Authorization in the JCR Repository

Here is how you can perform authentication in the JCR Repository:

```
<!--
 The ModeShape WebDAV implementation leverages the HTTP credentials to for
authentication
 and authorization within the JCR repository.  Unless the repository provides
for anonymous
 access, it makes no sense to try to log into the JCR repository without
credentials, so
 this constraint helps lock down the repository.

 This should generally not be modified.
-->
<security-constraint>
```

```
  <display-name>ModeShape WebDAV</display-name>
  <web-resource-collection>
    <web-resource-name>WebDAV</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <!--
      A user must be assigned this role to connect to any JCR repository, in
addition to
      needing the READONLY or READWRITE roles to actually read or modify the
data.
    -->
    <role-name>connect</role-name>
  </auth-constraint>
</security-constraint>

<!--
  Any auth-method will work for ModeShape.  BASIC is used this example for
simplicity.
-->
<login-config>
  <auth-method>BASIC</auth-method>
</login-config>

<!--
  This must match the role-name in the auth-constraint above.
-->
<security-role>
  <role-name>connect</role-name>
</security-role>
```

4.5. Using Repositories with JDBC in EAP

4.5.1. JDBC in EAP

The hierarchical database provides a JDBC-compliant API that allows clients to connect and query a repository via JDBC. The hierarchical database comes pre-packaged with a **`org.modeshape.jdbc`** module. This module contains a **`java.sql.Driver`** implementation that allows JDBC clients to connect to existing repositories.

4.5.2. Configure a Datasource and Driver

You can access a hierarchical database repository via JDBC to configure a datasource and a driver inside of JBoss EAP. The following example shows a configuration snippet from a JBoss EAP **`standalone.xml`** file, which exposes via JDBC, the workspace "extra" from a repository named "artifacts":

```
<datasource jndi-name="java:/datasources/ModeShapeDS" enabled="true" use-
java-context="true" pool-name="ModeShapeDS">
  <connection-url>jdbc:jcr:jndi:jcr?repositoryName=artifacts</connection-
url>
  <driver>modeshape</driver>
  <connection-property name="workspace">extra</connection-property>
```

```
  <security>
   <user-name>admin</user-name>
   <password>admin</password>
  </security>
</datasource>

<drivers>
  <driver name="modeshape" module="org.modeshape.jdbc">
     <driver-class>org.modeshape.jdbc.LocalJcrDriver</driver-class>
  </driver>
</drivers>
```

Configuring the hierarchical database JDBC driver requires the following attributes:

Table 4.3. JDBC driver attributes

name	A symbolic name for the JDBC driver for the datasource.
module	The JBoss EAP module name containing the JDBC driver implementation.
driver-class	The fully qualified class name of the **`java.sql.Driver`** implementation.

For each repository you want to access, you need to configure a DataSource in the JBoss EAP configuration file. In the example above, the following attributes are defined:

Table 4.4. JDBC driver attributes

jndi-name	The name under which the datasource should be registered in JNDI by JBoss EAP. Currently, JBoss EAP only allows datasources to be registered under a name beginning either with java:/ or java:jboss/.
connection-url	A JNDI URL that points the hierarchical database to an existing repository. The format of this URL is: **`jdbc:jcr:jndi:jcr:?repositoryName=`**
driver	The name of the JDBC driver.
security	The username and password that is passed to the connection, when attempting to access a repository. Inside JBoss EAP, these are taken from the **`modeshape-security`** domain.
connection-property	Any additional properties which can be passed to the connection. For example, to access a specific workspace of a repository, the workspace property can be defined.

4.5.3. Access Datasource from JNDI and Execute Queries

Once you configure a datasource and start the application server, you can access the datasource from JNDI and execute queries against the configured repository. Here is an example:

```
@Resource( mappedName = "datasources/ModeShapeDS" )
private DataSource modeshapeDS;
```

```
....

Connection connection = modeshapeDS.getConnection();
Statement statement = connection.createStatement();
ResultSet resultSet = statement.executeQuery("SELECT [jcr:primaryType],
[jcr:mixinTypes], [jcr:path], [jcr:name] FROM [nt:unstructured] ORDER BY
[jcr:path]");
```

For executing queries, use JCR-SQL2 query language. However, as you can not expose JCR Nodes directly via JDBC, the only way to return the path and score information is through additional columns in the result. But doing so is not compatible with JDBC applications that dynamically build queries based upon database metadata. Such applications require the columns to be properly described in database metadata, and the columns need to be used within queries. The hierarchical database attempts to solve these issues by directly supporting a number of "pseudo-columns" within JCR-SQL2 queries, wherever columns can be used. These "pseudo-columns" include:

- ***`jcr:score`***: This is a column of type DOUBLE that represents the full-text search score of the node, which is a measure of the node's relevance to the full-text search expression. The hierarchical database computes the scores for all queries, though the score for rows in queries that do not include a full-text search criteria may not be reliable.
- ***`jcr:path`***: This is a column of type PATH that represents the normalized path of a node, including same-name siblings. This is the same as what would be returned by the **`getPath()`** method of http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Node.htmlNode. Examples of paths include "/jcr:system" and "/foo/bar[3]".
- ***`jcr:name`***: This is a column of type NAME that represents the node name in its namespace-qualified form using namespace prefixes and excluding same-name-sibling indexes. Examples of node names include "jcr:system", "jcr:content", "ex:UserData", and "bar".
- ***`mode:localName`***: This is a column of type STRING that represents the local name of the node, which excludes the namespace prefix and same-name-sibling index. As an example, the local name of the "jcr:system" node is "system", while the local name of the "ex:UserData[3]" node is "UserData".
- ***`mode:depth`***: This is a column of type LONG that represents the depth of a node, which corresponds exactly to the number of path segments within the path. For example, the depth of the root node is 0, whereas the depth of the "/jcr:system/jcr:nodeTypes" node is 2.

These columns are exposed in the database metadata allowing potential clients to detect and use them.

4.6. Administering Repositories in JBoss EAP

4.6.1. Navigation with Management CLI

The CLI allows you to navigate the management model as if you are navigating a file system. Here are some helpful commands that can be run at the root of the management system. But each of these commands can be modified to run using relative paths also.

To navigate to the hierarchical database subsystem:

```
cd /subsystem=modeshape
```

To navigate to the hierarchical database repositories:

```
cd /subsystem=modeshape/repository
```

To navigate to a specific repository:

```
cd /subsystem=modeshape/repository=<repository-name>
```

To navigate to the hierarchical database web applications:

```
cd /subsystem=modeshape/webapp
```

To navigate to a specific web application:

```
cd /subsystem=modeshape/webapp=<web-app-war>
```

To navigate to a specific repository's CND sequencer:

```
cd /subsystem=modeshape/repository=<repository-name>/sequencer=cnd-sequencer
```

4.6.2. Managed Resource Commands

To view attribute values, metric values, and child managed nodes of a resource, like a repository, enter "ls" for a listing. A nicer way to see the same information in a much more readable form is to enter the following:

```
:read-resource
```

To obtain descriptions, possible attribute types, and possible child types:

```
:read-resource-description
```

To obtain current metric values:

```
:read-resource(include-runtime=true)
```

To obtain current attribute values recursively for child nodes so that you do not have to navigate to each child node:

```
:read-resource(recursive-depth=10)
```

4.6.3. Administering Repositories with JBoss Operations Network

You can administer hierarchical database repositories using JBoss Operations Network. For more information, see the *JBoss Operations Network* documentation.

Note

You can use JBoss Operations Network to deploy VDBs that are present in the hierarchical database repository.

Chapter 5. The REST Service

The hierarchical database's RESTful API was intended to be used by HTTP clients, so it is convenient to write a simple client application to read and write repository content using the RESTful API. However, since our trusty web browser is indeed a simple HTTP client, we can use it to directly interact with the RESTful API. It might not be pretty, but it works beautifully.

The RESTful API is nothing more than a simple JAX-RS web application that is packaged as a WAR file and that comes in 2 flavors:

- modeshape-web-jcr-rest-war - is a war file artifact available via Maven that can be deployed into any servlet container. To access it, include the following dependency in your project's POM:

```
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-web-jcr-rest-war</artifactId>
  <version>${modeshape.version}</version>
  <type>war</type>
</dependency>
```

- The JBoss EAP kit - once installed into JBoss EAP, it provides the RESTful API out-of-the-box, via a web application.

Both web applications use Basic HTTP Authentication and require a role named connect to be present in the authenticated user's set of roles.

The hierarchical database provides two different versions of the RESTful API:

1. REST Service 2.x - the version which was included in previous versions and which has been deprecated. However, for backwards compatibility it is still accessible using the v1 URL prefix: http://<host>:<port>/<context>/v1/
2. REST Service 3.x - a newer version which is an extension of the old one, plus a number of additional improvements.

5.1. REST Service 2.x

Represents the version of the RESTful API distributed with previous versions. It has been deprecated in this release, but is still available using the v1 URL prefix. It provides the following methods:

Retrieve a list of available repositories

URL : http://<host>:<port>/<context>/v1/

HTTP Method : GET

Produces : application/json; text/html; text/plain;

Default Output : text/plain

Response Code (if successful): OK

Response Format :

```
{
   "repo":
   {
```

```
        "repository":
        {
            "name": "repo",
            "resources":
            {
                "workspaces": "/resources/v1/repo"
            },
            "metadata":
            {
                "option.retention.supported": "false",
                "query.xpath.doc.order": "false",
                ...
          }
        }
    }
}
```

Retrieve a list of workspaces for a repository

URL : http://<host>:<port>/<context>/v1/<repository_name>

HTTP Method : GET

Produces : application/json; text/html; text/plain;

Default Output : text/plain

Response Code (if successful): OK

Response Format :

```
{
       "default":
       {
           "workspace":
           {
               "name": "default",
               "resources":
               {
                   "query": "/resources/v1/repo/default/query",
                   "items": "/resources/v1/repo/default/items"
               }
           }
       }
    }
```

Retrieve a node or a property

Retrieves an item at a given path.

URL : http://<host>:
<port>/<context>/v1/<repository_name>/<workspace_name>/items/<item_path>

HTTP Method : GET

Produces : application/json; text/html; text/plain;

Default Output : text/plain

Response Code (if successful): OK

Optional Query Parameters :

- depth - a numeric value indicating how many level of children should be retrieved under the node located at path. A negative value indicates all children
- mode:depth - same as the above

Response Format :

```
{
       "properties":
       {
           "jcr:primaryType": "mode:system"
       },
       "children":
       [
           "jcr:nodeTypes",
           "jcr:versionStorage",
           "mode:namespaces",
           "mode:locks"
       ]
    }
```

Create a node

Creates a node at the given path, using the body of request as JSON content

URL : http://<host>:
<port>/<context>/v1/<repository_name>/<workspace_name>/items/<node_path>

HTTP Method : POST

Produces : application/json; text/html; text/plain;

Default Output : text/plain

Request Content-Type : accepts any, but for this to work it has to be a valid JSON object

Response Code (if successful): CREATED

Optional Query Parameters :

- mode:includeNode - indicates if the entire node should be returned in the response or only the path to the new node.

Request Format :

```
{ "properties":{
    "jcr:primaryType":"nt:unstructured",
    "testProperty":"testValue",
    "multiValuedProperty":["value1", "value2"]
    },
    "children":{
        "childNode":{
            "properties":{
                "nestedProperty":"nestedValue"
```

```
            }
        }
    }
}
```

Response Format :

```
{"properties":{
    "jcr:primaryType":"nt:unstructured",
    "multiValuedProperty":["value1", "value2"],
    "testProperty":"testValue"
}, "children":{
    "childNode":{
        "properties":{
            "jcr:primaryType":"nt:unstructured",
            "nestedProperty":"nestedValue"
        }
    }
}}
```

Update a node or a property

Updates a node or a property at the given path, using the body of request as JSON content

URL : http://<host>:
<port>/<context>/v1/<repository_name>/<workspace_name>/items/<item_path>

HTTP Method : PUT

Produces : application/json; text/html; text/plain;

Default Output : text/plain

Request Content-Type : accepts any, but for this to work it has to be a valid JSON object

Response Code (if successful): OK

Request Format :

Node: same as the one used when creating

Property:

```
{"testProperty":"some_new_value"}
```

Response Format :

Node: same as one used when creating

Property:

```
{"testProperty":"some_new_value"}
```

Delete a node or a property

Deletes the node or the property at the given path.

URL : http://<host>:<port>/<context>/v1/<repository_name>/<workspace_name>/items/<item_path>

HTTP Method : DELETE

Produces : none

Response Code (if successful): OK

Execute a JCR query

Executes a JCR query in either: XPath, SQL or SQL2 format, returning a JSON object in response.

URL : http://<host>:<port>/<context>/v1/<repository_name>/<workspace_name>/query

HTTP Method : POST

Produces : application/json; text/html; text/plain;

Request Content-Type : application/jcr+sql; application/jcr+xpath; application/jcr+sql2; application/jcr+search

Default Output : text/plain

Response Code (if successful): OK

Optional Query Parameters :

- offset - the index in the result set where to start the retrieval of data
- limit - the maximum number of rows to return

Response Format :

```
{
       "types":
       {
           "nt:base.jcr:primaryType": "STRING",
           "nt:base.jcr:mixinTypes": "STRING",
           "nt:base.jcr:path": "STRING",
           "nt:base.jcr:name": "STRING",
           "nt:base.jcr:score": "DOUBLE",
           "nt:base.mode:localName": "STRING",
           "nt:base.mode:depth": "LONG"
       },
       "rows":
       [
           {
               "nt:base.jcr:primaryType": "mode:root",
               "nt:base.jcr:path": "/",
               "nt:base.jcr:name": "",
               "nt:base.jcr:score": "0.3535533845424652",
               "nt:base.mode:localName": "",
               "nt:base.mode:depth": "0"
           },
           {
               "nt:base.jcr:primaryType": "mode:locks",
               "nt:base.jcr:path": "/jcr:system/mode:locks",
               "nt:base.jcr:name": "mode:locks",
```

```
                "nt:base.jcr:score": "0.3535533845424652",
                "nt:base.mode:localName": "locks",
                "nt:base.mode:depth": "2"
            }
        ]
    }
```

5.2. REST Service 3.x

Represents the default version of the RESTful API distributed with the hierarchical database. It provides the following methods:

Retrieve a list of available repositories

URL : http://<host>:<port>/<context>

HTTP Method : GET

Produces : application/json; text/html; text/plain;

Default Output : text/html

Response Code (if successful): OK

Response Format :

```
{
    "repositories":[
        {
            "name":"repo",
            "workspaces":"http://localhost:8080/modeshape-rest",
            "metadata":{
                "custom.rep.name":"repo",
                "custom.rep.workspace.names":"default",}
                .....
            }
         }
     ]
}
```

Retrieve a list of workspaces for a repository

URL : http://<host>:<port>/<context>/<repository_name>

HTTP Method : GET

Produces : application/json; text/html; text/plain;

Default Output : text/html

Response Code (if successful): OK

Response Format :

```
{
    "workspaces":[
        {
```

```
            "name":"default",
            "repository":"http://localhost:8080/modeshape-rest",
            "items":"http://localhost:8080/modeshape-
rest/default/items",
            "query":"http://localhost:8080/modeshape-
rest/default/query",
            "binary":"http://localhost:8080/modeshape-
rest/default/binary",
            "nodeTypes":"http://localhost:8080/modeshape-
rest/default/nodetypes"
        }
    ]
}
```

Retrieve a node or a property

Retrieves an item at a given path.

URL : http://<host>:<port>/<context>/<repository_name>/<workspace_name>/items/<item_path>

HTTP Method : GET

Produces : application/json; text/html; text/plain;

Default Output : text/html

Response Code (if successful): OK

Optional Query Parameters :

- depth - a numeric value indicating how many level of children should be retrieved under the node located at path. A negative value indicates all children

Response Format :

```
{
    "self":"http://localhost:8080/modeshape-
rest/default/items/someNode",
    "up":"http://localhost:8080/modeshape-rest/default/items/",
    "id":"319a0554-3504-4984-b54b-3a9367caac92",
    "jcr:primaryType":"{http://www.modeshape.org/1.0}root",
    "jcr:uuid":"319a0554-3504-4984-b54b-3a9367caac92",
    "children":{
        "jcr:system":{
            "self":"http://localhost:8080/modeshape-
rest/default/items/jcr:system",
            "up":"http://localhost:8080/modeshape-
rest/default/items/",
            "id": "0a851519-e87d-4e02-b399-0503aa70ab3f"
        }
    }
}
```

Create a node

Creates a node at the given path, using the body of request as JSON content

URL : http://<host>:<port>/<context>/<repository_name>/<workspace_name>/items/<node_path>

HTTP Method : POST

Produces : application/json; text/html; text/plain;

Default Output : application/json

Request Content-Type : application/json

Response Code (if successful): CREATED

Request Format :

```
{
    "jcr:primaryType":"nt:unstructured",
    "testProperty":"testValue",
    "multiValuedProperty":["value1", "value2"],
    "children":{
        "childNode":{
            "nestedProperty":"nestedValue"
        }
    }
}
```

Response Format :

```
{
    "self":"http://localhost:8080/modeshape-
rest/default/items/testNode",
    "up":"http://localhost:8080/modeshape-rest/default/items/",
    "id":"bf171df0-daa2-481d-a48a-b3965cd69d9c",
    "jcr:primaryType":"{http://www.jcp.org/jcr/nt/1.0}unstructured",
    "multiValuedProperty":[
        "value1",
        "value2"
    ],
    "testProperty":"testValue",
    "children":{
        "childNode":{
            "self":"http://localhost:8080/modeshape-
rest/default/items/testNode/childNode",
            "up":"http://localhost:8080/modeshape-
rest/default/items/testNode",
            "id":"113e6eea-cbd2-4837-8344-5b28bbfd695c",
        }
    }
}
```

Update a node or a property

Updates a node or a property at the given path, using the body of request as JSON content

URL : http://<host>:<port>/<context>/<repository_name>/<workspace_name>/items/<item_path>

HTTP Method : PUT

Produces : application/json; text/html; text/plain;

Default Output : application/json

Request Content-Type : application/json

Response Code (if successful): OK

Request Format :

Node: same as the one used when creating

Property:

```
{"testProperty":"some_new_value"}
```

Response Format :

Node: same as one used when creating

Property:

```
{"testProperty":"some_new_value"}
```

Delete a node or a property

Deletes the node or the property at the given path. If a node is being deleted, this will also delete all of its descendants.

URL : http://<host>:<port>/<context>/<repository_name>/<workspace_name>/items/<item_path>

HTTP Method : DELETE

Produces : none

Response Code (if successful): NO_CONTENT

Retrieve a node by its identifier

Retrieves a node with a specified identifier. This is equivalent to the **`Session.getNodeByIdentifier(String)`** method, where the identifier is obtained from the **`id`** field (or the **`jcr:uuid`** field if the node is **`mix:referenceable`**) in a previous response. Remember that node identifiers are generated by the repository, are opaque (and are not always UUIDs), and always remains the same for a given node (even when moved or renamed) until the node is destroyed.

URL : http://<host>:<port>/<context>/<repository_name>/<workspace_name>/nodes/<node_id>

HTTP Method : GET

Produces : application/json; text/html; text/plain;

Default Output : text/html

Response Code (if successful): OK

Optional Query Parameters :

- depth - a numeric value indicating how many level of children should be retrieved under the node located at path. A negative value indicates all children

Response Format :

```
{
```

```
    "self":"http://localhost:8080/modeshape-
rest/default/items/someNode",
    "up":"http://localhost:8080/modeshape-rest/default/items/",
    "id":"319a0554-3504-4984-b54b-3a9367caac92",
    "jcr:primaryType":"{http://www.modeshape.org/1.0}root",
    "jcr:uuid":"319a0554-3504-4984-b54b-3a9367caac92",
    "children":{
        "jcr:system":{
            "self":"http://localhost:8080/modeshape-
rest/default/items/jcr:system",
            "up":"http://localhost:8080/modeshape-
rest/default/items/",
            "id": "0a851519-e87d-4e02-b399-0503aa70ab3f"
        }
    }
}
```

Update a node by its identifier

Updates a node with the given identifier, using the body of request as JSON content. The identifier must be obtained from the **`id`** field in a previous response.

URL : http://<host>:<port>/<context>/<repository_name>/<workspace_name>/nodes/<node_id>

HTTP Method : PUT

Produces : application/json; text/html; text/plain;

Default Output : application/json

Request Content-Type : application/json

Response Code (if successful): OK

Request Format :

Node: same as the one used when creating a node

Property:

```
{"testProperty":"some_new_value"}
```

Response Format :

Node: same as one used when creating a node

Property:

```
{"testProperty":"some_new_value"}
```

Delete a node by its identifier

Deletes the node with the given identifier, and all of its descendants. The identifier must be obtained from the **`id`** field in a previous response.

URL : http://<host>:<port>/<context>/<repository_name>/<workspace_name>/nodes/<node_id>

HTTP Method : DELETE

Produces : none

Response Code (if successful): NO_CONTENT

Execute a JCR query

Executes a JCR query in either: XPath, SQL or SQL2 format, returning a JSON object in response.

URL : http://<host>:<port>/<context>/<repository_name>/<workspace_name>/query

HTTP Method : POST

Produces : application/json; text/html; text/plain;

Request Content-Type : application/jcr+sql; application/jcr+xpath; application/jcr+sql2; application/jcr+search

Default Output : application/json

Response Code (if successful): OK

Optional Query Parameters :

- offset - the index in the result set where to start the retrieval of data
- limit - the maximum number of rows to return

Response Format :

```
{
    "columns":{
        "jcr:path":"STRING",
        "jcr:score":"DOUBLE",
        "foo":"STRING"
    },
    "rows":[
        {
            "jcr:path":"/{}testNode/{}child[2]",
            "jcr:score":"0.8575897812843323",
            "foo":"value",
            "mode:uri":"http://localhost:8080/modeshape-
rest/default/items/testNode/child[2]"
        },
        {
            "jcr:path":"/{}testNode/{}child[3]",
            "jcr:score":"0.8575897812843323",
            "foo":"value",
            "mode:uri":"http://localhost:8080/modeshape-
rest/default/items/testNode/child[3]"
        }
    ]
}
```

Create multiple nodes

Creates multiple nodes (bulk operation) in the repository, using a single session. If any of the nodes cannot be created, the entire operation fails.

URL : _http://<host>:<port>/<context>/<repository_name>/<workspace_name>/items

HTTP Method : POST

Produces : application/json; text/html; text/plain;

Default Output : application/json

Request Content-Type : application/json

Response Code (if successful): OK

Request Format :

```
{
    "testNode/child/subChild" : {
        "jcr:primaryType":"nt:unstructured",
        "testProperty":"testValue",
        "multiValuedProperty":["value1", "value2"]
    },
    "testNode/child" : {
        "jcr:primaryType":"nt:unstructured",
        "testProperty":"testValue",
        "multiValuedProperty":["value1", "value2"]
    },
    "testNode/otherChild" : {
        "jcr:primaryType":"nt:unstructured",
        "testProperty":"testValue",
        "multiValuedProperty":["value1", "value2"],
        "children":{
            "otherSubChild":{
                "nestedProperty":"nestedValue"
            }
        }
    }
}
```

Response Format :

```
[
    {
        "self":"http://localhost:8080/modeshape-
rest/default/items/testNode/child",
        "up":"http://localhost:8080/modeshape-
rest/default/items/testNode",
        "id":"0ef2edc9-c873-4a2f-805e-2950b98225c6",
        "jcr:primaryType":"
{http://www.jcp.org/jcr/nt/1.0}unstructured",
        "multiValuedProperty":[
            "value1",
            "value2"
        ],
        "testProperty":"testValue"
    },
    {
        "self":"http://localhost:8080/modeshape-
rest/default/items/testNode/child/subChild",
        "up":"http://localhost:8080/modeshape-
rest/default/items/testNode/child",
```

```
        "id":"fb6f4d82-33e1-4bc1-8048-d1f9a685779b",
        "jcr:primaryType":"
{http://www.jcp.org/jcr/nt/1.0}unstructured",
        "multiValuedProperty":[
            "value1",
            "value2"
        ],
        "testProperty":"testValue"
    },
    {
        "self":"http://localhost:8080/modeshape-
rest/default/items/testNode/otherChild",
        "up":"http://localhost:8080/modeshape-
rest/default/items/testNode",
        "id":"da12f5f9-4ab9-48d7-a159-07144e378d54",
        "jcr:primaryType":"
{http://www.jcp.org/jcr/nt/1.0}unstructured",
        "multiValuedProperty":[
            "value1",
            "value2"
        ],
        "testProperty":"testValue",
        "children":{
            "otherSubChild":{
                "self":"http://localhost:8080/modeshape-
rest/default/items/testNode/otherChild/otherSubChild",
                "up":"http://localhost:8080/modeshape-
rest/default/items/testNode/otherChild"
                "id":"21ea01f5-e41c-4aea-9087-e241e02a4b2d",
            }
        }
    }
]
```

Update multiple items

Updates multiple nodes and/or properties (bulk operation) in the repository, using a single session. If any of the items cannot be updated, the entire operation fails.

URL : _http://<host>:<port>/<context>/<repository_name>/<workspace_name>/items

HTTP Method : PUT

Produces : application/json; text/html; text/plain;

Default Output : application/json

Request Content-Type : application/json

Response Code (if successful): OK

Request Format : same as the one used when creating multiple nodes.

Response Format : same as the one used when creating multiple nodes.

Delete multiple items

Deletes multiple items (bulk operation) in the repository, using a single session. If any of the items cannot be removed, the entire operation fails.

URL : _http://<host>:<port>/<context>/<repository_name>/<workspace_name>/items

HTTP Method : DELETE

Produces : none;

Request Content-Type : application/json

Response Code (if successful): OK

Request Format :

```
["testNode/otherChild", "testNode/child",  "testNode/child/subChild"]
```

Retrieve a node type

Retrieves the information about a registered node type in the repository.

URL : http://<host>:
<port>/<context>/<repository_name>/<workspace_name>/nodetypes/node_type_name

HTTP Method : GET

Produces : application/json; text/html; text/plain;

Default Output : text/html

Response Code (if successful): OK

Response Format :

```
{
    "nt:base":{
        "mixin":false,
        "abstract":true,
        "queryable":true,
        "hasOrderableChildNodes":false,
        "propertyDefinitions":[
            {
                "jcr:primaryType":{
                    "requiredType":"Name",
                    "declaringNodeTypeName":"nt:base",
                    "mandatory":true,
                    "multiple":false,
                    "autocreated":true,
                    "protected":true,
                    "fullTextSearchable":true,
                    "onParentVersion":"COMPUTE"
                }
            },
            {
                "jcr:mixinTypes":{
                    "requiredType":"Name",
                    "declaringNodeTypeName":"nt:base",
                    "mandatory":false,
                    "multiple":true,
                    "autocreated":false,
```

```
                    "protected":true,
                    "fullTextSearchable":true,
                    "onParentVersion":"COMPUTE"
                }
            }
        ],
        "subTypes":[
            "http://localhost:8080/modeshape-
rest/default/nodetypes/mode:lock",
            "http://localhost:8080/modeshape-
rest/default/nodetypes/mode:locks",
             ....
        ]
    }
}
```

Import a CND file (via request content)

Imports a CND file into the Repository, using the entire request body stream as the content of the CND. If you were using curl , this would be the equivalent of curl -d

URL : _http://<host>:<port>/<context>/<repository_name>/<workspace_name>/nodetypes

HTTP Method : POST

Produces : application/json; text/html; text/plain;

Default Output : application/json

Response Code (if successful): OK

Response Format :

```
[
    {
        "nt:base":{
            "mixin":false,
            "abstract":true,
            "queryable":true,
            "hasOrderableChildNodes":false,
            "propertyDefinitions":[
                {
                    "jcr:primaryType":{
                        "requiredType":"Name",
                        "declaringNodeTypeName":"nt:base",
                        "mandatory":true,
                        "multiple":false,
                        "autocreated":true,
                        "protected":true,
                        "fullTextSearchable":true,
                        "onParentVersion":"COMPUTE"
                    }
                },
                {
                    "jcr:mixinTypes":{
                        "requiredType":"Name",
                        "declaringNodeTypeName":"nt:base",
```

```
                        "mandatory":false,
                        "multiple":true,
                        "autocreated":false,
                        "protected":true,
                        "fullTextSearchable":true,
                        "onParentVersion":"COMPUTE"
                    }
                }
            ],
            "subTypes":[
                "http://localhost:8080/modeshape-
rest/default/nodetypes/mode:lock",
                ...
            ]
        }
    },
    {
        "nt:unstructured":{
            "mixin":false,
            "abstract":false,
            "queryable":true,
            "hasOrderableChildNodes":true,
            "propertyDefinitions":[
                {
                    "*":{
                        "requiredType":"undefined",
                        "declaringNodeTypeName":"nt:unstructured",
                        "mandatory":false,
                        "multiple":true,
                        "autocreated":false,
                        "protected":false,
                        "fullTextSearchable":true,
                        "onParentVersion":"COPY"
                    }
                },
                {
                    "*":{
                        "requiredType":"undefined",
                        "declaringNodeTypeName":"nt:unstructured",
                        "mandatory":false,
                        "multiple":false,
                        "autocreated":false,
                        "protected":false,
                        "fullTextSearchable":true,
                        "onParentVersion":"COPY"
                    }
                }
            ],
            "superTypes":[
                "http://localhost:8080/modeshape-
rest/default/nodetypes/nt:base"
            ]
        }
    },
    {
        "mix:created":{
```

```
            "mixin":true,
            "abstract":false,
            "queryable":true,
            "hasOrderableChildNodes":false,
            "propertyDefinitions":[
                {
                    "jcr:created":{
                        "requiredType":"Date",
                        "declaringNodeTypeName":"mix:created",
                        "mandatory":false,
                        "multiple":false,
                        "autocreated":false,
                        "protected":true,
                        "fullTextSearchable":true,
                        "onParentVersion":"COPY"
                    }
                },
                {
                    "jcr:createdBy":{
                        "requiredType":"String",
                        "declaringNodeTypeName":"mix:created",
                        "mandatory":false,
                        "multiple":false,
                        "autocreated":false,
                        "protected":true,
                        "fullTextSearchable":true,
                        "onParentVersion":"COPY"
                    }
                }
            ],
            "subTypes":[
                "http://localhost:8080/modeshape-
rest/default/nodetypes/nt:hierarchyNode"
            ]
        }
    }
]
```

Import a CND file (via "multipart/form-data")

Imports a CND file into the Repository when the CND file came from a form submission, where the name of the HTML element is **file** . If you were using curl , this would be the equivalent of curl -F

URL : _http://<host>:<port>/<context>/<repository_name>/<workspace_name>/nodetypes

HTTP Method : POST

Produces : application/json; text/html; text/plain;

Request Content-Type : multipart/form-data

Default Output : application/json

Response Code (if successful): OK

Response Format : the same as when importing a CND via the request body.

Retrieve a binary property

Retrieves the content of a binary property from the repository, at a given path, by streaming it to the response.

URL : http://<host>:<port>/<context>/<repository_name>/<workspace_name>/binary/binary_property_path

HTTP Method : GET

Produces : the mime-type of the binary, or a default mime-type

Response Code (if successful): OK

Optional Query Parameters :

- mimeType - a string which can be provided by the client, in case it already knows the expected mimetype of the binary stream. Otherwise, the hierarchical database will try to detect the mimetype using its own detectors mechanism
- contentDisposition - a string which will be returned as the Content-Disposition response header. If none provide, the default is: attachment;filename=property_parent_name

Create a binary property (via request content)

Creates a new binary property in the repository, at the given path, using the entire request body stream as the content of the binary. If you were using curl , this would be the equivalent of curl -d

URL : http://<host>:<port>/<context>/<repository_name>/<workspace_name>/binary/binary_property_path

HTTP Method : POST

Produces : application/json; text/html; text/plain;

Default Output : application/json

Response Code (if successful): OK

Response Format :

```
{
    "testProperty":"http://localhost:8080/modeshape-
rest/default/binary/testNode/testProperty",
    "self":"http://localhost:8080/modeshape-
rest/default/items/testNode/testProperty",
    "up":"http://localhost:8080/modeshape-
rest/default/items/testNode"
}
```

Update a binary property (via request content)

Updates the content of a binary property in the repository, at the given path, using the entire request body stream as the content of the binary. If you were using curl , this would be the equivalent of curl -d

URL : http://<host>:<port>/<context>/<repository_name>/<workspace_name>/binary/binary_property_path

HTTP Method : POST, PUT

Produces : application/json; text/html; text/plain;

Default Output : application/json

Response Code (if successful): OK

Response Format : the same as in the case when creating a new binary property

Create/Update a binary property (via "multipart/form-data")

Creates or updates the content of a binary property in the repository, at the given path, when the content came from a form submission, where the name of the HTML element is **file** . If you were using curl , this would be the equivalent of curl -F

URL : http://<host>:<port>/<context>/<repository_name>/<workspace_name>/binary/binary_property_path

HTTP Method : POST

Produces : application/json; text/html; text/plain;

Default Output : application/json

Request Content-Type : multipart/form-data

Response Code (if successful): OK

Response Format : the same as in the case when creating a new binary property

Obtain a query plan for a JCR query

Obtain the query plan for an XPath, SQL or SQL2 query, returning the string representation of the query plan.

URL : http://<host>:<port>/<context>/<repository_name>/<workspace_name>/queryPlan

HTTP Method : POST

Produces : application/json; text/html; text/plain;

Default Output : text/plain

Request Content-Type : application/jcr+sql; application/jcr+xpath; application/jcr+sql2; application/jcr+search

Response Code (if successful): OK

Optional Query Parameters :

- offset - the index in the result set where to start the retrieval of data
- limit - the maximum number of rows to return

Response Format (as "application/json"):

```
{
    "statement":"SELECT * FROM [nt:unstructured] WHERE
ISCHILDNODE('\/testNode')",
    "language":"JCR-SQL2",
    "abstractQueryModel":"SELECT * FROM [nt:unstructured] WHERE
ISCHILDNODE([nt:unstructured],'\/testNode')",
    "queryPlan": [
         "Access [nt:unstructured]",
```

```
          "  Project [nt:unstructured] <PROJECT_COLUMNS=
[[nt:unstructured].[jcr:primaryType] AS
[nt:unstructured.jcr:primaryType], [nt:unstructured].[jcr:mixinTypes]
AS [nt:unstructured.jcr:mixinTypes], [nt:unstructured].[jcr:path] AS
[nt:unstructured.jcr:path], [nt:unstructured].[jcr:name] AS
[nt:unstructured.jcr:name], [nt:unstructured].[jcr:score] AS
[nt:unstructured.jcr:score], [nt:unstructured].[mode:localName] AS
[nt:unstructured.mode:localName], [nt:unstructured].[mode:depth] AS
[nt:unstructured.mode:depth]], PROJECT_COLUMN_TYPES=[STRING, STRING,
STRING, STRING, DOUBLE, STRING, LONG]>",
          "    Select [nt:unstructured]
<SELECT_CRITERIA=ISCHILDNODE([nt:unstructured],'\/testNode')>",
          "      Select [nt:unstructured] <SELECT_CRITERIA=
[nt:unstructured].[jcr:primaryType] = 'nt:unstructured'>",
          "        Source [nt:unstructured]
<SOURCE_NAME=__ALLNODES__, SOURCE_COLUMNS=[jcr:frozenUuid(STRING),
mode:sharedUuid(REFERENCE), mode:sessionScope(BOOLEAN),
jcr:defaultValues(STRING), mode:projectedNodeKey(STRING),
jcr:mixinTypes(STRING), jcr:frozenPrimaryType(STRING),
jcr:defaultPrimaryType(STRING), jcr:statement(STRING),
jcr:lastModifiedBy(STRING), jcr:mimeType(STRING),
jcr:hasOrderableChildNodes(BOOLEAN), jcr:etag(STRING),
jcr:encoding(STRING), jcr:root(REFERENCE), jcr:supertypes(STRING),
jcr:successors(REFERENCE), jcr:primaryItemName(STRING),
jcr:hold(STRING), jcr:workspace(STRING), jcr:description(STRING),
jcr:primaryType(STRING), mode:externalNodeKey(STRING),
mode:derivedFrom(STRING), mode:isHeldBySession(BOOLEAN),
jcr:baseVersion(REFERENCE), jcr:lastModified(DATE),
jcr:mergeFailed(REFERENCE), mode:derivedAt(DATE),
jcr:requiredPrimaryTypes(STRING), jcr:multiple(BOOLEAN),
mode:generated(BOOLEAN), jcr:activityTitle(STRING),
jcr:lifecyclePolicy(REFERENCE), jcr:isMixin(BOOLEAN),
jcr:availableQueryOperators(STRING),
jcr:childVersionHistory(REFERENCE), jcr:content(REFERENCE),
jcr:autoCreated(BOOLEAN), mode:alias(STRING), jcr:createdBy(STRING),
jcr:isFullTextSearchable(BOOLEAN), jcr:uuid(STRING),
jcr:onParentVersion(STRING), mode:expirationDate(DATE),
jcr:lockIsDeep(BOOLEAN), jcr:copiedFrom(REFERENCE),
jcr:isDeep(BOOLEAN), jcr:title(STRING), jcr:versionableUuid(STRING),
jcr:versionHistory(REFERENCE), jcr:isAbstract(BOOLEAN),
jcr:predecessors(REFERENCE), jcr:lockOwner(STRING),
mode:sha1(STRING), jcr:repository(STRING), jcr:created(DATE),
jcr:frozenMixinTypes(STRING), mode:lockedKey(STRING),
jcr:text(STRING), jcr:host(STRING), jcr:configuration(REFERENCE),
jcr:port(STRING), mode:workspace(STRING), jcr:nodeTypeName(STRING),
jcr:data(BINARY), jcr:isQueryable(BOOLEAN), jcr:language(STRING),
jcr:isQueryOrderable(BOOLEAN), jcr:mandatory(BOOLEAN),
jcr:isCheckedOut(BOOLEAN), jcr:protected(BOOLEAN),
jcr:sameNameSiblings(BOOLEAN), jcr:requiredType(STRING),
jcr:protocol(STRING), mode:lockingSession(STRING),
jcr:messageId(STRING), jcr:id(REFERENCE), mode:uri(STRING),
jcr:valueConstraints(STRING), jcr:retentionPolicy(REFERENCE),
jcr:activity(REFERENCE), jcr:currentLifecycleState(STRING),
jcr:path(STRING), jcr:name(STRING), jcr:score(DOUBLE),
```

```
mode:localName(STRING), mode:depth(LONG)],
SOURCE_ALIAS=nt:unstructured>"
        ]
}
```

Note that the JSON response contains several fields, including the original query statement, the language, the abstract query model (or AQM, which is always equivalent to the JCR-SQL2 form of the query), and the query plan (as an array of strings).

Response Format (as "text/plain"):

```
Access [nt:unstructured]
  Project [nt:unstructured] <PROJECT_COLUMNS=[[nt:unstructured].
[jcr:primaryType] AS [nt:unstructured.jcr:primaryType],
[nt:unstructured].[jcr:mixinTypes] AS
[nt:unstructured.jcr:mixinTypes], [nt:unstructured].[jcr:path] AS
[nt:unstructured.jcr:path], [nt:unstructured].[jcr:name] AS
[nt:unstructured.jcr:name], [nt:unstructured].[jcr:score] AS
[nt:unstructured.jcr:score], [nt:unstructured].[mode:localName] AS
[nt:unstructured.mode:localName], [nt:unstructured].[mode:depth] AS
[nt:unstructured.mode:depth]], PROJECT_COLUMN_TYPES=[STRING, STRING,
STRING, STRING, DOUBLE, STRING, LONG]>
    Select [nt:unstructured]
<SELECT_CRITERIA=ISCHILDNODE([nt:unstructured],'/testNode')>
      Select [nt:unstructured] <SELECT_CRITERIA=[nt:unstructured].
[jcr:primaryType] = 'nt:unstructured'>
        Source [nt:unstructured] <SOURCE_ALIAS=nt:unstructured,
SOURCE_NAME=__ALLNODES__, SOURCE_COLUMNS=[jcr:frozenUuid(STRING),
mode:sharedUuid(REFERENCE), mode:sessionScope(BOOLEAN),
jcr:defaultValues(STRING), mode:projectedNodeKey(STRING),
jcr:mixinTypes(STRING), jcr:frozenPrimaryType(STRING),
jcr:defaultPrimaryType(STRING), jcr:statement(STRING),
jcr:lastModifiedBy(STRING), jcr:mimeType(STRING),
jcr:hasOrderableChildNodes(BOOLEAN), jcr:etag(STRING),
jcr:encoding(STRING), jcr:root(REFERENCE), jcr:supertypes(STRING),
jcr:successors(REFERENCE), jcr:primaryItemName(STRING),
jcr:hold(STRING), jcr:workspace(STRING), jcr:description(STRING),
jcr:primaryType(STRING), mode:externalNodeKey(STRING),
mode:derivedFrom(STRING), mode:isHeldBySession(BOOLEAN),
jcr:baseVersion(REFERENCE), jcr:lastModified(DATE),
jcr:mergeFailed(REFERENCE), mode:derivedAt(DATE),
jcr:requiredPrimaryTypes(STRING), jcr:multiple(BOOLEAN),
mode:generated(BOOLEAN), jcr:activityTitle(STRING),
jcr:lifecyclePolicy(REFERENCE), jcr:isMixin(BOOLEAN),
jcr:availableQueryOperators(STRING),
jcr:childVersionHistory(REFERENCE), jcr:content(REFERENCE),
jcr:autoCreated(BOOLEAN), mode:alias(STRING), jcr:createdBy(STRING),
jcr:isFullTextSearchable(BOOLEAN), jcr:uuid(STRING),
jcr:onParentVersion(STRING), mode:expirationDate(DATE),
jcr:lockIsDeep(BOOLEAN), jcr:copiedFrom(REFERENCE),
jcr:isDeep(BOOLEAN), jcr:title(STRING), jcr:versionableUuid(STRING),
jcr:versionHistory(REFERENCE), jcr:isAbstract(BOOLEAN),
jcr:predecessors(REFERENCE), jcr:lockOwner(STRING),
mode:sha1(STRING), jcr:repository(STRING), jcr:created(DATE),
jcr:frozenMixinTypes(STRING), mode:lockedKey(STRING),
jcr:text(STRING), jcr:host(STRING), jcr:configuration(REFERENCE),
```

```
jcr:port(STRING), mode:workspace(STRING), jcr:nodeTypeName(STRING),
jcr:data(BINARY), jcr:isQueryable(BOOLEAN), jcr:language(STRING),
jcr:isQueryOrderable(BOOLEAN), jcr:mandatory(BOOLEAN),
jcr:isCheckedOut(BOOLEAN), jcr:protected(BOOLEAN),
jcr:sameNameSiblings(BOOLEAN), jcr:requiredType(STRING),
jcr:protocol(STRING), mode:lockingSession(STRING),
jcr:messageId(STRING), jcr:id(REFERENCE), mode:uri(STRING),
jcr:valueConstraints(STRING), jcr:retentionPolicy(REFERENCE),
jcr:activity(REFERENCE), jcr:currentLifecycleState(STRING),
jcr:path(STRING), jcr:name(STRING), jcr:score(DOUBLE),
mode:localName(STRING), mode:depth(LONG)]>
```

The text response only contains the string representation of the query plan.

Reordering nodes

Assuming you create a parent node POSTing the following request:

```
{
    "jcr:primaryType":"nt:unstructured",
    "children":{
        "child1":{
            "prop":"child1"
        },
        "child2":{
            "prop":"child2"
        },
        "child3":{
            "prop":"child3"
        }
    }
}
```

Then you can reorder its children by issuing a PUT request with the following format:

```
{
    "children":{
        "child3":{
        },
        "child2":{
        },
        "child1":{
        }
    }
}
```

Moving nodes

In order to move a node using the REST service, 2 steps are required:

1. Retrieve the node which should be moved and store its ID (the **`id`** member of the JSON response)
2. Edit the parent-to-be node (aka. the new parent) via a PUT request which contains the ID of the node:

```
{
    "children":{
        "child1":{
        },
        "child2":{
        },
        "child3":{
        },
        "41e666ff-0997-4ee0-9eb8-b41319f9f403": {
        }
    }
}
```

Chapter 6. Query and Search

The JCR API defines a way to query a repository for content that meets user-defined criteria. The JCR 2.0 API actually makes it possible for implementations to support multiple query languages, and the specification requires support for two languages: JCR-SQL2 and JCR-QOM. JCR 1.0 defined two other languages (XPath and JCR-SQL), though these languages were deprecated in JCR 2.0.

6.1. Query Languages

At this time, the hierarchical database supports five query languages:

- JCR-SQL2
- JCR-SQL
- XPath
- JCR-JQOM (programmatic API)
- full-text search (a language that reuses the full-text search expression grammar used in the second parameter of the **`CONTAINS(...)`** function of the JCR-SQL2 language)

It is best to pick the language for each query that expresses your application's needs. The JCR-SQL2 language is expressive, and is technically a superset of JCR-SQL. But sometimes it will be easier to specify path-oriented criteria using XPath. Or sometimes you only need to do full-text search, in which case the full-text search language is more appropriate.

Note

Not all JCR implementations execute their queries in the same way. Some (including Jackrabbit) have completely different execution paths for different languages, meaning queries in some languages are faster than equivalent queries expressed in other languages.

6.2. Creating Queries

There are two ways to create a JCR **`Query`** object. The first is by supplying a query expression and the name of the query language, and this can be done with the standard JCR API:

```
// Obtain the query manager for the session via the workspace ...
javax.jcr.query.QueryManager queryManager =
session.getWorkspace().getQueryManager();

// Create a query object ...
String language = ...  // e.g. javax.jcr.query.Query.JCR_SQL2
String expression = ...
javax.jcr.query.Query query = queryManager.createQuery(expression,language);
```

Before returning the **`Query`** , the hierarchical database finds a parser for the language given by the **`language`** parameter, and uses this parser to create a language-independent object representation of the query. (Note that any grammatical errors in the expression result in an immediate exception.) This object representation is what JCR 2.0 calls the "Query Object Model", or QOM. After parsing, the hierarchical database embeds the QOM into the **`Query`** object.

The second approach for creating a **`Query`** object is to programmatically build up the query using the **`QueryObjectModelFactory`** . Again, this uses the standard JCR API. Here's a simple example:

```
// Obtain the query manager for the session via the workspace ...
javax.jcr.query.QueryManager queryManager =
session.getWorkspace().getQueryManager();
javax.jcr.query.qom.QueryObjectModelFactory factory =
queryManager.getQOMFactory();

// Create the parts of a query object ...
javax.jcr.query.qom.Source selector = factory.selector(...);
javax.jcr.query.qom.Constraint constraints = ...
javax.jcr.query.qom.Column[] columns = ...
javax.jcr.query.qom.Ordering[] orderings = ...
javax.jcr.query.qom.QueryObjectModel model =
    factory.createQuery(selector,constraints,orderings,columns);

// The model is a query ...
javax.jcr.query.Query query = model;
```

Of course, the **`QueryObjectModelFactory`** can create lots variations of selectors, joins, constraints, and orderings. The hierarchical database fully supports this style of creating queries, and it even offers some very useful extensions (described below).

6.3. Executing Queries

As we mentioned above, all **`Query`** objects contain the object representation of the query, called the query object model. No matter which query language is used or whether the query was created programmatically, the hierarchical database uses the same kind of model objects to represent every single query.

So when the JCR client executes the query:

```
javax.jcr.query.Query query = ...

// Execute the query and get the results ...
javax.jcr.query.QueryResult result = query.execute();
```

The hierarchical database then takes the query's object model and runs it through a series of steps to plan, validate, optimize, and finally execute the query:

1. **Planning** - in this step, the hierarchical database converts the language-independent query object model into a canonical relational query plan that outlines the various relational operations that need to be performed for this query. The query plan forms a tree, with each leaf node representing an access query against the indexes. However, this plan is not quite ready to be used.

2. **Validation** - not all queries that are well-formed can be executed, so the hierarchical database then validates the canonical query plan to make sure that all named selectors exist, all named properties exist on the selectors, that all aliases are properly used, and that all identifiers are resolvable. If the query fails validation, an exception is thrown immediately.

3. **Optimization** - the canonical plan should mirror the actual query model, but it may not be the most simple or efficient plan. The hierarchical database runs the canonical plan through a rule-based optimizer to produce an optimum and executable plan. For example, one rule rewrites right outer joins as left outer joins. Another rule looks for identity joins (e.g., **`ISSAMENODE`** join criteria or equi-join criteria involving node identifiers), and if possible removes the join altogether (replacing it with

additional criteria) or copies criteria on one side of the join to the other. Another rule removes parts of the plan that (based upon criteria) will never return any rows. Yet another rule determines the best algorithm for joining tuples. Overall, there are about a dozen such rules, and all are intended to make the query plans more easily and efficiently executed.

4. **Execution** - the optimized plan is then executed: each access query in the plan is issued and the resulting tuples processed and combined to form the result set's tuples.

6.4. SQL Extensions

The hierarchical database adds several features to its support of the standard JCR-SQL and JCR-SQL2 grammars. These extensions include support for:

1. Additional join types with **`FULL OUTER JOIN`** and **`CROSS JOIN`**
2. **`UNION`** , **`INTERSECT`** , and **`EXCEPT`** set operations
3. Non-correlated subqueries in the **`WHERE`** clause; multiple subqueries can be used in a single query, and they can even be nested
4. Removing duplicate rows with **`SELECT DISTINCT ...`**
5. Limit the number of rows returned with **`LIMITcount`**
6. Skip initial rows with **`OFFSETnumber`**
7. Constrain the depth of a node with **`DEPTH(selectorName)`**
8. Constrain the path of a node with **`PATH(selectorName)`**
9. Constrain the references from a node with **`REFERENCE(selectorName.property)`** and **`REFERENCE(selectorName)`**
10. Ranges of criteria values using **`BETWEENlowerANDupper`** and optionally specifying whether to exclude the lower and/or upper values
11. Set criteria to specify multiple criteria values using **`IN`** and **`NOT IN`**
12. Use simple arithmetic in criteria and **`ORDER BY`** clauses, such as **`SCORE(type1)*3 + SCORE(type2)`**
13. Use pseudo-columns to include the path, score, node name, node local name, and node depth in result columns or in criteria

More detail of the particular extensions can be found in the JCR-SQL2 grammar.

Use these extensions within your JCR-SQL or JCR-SQL2 query expressions strings, and use the **standard JCR API** to obtain a **`Query`** :

```
// Obtain the query manager for the session via the workspace ...
javax.jcr.query.QueryManager queryManager =
session.getWorkspace().getQueryManager();

// Create a query object ...
String language = ...
String expression = ...  // USE THE EXTENSIONS HERE
javax.jcr.query.Query query = queryManager.createQuery(expression,language);
```

```
// And use the query ...
```

6.5. Query Object Model Extensions

The extensions in the JCR-SQL and JCR-SQL2 languages can also be used when building queries programmatically using the JCR Query Object Model API. The hierarchical database defines the **`org.modeshape.jcr.api.query.qom.QueryObjectModelFactory`** interface that extends the standard **`javax.jcr.query.qom.QueryObjectModelFactory`** interface, and which contains methods providing ways to construct a QOM with the extended features.

6.5.1. Join Types

The standard **`javax.jcr.query.qom.QueryObjectModelFactory`** interface uses a **`String`** to specify the join type:

```
package javax.jcr.query.qom;

public interface QueryObjectModelFactory {
    ...
    /**
     * Performs a join between two node-tuple sources.
     *
     * The query is invalid if 'left' is the same source as 'right'.
     *
     * @param left the left node-tuple source; non-null
     * @param right the right node-tuple source; non-null
     * @param joinType either QueryObjectModelConstants.JCR_JOIN_TYPE_INNER,
     *        QueryObjectModelConstants.JCR_JOIN_TYPE_LEFT_OUTER, or
     *        QueryObjectModelConstants.JCR_JOIN_TYPE_RIGHT_OUTER.
     * @param joinCondition the join condition; non-null
     * @return the join; non-null
     * @throws InvalidQueryException if a particular validity test is
possible on this method,
     *         the implemention chooses to perform that test (and not leave
it until later,
     *         on {@link #createQuery}), and the parameters given fail that
test
     * @throws RepositoryException if the operation otherwise fails
     */
    public Join join( Source left,
                      Source right,
                      String joinType,
                      JoinCondition joinCondition ) throws
InvalidQueryException, RepositoryException;
    ...
}
```

In addition to the three standard constants, the hierarchical database supports two additional constant values:

- **`javax.jcr.query.qom.QueryObjectModelConstants.JCR_JOIN_TYPE_INNER`**
- **`javax.jcr.query.qom.QueryObjectModelConstants.JCR_JOIN_TYPE_LEFT_OUTER`**
- **`javax.jcr.query.qom.QueryObjectModelConstants.JCR_JOIN_TYPE_RIGHT_OUTER`**

- **`org.modeshape.jcr.api.query.qom.QueryObjectModelConstants.JCR_JOIN_TYPE_CROSS`**
- **`org.modeshape.jcr.api.query.qom.QueryObjectModelConstants.JCR_JOIN_TYPE_FULL_OUTER`**

6.5.2. Set Operations

Creating a set query is very similar to creating a normal **`SELECT`** type query, but instead the following on **`org.modeshape.jcr.api.query.qom.QueryObjectModelFactory`** are used:

```
package org.modeshape.jcr.api.query.qom;

public interface QueryObjectModelFactory {
    ...
    /**
     * Creates a query with one or more selectors.
     *
     * @param source the node-tuple source; non-null
     * @param constraint the constraint, or null if none
     * @param orderings zero or more orderings; null is equivalent to a
zero-length array
     * @param columns the columns; null is equivalent to a zero-length array
     * @param limit the limit; null is equivalent to having no limit
     * @param isDistinct true if the query should return distinct values; or
false if no
     *        duplicate removal should be performed
     * @return the select query; non-null
     * @throws InvalidQueryException if a particular validity test is
possible on this method,
     *         the implemention chooses to perform that test and the
parameters given fail that
     *         test. See the individual QOM factory methods for the validity
criteria
     *         of each query element.
     * @throws RepositoryException if another error occurs.
     */
    public SelectQuery select( Source source,
                               Constraint constraint,
                               Ordering[] orderings,
                               Column[] columns,
                               Limit limit,
                               boolean isDistinct ) throws
InvalidQueryException, RepositoryException;

    /**
     * Creates a query command that effectively appends the results of the
right-hand query
     * to those of the left-hand query.
     *
     * @param left the query command that represents left-side of the set
operation;
     *        non-null and must have columns that are equivalent and union-
able to those
     *        of the right-side query
     * @param right the query command that represents right-side of the set
operation;
```

```
     *        non-null and must have columns that are equivalent and union-
able to those
     *        of the left-side query
     * @param orderings zero or more orderings; null is equivalent to a
zero-length array
     * @param limit the limit; null is equivalent to having no limit
     * @param all true if duplicate rows in the left- and right-hand side
results should
     *        be included, or false if duplicate rows should be eliminated
     * @return the select query; non-null
     * @throws InvalidQueryException if a particular validity test is
possible on this method,
     *         the implemention chooses to perform that test and the
parameters given fail
     *         that test. See the individual QOM factory methods for the
validity criteria
     *         of each query element.
     * @throws RepositoryException if another error occurs.
     */
    public SetQuery union( QueryCommand left,
                           QueryCommand right,
                           Ordering[] orderings,
                           Limit limit,
                           boolean all ) throws InvalidQueryException,
RepositoryException;

    /**
     * Creates a query command that returns all rows that are both in the
result of the
     * left-hand query and in the result of the right-hand query.
     *
     * @param left the query command that represents left-side of the set
operation;
     *        non-null and must have columns that are equivalent and union-
able to those
     *        of the right-side query
     * @param right the query command that represents right-side of the set
operation;
     *        non-null and must have columns that are equivalent and union-
able to those
     *        of the left-side query
     * @param orderings zero or more orderings; null is equivalent to a
zero-length array
     * @param limit the limit; null is equivalent to having no limit
     * @param all true if duplicate rows in the left- and right-hand side
results should
     *        be included, or false if duplicate rows should be eliminated
     * @return the select query; non-null
     * @throws InvalidQueryException if a particular validity test is
possible on this method,
     *         the implemention chooses to perform that test and the
parameters given fail
     *         that test. See the individual QOM factory methods for the
validity criteria
     *         of each query element.
     * @throws RepositoryException if another error occurs.
```

```
     */
    public SetQuery intersect( QueryCommand left,
                               QueryCommand right,
                               Ordering[] orderings,
                               Limit limit,
                               boolean all ) throws InvalidQueryException,
RepositoryException;

    /**
     * Creates a query command that returns all rows that are in the result
of the left-hand
     * query but not in the result of the right-hand query.
     *
     * @param left the query command that represents left-side of the set
operation;
     *        non-null and must have columns that are equivalent and union-
able to those
     *        of the right-side query
     * @param right the query command that represents right-side of the set
operation;
     *        non-null and must have columns that are equivalent and union-
able to those
     *        of the left-side query
     * @param orderings zero or more orderings; null is equivalent to a
zero-length array
     * @param limit the limit; null is equivalent to having no limit
     * @param all true if duplicate rows in the left- and right-hand side
results should
     *        be included, or false if duplicate rows should be eliminated
     * @return the select query; non-null
     * @throws InvalidQueryException if a particular validity test is
possible on this method,
     *         the implemention chooses to perform that test and the
parameters given fail
     *         that test. See the individual QOM factory methods for the
validity criteria
     *         of each query element.
     * @throws RepositoryException if another error occurs.
     */
    public SetQuery except( QueryCommand left,
                            QueryCommand right,
                            Ordering[] orderings,
                            Limit limit,
                            boolean all ) throws InvalidQueryException,
RepositoryException;
    ...
}
```

Note that the **`select(...)`** method returns a **`SelectQuery`** while the **`union(...)`** , **`intersect(...)`** and **`except(...)`** methods return a **`SetQuery`** . The **`SelectQuery`** and **`SetQuery`** interfaces are defined by the hierarchical database and both extend the **`QueryCommand`** interface. This interface is then used in the methods to create **`SetQuery`** .

The **`SetQuery`** object is not executable. To create the corresponding **`javax.jcr.Query`** object, pass the **`SetQuery`** to the following method on **`org.modeshape.jcr.api.query.qom.QueryObjectModelFactory`** :

```
package org.modeshape.jcr.api.query.qom;

public interface QueryObjectModelFactory {
    ...
    /**
     * Creates a set query.
     *
     * @param command set query; non-null
     * @return the executable query; non-null
     * @throws InvalidQueryException if a particular validity test is
possible on this method,
     *         the implemention chooses to perform that test and the
parameters given fail
     *         that test. See the individual QOM factory methods for the
validity criteria
     *         of each query element.
     * @throws RepositoryException if another error occurs.
     */
    public SetQueryObjectModel createQuery( SetQuery command ) throws
InvalidQueryException, RepositoryException;
    ...
}
```

The resulting **`SetQueryObjectModel`** extends **`javax.jcr.query.Query`** and **`SetQuery`** and can be executed and treated similarly to the standard **`javax.jcr.query.qom.QueryObjectModel`** (that also extends **`javax.jcr.query.Query`**).

6.5.3. Correlated Subqueries

The hierarchical database defines a **`Subquery`** interface that extends the standard **`javax.jcr.query.qom.StaticOperand`** interface, and thus can be used on the right-hand side of any **`Criteria`** :

```
public interface Subquery extends StaticOperand {
    /**
     * Gets the {@link QueryCommand} that makes up the subquery.
     *
     * @return the query command; non-null
     */
    public QueryCommand getQuery();
}
```

Subqueries can be created by passing a **`QueryCommand`** into this **`org.modeshape.jcr.query.qom.QueryObjectModelFactory`** method:

```
package org.modeshape.jcr.api.query.qom;

public interface QueryObjectModelFactory {
    ...
    /**
     * Creates a subquery that can be used as a {@link StaticOperand} in
another query.
     *
     * @param subqueryCommand the query command that is to be used as the
```

```
subquery
     * @return the constraint; non-null
     * @throws InvalidQueryException if a particular validity test is
possible on this method,
     *         the implemention chooses to perform that test (and not leave
it until later,
     *         on {@link #createQuery}), and the parameters given fail that
test
     * @throws RepositoryException if the operation otherwise fails
     */
    public Subquery subquery( QueryCommand subqueryCommand ) throws
InvalidQueryException, RepositoryException;
    ...
}
```

The resulting **`Subquery`** is a **`StaticOperand`** that can then be used to create a **`Criteria`** .

6.5.4. Removing Duplicate Rows

The **`org.modeshape.jcr.query.qom.QueryObjectModelFactory`** interface includes a variation of the standard **`QueryObjectModeFactory.select(...)`** method with an additional **`isDistinct`** flag that controls whether duplicate rows should be removed:

```
package org.modeshape.jcr.api.query.qom;

public interface QueryObjectModelFactory {
    ...
    /**
     * Creates a query with one or more selectors.
     *
     * @param source the node-tuple source; non-null
     * @param constraint the constraint, or null if none
     * @param orderings zero or more orderings; null is equivalent to a
zero-length array
     * @param columns the columns; null is equivalent to a zero-length array
     * @param limit the limit; null is equivalent to having no limit
     * @param isDistinct true if the query should return distinct values; or
false if no
     *        duplicate removal should be performed
     * @return the select query; non-null
     * @throws InvalidQueryException if a particular validity test is
possible on this method,
     *         the implemention chooses to perform that test and the
parameters given fail
     *         that test. See the individual QOM factory methods for the
validity criteria
     *         of each query element.
     * @throws RepositoryException if another error occurs.
     */
    public SelectQuery select( Source source,
                               Constraint constraint,
                               Ordering[] orderings,
                               Column[] columns,
                               Limit limit,
```

```
                                    boolean isDistinct ) throws
InvalidQueryException, RepositoryException;
    ...
}
```

6.5.5. Limit and Offset Results

The hierarchical database defines a **`Limit`** interface as a top-level object that can be used to create queries that limit the number of rows and/or skip a number of initial rows:

```
public interface Limit {

    /**
     * Get the number of rows skipped before the results begin.
     *
     * @return the offset; always 0 or a positive number
     */
    public int getOffset();

    /**
     * Get the maximum number of rows that are to be returned.
     *
     * @return the maximum number of rows; always positive, or equal to
Integer.MAX_VALUE if there is no limit
     */
    public int getRowLimit();

    /**
     * Determine whether this limit clause is necessary.
     *
     * @return true if the number of rows is not limited and there is no
offset, or false otherwise
     */
    public boolean isUnlimited();

    /**
     * Determine whether this limit clause defines an offset.
     *
     * @return true if there is an offset, or false if there is no offset
     */
    public boolean isOffset();
}
```

These range constraints can be constructed using this **`org.modeshape.jcr.query.qom.QueryObjectModelFactory`** method:

```
package org.modeshape.jcr.api.query.qom;

public interface QueryObjectModelFactory {
    ...
    /**
     * Evaluates to a limit on the maximum number of tuples in the results
and the
     * number of rows that are skipped before the first tuple in the
results.
```

```
     *
     * @param rowLimit the maximum number of rows; must be a positive
number, or Integer.MAX_VALUE if there is to be a
     *        non-zero offset but no limit
     * @param offset the number of rows to skip before beginning the
results; must be 0 or a positive number
     * @return the operand; non-null
     * @throws InvalidQueryException if a particular validity test is
possible on this method,
     *         the implemention chooses to perform that test (and not leave
it until later, on createQuery),
     *         and the parameters given fail that test
     * @throws RepositoryException if the operation otherwise fails
     */
    public Limit limit( int rowLimit,
                        int offset ) throws InvalidQueryException,
RepositoryException;
    ...
}
```

The **`Limit`** objects can then be used when creating queries using a variation of the standard **`QueryObjectModeFactory.select(...)`** defined in the **`org.modeshape.jcr.query.qom.QueryObjectModelFactory`** interface:

```
package org.modeshape.jcr.api.query.qom;

public interface QueryObjectModelFactory {
    ...
    /**
     * Creates a query with one or more selectors.
     *
     * @param source the node-tuple source; non-null
     * @param constraint the constraint, or null if none
     * @param orderings zero or more orderings; null is equivalent to a
zero-length array
     * @param columns the columns; null is equivalent to a zero-length array
     * @param limit the limit; null is equivalent to having no limit
     * @param isDistinct true if the query should return distinct values; or
false if no
     *        duplicate removal should be performed
     * @return the select query; non-null
     * @throws InvalidQueryException if a particular validity test is
possible on this method,
     *         the implemention chooses to perform that test and the
parameters given fail
     *         that test. See the individual QOM factory methods for the
validity criteria
     *         of each query element.
     * @throws RepositoryException if another error occurs.
     */
    public SelectQuery select( Source source,
                               Constraint constraint,
                               Ordering[] orderings,
                               Column[] columns,
                               Limit limit,
```

```
                                boolean isDistinct ) throws
InvalidQueryException, RepositoryException;
    ...
}
```

Similarly, the **`Limit`** objects can be passed to the hierarchical database **`except(...)`** , **`union(...)`** , **`intersect(...)`** methods, too.

6.5.6. Depth Constraints

The hierarchical database defines a **`DepthPath`** interface that extends the standard **`javax.jcr.query.qom.DynamicOperand`** interface, and thus can be used as part of a **`WHERE`** clause to constrain the depth of the nodes accessed by a selector:

```
public interface NodeDepth extends javax.jcr.query.qom.DynamicOperand {

    /**
     * Get the selector symbol upon which this operand applies.
     *
     * @return the one selector names used by this operand; never null
     */
    public String getSelectorName();
}
```

These range constraints can be constructed using this **`org.modeshape.jcr.query.qom.QueryObjectModelFactory`** method:

```
package org.modeshape.jcr.api.query.qom;

public interface QueryObjectModelFactory {
    ...
    /**
     * Evaluates to a LONG value equal to the depth of a node in the
specified selector.
     *
     * The query is invalid if selector is not the name of a selector in the
query.
     *
     * @param selectorName the selector name; non-null
     * @return the operand; non-null
     * @throws InvalidQueryException if a particular validity test is
possible on this method,
     *         the implemention chooses to perform that test (and not leave
it until later, on createQuery),
     *         and the parameters given fail that test
     * @throws RepositoryException if the operation otherwise fails
     */
    public NodeDepth nodeDepth( String selectorName ) throws
InvalidQueryException, RepositoryException;
    ...
}
```

6.5.7. Path Constraints

The hierarchical database defines a **`NodePath`** interface that extends the standard **`javax.jcr.query.qom.DynamicOperand`** interface, and thus can be used as part of a **`WHERE`** clause to constrain the path of nodes accessed by a selector:

```
public interface NodePath extends javax.jcr.query.qom.DynamicOperand {

    /**
     * Get the selector symbol upon which this operand applies.
     *
     * @return the one selector names used by this operand; never null
     */
    public String getSelectorName();
}
```

These range constraints can be constructed using this **`org.modeshape.jcr.query.qom.QueryObjectModelFactory`** method:

```
package org.modeshape.jcr.api.query.qom;

public interface QueryObjectModelFactory {
    ...
    /**
     * Evaluates to a PATH value equal to the prefix-qualified path of a
node in the specified selector.
     *
     * The query is invalid if selector is not the name of a selector in the
query.
     *
     * @param selectorName the selector name; non-null
     * @return the operand; non-null
     * @throws InvalidQueryException if a particular validity test is
possible on this method,
     *         the implemention chooses to perform that test (and not leave
it until later, on createQuery),
     *         and the parameters given fail that test
     * @throws RepositoryException if the operation otherwise fails
     */
    public NodePath nodePath( String selectorName ) throws
InvalidQueryException, RepositoryException;
    ...
}
```

6.5.8. Criteria on References From a Node

The hierarchical database defines a **`ReferenceValue`** interface that extends the standard **`javax.jcr.query.qom.DynamicOperand`** interface, and thus can be used as part of a **`WHERE`** or **`ORDER BY`** clause:

```
public interface ReferenceValue extends DynamicOperand {
    ...
    /**
     * Get the selector symbol upon which this operand applies.
     *
     * @return the one selector names used by this operand; never null
```

```
     */
    public String getSelectorName();

    /**
     * Get the name of the one reference property.
     *
     * @return the property name; or null if this operand applies to any
reference property
     */
    public String getPropertyName();
}
```

These reference value operand allow a query to easily place constraints on a particular REFERENCE property or (more importantly) any REFERENCE properties on the nodes. The former is a more simple alternative to using a regular comparison constraint with the REFERENCE property on one side and the **`jcr:uuid`** property on the other. The latter effectively means "where the node references (with any property) some other nodes", and this is something that standard JCR-SQL2 cannot represent.

They are created using these **`org.modeshape.jcr.query.qom.QueryObjectModelFactory`** methods:

```
package org.modeshape.jcr.api.query.qom;

public interface QueryObjectModelFactory {
    ...
    /**
     * Creates a dynamic operand that evaluates to the REFERENCE value of
the any property
     * on the specified selector.
     *
     * The query is invalid if:
     * - selector is not the name of a selector in the query, or
     * - property is not a syntactically valid JCR name.
     *
     * @param selectorName the selector name; non-null
     * @return the operand; non-null
     * @throws InvalidQueryException if a particular validity test is
possible on this method,
     *         the implemention chooses to perform that test (and not leave
it until later, on createQuery),
     *         and the parameters given fail that test
     * @throws RepositoryException if the operation otherwise fails
     */
    public ReferenceValue referenceValue( String selectorName ) throws
InvalidQueryException, RepositoryException;

    /**
     * Creates a dynamic operand that evaluates to the REFERENCE value of
the specified
     * property on the specified selector.
     *
     * The query is invalid if:
     * - selector is not the name of a selector in the query, or
     * - property is not a syntactically valid JCR name.
     *
     * @param selectorName the selector name; non-null
     * @param propertyName the reference property name; non-null
```

```
     * @return the operand; non-null
     * @throws InvalidQueryException if a particular validity test is
possible on this method,
     *         the implemention chooses to perform that test (and not leave
it until later, on createQuery),
     *         and the parameters given fail that test
     * @throws RepositoryException if the operation otherwise fails
     */
    public ReferenceValue referenceValue( String selectorName,
                                          String propertyName ) throws
InvalidQueryException, RepositoryException;
    ...
}
```

6.5.9. Range Criteria

The hierarchical database defines a **`Between`** interface that extends the standard **`javax.jcr.query.qom.Constraint`** interface, and thus can be used as part of a **`WHERE`** clause:

```
public interface Between extends Constraint {

    /**
     * Get the dynamic operand specification.
     *
     * @return the dynamic operand; never null
     */
    public DynamicOperand getOperand();

    /**
     * Get the lower bound operand.
     *
     * @return the lower bound; never null
     */
    public StaticOperand getLowerBound();

    /**
     * Get the upper bound operand.
     *
     * @return the upper bound; never null
     */
    public StaticOperand getUpperBound();

    /**
     * Return whether the lower bound is to be included in the results.
     *
     * @return true if the {@link #getLowerBound() lower bound} is to be
included, or false otherwise
     */
    public boolean isLowerBoundIncluded();

    /**
     * Return whether the upper bound is to be included in the results.
     *
     * @return true if the {@link #getUpperBound() upper bound} is to be
```

```
included, or false otherwise
     */
    public boolean isUpperBoundIncluded();
}
```

These range constraints can be constructed using this **`org.modeshape.jcr.query.qom.QueryObjectModelFactory`** method:

```
package org.modeshape.jcr.api.query.qom;

public interface QueryObjectModelFactory {
    ...
    /**
     * Tests that the value (or values) defined by the supplied dynamic
operand are
     * within a specified range. The range is specified by a lower and upper
bound,
     * and whether each of the boundary values is included in the range.
     *
     * @param operand the dynamic operand describing the values that are to
be constrained
     * @param lowerBound the lower bound of the range
     * @param upperBound the upper bound of the range
     * @param includeLowerBound true if the lower boundary value is not be
included
     * @param includeUpperBound true if the upper boundary value is not be
included
     * @return the constraint; non-null
     * @throws InvalidQueryException if a particular validity test is
possible on this method,
     *         the implemention chooses to perform that test (and not leave
it until later, on createQuery),
     *         and the parameters given fail that test
     * @throws RepositoryException if the operation otherwise fails
     */
    public Between between( DynamicOperand operand,
                            StaticOperand lowerBound,
                            StaticOperand upperBound,
                            boolean includeLowerBound,
                            boolean includeUpperBound ) throws
InvalidQueryException, RepositoryException;
    ...
}
```

To create a **`NOT BETWEEN ...`** criteria, create the **`Between`** criteria object, and then pass that into the standard **`QueryObjectModelFactory.not(Criteria)`** method.

6.5.10. Set Criteria

The hierarchical database defines a **`SetCriteria`** interface that extends the standard **`javax.jcr.query.qom.Constraint`** interface, and thus can be used as part of a **`WHERE`** clause:

```
public interface SetCriteria extends Constraint {

    /**
```

```
     * Get the dynamic operand specification for the left-hand side of the
set criteria.
     *
     * @return the dynamic operand; never null
     */
    public DynamicOperand getOperand();

    /**
     * Get the static operands for this set criteria.
     *
     * @return the static operand; never null and never empty
     */
    public Collection<? extends StaticOperand> getValues();
}
```

These set constraints can be constructed using this **`org.modeshape.jcr.query.qom.QueryObjectModelFactory`** method:

```
package org.modeshape.jcr.api.query.qom;

public interface QueryObjectModelFactory {
    ...
   /**
     * Tests that the value (or values) defined by the supplied dynamic
operand are
     * found within the specified set of values.
     *
     * @param operand the dynamic operand describing the values that are to
be constrained
     * @param values the static operand values; may not be null or empty
     * @return the constraint; non-null
     * @throws InvalidQueryException if a particular validity test is
possible on this method,
     *         the implemention chooses to perform that test (and not leave
it until later, on createQuery),
     *         and the parameters given fail that test
     * @throws RepositoryException if the operation otherwise fails
     */
    public SetCriteria in( DynamicOperand operand,
                           StaticOperand... values ) throws
InvalidQueryException, RepositoryException;
    ...
}
```

To create a **`NOT IN`** criteria, create the **`IN`** criteria to get a **`SetCriteria`** object, and then pass that into the standard **`QueryObjectModelFactory.not(Criteria)`** method.

6.5.11. Arithmetic Operands

The hierarchical database defines an **`ArithmeticOperand`** interface that extends the **`javax.jcr.query.qom.DynamicOperand`** , and thus can be used anywhere a **`DynamicOperand`** can be used.

```
public interface ArithmeticOperand extends DynamicOperand {

```

```
    /**
     * Get the operator for this binary operand.
     *
     * @return the operator; never null
     */
    public String getOperator();

    /**
     * Get the left-hand operand.
     *
     * @return the left-hand operator; never null
     */
    public DynamicOperand getLeft();

    /**
     * Get the right-hand operand.
     *
     * @return the right-hand operator; never null
     */
    public DynamicOperand getRight();
}
```

These can be constructed using additional **`org.modeshape.jcr.query.qom.QueryObjectModelFactory`** methods:

```
package org.modeshape.jcr.api.query.qom;

public interface QueryObjectModelFactory {
    ...
    /**
     * Create an arithmetic dynamic operand that adds the numeric value of
the two supplied operand(s).
     *
     * @param left the left-hand-side operand; not null
     * @param right the right-hand-side operand; not null
     * @return the dynamic operand; non-null
     * @throws InvalidQueryException if a particular validity test is
possible on this method,
     *         the implemention chooses to perform that test (and not leave
it until later, on createQuery),
     *         and the parameters given fail that test
     * @throws RepositoryException if the operation otherwise fails
     */
    public ArithmeticOperand add( DynamicOperand left,
                                  DynamicOperand right ) throws
InvalidQueryException, RepositoryException;

    /**
     * Create an arithmetic dynamic operand that subtracts the numeric value
of the second operand from the numeric value of the
     * first.
     *
     * @param left the left-hand-side operand; not null
     * @param right the right-hand-side operand; not null
     * @return the dynamic operand; non-null
     * @throws InvalidQueryException if a particular validity test is
```

```
possible on this method,
     *         the implemention chooses to perform that test (and not leave
it until later, on createQuery),
     *         and the parameters given fail that test
     * @throws RepositoryException if the operation otherwise fails
     */
    public ArithmeticOperand subtract( DynamicOperand left,
                                       DynamicOperand right ) throws
InvalidQueryException, RepositoryException;

    /**
     * Create an arithmetic dynamic operand that multplies the numeric value
of the first operand by the numeric value of the
     * second.
     *
     * @param left the left-hand-side operand; not null
     * @param right the right-hand-side operand; not null
     * @return the dynamic operand; non-null
     * @throws InvalidQueryException if a particular validity test is
possible on this method,
     *         the implemention chooses to perform that test (and not leave
it until later, on createQuery),
     *         and the parameters given fail that test
     * @throws RepositoryException if the operation otherwise fails
     */
    public ArithmeticOperand multiply( DynamicOperand left,
                                       DynamicOperand right ) throws
InvalidQueryException, RepositoryException;

    /**
     * Create an arithmetic dynamic operand that divides the numeric value
of the first operand by the numeric value of the
     * second.
     *
     * @param left the left-hand-side operand; not null
     * @param right the right-hand-side operand; not null
     * @return the dynamic operand; non-null
     * @throws InvalidQueryException if a particular validity test is
possible on this method,
     *         the implemention chooses to perform that test (and not leave
it until later, on createQuery),
     *         and the parameters given fail that test
     * @throws RepositoryException if the operation otherwise fails
     */
    public ArithmeticOperand divide( DynamicOperand left,
                                     DynamicOperand right ) throws
InvalidQueryException, RepositoryException;
    ...
}
```

6.6. Search and Text Extraction

The full-text search language and JCR-SQL2's full-text search constraint both have the ability to find nodes using a simpler search-engine-like expression with wildcards and phrases.

One can imagine how the hierarchical database performs these matches against a node's name and properties containing STRING, LONG, DATE, DOUBLE, DECIMAL, NAME, and PATH values. But for BINARY values, in order to determine whether the search expressions match, the hierarchical database has to determine what text is contained within each BINARY value. Indeed, the hierarchical database can only match against the BINARY value if it can extract the text from that value. This is where text extraction comes into play.

A **text extractor** is a component that knows how to extract searchable text from a BINARY value. Each text extract describes whether it can process files of a particular MIME type. If it can, the hierarchical database will (when necessary) call the extractor to obtain the searchable text for a supplied BINARY value.

Chapter 7. Query Language Grammars

The hierarchical database supports multiple query languages, including all four languages defined in the JCR 1.0 specification and JCR 2.0 specification .

7.1. JCR-SQL2

The **JCR-SQL2** query language is defined by the JCR 2.0 specification as a way to express queries using strings that are similar to SQL. This query language is an improvement over the earlier JCR-SQL language, providing among other things far richer specifications of joins and criteria.

7.1.1. Extensions to JCR-SQL2

The hierarchical database includes full support for the complete JCR-SQL2 query language defined by the specification. However, there are several extensions provided to make it even more powerful:

- Support for the **`FULL OUTER JOIN`** and **`CROSS JOIN`** join types, in addition to the **`LEFT OUTER JOIN`**, **`RIGHT OUTER JOIN`** and **`INNER JOIN`** types defined by JCR-SQL2. Note that **`JOIN`** is a shorthand for **`INNER JOIN`**. For detail, see the grammar for "joins" .
- Support for the **`UNION`** , **`INTERSECT`** , and **`EXCEPT`** set operations on multiple result sets to form a single result set. As with standard SQL, the result sets being combined must have the same columns. The **`UNION`** operator combines the rows from two result sets, the **`INTERSECT`** operator returns the difference between two result sets, and the **`EXCEPT`** operator returns the rows that are common to two result sets. Duplicate rows are removed unless the operator is followed by the **`ALL`** keyword. For detail, see the grammar for "set queries" .
- Removal of duplicate rows in the results, using **`SELECT DISTINCT ...`** expression. For detail, see the grammar for queries .
- Limiting the number of rows in the result set with the **`LIMIT count`** clause, where **`count`** is the maximum number of rows that should be returned. This clause may optionally be followed by the **`OFFSET number`** clause to specify the number of initial rows that should be skipped. For detail, see the grammar for "limits and offsets" .
- Additional dynamic operands **`DEPTH(selectorName)`** and **`PATH(selectorName)`** that enable placing constraints on the node depth and path, respectively. These dynamic operands can be used in a manner similar to **`NAME(selectorName)`** and **`LOCALNAME(selectorName)`** that are defined by JCR-SQL2. Note in each of these cases, the **`selectorName`** is optional if there is only one selector in the query. For detail, see the grammar for "dynamic operands" .
- Additional dynamic operand **`REFERENCE(selectorName.propertyName)`** and **`REFERENCE(selectorName)`** that enables placing constraints on one or any of the reference properties, respectively, and which can be used in a manner similar to the standard dynamic operand **`PropertyValue(selectorName.propertyName)`**. Note in each of these cases, the **`selectorName`** is optional if there is only one selector in the query, and that the **`propertyName`** can be excluded if the constraint should apply to all reference properties. For detail, see the grammar for "dynamic operands" .
- Support for the **`IN`** and **`NOT IN`** clauses to more easily and concisely supply multiple of discrete static operands. For example, **`WHERE ... [my:type].[prop1] IN (3,5,7,10,11,50) ...`**. For detail, see the grammar for "set constraints" .

- Support for the **BETWEEN** clause to more easily and concisely supply a range of discrete operands. For example, **WHERE ... [my:type].[prop1] BETWEEN 3 EXCLUSIVE AND 10 ...**. For detail, see the grammar for "between constraints" .
- Support for simple arithmetic in numeric-based criteria and order-by clauses. For example, **... WHERE SCORE(type1) + SCORE(type2) > 1.0** or **... ORDER BY (SCORE(type1) * SCORE(type2)) ASC, LENGTH(type2.property1) DESC**. For detail, see the grammar for "order-by clauses" .
- Support for (non-correlated) subqueries in the **WHERE** clause, wherever a static operand can be used. Subqueries can even be used within another subquery. All subqueries must return a single column, and each row's single value will be treated as a literal value. If the subquery is used in a clause that expects a single value (e.g., in a comparison), only the subquery's first row will be used. If the subquery is used in a clause that allows multiple values (e.g., **IN (...)**), then all of the subquery's rows will be used. For example, this expression **WHERE ... [my:type].[prop1] IN (SELECT [my:prop2] FROM [my:type2] WHERE [my:prop3] < '1000') AND ...** will use the results of the subquery as the literal values in the **IN** clause. See the "subqueries" section for more information.
- Support for several pseudo-columns (**jcr:path**, **jcr:score**, **jcr:name**, **mode:localName**, and **mode:depth**) that can be used in the **SELECT** , equijoin, and **WHERE** clauses. These pseudo-columns make it possible to return location-related and score information within the **QueryResult** 's rows. They also make queries look more like SQL, and thus may be more friendly and easier to use in existing SQL-aware client applications. See the "pseudo-columns" section for more information.
- Support for **NOT LIKE** as an operator in comparison criteria, and which is equivalent to wrapping a **LIKE** comparison criteria in a **NOT(...)** clause.

7.1.2. Extended JCR-SQL2 Grammar

The full grammar for the hierarchical database's extended JCR-SQL2 support is a strict superset of that defined by the JCR 2.0 specification. In other words, Any JCR-SQL2 query that uses the standard grammar it supported, as well as queries that make use of the provided extensions.

7.1.2.1. Queries

The top-level rule for the extended JCR-SQL2 grammar is **QueryCommand** , which consists of both **Query** and **SetQuery** :

```
QueryCommand ::= Query | SetQuery

SetQuery ::= Query ('UNION'|'INTERSECT'|'EXCEPT') ['ALL'] Query
                 { ('UNION'|'INTERSECT'|'EXCEPT') ['ALL'] Query }

Query ::= 'SELECT' ['DISTINCT'] columns
          'FROM' Source
          ['WHERE' Constraint]
          ['ORDER BY' orderings]
          [Limit]
```

The hierarchical database adds the concept of a set query, which is a query that performs a union , intersection , or complement of the results of two other queries. Set queries are common in SQL (which is essentially a set manipulation language) and are a very useful tool that would otherwise require significant processing of the results of multiple queries by the application. By supporting set queries, the application merely needs to declare that set operation be performed, and the hierarchical database will perform all the work before returning the results.

There is also the ability to use **`SELECT DISTINCT`**, which eliminates duplicate rows in a manner similar to SQL.

7.1.2.2. Source

A source is a named set of tuples, which in the hierarchical database corresponds to the nodes of a particular named node type. In other words, a source is equivalent to a table in a relational database. The available columns of a source are the named properties declared on the node type.

In the JCR-SQL2 grammar, a source is either a selector (a named node type) or a join specification:

```
Source ::= Selector | Join

Selector ::= nodeTypeName ['AS' selectorName]

nodeTypeName ::= Name
selectorName ::= /* A string that contains only SQL-legal characters,
                    and which can be used elsewhere in the query to
                    refer to the selector. */
```

See Also:

- Section 7.1.2.5, “Path and Name”

7.1.2.3. Joins

The JCR 2.0 specification does include joins in the standard JCR-SQL2 grammar, though the only defined types of joins included inner , left outer , and right outer joins. Because SQL also defines the useful full outer and cross join types, the hierarchical database adds support for these.

```
Join ::= left [JoinType] 'JOIN' right 'ON' JoinCondition
         /* If JoinType is omitted INNER is assumed. */

left ::= Source
right ::= Source

JoinType ::= Inner | LeftOuter | RightOuter | FullOuter | Cross

Inner ::= 'INNER' ['JOIN']

LeftOuter ::= 'LEFT JOIN' | 'OUTER JOIN' | 'LEFT OUTER JOIN'

RightOuter ::= 'RIGHT OUTER' ['JOIN']

RightOuter ::= 'FULL OUTER' ['JOIN']

RightOuter ::= 'CROSS' ['JOIN']

JoinCondition ::= EquiJoinCondition |
                  SameNodeJoinCondition |
                  ChildNodeJoinCondition |
                  DescendantNodeJoinCondition
```

Each of the four kinds of join conditions are described below.

join condition

An equijoin is a join that uses only equality comparisons in the join predicate (or join condition). Using any other operators (e.g., '**<**' or '**!=**') in the join condition disqualifies a query from being an equi-join.

Therefore, the rules for the equi-join condition are as follows:

```
EquiJoinCondition ::= selector1Name'.'property1Name
                      '=' selector2Name'.'property2Name

selector1Name ::= selectorName
selector2Name ::= selectorName
property1Name ::= propertyName
property2Name ::= propertyName

propertyName ::= Name
```

where the node type referenced by the selector identified in the query with the **`selector1Name`** must contain the property given by the **`property1Name`** literal, and similarly the node type referenced by the selector identified in the query with the **`selector2Name`** must contain the property given by the **`property2Name`** literal.

See also the "name rule" .

node join condition

An identity join is a special case of an equijoin, where the compared properties are node identifiers. Thus the join condition of an identity join constrains the node on one sides of the join to be the same node on the other side of the join. The standard JCR-SQL2 grammar defines a special function that makes this a little easier to use:

```
SameNodeJoinCondition ::= 'ISSAMENODE(' selector1Name
                                     ',' selector2Name
                                    [',' selector2Path] ')'

selector1Name ::= selectorName
selector2Name ::= selectorName
selector2Path ::= Path
```

See also the "path rule" .

Child-node join condition

A child-node join is one where the join condition constrains the node on the left side of the join to be a child of the node on the right side of the join. The standard JCR-SQL2 grammar defines a special function that makes it easier to specify such join conditions:

```
ChildNodeJoinCondition ::= 'ISCHILDNODE('
                           childSelectorName ','
                           parentSelectorName ')'

childSelectorName ::= selectorName
parentSelectorName ::= selectorName
```

Descendant-node join condition

A descendant-node join is one where the join condition constrains the node on the left side of the join to be a descendant of the node on the right side of the join. The standard JCR-SQL2 grammar defines a special function that makes it easier to specify such join conditions:

```
DescendantNodeJoinCondition ::= 'ISDESCENDANTNODE('
                                descendantSelectorName ','
                                ancestorSelectorName ')'

descendantSelectorName ::= selectorName
ancestorSelectorName ::= selectorName
```

See Also:

- Section 7.1.2.5, “Path and Name”

7.1.2.4. Constraints

The "query rule" included a **WHERE** clause that can define multiple constraints on the nodes included in the results. The standard JCR-SQL2 grammar defined several such constraints, including and , or , not , comparison , property existence , full-text search , same-node , child-node , and descendant-node constraints. The hierarchical database supports all of these, but adds two others: between and set constraints.

```
Constraint ::= ConstraintItem | '(' ConstraintItem ')'

ConstraintItem ::= And | Or | Not
                   Comparison | Between |
                   PropertyExistence |
                   SetConstraint |
                   FullTextSearch |
                   SameNode |
                   ChildNode |
                   DescendantNode
```

Each of these types of constraints are described below.

And constraint

An and constraint stipulates that a node (or record or tuple) is included only if two other constraints are both true.

```
And ::= constraint1 'AND' constraint2

constraint1 ::= Constraint
constraint2 ::= Constraint
```

Or constraint

An or constraint stipulates that a node (or record or tuple) is included if either of two other constraints are true.

```
Or ::= constraint1 'OR' constraint2

constraint1 ::= Constraint
constraint2 ::= Constraint
```

Not constraint

The not qualifier will negate another constraint, requiring that a node (or record or tuple) is included if the other constraint is not true.

```
Not ::= 'NOT' constraint

constraint ::= Constraint
```

Comparison constraint

A comparison constraint requires that the value for a node described by the dynamic operand on the left side of the operator is to be compared to a static literal value. The term "dynamic operand" is used in the JCR-SQL2 grammar because its value can only be determined during query evaluation.

```
Comparison ::= DynamicOperand Operator StaticOperand

Operator ::= '=' | '!=' | '<' | '<=' | '>' | '>=' | 'LIKE' | 'NOT
LIKE'
```

The behavior of the operators is dictated by the JCR 2.0 specification and matches how **`Value`** objects are compared:

- If the **`DynamicOperand`** evaluates to null, the constraint is not satisfied.
- If the '**`=`**' operator is used, the value that the **`DynamicOperand`** evaluates to must equal the **`StaticOperand`** value for the constraint to be satisfied.
- If the '**`!=`**' operator is used, the value that the **`DynamicOperand`** evaluates to must not equal the **`StaticOperand`** value for the constraint to be satisfied.
- If the '**`<`**' operator is used, the value that the **`DynamicOperand`** evaluates to must be less than the **`StaticOperand`** value for the constraint to be satisfied.
- If the '**`<=`**' operator is used, the value that the **`DynamicOperand`** evaluates to must be less than or equal to the **`StaticOperand`** value for the constraint to be satisfied.
- If the '**`>`**' operator is used, the value that the **`DynamicOperand`** evaluates to must be greater than the **`StaticOperand`** value for the constraint to be satisfied.
- If the '**`>=`**' operator is used, the value that the **`DynamicOperand`** evaluates to must be greater than or equal to the **`StaticOperand`** value for the constraint to be satisfied.
- If the '**`LIKE`**' operator is used, the constraint is only satisfied if the value that the **`DynamicOperand`** evaluates to match the pattern specified by the string literal **`StaticOperand`** , where in the pattern:
  - the character '**`%`**' matches zero or more characters, and
  - the character '_' (underscore) matches exactly one character, and
  - the string '**`\x`**' matches the character '**`x`**', and
  - all other characters match themselves

- If the **`NOT LIKE`** operator is used, the constraint is only satisfied if the value that the **`DynamicOperand`** evaluates to not match the pattern specified by the string literal **`StaticOperand`** , where in the pattern:
  - the character '**`%`**' matches zero or more characters, and
  - the character '_' (underscore) matches exactly one character, and
  - the string '**`\x`**' matches the character '**`x`**', and
  - all other characters match themselves

Also, note that, unlike SQL, the standard JCR-SQL2 grammar does not allow the left-hand side and right-hand sides of a comparison constraint to be swapped.

Between constraint

The between constraint is one of the extensions defined by the hierarchical database, and allows a query to more easily represent a range of static values than using only the constraints available in the standard JCR-SQL2 grammar. The between constraint is based on the similar expression in SQL.

```
Between ::= DynamicOperand ['NOT'] 'BETWEEN'
               lowerBound ['EXCLUSIVE'] 'AND'
               upperBound ['EXCLUSIVE']

lowerBound ::= StaticOperand
upperBound ::= StaticOperand
```

Property existence constraint

A property existence constraint stipulates that a property does indeed exist on a node that is of the node type specified by the named selector. the hierarchical database does allow the **`NOT`** qualifier to be excluded, which turns the constraint into a stipulation that the property does not exist on the node.

```
PropertyExistence ::= [selectorName'.']propertyName 'IS' ['NOT']
'NULL'

                     /* If only one selector exists in this query,
                        explicit specification of the selectorName
                        preceding the propertyName is optional */
```

Set constraint

Like the "between constraint", the set constraint is an extension to the standard JCR-SQL2 grammar that allows what would normally be a complicated combination of standard JCR-SQL2 constraints to be more easily represented with a single, simple expression. Again, this constraint is patterned after the similar expression in SQL.

```
SetConstraint ::= [selectorName '.']propertyName ['NOT'] 'IN'
                    '(' firstStaticOperand
                        {',' additionalStaticOperand }
                    ')'

                   /* If only one selector exists in this query,
                      explicit specification of the selectorName
```

```
                        preceding the propertyName is optional */

firstStaticOperand ::= StaticOperand
additionalStaticOperand ::= StaticOperand
```

Note that multiple static operands can be included in the comma-separated list.

Although this rule seems complicated, it is actually very straightforward. The following query selects all the properties defined on the **`acme:taggable`** node type, returning only those "taggable" nodes with a **`acme:tagname`** value of "tag1", "tag2", "tag3", or "tag4":

```
SELECT * FROM [acme:taggable] as tagged
WHERE tagged.[acme:tagName] IN ('tag1','tag2','tag3','tag4')
```

Even this trivial query is quite a bit simpler and easier to understand than if the query had used only the constraints defined by the standard JCR-SQL2 grammar:

```
SELECT * FROM [acme:taggable] as tagged
WHERE tagged.[acme:tagName] = 'tag1'
   OR tagged.[acme:tagName] = 'tag2'
   OR tagged.[acme:tagName] = 'tag3'
   OR tagged.[acme:tagName] = 'tag4'
```

Imagine how complicated a query might be with multiple joins, multiple criteria, and many values to be compared for one or several different properties.

text search constraint

```
FullTextSearch ::= 'CONTAINS('
                      ([selectorName'.']propertyName |
selectorName'.*')
                      ',' ''' fullTextSearchExpression''' ')'

                   /* If only one selector exists in this query,
                      explicit specification of the selectorName
                      preceding the propertyName is optional */

fullTextSearchExpression ::= FulltextSearch
```

The full-text search expression is a string literal that adheres to the "full-text search" grammar described below.

An example query selects all the properties defined on the **`acme:taggable`** node type, returning only those "taggable" nodes with a **`acme:tagname`** value that contains the "foo" term within the value:

```
SELECT * FROM [acme:taggable] as tagged
WHERE CONTAINS(tagged.[acme:tagName],'foo')
```

node constraint

The same-node constraint stipulates that the node appearing in the selector with the given name has a path that matches the literal path provided.

```
SameNode ::= 'ISSAMENODE(' [selectorName ','] Path ')'
```

```
                /* If only one selector exists in this query,
                   explicit specification of the selectorName
                   preceding the propertyName is optional */
```

Because this standard constraint clause is not really like traditional SQL, the hierarchical database defines a **`jcr:path`** "pseudo-column" that can be used in "comparison constraints" and that allows for using other comparison operators, including **`LIKE`**.

Child-node constraint

The child-node constraint stipulates that the node appearing in the selector with the given name is a child of a node with a path that matches the literal path provided.

```
ChildNode ::= 'ISCHILDNODE(' [selectorName ','] Path ')'

                /* If only one selector exists in this query,
                   explicit specification of the selectorName
                   preceding the propertyName is optional */
```

See also the **`jcr:path`** "pseudo-column" that can be used in "comparison constraints" and that allows for using other comparison operators, including **`LIKE`**. And because the right hand side (i.e., static operand) of a **`LIKE`** expression can involve wildcards, it may be easier and more understandable to use the pseudo-column.

Descendant-node constraint

The descendant-node constraint stipulates that the node appearing in the selector with the given name is a descendant of a node with a path that matches the literal path provided.

```
DescendantNode ::= 'ISDESCENDANTNODE(' [selectorName ','] Path ')'

                /* If only one selector exists in this query,
                   explicit specification of the selectorName
                   preceding the propertyName is optional */
```

See also the **`jcr:path`** "pseudo-column" that can be used in "comparison constraints" and that allows for using other comparison operators, including **`LIKE`**. And because the right hand side (i.e., static operand) of a **`LIKE`** expression can involve wildcards, it may be easier and more understandable to use the pseudo-column.

See Also:

- Section 7.1.2.6, “Static Operand”
- Section 7.1.2.7, “Dynamic Operand”

7.1.2.5. Path and Name

Many of the rules above have used paths and names, and the rules for these are defined as follows:

```
Name ::= '[' quotedName ']' | '[' simpleName ']' | simpleName

quotedName ::= /* A JCR Name (see the JCR specification) */
simpleName ::= /* A JCR Name that contains only SQL-legal
```

```
                  characters (namely letters, digits, and underscore) */

Path ::= '[' quotedPath ']' | '[' simplePath ']' | simplePath

quotedPath ::= /* A JCR Path that contains non-SQL-legal characters */
simplePath ::= /* A JCR Path (rather Name) that contains only SQL-legal
                    characters (namely letters, digits, and underscore) */
```

Note that JCR-SQL2 surrounds identifiers with square brackets (e.g., '**[**' and '**]**'), allowing names to contain a '**:**' character needed with namespaced names. If the names or paths only contain valid SQL characters, then they do not need to be quoted.

7.1.2.6. Static Operand

In the standard JCR-SQL2 grammar, a static operand appears on the right-hand side of an operator, and represents an expression whose value can be determined by static analysis of the query (e.g., when the query is parsed). In particular, a static operand in the standard JCR-SQL2 grammar comprised of either a literal value or a variable.

In SQL, however, the expression that appears on the right-hand side of an operator is not always able to be determined at query parse time. An example is a subquery, which appears on the right hand side but obviously can only be evaluated into values during query execution time. Since standard JCR-SQL2 does not include any such features, the term "static operand" is technically valid.

In addition to literal values and variables, the hierarchical database also supports "subqueries" appearing on the right-hand side of an operator. So this grammar continues to use the "static operand" term for easy comparison with the standard JCR-SQL2 grammar, but the term has a different (and expanded) semantic than in the standard grammar.

Therefore, the rules for what the hierarchical database allows on the right-hand side of an operator in a constraint is as follows:

```
StaticOperand ::= Literal | BindVariableValue | Subquery

Literal ::= CastLiteral | UncastLiteral

CastLiteral ::= 'CAST(' UncastLiteral ' AS ' PropertyType ')'

PropertyType ::= 'STRING' | 'BINARY' | 'DATE' |
                 'LONG' | 'DOUBLE' | 'DECIMAL' |
                 'BOOLEAN' | 'NAME' | 'PATH' |
                 'REFERENCE' | 'WEAKREFERENCE' |
                 'URI'

UncastLiteral ::= UnquotedLiteral |
                  ''' UnquotedLiteral ''' |
                  '"' UnquotedLiteral '"'

UnquotedLiteral ::= /* String form of a JCR Value,
                       as defined in the JCR specification */
```

Bind variable

The standard JCR-SQL2 grammar supports using variable names within a query, where the values for those variables are bound to the **`Query`** object before execution. In the query, the variable names are prefixed with a '**$**' character and are otherwise normal JCR name:

```
BindVariableValue ::= '$'bindVariableName

bindVariableName ::= /* A string that conforms to the JCR Name
syntax,
                        though the prefix does not need to be a
                        registered namespace prefix. */
```

So, consider this simple query that selects all the properties defined on the **`acme:taggable`** node type, and that returns only those "taggable" nodes with a **`acme:tagname`** value that matches the value of the **`tagValue`** variable:

```
SELECT * FROM [acme:taggable] as tagged
WHERE tagged.[acme:tagName] = $tagValue
```

This query could be evaluated using the JCR API as follows:

```
// Obtain the query manager for the session via the workspace ...
javax.jcr.Session session = // ...
javax.jcr.query.QueryManager mgr =
session.getWorkspace().getQueryManager();

// Create a query object ...
String language = ...
String expression = ...
javax.jcr.query.Query query =
queryManager.createQuery(expression,language);

// Bind a value to the variable ...
Value tag = session.getValueFactory().create("foo");
query.bindVariable("tagValue",tag);

// Execute the query and get the results ...
javax.jcr.query.QueryResult result = query.execute();
```

Obviously multiple variables can be used in a query expression, but a value must be bound to every variable before the **`Query`** object can be executed.

Subquery

The standard JCR-SQL2 grammar does not support subqueries. But subqueries are such a useful feature, so the hierarchical database supports using multiple subqueries within a single query. In fact, subqueries are nothing more than a **`QueryCommand`** , which if you'll remember is the top-level rule in the grammar. That means that subqueries can be any query, and you can even include subqueries within a subquery!

```
Subquery ::= '(' QueryCommand  ')' |
             QueryCommand
```

Strictly speaking, the hierarchical database only supports non-correlated subqueries, which means that they can actually be evaluated independently (outside the context of the containing query).

Additionally, because subqueries appear on the right-hand side of an operator, all subqueries must return a single column, and each row's single value will be treated as a literal value. If the subquery is used in a clause that expects a single value (e.g., in a comparison), only the subquery's first row will be used. If the subquery is used in a clause that allows multiple values (e.g., **`IN (...)`**), then

all of the subquery's rows will be used.

For example, in the following query fragment, the first value in each row of the subquery's results will be used within the **IN** clause of the outer query:

```
WHERE ... [my:type].[prop1] IN
   ( SELECT [my:prop2] FROM [my:type2]
     WHERE  [my:prop3] < '1000' )
AND ...
```

However, changing the **IN** clause to a comparison results in only the first value in the first row of the subquery's results being using in the comparison criteria:

```
WHERE ... [my:type].[prop1] =
   ( SELECT [my:prop2] FROM [my:type2]
     WHERE  [my:prop3] < '1000' )
AND ...
```

7.1.2.7. Dynamic Operand

In various constraints described above, the dynamic operand appears on the left-hand side of an operator, and signifies that the values can only be determined when the query is evaluated.

The standard JCR-SQL2 grammar defines seven kinds of dynamic operands: property value , length , node name , node local name , full-text search score , lowercase , and uppercase .

The hierarchical database supports all these types, but adds support for four more: reference value , node path , node depth , and simple arithmetic clauses . The hierarchical database also allows the dynamic operand to be surrounded by parentheses, which is sometimes convenient for complex queries.

The **DynamicOperand** rule in the extended grammar is:

```
DynamicOperand ::= PropertyValue | ReferenceValue |
                   Length | NodeName | NodeLocalName |
                   NodePath | NodeDepth |
                   FullTextSearchScore |
                   LowerCase | UpperCase |
                   Arithmetic |
                   '(' DynamicOperand ')'
```

Each of these types of dynamic operands is described below.

Property value operand

The property value operand always evaluates to the value(s) of the specified property on the selector.

```
PropertyValue ::= [selectorName'.'] propertyName

                  /* If only one selector exists in this query,
                     explicit specification of the selectorName
                     preceding the propertyName is optional. */
```

Note that if the property is multi-valued, the constraint will be satisfied if any of the property values works with the constraint. For example, if the **acme:tagNames** property is a multi-valued property

declared on the **acme:taggable** node type, then the following query will finds all **acme:taggable** nodes that has "foo" for at least one of the values of the **acme:tagNames** property:

```
SELECT * FROM [acme:taggable] as tagged
WHERE tagged.[acme:tagNames] = 'foo'
```

Reference value operand

One of the extensions is to support the a **REFERENCE(...)** dynamic operand, which enables placing constraints on one or any of the reference properties.

```
ReferenceValue ::= 'REFERENCE(' selectorName '.' propertyName ')' |
                   'REFERENCE(' selectorName ')' |
                   'REFERENCE()' |

                /* If only one selector exists in this query,
                   explicit specification of the selectorName
                   preceding the propertyName is optional.
                   Also, the property name may be excluded
                   if the constraint should apply to any
                   reference property.*/
```

The **REFERENCE** operand always evaluates to the identifier of the referenced nodes in one or all of the REFERENCE properties. Thus, all of the **REFERENCE** operands should be used with a **StaticOperand** that also evaluates to identifiers.

The **REFERENCE()** operand (with no selector name and no property name) evaluates to the identifiers of the nodes referenced by all of reference properties on the node in the only selector. The **REFERENCE(selectorName)** works the same way, but must be used if there is more than one selector in the query. Finally, the **REFERENCE(selectorName.propertyName)** evaluates to the identifiers of nodes referenced by the **propertyName** reference property on the nodes in the named selector.

For example, here is a query that finds all nodes that reference a set of nodes for which we already know the identifiers, **id1**, **id2**, and **id3**.

```
SELECT * FROM [nt:base]
WHERE REFERENCE() IN ('id1','id2','id3')
```

This operand works really well with subqueries or variables for the right-hand side. For example, here is a query finds all nodes that reference any of the nodes in the subgraph below the **/foo/bar/baz** node, where a subquery is used to find all nodes in the subgraph:

```
SELECT * FROM [nt:base]
WHERE REFERENCE() IN (
  SELECT [jcr:uuid] FROM [nt:base] AS refd
  WHERE ISDESCENDANT(refd,'/foo/bar/baz')
  )
```

This kind of query is impossible to do using standard JCR-SQL2 features, and shows some of the power of the extensions to JCR-SQL2.

Length operand

The length operand evaluates to the length (or lengths, if multi-valued) of a property. The length is defined to be:

- for a **`BINARY`** value, the number of bytes in the value, or
- for all other value types, the number of characters of the string resulting from a conversion of the value to a string.

The rule for the length operand is:

```
Length ::= 'LENGTH(' PropertyValue ')'
```

Node name operand

The node name operand always evaluates to the prefixed name of the node given by the supplied selector:

```
NodeName ::= 'NAME(' [selectorName] ')'

               /* If only one selector exists in this query,
                  explicit specification of the selectorName
                  is optional */
```

See also the **`jcr:name`** "pseudo-column", which enables accessing the JCR name of any node as if the name were a regular property on any node.

Node local name operand

The node name operand always evaluates to the local name of the node given by the supplied selector:

```
NodeLocalName ::= 'LOCALNAME(' [selectorName] ')'

               /* If only one selector exists in this query,
                  explicit specification of the selectorName
                  is optional */
```

See also the **`mode:localName`** "pseudo-column", which enables accessing the local name of any node as if the local name were a regular property.

Node depth operand

The node depth operand is an extension to the standard set of dynamic operands specific to the hierarchical database, and evaluates to the integer depth of the node given by the supplied selector. The depth of a node is defined to be the number of segments in the node's path. For example, the depth of the root node is 0, whereas the depth of the node at **`/foo/bar/baz`** is 3.

```
NodeDepth ::= 'DEPTH(' [selectorName] ')'

               /* If only one selector exists in this query,
                  explicit specification of the selectorName
                  is optional */
```

See also the **`mode:depth`** "pseudo-column", which enables accessing the depth of any node as if the depth were a regular property.

Node path operand

The node path operand is an extension to the standard set of dynamic operands specific to the hierarchical database, and evaluates to the path of the node given by the supplied selector.

```
NodePath ::= 'PATH(' [selectorName] ')'

               /* If only one selector exists in this query,
                  explicit specification of the selectorName
                  is optional */
```

See also the **`jcr:path`** "pseudo-column", which enables accessing the path of any node as if the path were a regular property.

Full text search score operand

The full-text search score operand evaluates to a **`DOUBLE`** value equal to the full-text search score of a node. The full-text search score ranks a selector's nodes by their relevance to the **`fullTextSearchExpression`** specified in a **`[FullTextSearch|#Fulltextsearchconstraint`** . The magnitude of the scores are implementation specific, but most implementations will produce higher scores with more relevant matching and lower scores for less-relevant matching.

```
FullTextSearchScore ::= 'SCORE(' [selectorName] ')'
               /* If only one selector exists in this query,
                  explicit specification of the selectorName
                  is optional */
```

See also the **`jcr:score`** "pseudo-column", which enables accessing the score of any node as if the score were a regular property.

Lowercase operand

The lowercase operand evaluates to the lower-case string value (or values, if multi-valued) of operand. If the operand does not evaluate to a string value, its value is first converted to a string.

```
LowerCase ::= 'LOWER(' DynamicOperand ')'
```

Uppercase operand

The uppercase operand evaluates to the upper-case string value (or values, if multi-valued) of operand. If the operand does not evaluate to a string value, its value is first converted to a string.

```
LowerCase ::= 'LOWER(' DynamicOperand ')'
```

Arithmetic operand

The arithmetic operand is an extension to the standard JCR-SQL2 grammar specific to the hierarchical database. It allows two other dynamic operands that evaluate to numeric values to be numerically combined using addition, subtraction, multiplication, or division.

```
Arithmetic ::= DynamicOperand ('+'|'-'|'*'|'/') DynamicOperand
```

For example, the following query restricts the results such that the sum of the score of nodes originating from separate selectors is greater than 1.0:

```
SELECT * FROM [acme:type1] AS type1
         JOIN [acme:type2] as type2 ON type1.prop1 < type2.prop2
         WHERE SCORE(type1) + SCORE(type2) > 1.0
```

So although it is possible to use in the **WHERE** clause, it is more likely to be used in the order-by clauses . For example, the following query orders the results based upon the difference in the scores of nodes in the two selectors:

```
SELECT * FROM [acme:type1] AS type1
         JOIN [acme:type2] as type2 ON type1.prop1 < type2.prop2
         ORDER BY ( SCORE(type1) - SCORE(type2) ) ASC,
                  LENGTH(type2.prop3) DESC
```

7.1.2.8. Ordering

The **ORDER BY** clause defined by the standard JCR-SQL2 grammar allows the order of the results to be dictated by the values evaluated at execution time based upon one or more "dynamic operands" . The rule for the expression is as follows:

```
orderings ::= Ordering {',' Ordering}

Ordering ::= DynamicOperand [Order]

Order ::= 'ASC' | 'DESC'
```

As with SQL, the **ASC** qualifier specifies that the ordering should be in ascending order, and is the default; likewise, the **DESC** qualifier specifies that the ordering should be in descending order.

See Also:

- Section 7.1.2.7, “Dynamic Operand”

7.1.2.9. Columns

The standard JCR-SQL2 grammar allows a query to include in the **SELECT** clause which property values should be returned and included in the results:

```
columns ::= (Column ',' {Column}) | '*'

Column ::= ([selectorName'.']propertyName ['AS' columnName]) |
           (selectorName'.*')

        /* If only one selector exists in this query,
           explicit specification of the selectorName
           is optional */

selectorName ::= Name
propertyName ::= Name
columnName ::= Name
```

When '*****' '" is used for the list of selected columns, the result set is expected to minimally include, for each

selector, a column for each single-valued non-residual property of the selector's node type, including those explicitly declared on the node type and those inherited from the node's supertypes.

For example, the result set for the following query would contain at least the **[jcr:primaryType]** column, since it is the only single-valued, non-residual property defined on the **[nt:base]** node type. The **[jcr:mixinTypes]** property is also non-residual, but the results need not include it since it is multi-valued.

```
SELECT * FROM [nt:base]
```

If there are multiple selectors, then **SELECT *** will include all of the selectable columns from each selector's node type. However, it is possible to request all of the selectable columns from some of the selectors, using the form. For example:

```
SELECT type1.*
FROM [acme:type1] AS type1
JOIN [acme:type2] as type2 ON type1.prop1 < type2.prop2
```

Note, however, that although only single-valued, non-residual properties are included when '*****' '" is used in the **SELECT** clause, it is possible to explicitly include residual properties. For example, the following query finds all nodes that have at least one "foo" value for the **acme:tagNames** property:

```
SELECT [acme:tagNames] AS tagName
FROM [nt:base] WHERE tagName = 'foo'
```

7.1.2.10. Limit and Offset

Neither the standard JCR-SQL2 grammar or the JCR API itself provide support for limiting the rows that are returned in the results. This is a common need, especially for applications that paginate the results, where each page shows a subset of the results.

Because this is such an essential feature that cannot be accomplished any other way, the hierarchical database adds support for specifying the maximum number of rows to return, and optionally specifying the number of initial rows that should be skipped. This extension follows the SQL syntax:

```
Limit ::= 'LIMIT' count [ 'OFFSET' offset ]
count ::= /* Positive integer value */
offset ::= /* Non-negative integer value */
```

The **LIMIT** clause is entirely optional, and if absent does not limit the result set rows in any way. However, if the **LIMIT count** clause is used, then the result set will contain no more than **count** rows. This **LIMIT** clause may optionally be followed by the **OFFSET number** clause, where **number** is the number of initial rows that should be skipped before the rows are included in the results.

7.1.2.11. Psuedo-Columns

The design of the JCR-SQL2 query language makes fairly heavy use of functions, including **SCORE()** , **NAME()** , **LOCALNAME()** , and various constraints. The hierarchical database provides several more useful functions, such as **PATH()** and **DEPTH()** , that follow the same patterns.

However, these functions have several disadvantages. First, they make the JCR-SQL2 language less "SQL-like", since SQL-92 and -99 do not define similar kinds of functions. (There are aggregate functions, like **COUNT** , **SUM** , etc., but they operate on a particular column in all tuples and are therefore more dissimilar than similar.) This means that applications that use SQL and SQL-like query languages are less likely to be able to build and issue JCR-SQL2 queries.

A second disadvantage of these functions is that JCR-SQL2 does not allow them to be used within the **SELECT** clause. As a result, the location-related and score information cannot be included as columns of values in the **QueryResult** rows. Instead, a client can only access this information by obtaining the **Node** object(s) for each row. Relying upon both the result set and additional Java objects makes it difficult to use the JCR query system. It also makes certain kinds of applications impossible.

For example, the hierarchical database's JDBC driver is designed to enable JDBC-aware applications to query repository content using JCR-SQL2 queries. The standard JDBC API cannot expose the **Node** objects, so the only way to return the path-related and score information is through additional columns in the result. While such columns could always "magically" appear in the result set, doing this is not compatible with JDBC applications that dynamically build the **SELECT** clauses of queries based upon database metadata. Such applications require the columns to be properly described in database metadata, and the columns need to be used within queries.

The hierarchical database attempts to solve these issues by directly supporting a number of "pseudo-columns" within JCR-SQL2 queries, wherever columns can be used. These "pseudo-columns" include:

- **jcr:score** is a column of type DOUBLE that represents the full-text search score of the node, which is a measure of the node's relevance to the full-text search expression. The hierarchical database does compute the scores for all queries, though the score for rows in queries that do not include a full-text search criteria may not be reliable.
- **jcr:path** is a column of type PATH that represents the normalized path of a node, including same-name siblings. This is the same as what would be returned by the getPath() method of Node. Examples of paths include "/jcr:system" and "/foo/bar3".
- **jcr:name** is a column of type NAME that represents the node name in its namespace-qualified form using namespace prefixes and excluding same-name-sibling indexes. Examples of node names include "jcr:system", "jcr:content", "ex:UserData", and "bar".
- **mode:localName** is a column of type STRING that represents the local name of the node, which excludes the namespace prefix and same-name-sibling index. As an example, the local name of the "jcr:system" node is "system", while the local name of the "ex:UserData3" node is "UserData".
- **mode:depth** is a column of type LONG that represents the depth of a node, which corresponds exactly to the number of path segments within the path. For example, the depth of the root node is 0, whereas the depth of the "/jcr:system/jcr:nodeTypes" node is 2.

All of these pseudo-columns can be used in the **SELECT** clause of any JCR-SQL2 query, and their use defines whether such columns appear in the result set. In fact, all of these pseudo-columns will be included when **SELECT *** clauses in JCR-SQL2 queries are expanded by the query engine. This means that every node type (even mixin node types that have no properties and are essentially markers) are represented by a queryable table with at least one column. However, unlike the older JCR-SQL query language, these pseudo-columns are never included in the result unless explicitly included or implicitly included with the **SELECT *** clause.

Note

Why did the hierarchical database use the **jcr** namespace prefix for some of the pseudo-columns, and **mode** for the others? The older JCR-SQL language defined the **jcr:score**, **jcr:path**, and **jcr:name** pseudo-columns, so we use the same names. The other columns were unique to the hierarchical database and are therefore defined with the **mode** namespace prefix.

Like any other column, all of these pseudo-columns can be also be used in the **WHERE** clause of any JCR-SQL2 query, even if they are not included in the **SELECT** clause. They can be used anywhere that a regular

column can be used, including within constraints and dynamic operands. The hierarchical database will automatically rewrite queries that use pseudo-columns in the dynamic operands of constraints to use the corresponding function, such as **`SCORE()`** , **`PATH()`** , **`NAME()`** , **`LOCALNAME()`** , and **`DEPTH()`** . Additionally, any property existence constraint using these pseudo-columns will always evaluate to 'true' (and thus the hierarchical database's query optimizer will always remove such constraints from the query plan).

The **`jcr:path`** pseudo-column may also be used on both sides of an "equijoin" constraint clause. For example, equijoin expressions similar to:

```
... selector1.[jcr:path] = selector2.[jcr:path] ...
```

will be automatically rewritten by the hierarchical database's optimizer to the following form:

```
... ISSAMENODE(selector1,selector2) ...
```

As with regular columns, the pseudo-columns must be qualified with the selector name if the query contains more than one selector.

7.1.3. Full-text Search Grammar

The grammar for the full-text search expressions used in the JCR-SQL2's "full-text search constraint" is as follows:

```
FulltextSearch ::= Disjunct {Space 'OR' Space Disjunct}

Disjunct ::= Term {Space Term}

Term ::= ['-'] SimpleTerm

SimpleTerm ::= Word | '"' Word {Space Word} '"'

Word ::= NonSpaceChar {NonSpaceChar}

Space ::= SpaceChar {SpaceChar}

NonSpaceChar ::= Char - SpaceChar /* Any Char except SpaceChar */

SpaceChar ::= ' '

Char ::= /* Any character */
```

This grammar supports expressions similar to what you might provide to an Internet search engine. It lists the terms or phrases that should appear (or not appear) in the applicable property value(s). Simple terms consist of a single word (with only non-space characters), while phrases can be surrounded with double quotes.

7.1.4. Example JCR-SQL2 Queries

7.1.4.1. Simple Queries

One of the simplest JCR-SQL2 queries finds all nodes in the current workspace of the repository:

```
SELECT * FROM [nt:base]
```

This query will return a result set containing the **`jcr:primaryType`** column, since the **`nt:base`** node type

defines only one single-valued, non-residual property called **`jcr:primaryType`**.

Note

The hierarchical database does not currently support returning multi-valued properties in result sets. This is permitted by the JCR 2.0 specification. The hierarchical database does, however, support using multi-valued properties in constraints and **`ORDER BY`** clauses.

Since our query used **`SELECT *`**, the hierarchical database also includes the five non-standard pseudo-columns mentioned above: **`jcr:path`**, **`jcr:score`**, **`jcr:name`**, **`mode:localName`**, and **`mode:depth`**. These columns are very convenient to have in the results, but also make certain criteria much easier than with the corresponding standard functions or those specific to the hierarchical database.

Queries can explicitly specify the columns that are to be returned in the results. The following query is very similar to the previous query and will return the same rows, but the result set will have only a single column and will not include any of the pseudo-columns:

```
SELECT [jcr:primaryType] FROM [nt:base]
```

The following query will return the same rows as in the previous two queries, but the **`SELECT`** clause explicitly includes only two of the pseudo-columns for the path and depth (which are computed from the nodes' locations):

```
SELECT [jcr:primaryType], [jcr:path], [mode:depth] FROM [nt:base]
```

In JCR-SQL2, a table representing a particular node type will have a column for each of the node type's property definitions, including those inherited from supertypes. For example, the **`nt:file`** node type, its **`nt:hierarchyNode`** supertype, and the **`mix:created`** mixin type are defined using the CND notation as follows:

```
[mix:created] mixin
  - jcr:created (date) protected
  - jcr:createdBy (string) protected

[nt:hierarchyNode] > mix:created abstract

[nt:file] > nt:hierarchyNode
  + jcr:content (nt:base) primary mandatory
```

Therefore, the table representing the **`nt:file`** node type will have 3 columns: the **`jcr:created`** and **`jcr:createdBy`** columns inherited from the **`mix:created`** mixin node type (via the **`nt:hierarchyNode`** node type), and the **`jcr:primaryType`** column inherited from the **`nt:base`** node type, which is the implicit supertype of the **`nt:hierarchyNode`** (and all node types).

The hierarchical database adheres to this behavior with the exception that a **`SELECT *`** will result in the additional pseudo-columns. Thus, this next query:

```
SELECT * FROM [nt:file]
```

is equivalent to this query:

```
SELECT [jcr:primaryType], [jcr:created], [jcr:createdBy],
       [jcr:path], [jcr:name], [jcr:score], [mode:localName], [mode:depth]
FROM [nt:file]
```

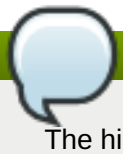
7.1.4.2. Using Columns in Constraints

Consider a query that selects some of the available columns from the **`nt:file`** table and uses a constraint to ensure the resulting file nodes have names that end in '.txt':

```
SELECT [jcr:primaryType], [jcr:created], [jcr:createdBy], [jcr:path] FROM
[nt:file]
WHERE LOCALNAME() LIKE '%.txt'
```

The hierarchical database also supports placing criteria against the **`mode:localName`** pseudo-column instead of using the **`LOCALNAME()`** function. Such a query is equivalent to the previous query and will produce the exact same results:

```
SELECT [jcr:primaryType], [jcr:created], [jcr:createdBy], [jcr:path]
FROM [nt:file]
WHERE [mode:localName] LIKE '%.txt'
```



Note

The hierarchical database's pseudo-columns are often far easier to use than the corresponding function-like constraints.

Although this query looks much more like SQL, the use of the '**[**' and '**]**' characters to quote the identifiers is not typical of a SQL dialect. The hierarchical database actually supports the using double-quote characters and square braces interchangeably around identifiers (although they must match around any single identifier). Again, this next query, which looks remarkably like any SQL-92 or -99 dialect, is functionally identical to the previous two queries:

```
SELECT "jcr:primaryType", "jcr:created", "jcr:createdBy", "jcr:path" FROM
"nt:file"
WHERE "mode:localName" LIKE '%.txt'
```

7.1.4.3. Inner Joins

In JCR-SQL2, a node will appear as a row in each table that corresponds to the node types defined by that node's primary type or mixin types, or any supertypes of these node types. In other words, a node will appear in the table corresponding to each node type for which **`Node.isNodeType(...)`** returns true.

For example, consider a node that has a primary type of **`nt:file`** but has an explicit mixin of **`mix:referenceable`** . This node will appear as a row in the all of these tables:

- **`nt:file`**
- **`mix:referenceable`**
- **`nt:hierarchyNode`**
- **`mix:created`**

- **nt:base**

However, the columns in each of these tables will differ. The **nt:file** node type has the **nt:hierarchyNode** , **mix:created** , and **nt:base** for supertypes, and therefore the table for **nt:file** contains columns for the property definitions on all of these types. But because **mix:referenceable** is not a supertype of **nt:file** , the table for **nt:file** will not contain a **jcr:uuid** column. To obtain a single result set that contains columns for all the properties of our node, we need to perform an identity join .

The next query shows how to return all properties for **nt:file** nodes that are also **mix:referenceable** :

```
SELECT file.*, ref.*
FROM [nt:file] AS file
JOIN [mix:referenceable] AS ref
  ON ISSAMENODE(file,ref)
```

Since wildcards were used in the **SELECT** clause, the hierarchical database expands the **SELECT** clause to include the columns for all (explicit and inherited) property definitions of each type plus pseudo-columns for each type, which is equivalent to:

```
SELECT file.[jcr:primaryType],
       file.[jcr:created],
       file.[jcr:createdBy],
       file.[jcr:path],
       file.[jcr:name],
       file.[jcr:score],
       file.[mode:localName],
       file.[mode:depth],
       ref.[jcr:path],
       ref.[jcr:name],
       ref.[jcr:score],
       ref.[mode:localName],
       ref.[mode:depth],
       ref.[jcr:uuid]
FROM [nt:file] AS file
JOIN [mix:referenceable] AS ref
  ON ISSAMENODE(file,ref)
```

Note because we are using an identity join, the **file.[jcr:path]** column will contain the same value as the **ref.[jcr:path]**.

Note

Fully-expand the **SELECT** clause to specify exactly the columns that you want, excluding the columns that return the same values or return values not needed by your application. This can also make the query a bit more efficient, since less data needs to be found and returned.

By the way, this is also what many well-written applications do when querying SQL databases.

Here is a query that does this by eliminating columns with duplicate values and using aliases that are simpler than the namespace-qualified names:

```
SELECT file.[jcr:primaryType] AS primaryType,
       file.[jcr:created] AS created,
       file.[jcr:createdBy] AS createdBy,
       ref.[jcr:uuid] AS uuid,
       file.[jcr:path] AS path,
       file.[jcr:name] AS name,
       file.[jcr:score] AS score,
       file.[mode:localName] AS localName,
       file.[mode:depth] AS depth
FROM [nt:file] AS file
JOIN [mix:referenceable] AS ref
  ON ISSAMENODE(file,ref)
```

Although this query looks much more like SQL, use of the '**[**' and '**]**' characters in JCR-SQL2 to quote the identifiers is not typical of a SQL dialect. Again, the hierarchical database supports the using double-quote characters and square braces interchangeably around identifiers (although they must match around any single identifier). This makes it easier for existing SQL-oriented tools and applications to work more readily, including applications that use the hierarchical database's JDBC driver to query a JCR repository.

This next query, which looks remarkably like any SQL-92 or -99 dialect, is functionally identical to the previous query. However, it uses double quotes and a pseudo-column identity constraint on **`jcr:path`** (which is identical in semantics and performance as the **`ISSAMENODE(...)`** constraint):

```
SELECT file."jcr:primaryType" AS primaryType,
       file."jcr:created" AS created,
       file."jcr:createdBy" AS createdBy,
       ref."jcr:uuid" AS uuid,
       file."jcr:path" AS path,
       file."jcr:name" AS name,
       file."jcr:score" AS score,
       file."mode:localName" AS localName,
       file."mode:depth" AS depth
FROM "nt:file" AS file
JOIN "mix:referenceable" AS ref
  ON file."jcr:path" = ref."jcr:path"
```

Note

When using joins and selecting multiple columns, use aliases on the columns to make it easier to reference those columns in constraints and ordering clauses.

7.1.4.4. Other Joins

These are examples of two-way inner joins, but the hierarchical database supports joining multiple tables together in a single query. The hierarchical database also supports a variety of joins, including:

- **`INNER JOIN`** (or **`JOIN`**)
- **`LEFT OUTER JOIN`**
- **`RIGHT OUTER JOIN`**
- **`FULL OUTER JOIN`**

- **CROSS JOIN**

7.1.4.5. Set Operations

The hierarchical database also supports several other query features beyond JCR-SQL2. One of these is support for set queries that use:

- **UNION** and **UNION ALL**
- **INTERSECT** and **INTERSECT ALL**
- **EXCEPT** and **EXCEPT ALL** .

Here is an example of a union:

```
SELECT [jcr:primaryType], [jcr:created], [jcr:createdBy], [jcr:path] FROM
[nt:file]
UNION
SELECT [jcr:primaryType], [jcr:created], [jcr:createdBy], [jcr:path] FROM
[nt:folder]
```

7.1.4.6. Subqueries

The hierarchical database also supports using (non-correlated) subqueries within the **WHERE** clause and wherever a static operand can be used. Subqueries can even be used within another subquery. **All subqueries, though, should return a single column** (all other columns will be ignored), and each row's single value will be treated as a literal value. If the subquery is used in a clause that expects a single row (e.g., in a comparison), only the subquery's first row will be used.

Subqueries in the hierarchical database are a powerful and easy way to use more complex criteria that is a function of the content in the repository, without having to resort to multiple queries and complex application logic, such as taking the results of one query and dynamically generating the criteria of another query.

Here's an example of a query that finds all **nt:file** nodes in the repository whose paths are referenced in the value of the **vdb:originalFile** property of the **vdb:virtualDatabase** nodes. (This query also uses the **$maxVersion** variable in the subquery.)

```
SELECT [jcr:primaryType], [jcr:created], [jcr:createdBy], [jcr:path]
FROM [nt:file]
WHERE PATH() IN (
   SELECT [vdb:originalFile] FROM [vdb:virtualDatabase]
   WHERE [vdb:version] <= $maxVersion
     AND CONTAINS([vdb:description],'xml OR xml maybe')
)
```

Without subqueries, this query would need to be broken into two separate queries: the first would find all of the paths referenced by the **vdb:virtualDatabase** nodes matching the version and description criteria, followed by one (or more) subsequent queries to find the **nt:file** nodes with the paths expressed as literal values (or variables).

Note

Using a subquery is not only easier to implement and understand, it is actually more efficient.

7.2. JCR-SQL

The JCR-SQL query language is defined by the JCR 1.0 specification as a way to express queries using strings that are similar to SQL. Support for the language is optional, and in fact this language was deprecated in the JCR 2.0 specification in favor of JCR-SQL2.

Important

As an aside, the hierarchical database's parser for JCR-SQL queries is actually a simplified and more limited version of the parser for JCR-SQL2 queries. All other processing, however, is done in exactly the same way.

The JCR 2.0 specification defines how nodes in a repository are mapped onto relational tables queryable through a SQL-like language, including JCR-SQL and JCR-SQL2. Each node type is mapped as a relational view with a single column for each of the node type's (residual and non-residual) property definitions. Conceptually, each node in the repository then appears as a record inside the view corresponding to the node type for which **`Node.isNodeType(nodeTypeName)`** would return true.

Since each node likely returns true from this method for multiple node type (e.g., the primary node type, the mixin types, and all supertypes of the primary and mixin node types), all nodes will likely appear as records in multiple views. And since each view only exposes those properties defined by (or inherited by) the corresponding node type, a full picture of a node will likely require joining the views for multiple node types. This special kind of join, where the nodes have the same identity on each side of the join, is referred to as an identity join, and is handled very efficiently by the hierarchical database.

7.2.1. Extensions to JCR-SQL

The hierarchical database includes support for the JCR-SQL language, and adds several extensions to make it even more powerful and useful:

- Support for the **`UNION`** , **`INTERSECT`** , and **`EXCEPT`** set operations on multiple result sets to form a single result set. As with standard SQL, the result sets being combined must have the same columns. The **`UNION`** operator combines the rows from two result sets, the **`INTERSECT`** operator returns the difference between two result sets, and the **`EXCEPT`** operator returns the rows that are common to two result sets. Duplicate rows are removed unless the operator is followed by the **`ALL`** keyword. For detail, see the grammar for set queries.
- Removal of duplicate rows in the results, using **`SELECT DISTINCT ...`**.
- Limiting the number of rows in the result set with the **`LIMIT count`** clause, where **`count`** is the maximum number of rows that should be returned. This clause may optionally be followed by the **`OFFSET number`** clause to specify the number of initial rows that should be skipped.
- Support for the **`IN`** and **`NOT IN`** clauses to more easily and concisely supply multiple of discrete static operands. For example, **`WHERE ... prop1 IN (3,5,7,10,11,50) ...`**.
- Support for the **`BETWEEN`** clause to more easily and concisely supply a range of discrete operands. For example, **`WHERE ... prop1 BETWEEN 3 EXCLUSIVE AND 10 ...`**.
- Support for (non-correlated) subqueries in the **`WHERE`** clause, wherever a static operand can be used. Subqueries can even be used within another subquery. All subqueries must return a single column, and each row's single value will be treated as a literal value. If the subquery is used in a clause that expects a single value (e.g., in a comparison), only the subquery's first row will be used. If the subquery is used in a

clause that allows multiple values (e.g., **`IN (...)`**), then all of the subquery's rows will be used. For example, this query **`WHERE ... prop1 IN ( SELECT my:prop2 FROM my:type2 WHERE my:prop3 < '1000' ) AND ...`** will use the results of the subquery as the literal values in the **`IN`** clause.

7.2.2. Extended JCR-SQL Grammar

The grammar for the JCR-SQL query language is actually a superset of that defined by the JCR 1.0 specification , and as such the complete grammar is included here.

Note

The grammar is presented using the same EBNF nomenclature as used in the JCR 1.0 specification. Terms are surrounded by '[' and ']' denote optional terms that appear zero or one times. Terms surrounded by '{' and '}' denote terms that appear zero or more times. Parentheses are used to identify groups, and are often used to surround possible values. Literals (or keywords) are denoted by single-quotes.

```
QueryCommand ::= Query | SetQuery

SetQuery ::= Query ('UNION'|'INTERSECT'|'EXCEPT') ['ALL'] Query
                 { ('UNION'|'INTERSECT'|'EXCEPT') ['ALL'] Query }

Query ::= Select From [Where] [OrderBy] [Limit]

Select ::= 'SELECT' ('*' | Proplist )

From ::= 'FROM' NtList

Where ::= 'WHERE' WhereExp

OrderBy ::= 'ORDER BY' propname [Order] {',' propname [Order]}

Order ::= 'DESC' | 'ASC'

Proplist ::= propname {',' propname}

NtList ::= ntname {',' ntname}

WhereExp ::= propname Op value |
             propname 'IS' ['NOT'] 'NULL' |
             like |
             contains |
             whereexp ('AND'|'OR') whereexp |
             'NOT' whereexp |
             '(' whereexp ')' |
             joinpropname '=' joinpropname |
             between |
             propname ['NOT'] 'IN' '(' value {',' value } ')'

Op ::= '='|'>'|'<'|'>='|'<='|'<>'

joinpropname ::= quotedjoinpropname | unquotedjoinpropname
```

```
quotedjoinpropname ::= ''' unquotedjoinpropname '''
unquotedjoinpropname ::= ntname '.jcr:path'

propname ::= quotedpropname | unquotedpropname
quotedpropname ::= ''' unquotedpropname '''
unquotedpropname ::= /* A property name, possible a pseudo-property:
jcr:score or jcr:path */

ntname ::= quotedntname | unquotedntname
quotedntname ::= ''' unquotedntname '''
unquotedntname ::= /* A node type name */

value ::= literal | subquery

literal ::= ''' literalvalue ''' | literalvalue
literalvalue ::= /* A property value (in standard string form) */

subquery ::= '(' QueryCommand ')' | QueryCommand

like ::= propname 'LIKE' likepattern [ escape ]
likepattern ::= ''' likechar { likepattern } '''
likechar ::= char | '%' | '_'

escape ::= 'ESCAPE' ''' likechar '''

char ::= /* Any character valid within the string representation of a value
            except for the characters % and _ themselves. These must be
escaped */

contains ::= 'CONTAINS(' scope ',' searchexp ')'
scope ::= unquotedpropname | '.'
searchexp ::= ''' exp '''
exp ::= ['-']term {whitespace ['OR'] whitespace ['-']term}
term ::= word | '"' word {whitespace word} '"'
word ::= /* A string containing no whitespace */
whitespace ::= /* A string of only whitespace*/

between ::= propname ['NOT'] 'BETWEEN' lowerBound ['EXCLUSIVE']
                                  'AND' upperBound ['EXCLUSIVE']
lowerBound ::= value
upperBound ::= value

Limit ::= 'LIMIT' count [ 'OFFSET' offset ]
count ::= /* Positive integer value */
offset ::= /* Non-negative integer value */
```

7.3. XPath

The JCR 1.0 specification uses the XPath query language because node structures in JCR are very analogous to the structure of an XML document. Thus, XPath provides a useful language for selecting and searching workspace content. And since JCR 1.0 defines a mapping between XML and a workspace view called the "document view", adapting XPath to workspace content is quite natural.

A JCR XPath query specifies the subset of nodes in a workspace that satisfy the constraints defined in the query. Constraints can limit the nodes in the results to be those nodes with a specific (primary or mixin) node

type, with properties having particular values, or to be within a specific subtree of the workspace. The query also defines how the nodes are to be returned in the result sets using column specifiers and ordering specifiers.

7.3.1. Extensions to XPath

The hierarchical database offers a bit more functionality in the **`jcr:contains(...)`** clauses than required by the specification. In particular, the second parameter specifies the search expression, and for these full-text search language expressions are accepted, including wildcard support.

Important

As an aside, the hierarchical database actually implements XPath queries by transforming them into the equivalent JCR-SQL2 representation. And the JCR-SQL2 language, although often more verbose, is much more capable of representing complex queries with multiple combinations of type, property, and path constraints.

7.3.2. Column Specifiers

JCR 1.0 specifies that support is required only for returning column values based upon single-valued, non-residual properties that are declared on or inherited by the node types specified in the type constraint. The hierarchical database follows this requirement, and does not specifying residual properties. However, the hierarchical database does allow multi-valued properties to be specified as result columns. And as per the specification, the hierarchical database always returns the **`jcr:path`** and **`jcr:score`** pseudo-columns.

The hierarchical database uses the last location step with an attribute axis to specify the properties that are to be returned as result columns. Multiple properties are specified with a union. For example, the following table shows several XPath queries and how they map to JCR-SQL2 queries.

XPath	JCR-SQL2
`//*`	`SELECT * FROM [nt:base]`
`//element(*,my:type)`	`SELECT * FROM [my:type]`
`//element(*,my:type)/@my:title`	`SELECT [my:title] FROM [my:type]`
`//element(*,my:type)/(@my:title | @my:text)`	`SELECT [my:title], [my:text] FROM [my:type]`
`//element(*,my:type)/(@my:title union @my:text)`	`SELECT [my:title], [my:text] FROM [my:type]`

Specifying result set columns

7.3.3. Type Constraints

JCR 1.0 specifies that support is required only for specifying constraints of one primary type, and it is optional to support specifying constraints on one (or more) mixin types. The specification also defines that the XPath **`element`** test be used to test against node types, and that it is optional to support**`element`** tests on location steps other than the last one. Type constraints are inherently inheritance-sensitive, in that a constraint against a particular node type 'X' will be satisfied by nodes explicitly declared to be of type 'X' or of subtypes of 'X'.

The hierarchical database does support using the **`element`** test to test against primary or mixin type. The hierarchical database also only supports using an **`element`** test on the last location step. For example, the following table shows several XPath queries and how they map to JCR-SQL2 queries.

XPath	JCR-SQL2
`//*`	`SELECT * FROM [nt:base]`
`//element(*,my:type)`	`SELECT * FROM [my:type]`
`/jcr:root/nodes/element(*,my:type)`	`SELECT * FROM [my:type]WHERE PATH([my:type])> LIKE '/nodes/%'` {{AND DEPTH([my:type]) = CAST(2 AS LONG) }}
`/jcr:root/nodes//element(*,my:type)`	`SELECT * FROM [my:type]WHERE PATH([my:type]) LIKE '/nodes/%'`
`/jcr:root/nodes//element(ex:nodeName,my:type)`	`SELECT * FROM [my:type]WHERE PATH([my:type]) LIKE '/nodes/%'AND NAME([my:type]) = 'ex:nodeName'`

Specifying type constraints

Note that the JCR-SQL2 language supported by the hierarchical database is far more capable of joining multiple sets of nodes with different type, property and path constraints.

7.3.4. Property Constraints

JCR 1.0 specifies that attribute tests on the last location step is required, but that predicate tests on any other location steps are optional.

The hierarchical database does support using attribute tests on the last location step to specify property constraints, as well as supporting axis and filter predicates on other location steps. For example, the following table shows several XPath queries and how they map to JCR-SQL2 queries.

XPath	JCR-SQL2
`//*[]`	`SELECT * FROM [nt:base]WHERE [nt:base].prop1 IS NOT NULL`
`//element(*,my:type)[@prop1]`	`SELECT * FROM [my:type]WHERE [my:type].prop1 IS NOT NULL`
`//element(*,my:type) [@prop1=xs:boolean('true')]`	`SELECT * FROM [my:type]WHERE [my:type].prop1 = CAST('true' AS BOOLEAN)`
`//element(*,my:type)[@id<1 and @name='john']`	`SELECT * FROM [my:type]WHERE id < 1 AND name = 'john'`
`//element(*,my:type)[a/b/@id]`	`SELECT * FROM [my:type]JOIN [nt:base] as nodeSet1ON ISCHILDNODE(nodeSet1, [my:type])JOIN [nt:base] as nodeSet2ON ISCHILDNODE(nodeSet2,nodeSet1)WHERE (NAME(nodeSet1) = 'a'` {{AND NAME(nodeSet2) = 'b') }} `AND nodeSet2.id IS NOT NULL]`
`//element(,my:type)[./{ }{}/*/@id]`	`SELECT * FROM [my:type]JOIN [nt:base] as nodeSet1ON ISCHILDNODE(nodeSet1, [my:type])JOIN [nt:base] as nodeSet2ON ISCHILDNODE(nodeSet2,nodeSet1)WHERE nodeSet2.id IS NOT NULLL`

XPath	JCR-SQL2
`//element(*,my:type)[.//@id]`	`SELECT * FROM [my:type]JOIN [nt:base] as nodeSet1ON ISDESCENDANTNODE(nodeSet1, [my:type])WHERE nodeSet2.id IS NOT NULLL`

Specifying property constraints

Section 6.6.3.3 of the JCR 1.0 specification contains an in-depth description of property value constraints using various comparison operators.

7.3.5. Path Constraints

JCR 1.0 specifies that exact, child node, and descendants-or-self path constraints be supported on the location steps in an XPath query.

The hierarchical database does support the four kinds of path constraints. For example, the following table shows several XPath queries and how they map to JCR-SQL2 queries.

XPath	JCR-SQL2
`/jcr:root/a[1]/b[2]`	`SELECT * FROM [nt:base]WHERE PATH([nt:base]) = '/a[1]/b[2]'`
`/jcr:root/a/b[*]`	`SELECT * FROM [nt:base]WHERE PATH([nt:base]) = '/a[%]/b[%]'`
`/jcr:root/a[1]/b[*]`	`SELECT * FROM [nt:base]WHERE PATH([nt:base]) = '/a[%]/b[%]'`
`/jcr:root/a[2]/b`	`SELECT * FROM [nt:base]WHERE PATH([nt:base]) = '/a[2]/b[%]'`
`/jcr:root/a/b[2]//c[4]`	`SELECT * FROM [my:type]WHERE PATH([nt:base]) = '/a[%]/b[2]/c[4]'OR PATH(nodeSet1) LIKE '/a[%]/b[\2]/%/c[\4]'`
`/jcr:root/a/b//c//d`	`SELECT * FROM [my:type]WHERE PATH([nt:base]) = '/a[%]/b[%]/c[%]/d[%]'OR PATH([nt:base]) LIKE '/a[%]/b[%]/%/c[%]/d[%]'OR PATH([nt:base]) LIKE '/a[%]/b[%]/c[%]/%/d[%]'OR PATH([nt:base]) LIKE '/a[%]/b[%]/%/c[%]/%/d[%]'`
`//element(*,my:type)[@id<1 and @name='john']`	`SELECT * FROM [my:type]WHERE id < 1 AND name = 'john'`
`/jcr:root/a/b//element(*,my:type)`	`SELECT * FROM [my:type]WHERE PATH([my:type]) = '/a[%]/b[%]/%'`

Specifying path constraints

Note that the JCR-SQL2 language supported by the hierarchical database is capable of representing a wider combination of path constraints, although the XPath expressions are easier to understand and significantly shorter.

Also, path constraints in XPath do not need to specify wildcards for the same-name-sibling (SNS) indexes, as XPath should naturally find all nodes regardless of the SNS index, unless the SNS index is explicitly specified. In other words, any path segment that does not have an explicit SNS index (or an SNS index of '[%]' or '[ ]') will match _all SNS index values. However, any segments in the path expression that have an explicit numeric SNS index will require an exact match. Thus this path constraint:

```
/a/b/c\[2]/d\[%]/\%/e\[_]
```

will effectively be converted into

```
/a[%]/b[%]/c\[2]/d\[%]/\%/e\[_]
```

This behavior is very different than how JCR-SQL and JCR-SQL2 path constraints are handled, since these languages interpret a lack of a SNS index as equating to '**[1]**'. To achieve the XPath-like matching, a query written in JCR-SQL or JCR-SQL2 would need to explicitly include '**[%]**' in each path segment where an SNS index literal is not already specified.

7.3.6. Ordering Specifiers

JCR 1.0 extends the XPath grammar to add support for ordering the results according to the natural ordering of the values of one or more properties on the nodes.

The hierarchical database does support zero or more ordering specifiers, including whether each specifier is ascending or descending. If no ordering specifiers are defined, the ordering of the results is not predefined and may vary (though ordering by score may be used by default). For example, the following table shows several XPath queries and how they map to JCR-SQL2 queries.

XPath	JCR-SQL2
`//element(,) order by @title`	`SELECT nodeSet1.titleFROM [nt:base] AS nodeSet1ORDER BY nodeSet1.title`
`//element(,) order by jcr:score()`	`SELECT *FROM [nt:base] AS nodeSet1ORDER BY SCORE(nodeSet1)`
`//element(*,my:type) order by jcr:score(my:type)`	`SELECT *FROM [my:type] AS nodeSet1ORDER BY SCORE(nodeSet1)`
`//element(,) order by @jcr:path`	`SELECT jcr:pathFROM [nt:base] AS nodeSet1ORDER BY PATH(nodeSet1)`
`//element(,) order by @title, @jcr:score`	`SELECT nodeSet1.titleFROM [nt:base] AS nodeSet1ORDER BY nodeSet1.title,SCORE(nodeSet1)`

Specifying result ordering

Note that the JCR-SQL2 language supported by the hierarchical database has a far richer **`ORDER BY`** clause, allowing the use of any kind of dynamic operand, including ordering upon arithmetic operations of multiple dynamic operands.

7.3.7. Miscellaneous

JCR 1.0 defines a number of other optional and required features, and these are summarized in this section.

- Only abbreviated XPath syntax is supported.

- Only the **`child`** axis (the default axis, represented by '/' in abbreviated syntax), **`descendant-or-self`** axis (represented by '//' in abbreviated syntax), **`self`** axis (represented by '.' in abbreviated syntax), and **`attribute`** axis (represent by '@' in abbreviated syntax) are supported.
- The **`text()`** node test is not supported.
- The **`element()`** node test is supported.
- The **`jcr:like()`** function is supported.
- The **`jcr:contains()`** function is supported.
- The **`jcr:score()`** function is supported.
- The **`jcr:deref()`** function is not supported.

7.4. JCR Java Query Object Model

JCR 2.0 introduces a new API for programmatically constructing a query. This API allows the client to construct the lower-level objects for each part of the query, and is a great fit for applications that would otherwise need to dynamically generate query expressions using fairly complicated string manipulation.

Using this API is a matter of getting the **`QueryObjectModelFactory`** from the session's **`QueryManager`** , and using the factory to create the various components, starting with the lowest-level components. Then these lower-level components can be passed to other factory methods to create the higher-level components, and so on, until finally the **`createQuery(...)`** method is called to return the **`QueryObjectModel`** .

Important

Although the JCR-SQL2 and Query Object Model API construct queries in very different ways, executing queries for the two languages is done in nearly the same way. The only difference is that a JCR-SQL2 query expression must be parsed into an abstract syntax tree (AST), whereas with the Query Object Model API your application is programmatically creating objects that effectively are the AST. From that point on, however, all subsequent processing is done in an identical manner for all the query languages.

Do not consider using the QOM API to get a performance benefit. The JCR-SQL2 parser is very efficient, and your application code will be far easier to understand and maintain. Where possible, use JCR-SQL2 query expressions.

7.4.1. Java Query Object Model Example

Here is a simple example that shows how this is done for the simple query **`SELECT * FROM [nt:unstructured] AS unstructNodes`**:

```
// Obtain the query manager for the session ...
javax.jcr.query.QueryManager queryManager =
session.getWorkspace().getQueryManager();

// Create a query object model factory ...
QueryObjectModelFactory factory = queryManager.getQOMFactory();
```

```
// Create the FROM clause: a selector for the [nt:unstructured] nodes ...
Selector source = factory.selector("nt:unstructured","unstructNodes");

// Create the SELECT clause (we want all columns defined on the node type)
...
Column[] columns = null;

// Create the WHERE clause (we have none for this query) ...
Constraint constraint = null;

// Define the orderings (we have none for this query)...
Ordering[] orderings = null;

// Create the query ...
QueryObjectModel query =
factory.createQuery(source,constraint,orderings,columns);

// Execute the query and get the results ...
// (This is the same as before.)
javax.jcr.QueryResult result = query.execute();
```

Obviously this is a lot more code than would be required to submit the fixed query expression, but the purpose of the example is to show how to use the Query Object Model API to build a query that you can easily understand. In fact, most Query Object Model queries will create the columns, orderings, and constraints using the **`QueryObjectModelFactory`** , whereas the example above assumes all of the columns, no orderings, and no constraints.

Once your application executes the **`QueryResult`** , processing the results is exactly the same as when using the JCR Query AP. This is because all of the query languages are represented internally and executed in exactly the same manner. For the sake of completion, here's the code to process the results by iterating over the nodes:

```
javax.jcr.NodeIterator nodeIter = result.getNodes();
while ( nodeIter.hasNext() ) {
    javax.jcr.Node node = nodeIter.nextNode();
                ...
}
```

or iterating over the rows in the results:

```
String[] columnNames = result.getColumnNames();
javax.jcr.query.RowIterator rowIter = result.getRows();
while ( rowIter.hasNext() ) {
    javax.jcr.query.Row row = rowIter.nextRow();
    // Iterate over the column values in each row ...
    javax.jcr.Value[] values = row.getValues();
    for ( javax.jcr.Value value : values ) {
                                ...
    }
    // Or access the column values by name ...
    for ( String columnName : columnNames ) {
        javax.jcr.Value value = row.getValue(columnName);
                                ...
    }
}
```

7.5. Full Text Search

There are times when a formal structured query language is overkill, and the easiest way to find the right content is to perform a search, like you would with a search engine such as Google or Yahoo! This is where the hierarchical database's **full-text search language** comes in, because it allows you to use the JCR query API but with a far simpler, Google-style search grammar.

This query language is actually defined by the JCR 2.0 specification as the full-text search expression grammar used in the second parameter of the **`CONTAINS(...)`** function of the JCR-SQL2 language. We have made it available as a first-class query language, such that a full-text search query supplied by the user, full-text-query , is equivalent to executing this JCR-SQL2:

```
SELECT * FROM [nt:base] WHERE CONTAINS([nt:base],'full-text-query')
```

This language allows a JCR client to construct a query to find nodes with property values that match the supplied terms. Nodes that "best" match the terms are returned before nodes that have a lesser match. Of course, the hierarchical database uses a complex system to analyze the node content and the query terms, and may perform a number of optimizations, such as (but not limited to) eliminating stop words (e.g., "the", "a", "and", etc.), treating terms independent of case, and converting words to base forms using a process called stemming (e.g., "running" into "run", "customers" into "customer").

Search terms can also include phrases by wrapping the phrase with double-quotes. For example, the search term **`table "customer invoice"`** would rank higher those nodes with properties containing the phrase "customer invoice" than nodes with properties containing only "customer" or "invoice".

Term in the query are implicitly AND-ed together, meaning that the matches occur when a node has property values that match all of the terms. However, it is also possible to put an "OR" in between two terms where either of those terms may occur.

By default, all terms are assumed to be positive terms, in the sense that the occurrence of the term will increase the rank of any nodes containing the value. However, it is possible to specify that terms should not appear in the results. This is called a negative term , and it reduces the rank of any node whose property values contain the value. To specify a negative term, prefix the term with a hyphen ('**-**').

Each term may also contain wildcards to specify the pattern to be matched (or negated). The hierarchical database supports two different sets of wildcards:

- '*****' matches zero or more characters, and '**?**' matches any single character; and
- '**%**' matches zero or more characters, and '_' matches any single character.

The former are wildcards that are more commonly used in various systems (including older JCR repository implementations), while the latter are the wildcards used in LIKE expressions in both JCR-SQL and JCR-SQL2. Both families are supported for convenience, and you can also mix and match the various wildcards, such as **`ta*bl_`** and **`ta?_ble*`**. (Of course, placing multiple '*****' or '**%**' characters next to each other offers no real benefit, as it is equivalent to a single '*****' or '**%**'.)

If you want to use these characters literally in a term and do not want them to be treated as wildcards, they must be escaped by prefixing them with a '{{}}' character. For example, this full text search expression:

```
table\* 'customer invoice\?'
```

will would rank higher those nodes with properties containing **`table*`** (including the unescaped asterisk as a wildcard) and those containing the phrase "customer invoice?" (including the unescaped question mark as a wildcard). To use a literal backslash character, escape it as well.

When using this query language, the **`QueryResult`** always contains the **`jcr:path`** and **`jcr:score`**

columns.

Warning

The hierarchical database handles leading and trailing wildcards in very different ways. When trailing wildcards are used, even a few characters preceding the wildcard can be used to quickly narrow down the potential results using the internal reverse indexes. However, when terms start with a wildcard the hierarchical database cannot use the internal reverse indexes to help narrow the results. Thus, performing a search with a leading wildcard must be done in a pretty inefficient manner in a process that is something analogous to a relational database's table scan. Where possible, avoid using leading wildcards in your search terms.

7.5.1. Full Text Search Grammar

The grammar for this full-text search language is specified in Section 6.7.19 of the JCR 2.0 specification , but it is also included here as a convenience.

Important

The grammar is presented using the same EBNF nomenclature as used in the JCR 2.0 specification. Terms are surrounded by matching square brackets (e.g., '**[**' and '**]**') denote optional terms that appear zero or one times. Terms surrounded by matching braces (e.g., '**}**' and '**{**') denote terms that appear zero or more times. Parentheses are used to identify groups, and are often used to surround possible values.

```
FulltextSearch ::= Disjunct {Space 'OR' Space Disjunct}

Disjunct ::= Term {Space Term}

Term ::= ['-'] SimpleTerm

SimpleTerm ::= Word | '"' Word {Space Word} '"'

Word ::= NonSpaceChar {NonSpaceChar}

Space ::= SpaceChar {SpaceChar}

NonSpaceChar ::= Char - SpaceChar /* Any Char except SpaceChar */

SpaceChar ::= ' '

Char ::= /* Any character */
```

As you can see, this is a pretty simple and straightforward query language. But this language makes it extremely easy to find all the nodes in the repository that match a set of terms.

Chapter 8. Built-in Node Types

The JCR 2.0 specification requires that repositories have a number of node types immediately available for use by client applications. The hierarchical database defines a number of additional node types that are installed into every repository. None of these node types can be changed or modified.

8.1. Standard Node Types

The following is the CND representation of the standard JCR built-in node types:

```
<jcr='http://www.jcp.org/jcr/1.0'>
<nt='http://www.jcp.org/jcr/nt/1.0'>
<mix='http://www.jcp.org/jcr/mix/1.0'>

// ----------------------------------------------------------------------
---
//                             Pre-defined Node Types
// ----------------------------------------------------------------------
---

[nt:base] abstract
  - jcr:primaryType (name) mandatory autocreated
      protected compute
  - jcr:mixinTypes (name) protected multiple compute

[nt:unstructured]
  orderable
  - * (undefined) multiple
  - * (undefined)
  + * (nt:base) = nt:unstructured sns version

[mix:created] mixin
  - jcr:created (date) protected
  - jcr:createdBy (string) protected

[nt:hierarchyNode] > mix:created abstract

[nt:file] > nt:hierarchyNode
  + jcr:content (nt:base) primary mandatory

[nt:linkedFile] > nt:hierarchyNode
  - jcr:content (reference) primary mandatory

[nt:folder] > nt:hierarchyNode
  + * (nt:hierarchyNode) version

[mix:referenceable] mixin
  - jcr:uuid (string) mandatory autocreated protected initialize

[mix:mimeType] mixin
  - jcr:mimeType (string)
  - jcr:encoding (string)

[mix:lastModified] mixin
```

```
  - jcr:lastModified (date)
  - jcr:lastModifiedBy (string)

[nt:resource] > mix:mimeType, mix:lastModified
  - jcr:data (binary) primary mandatory

[nt:nodeType]
  - jcr:nodeTypeName (name) mandatory protected copy
  - jcr:supertypes (name) multiple protected copy
  - jcr:isAbstract (boolean) mandatory protected copy
  - jcr:isMixin (boolean) mandatory protected copy
  - jcr:isQueryable (boolean) mandatory protected copy
  - jcr:hasOrderableChildNodes (boolean) mandatory protected copy
  - jcr:primaryItemName (name) protected copy
  + jcr:propertyDefinition (nt:propertyDefinition)  = nt:propertyDefinition
sns protected copy
  + jcr:childNodeDefinition (nt:childNodeDefinition) =
nt:childNodeDefinition sns protected copy

[nt:propertyDefinition]
  - jcr:name (name) protected
  - jcr:autoCreated (boolean) mandatory protected
  - jcr:mandatory (boolean) mandatory protected
  - jcr:isFullTextSearchable (boolean) mandatory protected
  - jcr:isQueryOrderable (boolean) mandatory protected
  - jcr:onParentVersion (string) mandatory protected
      < 'COPY', 'VERSION', 'INITIALIZE', 'COMPUTE',
        'IGNORE', 'ABORT'
  - jcr:protected (boolean) mandatory protected
  - jcr:requiredType (string) mandatory protected
    < 'STRING', 'URI', 'BINARY', 'LONG', 'DOUBLE', 'DECIMAL', 'BOOLEAN',
      'DATE', 'NAME', 'PATH', 'REFERENCE', 'WEAKREFERENCE', 'UNDEFINED'
  - jcr:valueConstraints (string) multiple protected
  - jcr:availableQueryOperators (name) mandatory multiple protected
  - jcr:defaultValues (undefined) multiple protected
  - jcr:multiple (boolean) mandatory protected

[nt:childNodeDefinition]
  - jcr:name (name) protected
  - jcr:autoCreated (boolean) mandatory protected
  - jcr:mandatory (boolean) mandatory protected
  - jcr:onParentVersion (string) mandatory protected
      < 'COPY', 'VERSION', 'INITIALIZE', 'COMPUTE',
        'IGNORE', 'ABORT'
  - jcr:protected (boolean) mandatory protected
  - jcr:requiredPrimaryTypes (name) = 'nt:base' mandatory protected multiple
  - jcr:defaultPrimaryType (name) protected
  - jcr:sameNameSiblings (boolean) mandatory protected

[nt:versionHistory] > mix:referenceable
  - jcr:versionableUuid (string) mandatory autocreated protected abort
  - jcr:copiedFrom (weakreference) protected abort < 'nt:version'
  + jcr:rootVersion (nt:version) = nt:version mandatory autocreated
protected abort
  + jcr:versionLabels (nt:versionLabels) = nt:versionLabels mandatory
autocreated protected abort
```

```
  + * (nt:version) = nt:version protected abort


[nt:versionLabels]
  - * (reference) protected abort < 'nt:version'

[nt:version] > mix:referenceable
  - jcr:created (date) mandatory autocreated protected abort
  - jcr:predecessors (reference) protected multiple abort < 'nt:version'
  - jcr:successors (reference) protected multiple abort < 'nt:version'
  - jcr:activity (reference) protected abort < 'nt:activity'
  + jcr:frozenNode (nt:frozenNode) protected abort

[nt:frozenNode] > mix:referenceable
  orderable
  - jcr:frozenPrimaryType (name) mandatory autocreated protected abort
  - jcr:frozenMixinTypes (name) protected multiple abort
  - jcr:frozenUuid (string) mandatory autocreated protected abort
  - * (undefined) protected abort
  - * (undefined) protected multiple abort
  + * (nt:base) protected sns abort

[nt:versionedChild]
  - jcr:childVersionHistory (reference) mandatory autocreated protected
abort < 'nt:versionHistory'

[nt:query]
  - jcr:statement (string)
  - jcr:language (string)

[nt:activity] > mix:referenceable
  - jcr:activityTitle (string) mandatory autocreated protected


[mix:simpleVersionable] mixin
  - jcr:isCheckedOut (boolean) = 'true' mandatory autocreated protected
ignore

[mix:versionable] > mix:simpleVersionable, mix:referenceable mixin
  - jcr:versionHistory (reference) mandatory protected ignore <
'nt:versionHistory'
  - jcr:baseVersion (reference) mandatory protected ignore < 'nt:version'
  - jcr:predecessors (reference) mandatory protected multiple ignore <
'nt:version'
  - jcr:mergeFailed (reference) protected multiple abort
  - jcr:activity (reference) protected < 'nt:version'
  - jcr:configuration (reference) protected ignore < 'nt:configuration'

[nt:configuration] > mix:versionable
  - jcr:root (reference) mandatory autocreated protected

[nt:address]
  - jcr:protocol (string)
  - jcr:host (string)
  - jcr:port (string)
  - jcr:repository (string)
```

```
  - jcr:workspace (string)
  - jcr:path (path)
  - jcr:id (weakreference)

[nt:naturalText]
  - jcr:text (string)
  - jcr:messageId (string)


// ------------------------------------------------------------------------
---
//                             Pre-defined Mixins
// ------------------------------------------------------------------------
---

[mix:etag] mixin
  - jcr:etag (string) protected autocreated

[mix:lockable] mixin
  - jcr:lockOwner (string) protected ignore
  - jcr:lockIsDeep (boolean) protected ignore

[mix:lifecycle] mixin
  - jcr:lifecyclePolicy (reference) protected initialize
  - jcr:currentLifecycleState (string) protected initialize

[mix:managedRetention] > mix:referenceable mixin
  - jcr:hold (string) protected multiple
  - jcr:isDeep (boolean) protected multiple
  - jcr:retentionPolicy (reference) protected

[mix:shareable] > mix:referenceable mixin

[mix:title] mixin
  - jcr:title (string)
  - jcr:description (string)

[mix:language] mixin
  - jcr:language (string)
```

8.2. Hierarchical Database Built-in Node Types

The following is the CND representation of built-in node types specific to the hierarchical database. Note that many of these outline the structure of nodes under the **`/jcr:system`** area of the repository and are protected (meaning clients can view but not directly modify their content).

```
//------------------------------------------------------------------------
--------
// N A M E S P A C E S
//------------------------------------------------------------------------
--------
<jcr = "http://www.jcp.org/jcr/1.0">
<nt  = "http://www.jcp.org/jcr/nt/1.0">
<mix = "http://www.jcp.org/jcr/mix/1.0">
<mode = "http://www.modeshape.org/1.0">
```

```
//------------------------------------------------------------------------
--------
// N O D E T Y P E S
//------------------------------------------------------------------------
--------

[mode:namespace] > nt:base
- mode:uri (string) primary protected version
- mode:generated (boolean) protected version

[mode:namespaces] > nt:base
+ * (mode:namespace) = mode:namespace protected version

[mode:nodeTypes] > nt:base
+ * (nt:nodeType) = nt:nodeType protected version

[mode:lock] > nt:base
- mode:lockedKey (string) protected ignore
- jcr:lockOwner (string) protected ignore
- mode:lockingSession (string) protected ignore
- mode:expirationDate (date) protected ignore
- mode:sessionScope (boolean) protected ignore
- jcr:isDeep (boolean) protected ignore
- mode:isHeldBySession (boolean) protected ignore
- mode:workspace (string) protected ignore

[mode:locks] > nt:base
+ * (mode:lock) = mode:lock protected ignore

[mode:versionHistoryFolder] > nt:base
+ * (nt:versionHistory) = nt:versionHistory protected ignore
+ * (mode:versionHistoryFolder) protected ignore

[mode:versionStorage] > mode:versionHistoryFolder

[mode:system] > nt:base
+ mode:namespaces (mode:namespaces) = mode:namespaces autocreated mandatory
protected abort
+ mode:locks (mode:locks) = mode:locks autocreated mandatory protected abort
+ jcr:nodeTypes (mode:nodeTypes) = mode:nodeTypes autocreated mandatory
protected abort
+ jcr:versionStorage (mode:versionStorage) = mode:versionStorage autocreated
mandatory protected abort

[mode:root] > nt:base, mix:referenceable orderable
- * (undefined) multiple version
- * (undefined) version
+ jcr:system (mode:system) = mode:system autocreated mandatory protected
ignore
+ * (nt:base) = nt:unstructured sns version

// This is the same as 'nt:resource' (which should generally be used
instead)...
[mode:resource] > nt:base, mix:mimeType, mix:lastModified
- jcr:data (binary) primary mandatory
```

```
[mode:share] > mix:referenceable    // Used for non-original shared nodes,
but never really exposed to JCR clients
- mode:sharedUuid (reference) mandatory protected initialize

[mode:hashed] mixin
- mode:sha1 (string)

// A marker node type that can be used to denote areas into which files can
be published.
// Published areas have optional titles and descriptions.
[mode:publishArea] > mix:title mixin

[mode:derived] mixin
- mode:derivedFrom (path) // the location of the original information from
which this was derived
- mode:derivedAt (date)  // the timestamp of the last change to the original
information from which this was derived
```

Chapter 9. Built-in Sequencers

9.1. Compact Node Type (CND) File Sequencer

The Compact Node Definition (CND) File Sequencer processes JCR CND files to extract node definitions with their property definitions, and inserts these into the repository using aliases of the JCR built-in types. The node structure generated by this sequencer is equivalent to the node structure used in **`/jcr:system/jcr:nodeTypes`** .

9.1.1. CND File Sequencer Example

As an example, consider the following CND file:

```
<mode = "http://www.modeshape.org/1.0">

[mode:example] mixin
- mode:name (string) multiple copy
+ mode:child (mode:example) = mode:example version
```

The resulting graph structure contains the node type information from the CND file above. Note that comments are not sequenced.

```
<mode:example jcr:primaryType=cnd:nodeType
              cnd:isQueryable=true
              cnd:hasOrderableChildNodes=false
              cnd:nodeTypeName=mode:example
              cnd:supertypes=[]
              cnd:isAbstract=false
              cnd:isMixin=true/>

    <cnd:propertyDefinition cnd:requiredType=STRING
                            jcr:primaryType=cnd:propertyDefinition
                            cnd:multiple=true
                            cnd:autoCreated=false
                            cnd:onParentVersion=COPY
                            cnd:mandatory=false
                            cnd:defaultValues=[]
                            cnd:isFullTextSearchable=true
                            cnd:isQueryOrderable=true
                            cnd:name=mode:name
                            cnd:availableQueryOperators=[]
                            cnd:protected=false
                            cnd:valueConstraints=[] />

    <cnd:childNodeDefinition jcr:primaryType=cnd:childNodeDefinition
                             cnd:sameNameSiblings=false
                             cnd:autoCreated=false
                             cnd:onParentVersion=VERSION
                             cnd:defaultPrimaryType=mode:example
                             cnd:mandatory=false
                             cnd:name=mode:child
                             cnd:protected=false
                             cnd:requiredPrimaryTypes=[mode:example] />
```

9.1.2. Using the CND File Sequencer

The CND File Sequencer can be added to the repository configuration like so:

```
{
    "name" : "CNDSequencer Test Repository",
    "sequencing" : {
        "removeDerivedContentWithOriginal" : true,
        "sequencers" : {
            "CND Sequencer" : {
                "description" : "CND Sequencer Same Location",
                "classname" : "CNDSequencer",
                "pathExpressions" : [
"default://(*.cnd)/jcr:content[@jcr:data]" ]
            }
        }
    }
}
```

As with other sequencers, you may use a more restrictive input path expression. For example, if you only want to sequence the CND files stored anywhere under the **`/global/nodeTypes/cnd`** area in the "metadata" workspace, then the path expression might be this:

```
metadata:/global/nodeTypes/cnd//(*.cnd)/jcr:content[@jcr:data]
```

9.2. Data Definition Language (DDL) File Sequencer

The Data Definition Language (DDL) File Sequencer is capable of parsing the more important DDL statements from SQL-92, Oracle, Derby, and PostgreSQL, and constructing a graph structure containing a structured representation of these statements. The resulting graph structure is largely the same for all dialects, though some dialects have non-standard additions to their grammar, and thus require dialect-specific additions to the graph structure.

The sequencer is designed to behave as intelligently as possible with as little configuration. Thus, the sequencer automatically determines the dialect used by a given DDL stream. This can be tricky, of course, since most dialects are very similar and the distinguishing features of a dialect may only be apparent in some of the statements.

To get around this, the sequencer uses a "best fit" algorithm: run the DDL stream through the parser for each of the dialects, and determine which parser was able to successfully read the greatest number of statements and tokens.

Note

It is possible to define which DDL dialects (or grammars) should be considered during sequencing using the "grammars" property in the sequencer configuration. Set the values of this property to the names of the grammars (e.g., "oracle", "postgres", "sql92", or "derby"), specified in the order they should be used. To use a custom DDL parser (not provided by the hierarchical database) provide the fully-qualified class name of the implementation class. If this custom parser implementation is not found on the default classpath, additional classpath URLs can be specified using the "classpath" property of the sequencer.

One useful feature of this sequencer is that, although only a subset of the (more common) DDL statements are supported, the sequencer is still extremely functional since it adds all statements into the output graph (the statement text and the position in the DDL file). Thus, if a DDL file contains statements the sequencer understands and statements the sequencer does not understand, the graph will still contain all statements, where those statements understood by the sequencer will have full detail. Since the underlying parsers are able to operate upon a single statement, it is possible to go back later (after the parsers have been enhanced to support additional DDL statements) and re-parse only those incomplete statements in the graph.

At this time, the sequencer supports SQL-92 standard DDL as well as dialects from Oracle, Derby, and PostgreSQL. It supports:

- Detailed parsing of CREATE SCHEMA, CREATE TABLE and ALTER TABLE.
- Partial parsing of DROP statements
- General parsing of remaining schema definition statements (i.e. CREATE VIEW, CREATE DOMAIN, etc. Note that the sequencer does not perform detailed parsing of SQL (i.e. SELECT, INSERT, UPDATE, etc....) statements.

9.2.1. DDL File Sequencer Example

Below is an example DDL schema definition statement containing table and view definition statements.

```
CREATE SCHEMA hollywood
CREATE TABLE films (title varchar(255), release date, producerName
varchar(255))
CREATE VIEW winners AS SELECT title, release FROM films WHERE producerName
IS NOT NULL;
```

The resulting graph structure contains the raw statement expression, pertinent table, column and key reference information and position of the statement in the text stream (e.g., line number, column number and character index) so the statement can be tied back to the original DDL:

```
<nt:unstructured jcr:name="statements"
                 jcr:mixinTypes = "mode:derived"
                 ddl:parserId="POSTGRES">
   <nt:unstructured jcr:name="hollywood"
jcr:mixinTypes="ddl:createSchemaStatement"
                    ddl:startLineNumber="1"
                    ddl:startColumnNumber="1"
                    ddl:expression="CREATE SCHEMA hollywood"
                    ddl:startCharIndex="0">
   <nt:unstructured jcr:name="films"
jcr:mixinTypes="ddl:createTableStatement"
                    ddl:startLineNumber="2"
                    ddl:startColumnNumber="5"
                    ddl:expression="CREATE TABLE films (title varchar(255),
release date, producerName varchar(255))"
                    ddl:startCharIndex="28"/>
   <nt:unstructured jcr:name="title" jcr:mixinTypes="ddl:columnDefinition"
                    ddl:datatypeName="VARCHAR"
                    ddl:datatypeLength="255"/>
   <nt:unstructured jcr:name="release" jcr:mixinTypes="ddl:columnDefinition"
                    ddl:datatypeName="DATE"/>
   <nt:unstructured jcr:name="producerName"
jcr:mixinTypes="ddl:columnDefinition"
```

```
                    ddl:datatypeName="VARCHAR"
                    ddl:datatypeLength="255"/>
   <nt:unstructured jcr:name="winners"
jcr:mixinTypes="ddl:createViewStatement"
                    ddl:startLineNumber="3"
                    ddl:startColumnNumber="5"
                    ddl:expression="CREATE VIEW winners AS SELECT title,
release FROM films WHERE producerName IS NOT NULL;"
                    ddl:queryExpression="SELECT title, release FROM films
WHERE producerName IS NOT NULL"
                    ddl:startCharIndex="113"/>
</nt:unstructured>
```

Note that all nodes are of type **`nt:unstructured`** while the type of statement is identified using mixins. Also, each of the nodes representing a statement contain: a **`ddl:expression`** property with the exact statement as it appeared in the original DDL stream; a **`ddl:startLineNumber`** and **`ddl:startColumnNumber`** property defining the position in the original DDL stream of the first character in the expression; and a **`ddl:startCharIndex`** property that defines the integral index of the first character in the expression as found in the DDL stream. All of these properties make sure the statement can be traced back to its location in the original DDL.

9.2.2. Using the DDL File Sequencer

To use the DDL File Sequencer, include the **`modeshape-sequencer-ddl`** JAR in your application and configure the repository to use this sequencer using something similar to:

```
{
    "name" : "DdlSequencer Test Repository",
    "sequencing" : {
        "removeDerivedContentWithOriginal" : true,
        "sequencers" : [
            {
                "description" : "Ddl sequencer test",
                "classname" : "DdlSequencer",
                "pathExpressions" : [
"default://(*.ddl)/jcr:content[@jcr:data] => default:/ddl" ]
            }
        ]
    }
}
```

This will use all of the built-in grammars (e.g., "sql92", "oracle", "postgres", and "derby"). To specify a different order or subset of the grammars, use the **`grammars`** parameter. Here's an example that uses the Standard grammar followed by the PostgreSQL grammar:

```
{
    "name" : "DdlSequencer Test Repository",
    "sequencing" : {
        "removeDerivedContentWithOriginal" : true,
        "sequencers" : [
            {
                "description" : "Ddl sequencer test",
                "classname" : "DdlSequencer",
                "grammars" : ["sql92", "postgres"],
Â                "pathExpressions" : [
```

```
"default://(*.ddl)/jcr:content[@jcr:data] => default:/ddl" ]
            }
        ]
    }
}
```

To use a custom implementation, use the fully-qualified name of the implementation class (which must have a no-arg constructor) as the name of the grammar:

```
{
    "name" : "DdlSequencer Test Repository",
    "sequencing" : {
        "removeDerivedContentWithOriginal" : true,
        "sequencers" : {
            "DDL Sequencer" : {
                "description" : "Ddl sequencer test",
                "classname" : "DdlSequencer",
                "grammars" : ["sql92", "postgres",
"org.example.ddl.MyCustomDdlParser"],
                "pathExpressions" : [
"default://(*.ddl)/jcr:content[@jcr:data] => default:/ddl" ]
            }
        }
    }
}
```

9.3. Text File Sequencer

Text sequencers extract data from text streams. There are separate sequencers for character-delimited sequencing and fixed width sequencing, but both treat the incoming text stream as a series of rows (separated by line-terminators, as defined in BufferedReader .readLine() with each row consisting of one or more columns. As noted above, each text sequencer provides its own mechanism for splitting the row into columns.

9.3.1. Abstract Text Sequencer

When using the Abstract Text Sequencer, the default row factory creates one node in the output location for each row sequenced from the source and adds each column with the row as a child node of the row node. The output graph takes the following form (all nodes have primary type **`nt:unstructured`**):

```
<graph root jcr:mixinTypes = mode:derived,
                mode:derivedAt="2011-05-13T13:12:03.925Z",
                mode:derivedFrom="/files/foo.dat">
     + text:row[1]
     |   + text:column[1] (jcr:mixinTypes = text:column, text:data =
<column1 data>)
     |   + ...
     |   + text:column[n] (jcr:mixinTypes = text:column, text:data =
<columnN data>)
     + ...
     + text:row[m]
         + text:column[1] (jcr:mixinTypes = text:column, text:data =
```

```
 <column1 data>)
          + ...
          + text:column[n] (jcr:mixinTypes = text:column, text:data =
 <columnN data>)
```

9.3.2. Abstract Text Sequencer Properties

The **`AbstractTextSequencer`** class provides a number of JavaBean properties that are common to both of the concrete text sequencer classes:

Table 9.1. Abstract Text Sequencer Properties

Property	Description
`commentMarker`	Optional property that, if set, indicates that any line beginning with exactly this string should be treated as a comment and should not be processed further. If this value is null, then all lines will be sequenced. The default value for this property is null
`maximumLinesToRead`	Optional property that, if set, limits the number of lines that will be read during sequencing. Additional lines will be ignored. If this value is non-positive, all lines will be read and sequenced. Comment lines are not counted towards this total. The default value of this property is -1 (indicating that all lines should be read and sequenced).
`rowFactoryClassName`	Optional property that, if set, provides the fully qualified name of a class that provides a custom implementation of the **`RowFactory`** interface. This class must have a no-argument, public constructor. If set, an instance of this class will be created each time that the sequencer sequences an input stream and will be used to provide the output structure of the graph. If this property is set to null, a default implementation will be used. The default value of this property is null.

9.3.3. Delimited Text Sequencer

The Delimited Text Sequencer splits rows into columns based on a regular expression pattern. Although the default pattern is a comma, any regular expression can be provided allowing for more sophisticated splitting patterns.

9.3.4. Delimited Text Sequencer Properties

The **`DelimitedTextSequencer`** class provides an additional JavaBean property to override the default regular expression pattern:

Table 9.2. DelimitedTextSequencer properties

Property	Description

Property	Description
splitPattern	Optional property that, if set, sets the regular expression pattern that is used to split each row into columns. This property may not be set to null and defaults to ",".

9.3.5. Using the Delimited Text Sequencer

To use the Delimited Text Sequencer, include the **modeshape-sequencer-text** JAR in your application and configure the repository to use this sequencer using something similar to:

```
{
    "name" : "Text Sequencers Test Repository",
    "sequencing" : {
        "removeDerivedContentWithOriginal" : true,
        "sequencers" : [
            {
                "name" : "Delimited text sequencer",
                "classname" : "delimitedtext",
                "pathExpression" : "default:/(*.csv)/jcr:content[@jcr:data]
=> /delimited",
                "commentMarker" : "#"
            }
        ]
    }
}
```

9.3.6. Fixed Width Text Sequencer

The Fixed Width Text Sequencer splits rows into columns based on predefined positions. The default setting is to have a single column per row.

9.3.7. Fixed Width Text Sequencer Properties

The **FixedWidthTextSequencer** class provides an additional JavaBean property to override the default start positions for each column.

Table 9.3. FixedWidthTextSequencer Properties

Property	Description
columnStartPositions	Optional property that, if set, specifies an array of integers where each value represents the start position of each column after the first (the start position for the first column never needs to be specified, since it is always '0'). The default value is an empty array, implying that each row should be treated as a single column. This property may not be set to null.

9.3.8. Using the Fixed Width Text Sequencer

To use the Fixed Width Text Sequencer, include the **`modeshape-sequencer-text`** JAR in your application configure the repository to use this sequencer using something similar to:

```
{
    "name" : "Text Sequencers Test Repository",
    "sequencing" : {
        "removeDerivedContentWithOriginal" : true,
        "sequencers" : {
            "Fixed Width Text Sequencer" : {
                "classname" : "fixedwidthtext",
                "pathExpressions" : [
"default:/(*.txt)/jcr:content[@jcr:data] => /fixed" ],
                "columnStartPositions" : [3,6],
                "commentMarker" : "#"
            }
        }
    }
}
```

9.4. Web Service Definition Language (WSDL) File Sequencer

The Web Service Definition Language (WSDL) File Sequencer can parse WSDL files that adhere to the W3C's Web Service Definition Language (WSDL) 1.1 specification, and output a representation of the WSDL file's messages, port types, bindings, services, types (including embedded XML Schemas), documentation, and extension elements (including HTTP, SOAP and MIME bindings). This derived information is intended to mirror the structure and semantics of the actual WSDL files while also making it possible for users to easily navigate, query and search over this derived information. This sequencer captures the namespace and names of all referenced components, and will resolve references to components appearing within the same file.

The WSDL specification allows for a fair amount of variation in WSDL files, and consequently this variation is reflected in the derived output structure.

9.4.1. WSDL File Sequencer Example

Consider an example WSDL file from the WSDL 1.1 specification :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<?xml version="1.0"?>
<definitions name="StockQuote"
             targetNamespace="http://example.com/stockquote.wsdl"
             xmlns:tns="http://example.com/stockquote.wsdl"
             xmlns:xsd1="http://example.com/stockquote.xsd"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

    <types>
       <schema targetNamespace="http://example.com/stockquote.xsd"
              xmlns="http://www.w3.org/2001/XMLSchema">
           <element name="TradePriceRequest">
              <complexType>
                  <all>
                      <element name="tickerSymbol" type="string"/>
                  </all>
```

```
            </complexType>
         </element>
         <element name="TradePrice">
            <complexType>
                <all>
                    <element name="price" type="float"/>
                </all>
            </complexType>
         </element>
      </schema>
   </types>

   <message name="GetLastTradePriceInput">
       <part name="body" element="xsd1:TradePriceRequest"/>
   </message>

   <message name="GetLastTradePriceOutput">
       <part name="body" element="xsd1:TradePrice"/>
   </message>

   <portType name="StockQuotePortType">
       <operation name="GetLastTradePrice">
          <input message="tns:GetLastTradePriceInput"/>
          <output message="tns:GetLastTradePriceOutput"/>
       </operation>
   </portType>

   <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
       <soap:binding style="document"
                     transport="http://schemas.xmlsoap.org/soap/http"/>
       <operation name="GetLastTradePrice">
          <soap:operation
                  soapAction="http://example.com/GetLastTradePrice"/>
          <input>
              <soap:body use="literal"/>
          </input>
          <output>
              <soap:body use="literal"/>
          </output>
       </operation>
   </binding>

   <service name="StockQuoteService">
       <documentation>My first service</documentation>
       <port name="StockQuotePort" binding="tns:StockQuoteBinding">
          <soap:address location="http://example.com/stockquote"/>
       </port>
   </service>
</definitions>
```

This WSDL definition includes an embedded XML Schema that defines the structure of two XML elements used in the web service messages, and it defines a 'StockQuotePortType' port type with input and output messages, a SOAP binding, and a SOAP service.

Within the **`wsdl:messages`** container node are all of the messages. In this case, there are two: the **`GetLastTradePriceInput`** input message and **`GetLastTradePriceOutput`** output message for the

GetLastTracePrice operation defined a bit later in the structure. Note how these messages contain the name, namespace URI, and REFERENCE to the corresponding **element** node in the embedded schema content. (If the element reference could not be resolved, REFERENCE property would not be set.)

Within the **wsdl:portTypes** container node are all of the port types. In this example, there is one: the **StockQuotePortType** that contains a single **GetLastTradePrice** operation. Here, the operation's input and output reference the corresponding message nodes vi the name, namespace URI, and REFERENCE property. Again, the REFERENCE property would not be set if the input and/or output use a message that is not in this WSDL file.

Within the **wsdl:bindings** container node are all of the bindings defined in the WSDL. In this example, there is a single binding that uses SOAP extensions, which describe all of the SOAP-specific information for the port type. The sequencer also supports HTTP and MIME extensions. And node how the input, output and faults of each binding operation reference (using the name, namespace URI, and REFERENCE properties) the corresponding input, output and fault (respectively) in the correct port type.

Finally, within the **wsdl:services** container node are all of the services defined in the WSDL. In this example, there is a single SOAP service that references the **StockQuotePortType** port type.

This example shows the basic structure this sequencer derives from WSDL 1.1 files. Not only does this structure mirror that of the actual WSDL file, but it makes this structure easy to navigate, search and query, especially when it includes the names and namespace URIs of the referenced components (and setting REFERENCE properties to the referenced component where possible).

9.4.2. WSDL File Sequencer Node Types

The WSDL 1.1 sequencer follows JCR best-practices by defining all nodes to have a primary type that allows any single or multi-valued property, meaning it is possible and valid for any node to have any property (with single or multiple values). This sequencer does not add any such properties or nodes, but you are free to annotate the structure as needed.

9.4.3. Using the WSDL File Sequencer

To use the WSDL File Sequencer, include the appropriate version of the Maven artifact with a **org.modeshape** group ID and **modeshape-sequencer-wsdl** artifact ID and configure your repository similar to:

```
{
    "name" : "WSDL Sequencer Test Repository",
    "sequencing" : {
        "removeDerivedContentWithOriginal" : true,
        "sequencers" : {
            "WSDL Sequencer" : {
                "classname" : "wsdlsequencer",
                "pathExpressions" : [
"default:/(*.wsdl)/jcr:content[@jcr:data] => /wsdl" ]
            }
        }
    }
}
```

9.5. Extensible Markup Language (XML) File Sequencer

The Extensible Markup Language (XML) File Sequencer stores the structure and data of an XML file into the repository. DTD, entity, comments, and other content are maintained by the sequencer in the output structure.

9.5.1. XML File Sequencer Example

For this XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.4//EN" "http://www.oasis-
open.org/docbook/xml/4.4/docbookx.dtd" [
<!ENTITY % RH-ENTITIES SYSTEM "Common_Config/rh-entities.ent">
<!ENTITY versionNumber "0.1">
<!ENTITY copyrightYear "2008">
<!ENTITY copyrightHolder "Red Hat Middleware, LLC.">]>
<?target content ?>
<?target2 other stuff ?>
<Cars xmlns:jcr="http://www.jcp.org/jcr/1.0">
    <!-- This is a comment -->
    <Hybrid>
        <car jcr:name="Toyota Prius"/>
    </Hybrid>
    <Sports>
    </Sports>
</Cars>
```

The sequencer will generate this content (assuming its output is redirected to **`xml/myxml`**)

```
<xml jcr:primaryType=nt:unstructured
 <myxml jcr:primaryType="modexml:document"
        jcr:mixinTypes="mode:derived"
        mode:derivedAt="2011-05-13T13:12:03.925Z"
        mode:derivedFrom="/files/docForReferenceGuide.xml"
        modedtd:name="book"
        modedtd:publicId="-//OASIS//DTD DocBook XML V4.4//EN"
        modedtd:systemId="http://www.oasis-
open.org/docbook/xml/4.4/docbookx.dtd">
  <modedtd:entity jcr:primaryType="modedtd:entity"
                  modedtd:name="%RH-ENTITIES"
                  modedtd:systemId="Common_Config/rh-entities.ent" />
  <modedtd:entity[2] jcr:primaryType="modedtd:entity"
                     modedtd:name="versionNumber"
                     modedtd:value="0.1" />
  <modedtd:entity[3] jcr:primaryType="modedtd:entity"
                     modedtd:name="copyrightYear"
                     modedtd:value="2008" />
  <modedtd:entity[4] jcr:primaryType="modedtd:entity"
                     modedtd:name="copyrightHolder"
                     modedtd:value="Red Hat Middleware, LLC." />
  <modexml:processingInstruction
jcr:primaryType="modexml:processingInstruction"

modexml:processingInstructionContent="content"
                                  modexml:target="target" />
  <modexml:processingInstruction[2]
```

```
jcr:primaryType="modexml:processingInstruction"

modexml:processingInstructionContent="other stuff"
                                     modexml:target="target2" />
  <Cars jcr:primaryType="modexml:element">
    <modexml:comment jcr:primaryType="modexml:comment"
                     modexml:commentContent="This is a comment" />
    <Hybrid jcr:primaryType="modexml:element">
      <car jcr:primaryType="modexml:element" />
    </Hybrid>
    <Sports jcr:primaryType="modexml:element" />
  </Cars>
</myxml>
```

9.5.2. XML File Sequencer CND

The CND used by this sequencer is provided below. Note that the XML sequencer will parse CDATA into its own node in the sequenced output even though the example above does not explicitly demonstrate this.

```
//------------------------------------------------------------------------
--------
// N A M E S P A C E S
//------------------------------------------------------------------------
--------
<jcr='http://www.jcp.org/jcr/1.0'>
<nt='http://www.jcp.org/jcr/nt/1.0'>
<mix='http://www.jcp.org/jcr/mix/1.0'>
<modexml='http://www.modeshape.org/xml/1.0'>
<modedtd='http://www.modeshape.org/dtd/1.0'>

//------------------------------------------------------------------------
--------
// N O D E T Y P E S
//------------------------------------------------------------------------
--------

[modexml:document] > nt:unstructured, mix:mimeType
  - modexml:cDataContent (string)

[modexml:comment] > nt:unstructured
  - modexml:commentContent (string)

[modexml:element] > nt:unstructured

[modexml:elementContent] > nt:unstructured
  - modexml:elementContent (string)

[modexml:cData] > nt:unstructured
  - modexml:cDataContent (string)

[modexml:processingInstruction] > nt:unstructured
  - modexml:processingInstruction (string)
  - modexml:target (string)

[modedtd:entity] > nt:unstructured
```

```
  - modexml:name (string)
  - modexml:value (string)
  - modexml:publicId (string)
  - modexml:systemId (string)
```

9.5.3. Using the XML File Sequencer

To use the XML File Sequencer, include **`modeshape-sequencer-xml.jar`** in your classpath and configure your repository similar to:

```
{
    "name" : "XML Sequencer Test Repository",
    "sequencing" : {
        "removeDerivedContentWithOriginal" : true,
        "sequencers" : {
            "XML Sequencer" : {
                "classname" : "xmlsequencer",
                "pathExpressions" : [
"default:/(*.xml)/jcr:content[@jcr:data] => /xml" ]
            }
        }
    }
}
```

9.6. XML Schema Document (XSD) File Sequencer

The XML Schema Document (XSD) File Sequencer can parse XML Schema Documents that adhere to the W3C's XML Schema Part 1 and Part 2 specifications, and output a representation of the XSD's attribute declarations, element declarations, simple type definitions, complex type definitions, import statements, include statements, attribute group declarations, annotations, other components, and even attributes with a non-schema namespace. This derived information is intended to accurately reflect the structure and semantics of the XSD files while also making it possible for users to easily navigate, query and search over this derived information. This sequencer captures the namespace and names of all referenced components, and will resolve references to components appearing within the same files.

The XML Schema specification is powerful, flexible, rich, and complicated. This means that many XML Schema Documents themselves are complicated. But it also means that there is a lot of variation in XSDs, and consequently there is a lot of variation in the output structure that this sequencer derives from XSD files.

9.6.1. XSD File Sequencer Example

Consider an example XML Schema Document taken from the XML Schema Primer :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:annotation>
    <xsd:documentation xml:lang="en">
     Purchase order schema for Example.com.
     Copyright 2000 Example.com. All rights reserved.
    </xsd:documentation>
  </xsd:annotation>
```

```
<xsd:element name="purchaseOrder" type="PurchaseOrderType"/>

<xsd:element name="comment" type="xsd:string"/>

<xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
    <xsd:element name="shipTo" type="USAddress"/>
    <xsd:element name="billTo" type="USAddress"/>
    <xsd:element ref="comment" minOccurs="0"/>
    <xsd:element name="items"  type="Items"/>
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>

<xsd:complexType name="USAddress">
  <xsd:sequence>
    <xsd:element name="name"   type="xsd:string"/>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city"   type="xsd:string"/>
    <xsd:element name="state"  type="xsd:string"/>
    <xsd:element name="zip"    type="xsd:decimal"/>
  </xsd:sequence>
  <xsd:attribute name="country" type="xsd:NMTOKEN"
                 fixed="US"/>
</xsd:complexType>

<xsd:complexType name="Items">
  <xsd:sequence>
    <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="productName" type="xsd:string"/>
          <xsd:element name="quantity">
            <xsd:simpleType>
              <xsd:restriction base="xsd:positiveInteger">
                <xsd:maxExclusive value="100"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="USPrice"  type="xsd:decimal"/>
          <xsd:element ref="comment"   minOccurs="0"/>
          <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
        </xsd:sequence>
        <xsd:attribute name="partNum" type="SKU" use="required"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>

<!-- Stock Keeping Unit, a code for identifying products -->
<xsd:simpleType name="SKU">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="\d{3}-[A-Z]{2}"/>
```

```
    </xsd:restriction>
  </xsd:simpleType>

</xsd:schema>
```

This schema defines the structure of several XML elements used to represent purchase orders, and describes an XML document such as the following:

```
<?xml version="1.0"?>
<purchaseOrder orderDate="1999-10-20">
   <shipTo country="US">
      <name>Alice Smith</name>
      <street>123 Maple Street</street>
      <city>Mill Valley</city>
      <state>CA</state>
      <zip>90952</zip>
   </shipTo>
   <billTo country="US">
      <name>Robert Smith</name>
      <street>8 Oak Avenue</street>
      <city>Old Town</city>
      <state>PA</state>
      <zip>95819</zip>
   </billTo>
   <comment>Hurry, my lawn is going wild<!/comment>
   <items>
      <item partNum="872-AA">
         <productName>Lawnmower</productName>
         <quantity>1</quantity>
         <USPrice>148.95</USPrice>
         <comment>Confirm this is electric</comment>
      </item>
      <item partNum="926-AA">
         <productName>Baby Monitor</productName>
         <quantity>1</quantity>
         <USPrice>39.98</USPrice>
         <shipDate>1999-05-21</shipDate>
      </item>
   </items>
</purchaseOrder>
```

9.6.2. XSD File Sequencer Node Types

The XSD sequencer follows JCR best-practices by defining all nodes to have a primary type that allows any single or multi-valued property, meaning it is possible and valid for any node to have any property (with single or multiple values). In fact, this feature is used when XSD files contain attributes with non-schema namespaces, which are then mapped onto properties with the attributes name and possibly-empty namespace. However, it is still useful to capture the metadata about what that node represents, and so the sequencer use explicit node type definitions and mixins for this.

9.6.3. Using the XSD File Sequencer

To use the XSD File Sequencer, include the appropriate version of the Maven artifact with a **`org.modeshape`** group ID and **`modeshape-sequencer-xsd`** artifacts ID. Alternatively, if you are using JAR files and manually setting up the classpath for your application, use the **`modeshape-sequencer-xsd-`**

2.7.0.Final-jar-with-dependencies.jar file. Then, define a sequencing configuration, using something similar to this:

```
{
    "name" : "XSD Sequencer Test Repository",
    "sequencing" : {
        "removeDerivedContentWithOriginal" : true,
        "sequencers" : {
            "XSD Sequencer" : {
                "classname" : "xsdsequencer",
                "pathExpressions" : [
"default:/(*.xsd)/jcr:content[@jcr:data]" ]
            }
        }
    }
}
```

9.7. ZIP File Sequencer

The ZIP file sequencer extracts the files and folders contained in the ZIP archive file, extracting the files and folders into the repository using JCR's **nt:file** and **nt:folder** built-in node types. The structure of the output thus matches the logical structure of the contents of the ZIP file.

Example

This sequencer generates a graph structure that maps to the files and folders in the ZIP file. An example (listed in the JCR document view) from sequencing a ZIP file written into **/a/foo** and containing one file, **/x/y/z.txt** is provided below:

```
<foo jcr:primaryType="zip:file"
     jcr:mixinTypes="mode:derived">
  <x jcr:primaryType="nt:folder"
     jcr:created="2011-05-12T20:07Z"
     jcr:createdBy="currentJcrUser">
    <y jcr:primaryType="nt:folder"
       jcr:created="2011-05-12T20:09Z"
       jcr:createdBy="currentJcrUser">
      <z.txt jcr:primaryType="nt:file">
        <jcr:content jcr:primaryType="nt:resource"
                     jcr:data="This is the file content"
                     jcr:lastModified="2011-05-12T20:12Z"
                     jcr:lastModifiedBy="currentJcrUser"
                     jcr:mimeType="text/plain" />
      </z.txt>
    </y>
  </x>
</foo>
```

The CND for the **zip:file** node type is listed below.

```
[zip:file] > nt:folder, mix:mimeType
```

9.7.1. Using the ZIP File Sequencer

To use this sequencer, include the **`modeshape-sequencer-zip`** JAR in your application and configure the repository similar to:

```
{
    "name" : "ZIP Sequencer Test Repository",
    "sequencing" : {
        "removeDerivedContentWithOriginal" : true,
        "sequencers" : {
            "ZIP Sequencer" : {
                "classname" : "zipsequencer",
                "pathExpressions" : [
"default:/(*.zip)/jcr:content[@jcr:data] => /zip" ]
            }
        }
    }
}
```

Chapter 10. Built-in Connectors

The hierarchical database comes with several connectors so that you can set up repositories that federate data from external systems.

10.1. File System Connector

This connector exposes files and folders on the file system as **`nt:file`** and **`nt:folder`** nodes in the repository. To use, configure an external source for a given file system (or area of the repository); each external source can be set up as read-only (to only expose the file system's existing files and folders) or as writable (to allow JCR clients to create/update/delete files and folders on the file system).

The File System Connector maps **`nt:file`** and **`nt:folder`** properties directly to the attributes on the file system's files and folders. By default, the hierarchical database will store these extra properties in the same Infinispan cache where the normal content is stored, though such content will be lost if files and folders are moved or renamed outside of the hierarchical database. Several other options are possible, including storing these extra properties on the file system using "sidecar" files that are named similarly to and stored adjacent to the target file or folder. See the **`extraPropertiesStorage`** attribute description below for more detail.

The connector does not currently monitor the file system for newly created files or folders, and therefore no events are created. However, navigation will always expose the current files/folder nodes within a folder. The hierarchical database can index the content so that the projected **`nt:file`** , **`nt:folder`** , and **`nt:resource`** nodes can be queried, but this must be done manually via the Workspace API's **`reindex`** methods.

Note

As of this release, the file system connector is pageable , which means it can efficiently expose folders that contain large numbers of items. Paging is a tradeoff between loading the parent node faster (by having smaller numbers of child references) and having to go back to the connector more frequently. By default, the connector includes only 20 items per page, so the page size can be adjusted to best suit your application's needs.

The connector classname is **`org.modeshape.connector.filesystem.FileSystemConnector`**, and there are several attributes that should be configured on each external source:

Attribute Name	Description
`directoryPath`	The path to the file or folder that is to be accessed by this connector.

Attribute Name	Description
`extraPropertyStorage`	An optional string flag that specifies how this source handles "extra" properties that are not stored via file system attributes. The value should be one of the following: - **`store`** - Any extra properties are stored in the same Infinispan cache where the content is stored. This is the default and is used if the actual value does not match any of the other accepted values. - **`json`** - Any extra properties are stored in a JSON file next to the file or directory. - **`legacy`** - Any extra properties are stored in a file next to the file or directory. This is generally discouraged unless you were using a previous version of the hierarchical database and have a directory structure that already contains these files. - **none** - An exception is thrown if the nodes contain any extra properties.
`inclusionPattern`	Optional property that specifies a regular expression that is used to help determine which files and folders in the underlying file system are exposed through this connector. The connector will expose only those files and folders with a name that matches the provided regular expression (as long as they also are not excluded by the **`exclusionPattern`**). If no inclusion pattern is specified, then the connector will include all files and folders that are not excluded via the **`exclusionPattern`** .
`exclusionPattern`	Optional property that specifies a regular expression that is used to help determine which files and folders in the underlying file system are not exposed through this connector. Files and folders with a name that matches the provided regular expression will not be exposed by this source.
`addMimeTypeMixin`	A boolean flag that specifies whether this connector should add the **`mix:mimeType`** mixin to the **`nt:resource`** nodes to include the **`jcr:mimeType`** property. If set to true, the MIME type is computed immediately when the **`nt:resource`** node is accessed, which might be expensive for larger files. This is false by default.
`readOnly`	A boolean flag that specifies whether this source can create/modify/remove files and directories on the file system to reflect changes in the JCR content. By default, sources are not read-only.

Attribute Name	Description
cacheTtlSeconds	Optional property that specifies the default maximum number of seconds (i.e., time to live) that a node returned by this connector should be cached in the workspace cache before being expired. By default, the connector will not set a special value, and the repository will determine how long the node is cached in the workspace cache.
isQueryable	Optional property that specifies whether or not the content exposed by this connector should be indexed by the repository. This acts as a global flag, allowing a specific connector to mark its entire content as non-queryable. By default, all content exposed by a connector is queryable.
pageSize	**(Added in this release)** Optional property that controls the number of children that the connector should include in a single page; the default is 20. For example, if a folder contains 200 items (e.g., files or folders) and the page size is 20, then the connector will include in the document representing this folder only the properties of the folder and the first 20 items (that are readable, that satisfy the inclusion pattern, and that does not match the exclusion pattern). As additional children are needed (e.g., as the hierarchical database client navigates or accesses the folder's child nodes), the hierarchical database will request additional pages, each with up to 20 items.

By default, the file system connector will expose all of the files and folders that are underneath the specified directory and readable by the Java process, and it will allow hierarchical database clients using the JCR API to change, remove, or even create new files and folders. Additionally, any "extra properties" (e.g., those that are not directly mappable to file system attributes, such as **jcr:primaryType**, **jcr:created**, **jcr:lastModified**, and **jcr:data**) will be stored not on the file system but in the same Infinispan cache that the repositories own internal (non-federated) content is stored. The connector will also use pages to efficiently work with folders with large numbers of items.

If other behavior is desired, set the connector's properties to non-default values. For example, if hierarchical database clients are not allowed to modify, create, or remove file and folder nodes, then the connector should be configured with **readOnly** set to **true** . Or, if only certain files and folders are to be exposed, set the **inclusionPattern** and **exclusionPattern** to regular expressions that the connector can use to know whether to include or exclude files and folders by name. Note that any file or folder will only be exposed by the connector when the file/folder is readable and when its name satisfies the **inclusionPattern**and does not satisfy the exclusion pattern.

The connector is often used to expose as content in a repository the existing files and folders on the file system. Since the connector does not access any OS-specific file attributes, the connector maps each existing file and folder as follows:

- A folder is represented in the hierarchical database as a node with a primary type of **nt:folder**, no mixin types, and the **jcr:created** timestamp set to the last modified timestamp given by the file system. The node will contain a child for each file and folder that are to be exposed (as discussed above).
- A file is represented in the hierarchical database as a node with a primary type of **nt:file**, no mixin types, and the **jcr:created** timestamp set to the last modified timestamp given by the file system. The node will contain a single child node named **jcr:content** that represents the content of the file, and

which has a primary type of **`nt:resource`** and the **`jcr:lastModified`** timestamp set to the file system's last modified timestamp for the file. If the connector is configured with **`addMimeTypeMixin`** set to **`true`** , then the hierarchical database will also attempt to determine the MIME type for the file's content and, if determined, add the **`mix:mimeType`** mixin and the **`jcr:mimeType`** property to the **`jcr:content`** node.

Here is a sample configuration that projects the **`//a/b/c`** directory onto a node the repository at **`/files`**, with the above (default) behavior:

```
{
    ...
    "externalSources" : {
        "local-git-repo" : {
            "classname" :
"org.modeshape.connector.filesystem.FileSystemConnector",
            "directoryPath" : "/a/b/c/",
            "projections" : \[ "/files" \]
        }
    }
    ...
}
```

Here is a slightly different configuration that is read-only, that excludes any files or folders with names that end with "{{.tmp}" (and have at least one character before this suffix), and that includes the automatically-detected MIME type:

```
{
    ...
    "externalSources" : {
        "local-git-repo" : {
            "classname" :
"org.modeshape.connector.filesystem.FileSystemConnector",
            "directoryPath" : "/a/b/c/",
            "projections" : \[ "/files" \],
            "readOnly" : true,
            "addMimeTypeMixin" : true,
            "exclusionPattern" : ".+[.]tmp$"
        }
    }
    ...
}
```

Of course, some applications may want to set additional properties and/or mixins. When the connector is writable (e.g., not read-only), the connector can store these properties in one of several places, based upon the **`extraPropertyStorage`** configuration property. By default, these extra properties are stored in the same Infinispan cache where the hierarchical database repository stores the rest of its internal (non-federated) content. This is convenient, but can lead to orphaned documents in the Infinispan cache should files and folder be removed outside of the hierarchical database.

Alternatively, the connector can store these extra properties on the file system. Any extra properties on a file or folder will be stored in a "sidecar" next to the corresponding file or folder and named similarly to the corresponding file or folder but with a special suffix. If stored as a JSON file, the suffix will be **`.modeshape.json`**, or if stored as a text file the suffix will be **`.modeshape`**. (The text format is the same as

that used in the previous release, but is provided only for backward compatibility. Where possible, choose the JSON format.) Extra properties on the **`jcr:content`** child of **`nt:file`** nodes are stored in a different sidecar file, named similarly to the corresponding file but with the **`.content.modeshape.json`** or **`.content.modeshape`** suffix. Note that these sidecar files are never exposed as nodes by the connector.

It is even possible to prevent updating or creating files and folders with extra properties. To do this, configure the connector with the **`extraPropertyStorage`** property set to **`none`**.

Here is another sample configuration for a connector that works the same as the earlier configuration except that it is now storing extra properties in a JSON sidecar:

```
{
    ...
    "externalSources" : {
        "local-git-repo" : {
            "classname" :
"org.modeshape.connector.filesystem.FileSystemConnector",
            "directoryPath" : "/a/b/c/",
            "projections" : \[ "/files" \],
            "readOnly" : true,
            "addMimeTypeMixin" : true,
            "exclusionPattern" : ".+[.]tmp$",
            "extraPropertyStorage" : "json"
        }
    }
    ...
}
```

10.2. Git Connector

This read-only connector exposes the branches, tags, and commits in a local Git repository as nodes within a repository. The structure is pre-defined by the connector so that the branches, tags, commits, and their files and folders are all accessible via navigation, via identifiers, or via query (if configured).

The connector classname is **`org.modeshape.connector.git.GitConnector`**, and there are several attributes that should be configured on each external source:

Attribute Name	Description
`directoryPath`	The path to the folder that is or contains the .git data structure is to be accessed by this connector. This is required.
`includeMimeType`	A boolean flag denoting whether the MIME types for the files should be determined and included as a property on the node. This is 'false' by default.

Attribute Name	Description
`remoteName`	The alias used by the local Git repository for the remote repository. The default is **`origin`**, which is common in Git repositories. If the value contains commas, the value contains an ordered list of remote aliases that should be accessed; the first one to match an existing remote will be used. The remote names are used to know which branches should be exposed: if at least one remote name is given, then only the branches in the remote(s) will be exposed; if no remotes are given, then all local branches will be exposed.
`queryableBranches`	An array with the names of the branches that should be queryable by the repository. By default, only the **`master`** branch is queryable. Set this to an empty array if no branches are to be queryable.
`cacheTtlSeconds`	Optional property that specifies the default maximum number of seconds (i.e., time to live) that a node returned by this connector should be cached in the workspace cache before being expired. By default, the connector will not set a special value, and the repository will determine how long the node is cached in the workspace cache.

Here is a sample configuration that projects the Git repository located at **`/home/jsmith/git/MyRepo`** on the local file system into the repository under the **`/git/MyRepo`** node, which will have a primary type of **`git:root`**. The **`master`** and **`2.x`** branches will be included in the hierarchical database indexes when the content is reindexed, and MIME types will be included on all **`git:resource`** nodes (that is, the **`jcr:content`** child of the **`git:file`** nodes). The list of branches and tags will include those on the **`upstream`** and **`origin`** remotes.

```
{
    ...
    "externalSources" : {
        "local-git-repo" : {
            "classname" : "org.modeshape.connector.git.GitConnector",
            "directoryPath" : "/home/jsmit/git/MyRepo/",
            "remoteName" : "upstream,origin",
            "includeMimeType" : true,
            "queryableBranches" : ["master","2.x"],
            "projections" : \[ "/git/MyRepo" \]
        }
    }
    ...
}
```

And here is a description of the repository structure:

Path	Description
`/branches/{branchName}`	The list of branches.
`/tags/{tagName}`	The list of tags.

Path	Description
/commits/{branchOrTagNameOrCommit\/{objectId}	The history of commits on the branch, tag or object ID name "{ **branchOrTagNameOrCommit** }", where "{ **objectId** }" is the object ID of the commit.
/commit/{branchOrTagNameOrCommit}	The information about a particular branch, tag or commit "{ **branchOrTagNameOrCommit** }".
/tree/{branchOrTagOrObjectId}/{filesAndFolders}/...	The structure of the directories and files in the specified branch, tag or commit "{ **branchOrTagNameOrCommit** }".

The node types used by the connector are specified here . Some of the more important node types include:

Node Type	Description
git:committed	A mixin that defines the **git:objectId** (SHA-1 hash), **git:author** , **git:committer** , **git:committed** (date), and **git:title** properties that appear on all "committed" nodes.
git:file	The primary node type for a node representing a file in a Git repository. Extends both **nt:file** and **git:committed** .
git:folder	The primary node type for a node representing a folder in a Git repository. Extends both **nt:folder** and **git:committed** .
git:resource	The primary node type for a node representing the **jcr:content** child of **git:file** nodes, where content-related information is placed. Extends both **nt:resource** and **git:committed** .
git:branch	The primary node type for a node representing a Git branch.
git:tag	The primary node type for a node representing a Git tag.
git:commit	The primary node type for a node representing a Git commit.
git:branches	The primary node type for the node that contains the list of **git:branch** nodes.
git:tags	The primary node type for the node that contains the list of **git:tag** nodes.
git:commits	The primary node type for the node that contains a list of **git:commit** nodes.
git:root	The primary node type for the top-level node of the repository.

10.3. CMIS Connector

This connector exposes the content of a CMIS repository.

The Content Management Interoperability Services (CMIS) standard defines a domain model and Web Services, Restful AtomPub and browser (JSON) bindings that can be used by applications to work with one or more Content Management repositories/systems.

The CMIS connector is designed to be layered on top of existing Content Management systems. It is intended to use Apache Chemistry API to access services provided by Content Management system and incorporate those services into the hierarchical database content repository.

The connector class name is **`org.modeshape.connector.cmis.CmisConnector`**, and there are several attributes that should be configured on each external source:

Attribute Name	Description
`aclService`	URL of the Access list service binding entry point. The ACL Services are used to discover and manage Access Control Lists.
`discoveryService`	URL of the Discovery service binding entry point. Discovery service executes a CMIS query statement against the contents of the repository.
`multifilingService`	URL of the Multi-filing service binding entry point. The Multi-Ã¯Â¬Âling Services are used to Ã¯Â¬Âle/un-Ã¯Â¬Âle objects into/from folders.
`navigationService`	URL of the Navigation service binding entry point. The Navigation service gets the list of child objects contained in the speciÃ¯Â¬Âed folder.
`objectService`	URL of the Object service binding entry point. Creates a document object of the speciÃ¯Â¬Âed type (given by the cmis:objectTypeId property) in the (optionally) speciÃ¯Â¬Âed location
`policyService`	URL of the Policy service binding entry point. Applies a speciÃ¯Â¬Âed policy to an object.
`relationshipService`	URL of the Relationship service binding entry point. Gets all or a subset of relationships associated with an independent object.
`repositoryService`	URL of the Repository service binding entry point. Returns a list of CMIS repositories available from this CMIS service endpoint.
`versioningService`	URL of the Policy service binding entry point. Create a private working copy (PWC) of the document.
`readOnly`	A boolean flag that specifies whether this source can create/modify/remove files and directories on the file system to reflect changes in the JCR content. By default, sources are not read-only.
`cacheTtlSeconds`	Optional property that specifies the default maximum number of seconds (i.e., time to live) that a node returned by this connector should be cached in the workspace cache before being expired. By default, the connector will not set a special value, and the repository will determine how long the node is cached in the workspace cache.
`isQueryable`	Optional property that specifies whether or not the content exposed by this connector should be indexed by the repository. This acts as a global flag, allowing a specific connector to mark its entire content as non-queryable. By default, all content exposed by a connector is queryable.

Here is a sample configuration that projects the CMIS repository into the Modeshape repository under the **`/cmis/`** node

```
{
    ...
    "externalSources" : {
        "cmis" : {
            "classname" : "org.modeshape.connector.CmisConnector",
            "cacheTtlSeconds" : 5,
            "aclService" : "http://localhost:8080/services/ACLService?wsdl",
            "discoveryService" :
"http://localhost:8080/services/DiscoveryService?wsdl",
            "multifilingService" :
"http://localhost:8080/services/MultifilingService?wsdl",
            "navigationService" :
"http://localhost:8080/services/NavigationService?wsdl",
            "objectService" : "http://localhost:8080/services/ObjectService?
wsdl",
            "policyService" : "http://localhost:8080/services/PolicyService?
wsdl",
            "relationshipService" :
"http://localhost:8080/services/RelationshipService?wsdl",
            "repositoryService" :
"http://localhost:8080/services/RepositoryService?wsdl",
            "versioningService" :
"http://localhost:8080/services/VersioniongService?wsdl",
            "repositoryId" : "A1",
            "projections" : [ "default:/cmis => /" ]
        }
    }
    ...
}
```

Here is the same configuration except that a variable is used so that the actual URLs can be set with a system property:

```
{
    ...
    "externalSources" : {
        "cmis" : {
            "classname" : "org.modeshape.connector.CmisConnector",
            "cacheTtlSeconds" : 5,
            "aclService" : "${custom.cmis.services.url}/ACLService?wsdl",
            "discoveryService" :
"${custom.cmis.services.url}/DiscoveryService?wsdl",
            "multifilingService" :
"${custom.cmis.services.url}/MultifilingService?wsdl",
            "navigationService" :
"${custom.cmis.services.url}/NavigationService?wsdl",
            "objectService" : "${custom.cmis.services.url}/ObjectService?
wsdl",
            "policyService" : "${custom.cmis.services.url}/PolicyService?
wsdl",
            "relationshipService" :
"${custom.cmis.services.url}/RelationshipService?wsdl",
            "repositoryService" :
"${custom.cmis.services.url}/RepositoryService?wsdl",
            "versioningService" :
"${custom.cmis.services.url}/VersioniongService?wsdl",
```

```
            "repositoryId" : "A1",
            "projections" : [ "default:/cmis => /" ]
        }
    }
    ...
}
```

The Repository structure is defined as follows

Path	Description
/repository_info	The description of the CMIS repository
/filesAndFolders	The structure of the folders and files in the projected repository

Node types used by connectors are specified by JCR specifications or imported from CMIS repository itself. Most important node types are as follows:

Node Type	Description
nt:folder	The primary node type for the node representing CMIS folder
nt:file	The primary node type for the node representing CMIS document
nt:resource	The primary node type for the node representing binary content of the CMIS document
cmis:repository	The primary node type for the node representing information of CMIS repository itself

Chapter 11. Built-in Text Extractors

The hierarchical database comes with a single text extractor. All you have to do is configure it and be ready to work with the generated output.

11.1. Tika Text Extractor

This text extractor uses the Tika library to extract text from a variety of file formats. It will automatically discover all of the Tika Parser implementations that are defined in **`META-INF/services/org.apache.tika.parser.Parser`** text files accessible via the current classloader and that contain the class names of the Parser implementations (one class name per line in each file). In other words, ensure that the Tika libraries for the appropriate file formats are on the classpath, and the text extractor will be able to use them all.

This text extractor can be configured in a hierarchical database configuration by specifying several optional properties:

- **`excludedMimeTypes`** - The comma- or whitespace-separated list of MIME types that should be excluded from text extraction, even if there is a Tika Parser available for that MIME type. By default, the MIME types for package files are excluded, though explicitly setting any excluded MIME types will override these default.
- **`includedMimeTypes`** - The comma- or whitespace-separated list of MIME types that should be included in text extraction. This extractor will ignore any MIME types in this list that are not covered by Tika Parser implementations.

To use this extractor, include the **`modeshape-extractor-tika`** JAR and the appropriate required Tika JARs are on the classpath (or via Maven) and configure the repository in a similar fashion to:

```
{
    "name" : "Sample Config",
    "query" : {
        "textExtracting": {
            "extractors" : {
                "tikaExtractor":{
                    "name" : "General content-based extractor",
                    "classname" : "tika",
                }
            }
         },
    }
}
```

Chapter 12. Monitoring

12.1. Public API

The hierarchical database now includes as part of its public API a set of interfaces that your application can use to monitor the activities and health of your repository. We did this because the standard JCR API does not cover monitoring at all, and we thought it is useful enough to make it available.

12.2. Metrics

The hierarchical database can capture a number of different measurements, called **metrics** , and these are broken into two categories: duration-based metrics (how long something takes) and simple value metrics.

Duration metrics are represented by the **`org.modeshape.jcr.api.monitor.DurationMetric`** enumeration, and include:

Metric	Description
Query execution time	The amount of time required to execute a query.
Session duration	The length of time that a session is used before being closed.
Sequencer duration	The length of time required to sequence a node, produce the output, and save the changes to the workspace.

Value metrics are represented by the **`org.modeshape.jcr.api.monitor.ValueMetric`** enumeration, and include:

Metric	Style	Description
Active sessions	continuous	The number of active sessions.
Active queries	continuous	The number of active queries.
Workspace count	continuous	The number of workspaces.
Session-scoped locks	continuous	The number of session-scoped locks held by clients.
Open-scoped locks	continuous	The number of open-scoped locks held by clients.
Listener count	continuous	The number of listeners registered with active sessions.
Event queue size	continuous	The number of events that are enqueued for processing and sending to listeners.
Event count	incremental	The number of events that have been sent to at least one listener.
Changed nodes	incremental	The number of nodes that were created, updated, or deleted.
Session saves	incremental	The number of **`Session.save()`** calls.
Sequencer queue size	continuous	The number of sequencing operations that are enqueued.
Sequenced nodes	incremental	The number of nodes sequenced.

Values for each of these metrics is captured every 5 seconds, where the continuous metrics are recorded as is (the values continue from one measurement to the next), while the incremental metrics represent distinct perturbations (or increments) from 0.

12.3. Windows and Statistics

As mentioned above, the hierarchical database measures the values for each metric every 5 seconds. But it would take vast amounts of space to keep all these measurements around for long periods of time. Instead, the hierarchical database calculates the **statistics** for various intervals, and then rolls up the statistics into different **time windows** .

The statistics are straightforward:

Statistic	Data type	Description
Count	int	The number of samples.
Maximum	long	The maximum value from the samples.
Minimum	long	The minimum value from the samples.
Mean	double	The mean (or average) value from the samples.
Variance	double	The average of the squared differences from the mean .
Standard Deviation	double	A measure of how spread out the samples are and is the square root of the variance

and are represented in the API by the **`org.modeshape.jcr.api.monitor.Statistics`** interface (with getter methods for each statistic). The statistics were chosen because multiple **`Statistics`** objects can easily be rolled-up into a single **`Statistic`** object.

The rollup process is pretty simple. For each metric:

- The value is captured every 5 seconds, and recorded a **`Statistics`** instance with a single sample. This is repeated 9 more times.
- After 60 seconds, the 10 **`Statistics`** objects recorded in the previous step are rolled-up into a single **`Statistics`** object for this minute. This is repeated 59 more times.
- After 60 minutes, the 60 **`Statistics`** objects recorded in the previous step are rolled-up into a single **`Statistics`** object for this hour. This is repeated 23 more times.
- After 24 hours, the 24 **`Statistics`** objects recorded in the previous step are rolled-up into a single **`Statistics`** object for the day. This is repeated 7 more times.
- After 7 days, the 7 **`Statistics`** objects recorded in the previous step are rolled-up into a single **`Statistics`** object for the week. This is repeated 52 more times.

Each of these periods represents a **window in time** during with the **`Statistics`** are captured:

Window timeframe	Description
60 seconds	A **`Statistics`** for each of the ten 5-second intervals during the last minute.
60 minutes	A **`Statistics`** for each minute during the last hour.

Window timeframe	Description
24 hours	A **`Statistics`** for each hour during the last day.
7 days	A **`Statistics`** for each day during the last week.
52 weeks	A **`Statistics`** for each week during the last year.

The **`org.modeshape.jcr.api.monitor.Window`** enumeration is used to represent each of these windows in time.

12.4. Histories

The set of **`Statistics`** objects for a particular metric during a **`Window`** is called the **history** of the metric, which is represented by the **`org.modeshape.jcr.api.monitor.History`** interface:

```
public interface History {

    /**
     * Get the kind of window.
     *
     * @return the window type; never null
     */
    public Window getWindow();

    /**
     * Get the total duration of this history window.
     *
     * @param unit the desired time unit; if null, then {@link
TimeUnit#SECONDS} is used
     * @return the duration
     */
    public long getTotalDuration( TimeUnit unit );

    /**
     * Get the timestamp (including time zone information) at which this
history window starts.
     *
     * @return the time at which this window starts
     */
    public DateTime getStartTime();

    /**
     * Get the timestamp (including time zone information) at which this
history window ends.
     *
     * @return the time at which this window ends
     */
    public DateTime getEndTime();

    /**
     * Get the statistics for that make up the history.
     *
     * @return the statistics; never null, but the array may contain null if
the window is
```

```
     *         longer than the lifetime of the repository
     */
    Statistics[] getStats();
}
```

Note

The **`org.modeshape.jcr.api.value.DateTime`** interface is an immutable representation of an instant in time. It includes timezone information and methods for converting or obtaining the various representations and/or parts of the instant. It is based upon initial work by the JSR-310 effort, and is far superior to the mutable and difficult-to-use **`java.util.Calendar`** class.

12.5. Repository Monitor

The **`org.modeshape.jcr.api.monitor.RepositoryMonitor`** interface can then be used to get the available metrics and windows, as well as obtaining the history for a given metric and window:

```
public interface RepositoryMonitor {

    /**
     * Get the ValueMetric enumerations that are available for use by the
caller
     * with {{getHistory(ValueMetric, Window)}}.
     *
     * @return the immutable set of ValueMetric instances; never null but
possibly
     *         empty if the caller has no permissions to see any value
metrics
     */
    Set<ValueMetric> getAvailableValueMetrics();

    /**
     * Get the DurationMetric enumerations that are available for use by the
caller
     * with {{getHistory(DurationMetric, Window)}}.
     *
     * @return the immutable set of DurationMetric instances; never null but
possibly
     *         empty if the caller has no permissions to see any value
metrics
     */
    Set<DurationMetric> getAvailableDurationMetrics();

    /**
     * Get the Window enumerations that are available for use by the caller
with
     * {{getHistory(DurationMetric, Window)}} and {{getHistory(ValueMetric,
Window)}}.
     *
     * @return the immutable set of DurationMetric instances; never null but
possibly
     *         empty if the caller has no permissions to see any value
```

```
metrics
     */
    Set<Window> getAvailableWindows();

    /**
     * Get the statics for the specified value metric during the given
window in time.
     * The oldest statistics will be first, while the newest statistics will
be last.
     *
     * @param metric the value metric; may not be null
     * @param windowInTime the window specifying which statistics are to be
returned;
     *        may not be null
     * @return the history of the metrics; never null but possibly empty if
there are
     *         no statistics being captures for this repository
     * @throws AccessDeniedException if the session does not have privileges
to monitor the repository
     * @throws RepositoryException if there is an error obtaining the
history
     */
    public History getHistory( ValueMetric metric,
                               Window windowInTime )
                   throws AccessDeniedException, RepositoryException;

    /**
     * Get the statics for the specified duration metric during the given
window in time.
     * The oldest statistics will be first, while the newest statistics will
be last.
     *
     * @param metric the duration metric; may not be null
     * @param windowInTime the window specifying which statistics are to be
returned;
     *        may not be null
     * @return the history of the metrics; never null but possibly empty if
there are
     *         no statistics being captures for this repository
     * @throws AccessDeniedException if the session does not have privileges
to monitor the repository
     * @throws RepositoryException if there is an error obtaining the
history
     */
    public History getHistory( DurationMetric metric,
                               Window windowInTime )
                   throws AccessDeniedException, RepositoryException;

    /**
     * Get the longest-running activities recorded for the specified metric.
     * The results contain the duration records in order of increasing
duration,
     * with the activity with the longest duration appearing last in the
array.
     *
     * @param metric the duration metric; may not be null
```

```
     * @return the activities with the longest durations; never null but
possibly
     *         empty if no such activities were performed
     * @throws AccessDeniedException if the session does not have privileges
to monitor the repository
     * @throws RepositoryException if there is an error obtaining the
history
     */
    public DurationActivity[] getLongestRunning( DurationMetric metric )
                  throws AccessDeniedException, RepositoryException;
```

And finally, your application can get the **`RepositoryMonitor`** instance from the Session's workspace, using the **`org.modeshape.jcr.api.Workspace`** interface that extends the standard **`javax.jcr.Workspace`** interface:

```
Session session = ...
org.modeshape.jcr.api.Workspace workspace =
(org.modeshape.jcr.api.Workspace)session.getWorkspace();
RepositoryMonitor monitor =
workspace.getRepositoryManager().getRepositoryMonitor();
```

12.6. Monitoring Examples

12.6.1. Active Sessions During the Last Hour

This example shows how to get the history containing the number of active sessions during each minute of the last hour:

```
RepositoryMonitor monitor =
workspace.getRepositoryManager().getRepositoryMonitor();
History history =
monitor.getHistory(ValueMetric.SESSION_COUNT,Window.PREVIOUS_60_MINUTES);

// Use the history information to build a graph and determine the axes labels
...
int duration = history.getTotalDuration(TimeUnit.MINUTES);  // will be '60'
DateTime started = history.getStartTime();
DateTime ended = history.getEndTime();
Statistics[] stats = history.getStats();  // will contain 60 elements
```

Here, each **`Statistics`** object represents the number of active sessions that existed during each minute. If, for example, all the sessions were closed in the second-to-last minute, then the second-to-last **`Statistics`** object will reflect some of them closing, while the first **`Statistics`** object will have average, maximum, and minimum values of 0.

12.6.2. Query Durations During the Last Day

This example shows how to obtain the statistics for the durations of queries executed during the last 24 hours:

```
RepositoryMonitor monitor =
workspace.getRepositoryManager().getRepositoryMonitor();
History history =
```

```
monitor.getHistory(DurationMetric.QUERY_EXECUTION_TIME,Window.PREVIOUS_24_HO
URS);

// Use the history information to build a graph and determine the axes labels
...
int duration = history.getTotalDuration(TimeUnit.MINUTES);  // will be
'1440' (or 24 x 60 )
DateTime started = history.getStartTime();
DateTime ended = history.getEndTime();
Statistics[] stats = history.getStats();  // will contain 24 elements
```

Each **`Statistics`** object will represent the number, average, maximum, minimum, variance, and standard deviation for the queries that were executed during an hour of the last 24 hours.

12.6.3. Worst Performing Queries During the Last Day

In the same way that we can obtain the statistics for the queries that were submitted during the last 24 hours, we can also obtain information about the longest-running queries:

```
RepositoryMonitor monitor =
workspace.getRepositoryManager().getRepositoryMonitor();
// Get the 'DurationActivity' object for each long-running query, where the
longest is last ...
DurationActivity[] longestQueries =
monitor.getLongestRunning(DurationMetric.QUERY_EXECUTION_TIME);

for ( DurationActivity queryActivity : longestQueries ) {
    long duration = queryActivity.getDuration(TimeUnit.MILLISECONDS);
    Map<String,String> payload = queryActivity.getPayload();
    String query = payload.get("query");
}
```

12.6.4. Event Queue Backlog During the Last Hour

This example shows how to get the history containing the number of events in the event queue during each minute of the last hour:

```
RepositoryMonitor monitor =
workspace.getRepositoryManager().getRepositoryMonitor();
History history =
monitor.getHistory(ValueMetric.EVENT_QUEUE_SIZE,Window.PREVIOUS_60_MINUTES);

// Use the history information to build a graph and determine the axes labels
...
int duration = history.getTotalDuration(TimeUnit.MINUTES);  // will be '60'
DateTime started = history.getStartTime();
DateTime ended = history.getEndTime();
Statistics[] stats = history.getStats();  // will contain 60 elements
```

Here, each **`Statistics`** object represents the number of events that are in the queue during each minute. If, for example, the number of events is increasing during each minute, then the hierarchical database is falling behind in notifying the listeners. This likely will happen when sessions are making frequent changes, while registered listeners are taking too long to process the event.

Note

Listeners are not supposed to take too long to process the event, since one thread is being used to notify all listeners. If your listeners are taking too long, consider managing the queuing in a separate **`java.util.concurrent.Executor`**, where the actual work is performed on separate threads.

Also, be careful if the listener looks up content using a session. Generally speaking, it is not good practice for a listener to reuse the same session on which it is registered, since all listeners will share the same session. The hierarchical database is thread-safe, but any changes made by one listener will be visible to other listeners.

Chapter 13. Backup and Restore

13.1. Backup and Restore Overview

Backup and restore functions are provided that enable repository administrators to create backups of an entire repository (even when the repository is in use), and to then restore a repository to the state reflected by a particular backup. This works regardless of where the repository content is persisted.

There are several reasons why you might want to restore a repository to a previous state:

Failure

For example, the application or the process it is running in might stop unexpectedly, the hardware on which the process is running might fail, or the persistent store might have a catastrophic failure (although this would most likely be backed up already).

Transfer

For example, backups of a running repository can be used to transfer content to a new repository hosted in a different location. It might be possible to manually transfer the persisted content (for example, in a database or on the file system), but the process of doing so varies for different persistence options.

Ease of Access

For example, the hierarchical database can be configured to use a distributed in-memory data grid that already maintains its own copies for ensuring high availability, and therefore the data grid might not persist anything to disk. In such cases, content is stored on the data grid's virtual heap, and getting access to it without the hierarchical database may be difficult.

Configuration Change

For example, you may initially configure your repository to use a particular persistence approach but, over time as the repository grows, you want to move to a different, more scalable (but perhaps more complex) persistence approach.

Migration

Finally, the backup and restore feature can be used to migrate to a new major version of the hierarchical database.

13.2. Migrating from a Previous Release

Backup and restore can be used to migrate content stored in the provided hierarchical database system.

This is the proposed way for users to migrate such data for release 6.

Note

When migrating from a version of JBoss Data Virtualization prior to version 6.0.0, a migration tool will be provided to backup the repository from the previous installation. The backup can be restored using the `restoreRepository` method on the new (and empty) repository.

13.3. The Repository Manager

13.3. The Repository Manager

The org.modeshape.jcr.api.RepositoryManager interface contains the backup and restore functions.

```
public interface RepositoryManager {

    ...

    Problems backupRepository( File backupDirectory ) throws
RepositoryException;

    ...

    Problems restoreRepository( File backupDirectory ) throws
RepositoryException;

    ...

}
```

The following code demonstrates how to access the repository manager from a standard authenticated JCR session:

```
javax.jcr.Repository repository = ...
javax.jcr.Credentials credentials = ...
String workspaceName = ...
javax.jcr.Session session = repository.login(credentials,workspaceName);
org.modeshape.jcr.api.Session msSession =
(org.modeshape.jcr.api.Session)session;
org.modeshape.jcr.api.RepositoryManager repoMgr =
((org.modeshape.jcr.api.Session)session).getWorkspace().getRepositoryManager
();
```

Note

Which workspace is used by the session is not important.

13.4. Backup a Repository

The **`backupRepository`** method provided by the org.modeshape.jcr.api.RepositoryManager interface is used to create a backup of the entire repository, including all workspaces that existed when the backup was initiated.

This method blocks until the backup is completed, so it is the caller's responsibility to invoke the method asynchronously if that is desired.

When this method is called on a repository that is being actively used, all of the changes made while the backup process is underway will be included. At some point near the end of the backup process, however, additional changes will be excluded from the backup. This means that each backup contains a fully-consistent snapshot of the entire repository as it existed near the time at which the backup completed.

The following code demonstrates usage of the backup method:

```
org.modeshape.jcr.api.RepositoryManager repoMgr = ...
java.io.File backupDirectory = ...
Problems problems = repoMgr.backupRepository(backupDirectory);
if ( problems.hasProblems() ) {
    System.out.println("Problems restoring the repository:");
    // Report the problems (we'll just print them out) ...
    for ( Problem problem : problems ) {
       System.out.println(problem);
    }
} else {
    System.out.println("The backup was successful");
}
```

Each backup is stored on the file system in a directory that contains a series of GZIP-ed files (each containing representations of approximately 100K nodes) and a subdirectory in which all the large BINARY values are stored.

Note

It is the application's responsibility to initiate each backup operation. There currently is no way to configure a scheduled backup. Doing so would add significant complexity.

13.5. Restore a Repository

Once you have a complete backup on disk, you can then restore a repository back to the state captured within the backup using the **`restoreRepository`** method provided by the org.modeshape.jcr.api.RepositoryManager interface.

To do this, start a repository (or perhaps a new instance of a repository with a different configuration) and, before it is used by any applications, restore the content.

The following code demonstrates usage of the restore method:

```
org.modeshape.jcr.api.RepositoryManager repoMgr = ...
java.io.File backupDirectory = ...
Problems problems = repoMgr.restoreRepository(backupDirectory);
if ( problems.hasProblems() ) {
    System.out.println("Problems backing up the repository:");
    // Report the problems (we'll just print them out) ...
    for ( Problem problem : problems ) {
         System.out.println(problem);
    }
} else {
    System.out.println("The restoration was successful");
}
```

Once a restore succeeds, the newly-restored repository will be restarted and ready for use.

Chapter 14. Security

The hierarchical database delegates all authentication and authorization to the providers with which a repository is configured. The hierarchical database includes a few providers, but it is also possible to create custom authentication and/or authorization providers.

One exception is the access control feature, new in this latest release, that provides a way to use the standard JCR API to define node-level access control lists (ACLs) that augment the normal authorization mechanism. These fine-grained access controls are handled entirely within the hierarchical database, stored within the normal repository content, and built on top of the existing authentication and authorization providers.

14.1. Authentication and Authorization

In order to create a **`Session`** , a client application must authenticate their identity by logging in and providing a **`javax.jcr.Credential`** . The hierarchical database passes this credential to a series of **`AuthenticationProvider`** components. The first provider to accept the credential will result in the hierarchical database authenticating the caller and returning a valid **`Session`** .

The authorizing provider, as part of the authentication step, returns an internal **`SecurityContext`** that is associated with that session. This **`SecurityContext`** is then used to determine whether the session is authorized to read, write, or administer the repository. These are coarse-grained roles that apply to all content; for example, if a session only has the read role, then it can read all repository content but can write or administer no content.

The names of the three roles are **`readonly`**, **`readwrite`**, and **`admin`**.

See Also:

- Section 15.1, “Custom Authentication and Authorization Modules”

14.2. Anonymous Sessions

The hierarchical database does make it possible for clients to create anonymous sessions. These are never authenticated, and they are generally given only the **`readonly`** role. Of course, you can choose to configure anonymous sessions to use any of the three roles, though be careful granting more than **`readonly`**.

When a client attempts to authenticate normally by supplying credentials, should that authentication fail, the repository can do one of two things:

- fail by throwing an exception
- return an anonymous session

This is often useful in applications that want to always provide at least some read-only functionality for all users.

14.3. JAAS

The **`org.modeshape.jcr.security.JaasProvider`** class is configured to use a specific JAAS policy to perform all authentication and role-based authorization. This is the easiest to use, since most application servers will come with JAAS support and even Java SE applications can pretty easily set up one of the available JAAS implementations.

Note

If no providers are explicitly configured, the JAAS provider is automatically enabled with the "modeshape-jcr" policy.

14.4. JAAS Configuration

Each JAAS implementation will be configured differently. In the case of the PicketBox implementation, configuration is done via a **`jaas.conf.xml`** file on the classpath. There are quite a few modules to choose from, including LDAP, database, XACML, and even a simple file-based option. Here is an example of a **`jaas.conf.xml`** file that uses the users and roles defined in local files:

```
<?xml version='1.0'?>
<policy xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:jboss:security-config:5.0"
xmlns="urn:jboss:security-config:5.0">
        <application-policy name="modeshape-jcr">
                <authentication>
                        <login-module
code="org.jboss.security.auth.spi.UsersRolesLoginModule" flag="required">
                <module-option
name="usersProperties">security/users.properties</module-option>
                <module-option
name="rolesProperties">security/roles.properties</module-option>
                        </login-module>
                </authentication>
        </application-policy>
</policy>
```

This file sets up a JAAS policy named **`modeshape-jcr`** that uses the User-Roles Login Module, and defines the users and passwords in the **`security/users.properties`** file and the roles in the **`security/roles.properties`** file.

The users file contains a line for each user, of the form **`username=password`**. The roles file also contains a line for each user, but this format is a little more complicated:

```
{{<username>=<role>\[,<role>,...\]}}
```

where:

- **`<username>`** is the name of the user,
- **`<role>`** is an expression describing a role for the user and which adheres to the format**`<role>=<roleName>[.<workspaceName]`**, where:
  - **`<roleName>`** is one of **admin**, **readonly**, **readwrite**, or (for WebDAV and RESTful access) **connect**
  - **`<workspaceName>`** is the name of the repository workspace to which the role is granted; if absent, the role will be granted for all workspaces in the repository

For example, the following line provides all roles to user 'jsmith' for all workspaces in the configured repository:

```
jsmith=admin,connect,readonly,readwrite
```

while

```
jsmith=connect,readonly,readwrite.ws1
```

provides connect and read access to all workspaces, but only write access to the **`ws1`** workspace.

14.5. Servlet Authentication

You can configure a repository to this provider, and then have your applications create a **`org.modeshape.jcr.api.ServletCredentials`** instance with the servlet's **`HttpServletRequest`** . The hierarchical database will then delegate all authentication and role-based authorization to the servlet container. Again, the roles are expected to be **`readonly`**, **`readwrite`** and **`admin`**.

Note

If no providers are explicitly configured, the Servlet provider is automatically enabled if the servlet API is on the classpath.

14.6. Access Controls

Recall that the aforementioned role-based authorizations apply to a whole repository or workspace, and thus are referred to as coarse-grained authorization. This simple approach is perfectly acceptable for many applications. However, with the latest release, it is possible to use fine-grained authorization to determine what operations are allowed on specific nodes or subtrees. The API to set up and manage these fine-grained permissions and access control lists is actually part of the standard JCR 2.0 API.

Note that an authenticated user must have already be granted the coarse-grained roles for a repository before any fine-grained access controls are even evaluated. This means that, for example, even if an authenticated user is granted a privilege to modify the properties of a node, that means nothing unless the user has one of the roles that allows writing or changing content. In other words, when using fine-grained access controls, the hierarchical database will require that both the coarse-grained and fine-grained authorizations allow the requested action.

14.7. Privileges

The JCR 2.0 API defines the following privileges:

Privilege	Description
`jcr:read`	The privilege to retrieve a node and get its properties and their values.
`jcr:modifyProperties`	The privilege to create, remove and modify the values of the properties of a node.
`jcr:addChildNodes`	The privilege to create child nodes of a node.
`jcr:removeNode`	The privilege to remove a node.

Privilege	Description
jcr:removeChildNodes	The privilege to remove child nodes of a node. In order to actually remove a node requires **jcr:removeNode** on that node and **jcr:removeChildNodes** on the parent node.
jcr:write	An aggregate privilege that contains: **jcr:modifyProperties** , **jcr:addChildNodes** , **jcr:removeNode** , and **jcr:removeChildNodes** .
jcr:readAccessControl	The privilege to read the access control settings of a node.
jcr:modifyAccessControl	The privilege to modify the access control settings of a node.
jcr:lockManagement	The privilege to lock and unlock a node.
jcr:versionManagement	The privilege to perform versioning operations on a node.
jcr:nodeTypeManagement	The privilege to add and remove mixin node types and change the primary node type of a node.
jcr:retentionManagement	The privilege to perform retention management operations on a node.
jcr:lifecycleManagement	The privilege to perform lifecycle operations on a node.
jcr:all	An aggregate privilege that contains: **jcr:read** , **jcr:write** , **jcr:readAccessControl** , **jcr:modifyAccessControl** , **jcr:lockManagement** , **jcr:versionManagement** , **jcr:nodeTypeManagement** , **jcr:retentionManagement** , and **jcr:lifecycleManagement**

See the **javax.jcr.security.AccessControlManager** API for methods to determine the privileges supported by the repository on any given node and for manually determining whether the session has particular privileges on any given node.

14.8. Principals

Privileges are assigned to specific principals , which can either represent usernames or the names of groups. A principal is represented in the API via the **javax.security.Principal** , which can be any implementation. (The hierarchical database primarily uses the principal's name.)

An authenticated user is considered a member of a group if the **AuthorizationProvider** or **AdvancedAuthorizationProvider** implementations return **true** for **hasRole(groupName)** .

14.9. Access Control Policies

The privileges granted to a user can be controlled by assigning an access control policy to nodes. Before the access to a node can be controlled, however, it must have the **mode:accessControllable** mixin. Each such node has one or more access control policies to which additional access control entries (e.g., a principal-permissions pair) can be added.

For example, the following code fragment shows how to define an access control policy on a specific node

(and its descendants):

```
String path = "/Cars/Luxury";
String[] privileges = new String[]{Privilege.JCR_READ, Privilege.JCR_WRITE,
Privilege.JCR_MODIFY_ACCESS_CONTROL};
Principal principal = ...  /* any implementation, referring to a username or
group name */
Session session = ...
AccessControlManager acm = session.getAccessControlManager();

// Convert the privilege strings to Privilege instances ...
Privilege[] permissions = new Privilege[privileges.length];
for (int i = 0; i < privileges.length; i++) {
    permissions[i] = acm.privilegeFromName(privileges[i]);
}

AccessControlList acl = null;
AccessControlPolicyIterator it = acm.getApplicablePolicies(path);
if (it.hasNext()) {
    acl = (AccessControlList)it.nextAccessControlPolicy();
} else {
    acl = (AccessControlList)acm.getPolicies(path)[0];
}
acl.addAccessControlEntry(principal, permissions);

acm.setPolicy(path, acl);
session.save();
```

From this point on, when a session is created by authenticating as a user with the supplied principal (e.g., username or group membership), then that session will be allowed to read, write and modify access controls on the **`/Cars/Luxury`** node or its descendants (unless otherwise restricted with access controls). Again, this presume that the authentication session already has the coarse-grained roles for reading and writing content in this particular workspace.

Note

Creating an access control entry for a principal that does not exist is not useful, but it is not dangerous, either. Evaluation of access controls requires that the entry match the current session's username or roles (for groups); other principals are never considered.

See the **`javax.jcr.security.AccessControlManager`** API and the JSR-283 for more information about defining and using access control policies.

Chapter 15. Extending the Hierarchical Database

You can create customized extensions to the hierarchical database. To do so you will need to write Java code, and to use Maven 3 for the build system.

15.1. Custom Authentication and Authorization Modules

15.1.1. The AuthenticationProvider Interface

The hierarchical database defines a simple interface for authenticating users. Each repository can have multiple authentication modules, and a client is authenticated as soon as one of the modules accepts the credentials. The interface is quite simple:

```
/**
 * An interface used by a ModeShape Repository for authenticating users when
they create new sessions
 * using Repository.login(javax.jcr.Credentials, String)} and related
methods.
 */
public interface AuthenticationProvider {

    /**
     * Authenticate the user that is using the supplied credentials. If the
supplied credentials are authenticated, this
     * method should construct an ExecutionContext that reflects the
authenticated environment, including the context's
     * valid SecurityContext security context that will be used for
authorization throughout.
     *
     * Note that each provider is handed a map into which it can place name-
value pairs that will be used in the
     * Session attributes of the Session that results from this
authentication attempt.
     * ModeShape will ignore any attributes if this provider does not
authenticate the credentials.
     *
     *
     * @param credentials the user's JCR credentials, which may be an
AnonymousCredentials if authenticating as an
     *        anonymous user
     * @param repositoryName the name of the JCR repository; never null
     * @param workspaceName the name of the JCR workspace; never null
     * @param repositoryContext the execution context of the repository,
which may be wrapped by this method
     * @param sessionAttributes the map of name-value pairs that will be
placed into the Session attributes; never null
     * @return the execution context for the authenticated user, or null if
this provider could not authenticate the user
     */
    ExecutionContext authenticate( Credentials credentials,
                                   String repositoryName,
                                   String workspaceName,
                                   ExecutionContext repositoryContext,
```

```
                                        Map<String, Object> sessionAttributes
);

}
```

All the parameters are supplied by the hierarchical database and contain everything necessary to authenticate a client attempting to create a new JCR Session.

Implementations are expected to return a new **`ExecutionContext`** instance for the user, and this can be created from the repository's execution context by calling **`repositoryContext.with(securityContext)`** , where **`securityContext`** is a custom implementation of the **`org.modeshape.jcr.security.SecurityContext`** interface that returns information about the authenticated user:

```
/**
 * A security context provides a pluggable means to support disparate
authentication and authorization mechanisms that specify the
 * user name and roles.
 *
 * A security context should only be associated with the execution context
after authentication has occurred.
 */
@NotThreadSafe
public interface SecurityContext {

    /**
     * Return whether this security context is an anonymous context.
     * @return true if this context represents an anonymous user, or false
otherwise
     */
    boolean isAnonymous();

    /**
     * Returns the authenticated user's name
     * @return the authenticated user's name
     */
    String getUserName();

    /**
     * Returns whether the authenticated user has the given role.
     * @param roleName the name of the role to check
     * @return true if the user has the role and is logged in; false
otherwise
     */
    boolean hasRole( String roleName );

    /**
     * Logs the user out of the authentication mechanism.
     * For some authentication mechanisms, this will be implemented as a no-
op.
     */
    void logout();
}
```

Note that if you want to provide authorization functionality, then your **`SecurityContext`** implementation must also implement **`AuthorizationProvider`** or **`AdvancedAuthorizationProvider`** .

15.1.2. The AuthorizationProvider Interface

The hierarchical database uses its **`org.modeshape.jcr.security.AuthorizationProvider`** interface to determine whether a Session has the appropriate privileges to perform reads and writes.

```
/**
 * An interface that can authorize access to specific resources within
repositories.
 */
public interface AuthorizationProvider {

    /**
     * Determine if the supplied execution context has permission for all of
the named actions in the named workspace.
     * If not all actions are allowed, the method returns false.
     *
     * @param context the context in which the subject is performing the
actions on the supplied workspace
     * @param repositoryName the name of the repository containing the
workspace content
     * @param repositorySourceName <i>This is no longer used and will always
be the same as the repositoryName</i>
     * @param workspaceName the name of the workspace in which the path
exists
     * @param path the path on which the actions are occurring
     * @param actions the list of {@link ModeShapePermissions actions} to
check
     * @return true if the subject has privilege to perform all of the named
actions on the content at the supplied
     *         path in the given workspace within the repository, or false
otherwise
     */
    boolean hasPermission( ExecutionContext context,
                           String repositoryName,
                           String repositorySourceName,
                           String workspaceName,
                           Path path,
                           String... actions );
}
```

You can have your **`SecurityContext`** implementation also implement this interface, and return **`true`** whenever the session is allows to perform the requested operations.

15.1.3. The AdvancedAuthorizationProvider Interface

The hierarchical database uses its **`org.modeshape.jcr.security.AdvancedAuthorizationProvider`** interface to determine whether a Session has the appropriate privileges to perform reads and writes.

```
/**
 * An interface that can authorize access to specific resources within
repositories. Unlike the more basic and simpl
 * AuthenticationProvider, this interface allows an implementation to get at
additional information with each call to
 * hasPermission(Context, Path, String...).
```

```
 *
 * In particular, the supplied Context instance contains the Session that is
calling this provider, allowing the
 * provider implementation to access authorization-specific content within
the repository to determine permissions for other
 * repository content.
 *
 * In these cases, calls to the session to access nodes will result in their
own calls to hasPermission(Context, Path, String...).
 * Therefore, such implementations need to handle these special
authorization-specific content permissions in an explicit fashion.
 * It is also adviced that such providers cache as much of the
authorization-specifc content as possible, as the
 * hasPermission(Context, Path, String...) method is called frequently.
 */
public interface AdvancedAuthorizationProvider {

    /**
     * Determine if the supplied execution context has permission for all of
the named actions in the given context. If not all
     * actions are allowed, the method returns false.
     *
     * @param context the context in which the subject is performing the
actions on the supplied workspace
     * @param absPath the absolute path on which the actions are occurring,
or null if the permissions are at the workspace-level
     * @param actions the list of {@link ModeShapePermissions actions} to
check
     * @return true if the subject has privilege to perform all of the named
actions on the content at the supplied path in the
     *         given workspace within the repository, or false otherwise
     */
    boolean hasPermission( Context context,
                           Path absPath,
                           String... actions );
}
```

where **`Context`** is a new nested interface nested in **`AdvancedAuthorizationProvider`** :

```
    /**
     * The context in which the calling session is operating, and which
contains session-related information that a provider
     * implementation may find useful.
     */
    public static interface Context {
        /**
         * Get the execution context in which this session is running.
         *
         * @return the session's execution context; never null
         */
        public ExecutionContext getExecutionContext();

        /**
         * Get the session that is requesting this authorization provider to
         * {@link AdvancedAuthorizationProvider#hasPermission(Context, Path,
String...) determine permissions}. Provider
```

```
         * implementations are free to use the session to access nodes
<i>other</i> than those for which permissions are being
         * determined. For example, the implementation may access other
<i>authorization-related content</i> inside the same
         * repository. Just be aware that such accesses will generate
additional calls to the
         * {@link AdvancedAuthorizationProvider#hasPermission(Context, Path,
String...)} method.
         *
         * @return the session; never null
         */
        public Session getSession();

        /**
         * Get the name of the repository that is being accessed.
         *
         * @return the repository name; never null
         */
        public String getRepositoryName();

        /**
         * Get the name of the repository workspace that is being accessed.
         *
         * @return the workspace name; never null
         */
        public String getWorkspaceName();
    }
```

You can have your **`SecurityContext`** implementation also implement this interface, and return **`true`** whenever the session is allows to perform the requested operations.

15.1.4. Configure a Repository to Use Your Custom Modules

Once you've implemented the interfaces and placed the classes on the classpath, all you have to do is then configure your repositories to use your modules. As noted in the "configuration overview" , there is a nested document in the JSON configuration file in the **`security`** field, and this section lists the authentication module implementations in the order that they should be used . For example:

```
    ...
    "security" : {
        "anonymous" : {
            "username" : "<anonymous>",
            "roles" : ["readonly","readwrite","admin"],
            "useOnFailedLogin" : false
        },
        "providers" : [
            {
                "name" : "My Custom Security Provider",
                "classname" : "com.example.MyAuthenticationProvider",
            },
            {
                "classname" : "JAAS",
                "policyName" : "modeshape-jcr",
```

```
            }
        ]
    },
    ...
```

This configuration enables the use of anonymous logins (although it disables a failed authentication attempt from downgrading to an anonymous session with the **`useOnFailedLogin`** as **`false`**), and configures two authentication providers: the **`MyAuthenticationProvider`** implementation will be used first, and if that does not authenticate the repository will delegate to the built-in JAAS provider. (Note that built-in providers can be referenced with an alias in the **`classname`** field rather than the fully-qualified classname.) Anonymous authentication is always performed last.

15.2. Custom Sequencers

15.2.1. The Sequencer Framework

A sequencer is actually a plain old Java object (POJO). To create a sequencer, create a Java class that extends a single abstract class, called **`Sequencer`** :

```
package org.modeshape.jcr.api.sequencer;

import javax.jcr.Node;
import javax.jcr.Property;
import javax.jcr.RepositoryException;

public abstract class Sequencer {

    ...

    /**
     * Execute the sequencing operation on the specified property, which has
recently
     * been created or changed.
     *
     * Each sequencer is expected to process the value of the property,
extract information
     * from the value, and write a structured representation (in the form of
a node or a
     * subgraph of nodes) using the supplied output node. Note that the
output node
     * will either be:
     *   1. the selected node, in which case the sequencer was configured to
generate the
     *      output information directly under the selected input node; or
     *   2. a newly created node in a different location than node being
sequenced (in
     *      this case, the primary type of the new node will be
'nt:unstructured', but
     *      the sequencer can easily change that using
Node.setPrimaryType(String) ).
     *
     * The implementation is expected to always clean up all resources that
it acquired,
     * even in the case of exceptions.
```

```
     *
     * @param inputProperty the property that was changed and that should be
used as
     *        the input; never null
     * @param outputNode the node that represents the output for the derived
information;
     *        never null, and will either be a new node if the output is
being placed
     *        outside of the selected node, or will not be new when the
output is to be
     *        placed on the selected input node
     * @param context the context in which this sequencer is executing, and
which may
     *        contain additional parameters useful when generating the
output structure; never null
     * @return true if the sequencer's output should be saved, or false
otherwise
     * @throws Exception if there was a problem with the sequencer that
could not be handled.
     *         All exceptions will be logged automatically as errors by
ModeShape.
     */
    public abstract boolean execute( Property inputProperty,
                                     Node outputNode,
                                     Context context ) throws Exception;

    /**
     * Initialize the sequencer. This is called automatically by ModeShape,
and
     * should not be called by the sequencer.
     * <p>
     * By default this method does nothing, so it should be overridden by
     * implementations to do a one-time initialization of any internal
components.
     * For example, sequencers can use the supplied 'registry' and
     * 'nodeTypeManager' objects to register custom namesapces and node
types
     * required by the generated content.
     * </p>
     *
     * @param registry the namespace registry that can be used to register
     * custom namespaces; never null
     * @param nodeTypeManager the node type manager that can be used to
register
     * custom node types; never null
     */
    public void initialize( NamespaceRegistry registry,
                            NodeTypeManager nodeTypeManager ) {
    }

}
```

The abstract class also contains fields and getters (not shown above) for the name, description, and path expressions that are automatically set by the hierarchical database during repository initialization. The **`initialize(...)`** method is run upon repository initialization and can be overridden by an implementation to register (if required) any custom namespaces and node types required by the sequencer's generated

output.

Note

The **outputNode** might belong to a different **javax.jcr.Session** object than the **inputProperty** , if the input and output paths of the sequencer configuration specify different workspaces. Therefore, be careful that all changes are made using the output node and its session.

The inputs to the sequencer depend on how it is configured, but often the **inputProperty** represents the **jcr:data** BINARY property on the **jcr:content** child of an **nt:file** node. The **outputNode** , however, will be one of two things:

1. If there is no output path in the path expression, then the sequenced output is to be placed directly under the selected node, so therefore the **outputNode** will be the existing node being sequenced. In this case, the sequencer should place all content under the output node. In this case, the sequencers are not allowed to change the primary type.
2. Otherwise, the sequenced output is to be placed in a different location than the selected node. In this case, the hierarchical database uses the name of the selected node and creates a new node under the output path. This new node will have a primary type of **nt:unstructured**, but sequencers are allowed to change the primary type.

The final parameter to the **execute(...)** method is the **SequencerContext** object, which is an interface containing some extra information often useful when sequencing files:

```
package org.modeshape.jcr.api.sequencer;

import java.util.Calendar;

/**
 * The sequencer context represents the complete context of a sequencer
invocation.
 * Currently, this information includes the current time of execution.
 */
public interface SequencerContext {

    /**
     * Get the timestamp of the sequencing. This is always the timestamp of
the
     * change event that is being processed.
     *
     * @return timestamp the "current" timestamp; never null
     */
    Calendar getTimestamp();

}
```

15.3. Custom Text Extractors

15.3.1. The Text Extraction Framework

A text extractor is actually a plain old Java object (POJO). To create an extractor, you create a Java class that extends a single abstract class, called **`TextExtractor`** :

```
package org.modeshape.jcr.api.text;

import javax.jcr.Node;
import javax.jcr.Property;
import javax.jcr.RepositoryException;

public abstract class TextExtractor {

    ...

    /**
     * Determine if this extractor is capable of processing content with the
supplied MIME type.
     * @param mimeType the MIME type; never null
     * @return true if this extractor can process content with the supplied
MIME type, or false otherwise.
     */
    public abstract boolean supportsMimeType( String mimeType );

    /**
     * Extract text from the given {@link Binary}, using the given output to
record the results.
     * @param binary the binary value that can be used in the extraction
process; never <code>null</code>
     * @param output the output from the sequencing operation; never
<code>null</code>
     * @param context the context for the sequencing operation; never
<code>null</code>
     * @throws Exception if there is a problem during the extraction process
     */
    public abstract void extractFrom( Binary binary,
                                      TextExtractor.Output output,
                                      Context context ) throws Exception;

    /**
     * Allows subclasses to process the stream of binary value property in
"safe" fashion, making sure the stream is closed at the
     * end of the operation.
     * @param binary a {@link org.modeshape.jcr.api.Binary} who is expected
to contain a non-null binary value.
     * @param operation a {@link
org.modeshape.jcr.api.text.TextExtractor.BinaryOperation} which should work
with the stream
     * @param <T> the return type of the binary operation
     * @return whatever type of result the stream operation returns
     * @throws Exception if there is an error processing the stream
     */
    protected final <T> T processStream( Binary binary,
                                         BinaryOperation<T> operation )
throws Exception {
        ...
    }
```

```
    /**
     * Interface which can be used by subclasses to process the input stream
of a binary property.
     * @param <T> the return type of the binary operation
     */
    protected interface BinaryOperation<T> {
        T execute( InputStream stream ) throws Exception;
    }

    /**
     * Interface which provides additional information to the text
extractors, during the extraction operation.
     */
    public interface Context {
        String mimeTypeOf( String name,
                           Binary binaryValue ) throws RepositoryException,
IOException;
    }

    /**
     * The interface passed to a TextExtractor to which the extractor should
record all text content.
     */
    public interface Output {
        /**
         * Record the text as being extracted. This method can be called
multiple times during a single extract.
         * @param text the text extracted from the content.
         */
        void recordText( String text );
    }
}
```

The abstract class also contains fields and getters (not shown above) for the name and logger that are automatically set by the hierarchical database during repository initialization.

There are two abstract methods that must be implemented: **`supportsMimeType(...)`** and **`extractFrom(...)`** . The first is fairly obvious: return true for all of the MIME types for which the extractor is capable of processing. The **`extractFrom`** method is the meat of the implementation, and should process the BINARY value's contents and write the searchable text to the supplied **`Output`** object.

Note that the **`processStream(...)`** method is a utility that can be called by the **`extractFrom`** and that properly opens the BINARY value's stream, processes the content, and ensures that the stream is always closed. Your implementation can therefore implement the **`extractFrom`** method as follows:

```
public void extractFrom( final Binary binary,
                         final TextExtractor.Output output,
                         final Context context ) throws Exception {
    processStream(binary, new BinaryOperation<Object>() {
        @Override
        public Object execute( InputStream stream ) throws Exception {
            // Custom logic to read the stream and write to 'output'
            return null;
        }
    });
}
```

This can make your implementation a little easier, but feel free to implement the **`extractFrom`** method directly process the stream.

15.4. Custom Connectors

15.4.1. The Connector Framework

A connector is actually a plain old Java object (POJO). To create a connector, create a Java class that extends one of the following abstract classes:

- **`ReadOnlyConnector`** - extend this class when the hierarchical database clients will never be able to manipulate, create or remove any content exposed by the connector.
- **`WritableConnector`** - extend this class when the hierarchical database clients may be able to manipulate, create and/or remove content exposed by the connector. Note that each time this connector is configured, it can still be made to be read-only.

A connector operates by accessing an external system and dynamically creating nodes that represent information in that external system. The nodes must form a single tree, although how that tree is structured and what the nodes actually look like is completely up to the connector implementation.

15.4.2. Documents

While a connector conceptually exposes nodes, technically it exchanges representations of nodes (and other information, like sublists of children). These representations take the form of Java Document objects that are semantically like JSON and BSON documents. The connector SPI does this for a number of reasons:

- Firstly, the hierarchical database actually stores its own internal (non-federated) nodes as Documents, so connectors are actually working with the same kind of internal Document instances that the hierarchical database uses.
- Secondly, a Document is easily converted to and from JSON (and BSON), making it potentially very easy to write a connector that accesses a remote system.
- Thirdly, constructs other than nodes can be represented as documents; for example, a connector can be pageable, meaning it breaks the list of child node references into multiple pages that are read with separate requests, allowing the connector to efficiently expose large numbers of children under a single node.
- Finally, the node's identifier, properties, child node references, and other information specific to the hierarchical database are stored in specific fields within a Document, but additional fields can be used by the connector and hidden from hierarchical database clients. Though this makes little sense for a read-only connector, a writable connector might include such hidden fields when reading nodes so that when the document comes back to the connector those hidden fields are still available.

Before studying the Documents you shall study the methods your connector implementation will need to implement.

15.4.3. Read Only Connector

The following code fragment shows the methods that a **`ReadOnlyConnector`** subclass must implement.

```
package org.modeshape.jcr.federation.spi;

```

```
import java.io.IOException;
import java.util.Collection;
import javax.jcr.NamespaceRegistry;
import javax.jcr.RepositoryException;
import org.infinispan.schematic.document.Document;
import org.modeshape.jcr.api.nodetype.NodeTypeManager;

public abstract class ReadOnlyConnector extends Connector {

    ...

   /**
     * Initialize the connector. This is called automatically by ModeShape
once for each Connector instance,
     * and should not be called by the connector. By the time this method is
called, ModeShape will have
     * already set the {{ExecutionContext}}, {{Logger}}, connector name,
repository name {@link #context},
     * and any fields that match configuration properties for the connector.
     *
     * By default this method does nothing, so it should be overridden by
implementations to do a one-time
     * initialization of any internal components. For example, connectors
can use the supplied {{registry}}
     * and {{nodeTypeManager}} parameters to register custom namespaces and
node types used by the exposed nodes.
     *
     * This is also an excellent place for connector to validate the
connector-specific fields set by ModeShape
     * via reflection during instantiation.
     *
     * @param registry the namespace registry that can be used to register
custom namespaces; never null
     * @param nodeTypeManager the node type manager that can be used to
register custom node types; never null
     * @throws RepositoryException if operations on the {@link
NamespaceRegistry} or {@link NodeTypeManager} fail
     * @throws IOException if any stream based operations fail (like
importing cnd files)
     */
    public void initialize( NamespaceRegistry registry,
                            NodeTypeManager nodeTypeManager ) throws
RepositoryException, IOException {
    }

    /**
     * Returns the id of an external node located at the given external path
within the connector's
     * exposed tree of content.
     *
     * @param externalPath a non-null string representing an external path,
or "/" for the top-level
     *        node exposed by the connector
     * @return either the id of the document or null
     */
    public abstract String getDocumentId( String externalPath );
```

```
    /**
     * Returns a Document instance representing the document with a given
id. The document should have
     * a "proper" structure for it to be usable by ModeShape.
     *
     * @param id a {@code non-null} string
     * @return either an {@link Document} instance or {@code null}
     */
    public abstract Document getDocumentById( String id );

    /**
     * Return the path(s) of the external node with the given identifier.
The resulting paths are from the
     * point of view of the connector. For example, the "root" node exposed
by the connector wil have a
     * path of "/".
     *
     * @param id a null-null string
     * @return the connector-specific path(s) of the node, or an empty
document if there is no such
     * document; never null
     */
    public abstract Collection<String> getDocumentPathsById( String id );

    /**
     * Checks if a document with the given id exists in the end-source.
     *
     * @param id a non-null string.
     * @return {{true}} if such a document exists, {{false}} otherwise.
     */
    public abstract boolean hasDocument( String id );

    ...
}
```

Not shown are fields, getters, and other implemented methods that your methods will almost certainly use. For example, a **`Document`** is a read-only representation of a JSON document, and they can be created by calling the **`newDocument(id)`** method with the document's identifier, using the resulting **`DocumentWriter`** to set/remove/add fields (and nested documents), and calling the writer's **`document()`** method to obtain the read-only **`Document`** instance.

The **`DocumentWriter`** interface provides dozens of methods for getting and setting node properties and child node references. Here's some code that uses a document writer to construct a node representation with a few properties:

```
   String id = ...
   DocumentWriter writer = newDocument(id);
   writer.setPrimaryType("lib:book");
   writer.addMixinType("lib:tagged");
   writer.addProperty("lib:isbn, "0486280616");
   writer.addProperty("lib:format, "paperback");
   writer.addProperty("lib:author", "Mark Twain");
   writer.addProperty("lib:title", "The Adventures of Huckleberry Finn");
```

```
    writer.addProperty("lib:tags", "fiction", "classic", "americana");
    // Add a single child named 'tableOfContents' with its own identifier
    writer.addChild(id + "/toc","tableOfContents");
    Document doc = writer.document();
```

As you can see, creating documents is pretty straightforward.

Identifiers of documents are simple strings that are expected to uniquely and durably identify a document. However, the content of that string is entirely up to the connector implementations. If the external system already has the notion of unique identifiers, it might be easiest to reuse a string representation of those identifiers. For example, a database might have a unique key within a given table, whereas a Git repository uses SHA-1 hashes for identifiers of commits, branches, tags, etc. Some external systems (like file systems) do not have a concept of unique identifiers, and in such cases the connector should devise its own identifier mechanism is durable and reliable.

15.4.4. Properties, Paths, Names, and Values

Most of the time, you can use string property names and property values that are String, Calendar, URL, or Numeric instances, and the hierarchical database will convert to an internal object representation. However, the hierarchical database provides object definitions of JCR names, paths, values, and properties. These classes are often much easier to work with than the String names and paths, and they're easy to create using namespace-aware factories. The **`ValueFactories`** interface is a container for type-specific factories accessible with various getter methods. Here's an example of creating a **`Path`** value from a string and then using the **`Path`** methods to get at the already-parsed segments of the path:

```
    String str = "/a/b/c/cust:d";
    PathFactory pathFactory = factories().getPathFactory();
    Path path = pathFactory.create(str);
    for ( Segment segment : path ) {
        Name name = segment.getName();
        String localName = name.getLocalName();
        String namespaceUri = name.getNamespaceUri();
        if ( segment.hasIndex() ) {
            String snsIndex = segment.getIndex();
        }
    }
    Path parentPath = path.getParent();
    ...
```

The process of using a factory to create **`Name`** , **`Binary`** , **`DateTime`** , and all other JCR-compliant values is similar.

Properties are slightly different, since they are a bit more structured. The hierarchical database provides a **`PropertyFactory`** that can create single- or multi-valued **`Property`** instances given a name and one or more values. Here's some simple code that shows how to create a single-valued property:

```
    PropertyFactory propFactory = propertyFactory();
    Name propName = nameFactory().create("lib:title");
    String propValue = factories().stringFactory("The Adventures of
Huckleberry Finn");
    Property prop = propFactory.create(propName,propValue);
```

All **`Property`** , **`Name`** , **`Path`** , **`DateTime`** , and **`Binary`** instances are immutable, meaning you can pass them around without worrying about whether the receiver might modify them. Also, the factories will often pick implementation classes that are tailored for the specific value. For example, there are separate

implementations for the root path, single-segment paths, paths created from a parent path, single-valued properties, empty properties, and multi-valued properties.

15.4.5. Writable Connector

The following code fragment shows the methods that a **`WritableConnector`** subclass must implement.

```
package org.modeshape.jcr.federation.spi;

import java.io.IOException;
import java.util.Collection;
import javax.jcr.NamespaceRegistry;
import javax.jcr.RepositoryException;
import org.infinispan.schematic.document.Document;
import org.modeshape.jcr.api.nodetype.NodeTypeManager;

public abstract class WritableConnector extends Connector {

    ...

   /**
     * Initialize the connector. This is called automatically by ModeShape
once for each Connector instance,
     * and should not be called by the connector. By the time this method is
called, ModeShape will have
     * already set the {{ExecutionContext}}, {{Logger}}, connector name,
repository name {@link #context},
     * and any fields that match configuration properties for the connector.
     *
     * By default this method does nothing, so it should be overridden by
implementations to do a one-time
     * initialization of any internal components. For example, connectors
can use the supplied {{registry}}
     * and {{nodeTypeManager}} parameters to register custom namespaces and
node types used by the exposed nodes.
     *
     * This is also an excellent place for connector to validate the
connector-specific fields set by ModeShape
     * via reflection during instantiation.
     *
     * @param registry the namespace registry that can be used to register
custom namespaces; never null
     * @param nodeTypeManager the node type manager that can be used to
register custom node types; never null
     * @throws RepositoryException if operations on the {@link
NamespaceRegistry} or {@link NodeTypeManager} fail
     * @throws IOException if any stream based operations fail (like
importing cnd files)
     */
    public void initialize( NamespaceRegistry registry,
                            NodeTypeManager nodeTypeManager ) throws
RepositoryException, IOException {
    }

    /**
     * Returns the id of an external node located at the given external path
```

```
within the connector's
     * exposed tree of content.
     *
     * @param externalPath a non-null string representing an external path,
or "/" for the top-level
     *        node exposed by the connector
     * @return either the id of the document or null
     */
    public abstract String getDocumentId( String externalPath );

    /**
     * Returns a Document instance representing the document with a given
id. The document should have
     * a "proper" structure for it to be usable by ModeShape.
     *
     * @param id a {@code non-null} string
     * @return either an {@link Document} instance or {@code null}
     */
    public abstract Document getDocumentById( String id );

    /**
     * Return the path(s) of the external node with the given identifier.
The resulting paths are
     * from the point of view of the connector. For example, the "root" node
exposed by the connector
     * will have a path of "/".
     *
     * @param id a null-null string
     * @return the connector-specific path(s) of the node, or an empty
document if there is no such
     *         document; never null
     */
    public abstract Collection<String> getDocumentPathsById( String id );

    /**
     * Checks if a document with the given id exists in the end-source.
     *
     * @param id a non-null string.
     * @return {{true}} if such a document exists, {{false}} otherwise.
     */
    public abstract boolean hasDocument( String id );

    /**
     * Removes the document with the given id.
     *
     * @param id a non-null string.
     * @return {{true}} if the document was removed, or {{false}} if there
was no document with the
     *         given id
     */
    public abstract boolean removeDocument( String id );

    /**
     * Stores the given document.
     *
     * @param document a non-null Document instance.
```

```
     * @throws DocumentAlreadyExistsException if there is already a new
document with the same identifier
     * @throws DocumentNotFoundException if one of the modified documents
was removed by another session
     */
    public abstract void storeDocument( Document document );

    /**
     * Updates a document using the provided changes.
     *
     * @param documentChanges a non-null DocumentChanges object which
contains
     *        granular information about all the changes.
     */
    public abstract void updateDocument( DocumentChanges documentChanges );

    /**
     * Generates an identifier which will be assigned when a new document
(aka. child) is created under an
     * existing document (aka.parent). This method should be implemented
only by connectors which support
     * writing.
     *
     * @param parentId a non-null string which represents the identifier of
the parent under which the new
     *        document will be created.
     * @param newDocumentName a non-null Name which represents the name that
will be given
     *        to the child document
     * @param newDocumentPrimaryType a non-null Name which represents the
child document's
     *        primary type.
     * @return either a non-null string which will be assigned as the new
identifier, or null which means
     *         that no "special" id format is required. In this last case,
the repository will
     *         auto-generate a random id.
     * @throws org.modeshape.jcr.cache.DocumentStoreException if the
connector is readonly.
     */
    public abstract String newDocumentId( String parentId,
                                          Name newDocumentName,
                                          Name newDocumentPrimaryType );
    ...
}
```

A **`WritableConnector`** has to implement all of the read-related methods that a **`ReadOnlyConnector`** must implement and a handful of write-related methods for removing, updating, and storing new documents (nodes).

In the same way that the hierarchical database provides a **`DocumentWriter`**, there is also a **`DocumentReader`** that has methods to easily read properties, primary type, mixin types, and child references:

```
    Document doc = ...
    DocumentReader reader = readDocument(doc);
```

```
    String id = reader.getDocumentId();
    String primaryType = reader.getPrimaryTypeName();
    Map<Name, Property> properties = reader.getProperties();
    // Get the ordered list of child references ...
    LinkedHashMap<String,Name> childReferences = reader.getChildrenMap();
    for ( Map<String,Name>.Entry childRef : childReferences.entrySet() ) {
        String key = childRef.getKey();
        String name = childRef.getValue();
    }
```

15.4.6. Extra Properties

The hierarchical database provides a framework for storing "extra properties" that cannot be stored in the external system. For example, the "file system connector" cannot naturally map arbitrary properties to file attributes, and instead uses a variety of techniques to stores these extra properties.

By default, the hierarchical database can store the extra properties inside the same Infinispan cache where the repository's own internal (non-federated) content is stored. However, this may not be ideal, and so a connector can provide its own implementation of the **`ExtraPropertiesStore`** interface:

```
package org.modeshape.jcr.federation.spi;

/**
 * Store for extra properties, which a {@link Connector} implementation can
use to store and retrieve
 * "extra" properties on a node that cannot be persisted in the external
system. Generally, a connector
 * should store as much as possible in the external system. However, not all
systems are capable of
 * persisting any and all properties that a JCR client may put on a node. In
such cases, the connector
 * can store these "extra" properties (that it does not persist) in this
properties store.
 */
public interface ExtraPropertiesStore {

    static final Map<Name, Property> NO_PROPERTIES = Collections.emptyMap();

    /**
     * Store the supplied extra properties for the node with the supplied
ID. This will overwrite any properties
     * that were previously stored for the node with the specified ID.
     *
     * @param id the identifier for the node; may not be null
     * @param properties the extra properties for the node that should be
stored in this storage area, keyed by
     *        their name
     */
    void storeProperties( String id,
                          Map<Name, Property> properties );

    /**
     * Update the supplied extra properties for the node with the supplied
ID.
     *
```

```
     * @param id the identifier for the node; may not be null
     * @param properties the extra properties for the node that should be
stored in this storage area, keyed by
     *        their name; any entry that contains a null Property will
define a property that should be removed
     */
    void updateProperties( String id,
                           Map<Name, Property> properties );

    /**
     * Retrieve the extra properties that were stored for the node with the
supplied ID.
     *
     * @param id the identifier for the node; may not be null
     * @return the map of properties keyed by their name; may be empty, but
never null
     */
    Map<Name, Property> getProperties( String id );

    /**
     * Remove all of the extra properties that were stored for the node with
the supplied ID.
     *
     * @param id the identifier for the node; may not be null
     * @return true if there were properties stored for the node and now
removed, or false if there were no extra
     *         properties stored for the node with the supplied key
     */
    boolean removeProperties( String id );
}
```

Then to use the extra properties store, simple call in the connector's **`initialize(...)`** method the **`setEtraPropertiesStore(ExtraPropertiesStore store)`** method with an instance of your custom store. Then, in your **`store(Document)`** and **`update(Document)`** methods, record these extra properties. There are multiple ways of doing this, but here are a few:

```
    ExtraProperties extraProperties = extraPropertiesFor(id, false);
    // Add a single property ...
    Property p1 = ...
    extraProperties.add(p1);
    // Or add multiple properties at once ...
    Map<Name,Property> properties = ...
    extraProperties.addAll(properties).except("jcr:primaryType",
"jcr:created");
    extraProperties.save();
```

15.4.7. Pageable Connectors

A **`Document`** that represents a node will contain references to all the children of that node. These references are relatively small (the ID and name of the child), and for many connectors this is sufficient and fast enough. However, when the number of children under a node starts to increase, building the list of child references for a parent node can become noticeable and even burdensome, especially when few (if any) of the child references may ultimately be resolved into their node representations.

A pageable connector is one that exposes the children of nodes in a "page by page" fashion, where the

parent node only contains the first page of child references and subsequent pages are loaded only if needed. This turns out to be quite effective, since when clients navigate a specific path (or ask for a specific child of a parent by its name) the hierarchical database does not need to use the child references in a node's document and can instead have the connector resolve such (relative or absolute external) paths into an identifier and then ask for the document with that ID.

Therefore, the only time the child references are needed are when clients iterate over the children of a node. A pageable connector will only be asked for as many pages as needed to handle the client's iteration, making it very efficient for exposing a node structure that can contain nodes with numerous children.

To make your **`ReadOnlyConnector`** or **`WritableConnector`** support paging, implement the **`Pageable`** interface:

```
package org.modeshape.jcr.federation.spi;

public interface Pageable {

    /**
     * Return a document which represents a page of children. The document
for the parent node
     * should include as many children as desired, and then include a
reference to the next
     * page of children with the {{PageWriter#addPage(String, String, long,
long)}} method.
     * Each page returned by this method should also include a reference to
the next page.
     *
     * @param pageKey a non-null {@link PageKey} instance, which offers
information about the
     *                page that should be retrieved.
     * @return either a non-null page document or {@code null} indicating
that such a page
     *         doesn't exist
     */
    Document getChildren( PageKey pageKey );
}
```

The hierarchical database then knows that the document for the parent will contain only some of the children and how to access each page of children as needed.

For example, here's an example of code that might be used in a connector's **`getDocumentById(...)`** method to include some of the children in the parent node's document and to include a reference to a second page of children. This uses an imaginary **`Book`** class that is presumed to represent information about a book in a library:

```
   String id = "category/Americana";
   DocumentWriter writer = newDocument(id);
   writer.setPrimaryType("lib:category");
   writer.addProperty("lib:description", "Classic American literature");
   // Get the books in this category ...
   Collection<Book> books = getBooksInCategory("Americana");
   // Put just 20 in this document ...
   count = 0;
   for ( Book book : books ) {
       writer.addChild(book.getId(),book.getTitle());
       if ( ++count == 20 ) break;
```

```
    }
    if ( count == 20 ) {
        // There were more than 20 books, so add a reference to the next page
        // that starts with the 20th book ...
        writer.addPage(id, 20, 20, books.size());
    }
    Document doc = writer.document();
```

Then, the connector's **`getPage(...)`** method would implement getting the child references for a particular page:

```
public Document getPage( PageKey pageKey ) {
    String parentId = pageKey.getParentId();
    int offset = pageKey.getOffsetInt();
    String category = parentId.substring(9);  // we assume this is
"category/{categoryName}"
    DocumentWriter writer = newDocument(parentId);
    // Get the next 20 books in this category plus one so we know there are
more ...
    List<Book> books =
getBooksInCategory("Americana").sublist(offset,offset+20+1); // no error
checking here!
    for ( Book book : books ) {
        writer.addChild(book.getId(),book.getTitle());
        if ( ++count == 20 ) break;
    }
    if ( count == 20 ) {
        // There were more than 20 books, so add a reference to the next page
        // that starts with the 20th book ...
        writer.addPage(id, 20, 20, books.size());
    }
    Document doc = writer.document();
}
```

As you can see, the logic of **`getPage(...)`** is actually very similar to the logic that adds children in the **`getDocumentById(...)`** method, and your connector might find it useful to abstract this into a single helper method.

Appendix A. Appendix

A.1. File Locations

The hierarchical database is installed with JBoss Data Virtualization and JBoss Fuse Service Works by default.

Associated files are located within the JBoss Enterprise Application Platform directory structure as follows:

```
/docs
   /schema
      modeshape_1_0.xsd
/domain
  /configuration
      domain-modeshape.xml
/modules
   /javax/jcr/*
   /org/modeshape/*
   /org/hibernate/*
   /org/infinispan
   /org/apache/*
/standalone
   /configuration/
      standalone-modeshape.xml
      standalone-modeshape-ha.xml
```

Appendix B. Initial Content

To help setup a simple, out-of-the-box repository pre-populated with some content, the hierarchical database provides a way to configure such content using a simple xml format. This content can be imported either in a specific workspace, or imported by default in all predefined or new workspaces.

Note

Initial content is imported only the first time a repository starts up into the predefined workspaces or when a new workspace is created, if that workspace was configured as such.

Warning

The initial content feature is intended to allow the import of a simple structure and is not intended for large volumes of data or complex data structures. There are other, more powerful mechanisms like backup & restore or JCR XML import/export that may be better suited to those cases.

B.1. XML Format

Each initial content XML must define a single root node called **`jcr:root`** under the namespace **`http://www.jcp.org/jcr/1.0`** . This represents the root node of a workspace and all content is imported below it.

Example B.1. Example

```
<?xml version="1.0" encoding="UTF-8"?>
<jcr:root xmlns:jcr="http://www.jcp.org/jcr/1.0">
    <folder jcr:mixinTypes="mix:created, mix:lastModified"
jcr:primaryType="nt:folder">
        <file1 jcr:primaryType="nt:file">
            <jcr:content/>
        </file1>
        <file2 jcr:primaryType="nt:file">
            <jcr:content/>
        </file2>
    </folder>
</jcr:root>
```

Each node has by default, the same name as the XML element which defines it and the properties the attributes of the XML element. Beside any number of custom properties, the JCR properties: **`jcr:name`** , **`jcr:primaryType`** and **`jcr:mixinTypes`** are supported, allowing for a node to have custom name, type and/or mixins. If not specified, the default node type of the created node will be **`nt:unstructured`** .

```
<?xml version="1.0" encoding="UTF-8"?>
<jcr:root xmlns:jcr="http://www.jcp.org/jcr/1.0">
    <Cars>
        <Hybrid>
```

```
            <car jcr:name="Toyota Prius" maker="Toyota" model="Prius"/>
            <car jcr:name="Toyota Highlander" maker="Toyota"
model="Highlander"/>
            <car jcr:name="Nissan Altima" maker="Nissan" model="Altima"/>
        </Hybrid>
        <Sports>
            <car jcr:name="Aston Martin DB9" maker="Aston Martin"
model="DB9"/>
            <car jcr:name="Infiniti G37" maker="Infiniti" model="G37"/>
        </Sports>
    </Cars>
</jcr:root>
```

It is also possible to override the name of the nodes by defining the **`jcr:name`** attribute, which will then be used instead of the XML element's name.

B.2. Configuring Initial Content

The configuration necessary for a repository to make use of the initial content is the following:

```
{
    "name" : "Repository with initial content",
    "storage" : {
        "transactionManagerLookup" :
"org.infinispan.transaction.lookup.DummyTransactionManagerLookup"
    },
    "workspaces" : {
        "predefined" : ["ws1", "ws2"],
        "default" : "default",
        "allowCreation" : true,
        "initialContent" : {
            "ws1" : "xmlImport/docWithMixins.xml",
            "ws2" : "xmlImport/docWithCustomType.xml",
            "default" : "xmlImport/docWithoutNamespaces.xml",
            "ws4" : "",
            "ws5" : "xmlImport/docWithCustomType.xml",
            "*" : "xmlImport/docWithMixins.xml"
        }
    }
}
```

One needs to define an **`initialContent`** object inside the **`workspaces`** object, with the following content:

- each attribute name inside the **`initialContent`** object, with the exception of the **`*`** string, will be treated as the name of a workspace and will have precedence over anything else. This includes the empty string, which can be used to explicitly configure workspace without any initial content, when a default is defined (see below)
- the **`*`** character is interpreted as "default content" which means that any predefined or newly created workspaces, that are not configured explicitly, will make use of this content
- the value of each attribute must be a simple string (including the empty string) which represents the URL of an XML file located in the runtime classpath

Appendix C. Binary Values

The hierarchical database is now capable of handling binary values that are larger than available memory. This is because it never loads the whole value onto the heap, but instead streams the value to and from the persistent store. You can also configure where the hierarchical database stores the binary values independently of where the rest of the content is stored.

The hierarchical database stores all binary content by its SHA-1 hash. The SHA-1 cryptographic hash function is not used for security purposes, but is instead used because the SHA-1 can reliably be determined entirely from the content itself, and because two binary contents will only have the same SHA-1 if they are indeed identical. Thus, the SHA-1 hash of some binary content serves as an excellent key for storing and referencing that content.

Using the SHA-1 hash as the identifier for the binary content also means that the hierarchical database never needs to store a given binary content more than once, no matter how many nodes or properties refer to it. It also means that if your JCR client already knows (or can compute) the SHA-1 of a large value, the JCR client can use APIs specific to the hierarchical database to easily determine if that value has already been stored in the repository.

C.1. Extended Binary Interface

The hierarchical database public API defines the **`org.modeshape.jcr.api.Binary`** interface as a simple extension to the standard **`javax.jcr.Binary`** interface. The hierarchical database adds useful methods to get the SHA-1 hash (as a binary array and as a hexadecimal string) and the MIME type for the content:

```
@Immutable
public interface Binary extends javax.jcr.Binary {

    /**
     * Get the SHA-1 hash of the contents. This hash can be used to
determine whether two
     * Binary instances contain the same content.
     *
     * Repeatedly calling this method should generally be efficient, as it
most implementations
     * will compute the hash only once.
     *
     * @return the hash of the contents as a byte array, or an empty array
if the hash could
     * not be computed.
     * @see #getHexHash()
     */
    byte[] getHash();

    /**
     * Get the hexadecimal form of the SHA-1 hash of the contents. This hash
can be used to
     * determine whether two Binary instances contain the same content.
     *
     * Repeatedly calling this method should generally be efficient, as it
most implementations
     * will compute the hash only once.
     *
     * @return the hexadecimal form of the getHash(), or a null string if
the hash could
```

```
     * not be computed or is not known
     * @see #getHash()
     */
    String getHexHash();

    /**
     * Get the MIME type for this binary value.
     *
     * @return the MIME type, or null if it cannot be determined (e.g., the
Binary is empty)
     * @throws IOException if there is a problem reading the binary content
     * @throws RepositoryException if an error occurs.
     */
    String getMimeType() throws IOException, RepositoryException;

    /**
     * Get the MIME type for this binary value.
     *
     * @param name the name of the binary value, useful in helping to
determine the MIME type
     * @return the MIME type, or null if it cannot be determined (e.g., the
Binary is empty)
     * @throws IOException if there is a problem reading the binary content
     * @throws RepositoryException if an error occurs.
     */
    String getMimeType( String name ) throws IOException,
RepositoryException;
}
```

All **`javax.jcr.Binary`** values returned will implement this public interface, so you can cast the values to gain access to the additional methods.

C.2. Importing and Exporting

When exporting content from a workspace with large **`Binary`** values, be sure to export using JCR's System View format. Only the System View treats properties as child elements. This allows each large value to be streamed (using buffered streams) into the XML element's content as a Base64-encoded string. Importing can also take advantage of streaming.

Exporting content using JCR's Document View results in all properties being treated as XML attributes, and various XML processing libraries treat large attributes poorly (e.g., using values that are in-memory **`String`** objects). Another critical disadvantage of the Document View is that it is unable to represent multi-valued properties, since attributes can have only one value.

Appendix D. Scaling for Many Child Nodes

The hierarchical database efficiently handles situations in which a single node has a large number (>100K) of child nodes. It does this by segmenting the parent's list of child references into multiple blocks, where each block is small enough to manage.

The hierarchical database actually performs this optimization in the background rather than do it during the Session's save() operation. As a consequence, the actual number of child references stored in any block might vary significantly from the "optimal" value. While the hierarchical database is capable of handling blocks of any size, performance when dealing with very large numbers of child nodes will be improved when the block sizes are optimized.

Accessing by Path

Navigating to a node by using its path is perhaps one of the most common access patterns in JCR. This uses the **`Node.getNode(String)`** method that takes a relative path, finding a particular child node with the supplied name and same-name-sibling index. The hierarchical database internally indexes the children in each block by both names, so finding nodes by name (and SNS) are as fast as possible, even if multiple blocks need to be accessed.

Iterating

Another common access pattern is to iterate over some or all of a parent node's children, using the **`Node.getNodes()`** and **`Node.getNodes(String)`** methods. The resulting **`NodeIterator`** will transparently access the children one block at a time, and will continue with all blocks until the last child reference is found or until the caller halts the iteration.

Accessing by Identifier

Another common access pattern is to find a node by identifier, using the **`Session.getNodeByIdentifier(String)`** method. The hierarchical database handles this request by directly finding the node by its identifier, and only needs to access the parent's (or ancestors') child references only when the node's name or path is requested by the caller (via the **`Node.getName()`** or **`Node.getPath()`** methods).

Appendix E. Infinispan Configuration

In all cases, you still need to configure Infinispan. There are a few things to keep in mind:

- Minimally, the cache used by a repository needs to be **transactional** , since the hierarchical database internally uses transactions and works with client-initiated or container-managed JTA transactions.
- Applications that may be concurrently updating the same nodes should use Infinispan configured to use **pessimistic locking** . By default Infinispan will use optimistic locking; this is more efficient for applications that do not update the same nodes, but concurrently updating the same nodes with optimistic locking may very well cause some updates to be lost. If you're not sure, use pessimistic locking.

Sample Infinispan configuration using a FileCacheStore is provided below:

Example E.1. Infinispan Pessimistic Locking

```
<local-cache name="sample">
   <!-- ModeShape requires transactions -->
   <transaction mode="NON_XA" locking="PESSIMISTIC"/>
   <!-- Use a cache with file-backed write-through storage. File-backed
storage is simple, but not necessarily the fastest. -->
   <file-store passivation="false" path="modeshape/store/sample" relative-
to="jboss.server.data.dir" purge="false"/>
</local-cache>
```

Appendix F. Registering Custom Node Types

As described in the section on defining custom node types, the JCR 2.0 specification defines the Compact Node Definition (CND) format to easily and compactly specify node type definitions, but uses this format only within the specification. Instead, the only standard API for registering custom node types is via the standard programmatic API.

The hierarchical database fully supports this standard API, but it also defines a non-standard API for reading node type definitions from either CND files or the older Jackrabbit XML format. This non-standard API is described in this section.

F.1. Registering Node Types Using CND Files

The hierarchical database defines in its public API a **`org.modeshape.jcr.nodetype.NodeTypeManager`** interface that extends the standard **`javax.jcr.nodetype.NodeTypeManager`** interface:

```
public interface NodeTypeManager extends javax.jcr.nodetype.NodeTypeManager
{

    /**
     * Read the supplied stream containing node type definitions in the
standard JCR 2.0 Compact Node Definition (CND) format or
     * non-standard Jackrabbit XML format, and register the node types with
this repository.
     *
     * @param stream the stream containing the node type definitions in CND
format
     * @param allowUpdate a boolean stating whether existing node type
definitions should be modified/updated
     * @throws IOException if there is a problem reading from the supplied
stream
     * @throws InvalidNodeTypeDefinitionException if the
<code>NodeTypeDefinition</code> is invalid.
     * @throws NodeTypeExistsException if <code>allowUpdate</code> is
<code>false</code> and the <code>NodeTypeDefinition</code>
     *         specifies a node type name that is already registered.
     * @throws UnsupportedRepositoryOperationException if this
implementation does not support node type registration.
     * @throws RepositoryException if another error occurs.
     */
    void registerNodeTypes( InputStream stream,
                            boolean allowUpdate )
        throws IOException, InvalidNodeTypeDefinitionException,
NodeTypeExistsException, UnsupportedRepositoryOperationException,
        RepositoryException;

    /**
     * Read the supplied file containing node type definitions in the
standard JCR 2.0 Compact Node Definition (CND) format or
     * non-standard Jackrabbit XML format, and register the node types with
this repository.
     *
     * @param file the file containing the node types
```

```
     * @param allowUpdate a boolean stating whether existing node type
definitions should be modified/updated
     * @throws IOException if there is a problem reading from the supplied
stream
     * @throws InvalidNodeTypeDefinitionException if the
<code>NodeTypeDefinition</code> is invalid.
     * @throws NodeTypeExistsException if <code>allowUpdate</code> is
<code>false</code> and the <code>NodeTypeDefinition</code>
     *         specifies a node type name that is already registered.
     * @throws UnsupportedRepositoryOperationException if this
implementation does not support node type registration.
     * @throws RepositoryException if another error occurs.
     */
    void registerNodeTypes( File file,
                            boolean allowUpdate ) throws IOException,
RepositoryException;

    /**
     * Read the supplied stream containing node type definitions in the
standard JCR 2.0 Compact Node Definition (CND) format or
     * non-standard Jackrabbit XML format, and register the node types with
this repository.
     *
     * @param url the URL that can be resolved to the file containing the
node type definitions in CND format
     * @param allowUpdate a boolean stating whether existing node type
definitions should be modified/updated
     * @throws IOException if there is a problem reading from the supplied
stream
     * @throws InvalidNodeTypeDefinitionException if the
<code>NodeTypeDefinition</code> is invalid.
     * @throws NodeTypeExistsException if <code>allowUpdate</code> is
<code>false</code> and the <code>NodeTypeDefinition</code>
     *         specifies a node type name that is already registered.
     * @throws UnsupportedRepositoryOperationException if this
implementation does not support node type registration.
     * @throws RepositoryException if another error occurs.
     */
    void registerNodeTypes( URL url,
                            boolean allowUpdate ) throws IOException,
RepositoryException;
}
```

Cast the **`NodeTypeManager`** instance obtained from the **`Workspace.getNodeTypeManager()`** method:

```
Session session = ...
Workspace workspace = session.getWorkspace();
org.modeshape.jcr.api.nodetype.NodeTypeManager nodeTypeMgr =
     (org.modeshape.jcr.api.nodetype.NodeTypeManager)
workspace.getNodeTypeManager();

// Then register the node types in one or more CND files
// using a Java File object ...
File myCndFile = ...
nodeTypeManager.registerNodeTypes(myCndFile,true);
```

```
// or a URL that is resolvable to a CND file ...
URL myCndUrl = ...
nodeTypeManager.registerNodeTypes(myCndUrl,true);

// or an InputStream to the content of a CND file ...
InputStream myCndStream = ...
nodeTypeManager.registerNodeTypes(myCndStream,true);
```

Alternatively, you can cast the result of the **`Session.getWorkspace()`** method to **`org.modeshape.jcr.api.Workspace`** , which overrides the **`getNodeTypeManager()`** method to return **`org.modeshape.jcr.api.nodetype.NodeTypeManager`** :

```
Session session = ...
org.modeshape.jcr.api.Workspace workspace = (org.modeshape.jcr.api.Workspace)
session.getWorkspace();
org.modeshape.jcr.api.nodetype.NodeTypeManager nodeTypeMgr =
workspace.getNodeTypeManager();

// Then register the node types in one or more CND files ...
```

F.2. Registering CND Files via Configuration

In addition to using the hierarchical database public API as described above, it is possible to configure a repository to import, at startup, one or more CND files using the following format:

```
{
    "name" : "Repository with node types",
    "storage" : {
        "transactionManagerLookup" :
"org.infinispan.transaction.lookup.DummyTransactionManagerLookup"
    },
    "workspaces" : {
        "predefined" : ["ws1", "ws2"],
        "default" : "default",
        "allowCreation" : true
    },
    "node-types" : ["cnd/cars.cnd", "cnd/aircraft.cnd"]
}
```

where the **`node-types`** attribute accepts an array of strings, representing paths to CND files, accessible at runtime.

Note

If CND files are configured to be imported at repository startup, they will overwrite each time any pre-existing node types with the same name that have been registered previously.

F.3. Jackrabbit XML Format

The hierarchical database also supports the older non-standard Jackrabbit format for defining node types, and only to make it easier for people to switch from Jackrabbit to the hierarchical database. The Jackrabbit

2.x no longer uses this format, and Jackrabbit 1.x only used this XML format for built-in node types and discouraged users from modifying it. However, some users of Jackrabbit 1.x still added their custom node types to this file.

Warning

Use the standard CND format wherever possible, and use this non-standard XML format only if you're trying to switch from Jackrabbit to the hierarchical database (with as little work as possible). If you are using the hierarchical database, you will need to convert your XML files to CND files.

The DTD for the non-standard XML files can be found here .

Appendix G. Revision History

Revision 6.3.0-04	**Fri Sep 9 2016**	**David Le Sage**
Updates for release 6.3		
Revision 6.2.0-03	**Fri Nov 6 2015**	**David Le Sage**
Updates for release 6.2		