nuxeo | Professional Open Source ECM

Nuxeo SQL Storage

Specification

Last Modification	2008-08-29
Project Code	NUXEO
Document ID	NXSQL
Version	1.1
Copyright	Copyright © 2008 Nuxeo. All Rights Reserved.

History

Version	Date	Participant(s)	Comments
0.5	2008-03-25	• Florent Guillaume	First released version
0.6	2008-05-02	• Florent Guillaume	Improved storage
0.9	2008-05-05	• Florent Guillaume	Reformatting
1.0	2008-05-28	• Florent Guillaume	Expanded file storage section
1.1	2008-08-29	• Florent Guillaume	Describe current implementation

Table of Contents

1 Introduction

This document presents the design and implementation (as of 2008-08-29) of a storage model for Nuxeo EP to persist information in an SQL database.

The goals of this storage model are:

- store information in standard SQL databases,
- use "natural" object mapping to tables,
- be fast,

And in the future:

- support a full versioning API, including tree versioning,
- support full-text searches on databases having that capability,
- support storage of blobs as database objects or as filesystem files,
- have some flexibility in the storage model to optimize certain cases at configuration time.

2 Mapping Nuxeo to nodes and properties

The Nuxeo model is mapped internally to a model based on a hierarchy of nodes and properties. This model is similar to the basic JCR (JSR-170) data model.

2.1 Nodes, properties, children

A node represents a complex value having several properties.

The properties of a node can be either simple (scalars, including binaries), or collections of scalars (lists usually).

A node can also have children. A complex property of a Nuxeo document is mapped to child nodes.

2.2 Children

The parent-child information for nodes is stored in a so-called "hierarchy" table.

The normal children of a document are mapped to child nodes of the document node.

There are therefore two kinds of children: child documents and complex types. They have to be quickly distinguished in order to:

- find all child documents and only them,
- find all complex properties of a document and only them,
- resolve name collisions.

To distinguish the two, the fragment table holding hierarchy information has an additional column holding a "is complex property" flag.

2.3 Fragment tables

A fragment table is a table holding information corresponding to one schema (simple fragment), or a table corresponding to a multi-valued property of a schema (collection fragment).

For a simple fragment, each of the table's columns correspond to a simple property of the represented schema. One row corresponds to one document using that schema.

For a collection fragment, the set of values for the multi-valued property is represented using as many rows as needed.

A node is the set of fragments corresponding to the schemas of that node.

2.4 Fields mapping

Nuxeo fields are mapped to properties or to child nodes:

- a simple type (scalar or array of scalars) is mapped to a property (simple or collection) of the document node,
- a complex type is mapped to a child node of the document node. There are two kinds of complex types to consider:
  - lists are mapped to a set of ordered complex property children,
  - maps are mapped to a node whose node type corresponds to the schema of the map.

2.5 Security

Security information is stored as an ACL which is a collection of simple ACEs holding basic rights information. This collection is stored in a dedicated table in a similar way to lists of scalars, except that the value is split over several column to represent the rich ACE values.

3 Fragment SQL mapping

3.1 Identifiers

Each node has a unique identifier. In standard Nuxeo EP, it is a UUID that is randomly generated by the storage backend. The backend also supports using an integer assigned from an auto-incremented database sequence, but this is not used by Nuxeo at the moment.

All the fragments making up a given node use the node identifier in their `id` column.

For clarity in the rest of this document simple integers are used, but Nuxeo actually uses UUIDs by default.

3.2 Repositories table

This table hold the root identifier for each repository. Usually Nuxeo has only one repository per database, which is named “default”.

Table **`repositories`**:

id	name
1	default

3.3 Hierarchy table

There are two kinds of nodes: placeful ones (those who have a location in the containment hierarchy), and placeless ones (version frozen nodes).

Each node has a row in the main `hierarchy` table defining its containment information if it is placeful, or just holding its name if it is placeless. The same tables holds ordering information for ordered children.

Table **`hierarchy`**:

id	parentid	pos	name	...
1			“”	
1234	1	0	workspace	
5678	1234	0	mydoc	

Note that:

- the `id` column is used as a foreign key reference with “on delete cascade” from all other fragment tables that refer to it,
- the `pos` is NULL for non-ordered children,
- the `parentid` and `pos` are NULL for placeless nodes,
- the `name` is an empty string for the hierarchy's root.

For performance reasons (denormalization) this table has actually more columns; they are detailed below.

3.4 Type information

The node types are accessed from the main `hierarchy` table.

When retrieving a node by its id, the primary type is consulted first and, according to the value found, applicable fragments are deduced, to give a full information of all the fragment tables that apply to this node.

Table **hierarchy** (continued):

id	...	isproperty	primarytype	...
1		FALSE	Root	
1234		FALSE	Bar	
5678		FALSE	Mytype	

The isproperty column holds a boolean that distinguishes normal children from complex properties.

3.5 Simple fragment tables

Each Nuxeo schema corresponds to one table. The table's columns are all the single-valued properties of the corresponding schema. Multi-valued properties are stored in a separate table each.

A myschema fragment (corresponding to a Nuxeo schema with the same name) will have the following table:

Table **myschema**:

id	title	description	created
5678	Mickey	The Mouse	2008-08-01 12:56:15.000

A consequence is that to retrieve the content of a node, a SELECT will have to be done in each of the tables corresponding to the node type and all its inherited node types. However lazy retrieval of a node's content means that in many cases only a subset of these tables will be needed.

3.6 Collection fragment tables

A multi-valued property is represented as data from a separate array table holding the values and their order. For instance, the property “my:subjects” of the schema “myschema” with prefix “my” will be stored in the following table:

Table **my_subjects**:

id	pos	value
5678	0	USA
5678	1	CTU

3.7 Specialized tables

The following tables correspond to features that are not expressed primarily as Nuxeo schemas.

3.7.1 Files and binaries

The file (content) abstraction in Nuxeo is treated by the storage as any other schema, except that one of the column hold a “binary” value. This binary value corresponds indirectly to the content of the file.

Table **content**:

id	mime-type	encoding	data	length	digest
8501	application/pdf		ebca0d868ef3	344256	
8502	text/plain	ISO-8859-1	5f3b55a834a0	541	
8503	text/plain	ISO-8859-1	5f3b55a834a0	541	

Table **file**:

id	filename
4061	report.pdf
4062	test.txt
4063	test_copy.txt

The filename is stored in a separate table just because the current Nuxeo schemas are split that way (the filename is a property of the document, but the content is a child complex property). In the future this could be optimized by having better schemas.

The `data` column of the `content` table refers to a binary type. All binary storage is done on the server filesystem according to the value stored in the data column, which is a cryptographic hash of the binary, in order to check for uniqueness and share identical binaries.

On the server filesystem, a binary is stored in a set of multi-level directories based on the has, to spread storage. For instance the binary with the has `ebca0d868ef3` will be stored in a file with path `data/eb/ca/0d/ebca0d868ef3` under the binaries root.

3.7.2 Locking

The locking in the basic Nuxeo model is simple, and mostly managed by the application level.

A lock can be shallow (applies only to the node on which it's placed) or deep (applies to all underlying nodes)

Table **locks**:

id	lock
5670	Administrator:20 Aug 2008
5678	Administrator:20 Aug 2008
9944	Administrator:21 Aug 2008

In the future this table will be extended with a timestamp (for timeouts), a token (for open-scoped locks), and further information (like a deep/shallow flag).

3.7.3 Versioning

Versioning uses identifiers for several concepts:

- live node id,
- version id,
- versionable id.

The *live node id* is the identifier of a node that may be subject to versioning.

The *version id* is the identifier of the frozen node copy that is created when a version was snapshotted.

The *versionable id* is the identifier of the original live node of a version, but keeps its meaning even after the live node may be deleted. Several frozen version nodes may come from the same live node, and therefore have the same versionable id.

Version nodes don't have a parent (they are placeless), but have more meta-information (`versionableid`, various information) than live nodes. Live nodes hold information about the version they are derived from (base version).

Table **`hierarchy`** (continued):

id	...	baseversionid	ischeckedin	majorversion	minorversion
5675		6120	TRUE	1	0
5678		6143	FALSE	1	1
5710			FALSE		
6120				1	0
6121				1	1
6143				4	3

Note that:

- the `baseversionid` represents the version from which a checked out or checked in document originates. A new document that has never been checked in has a NULL `baseversionid`,
- a version node has a NULL `ischeckedin` value,
- this information is inlined in the `hierarchy` table for performance reasons.

Table **`versions`**:

id	versionableid	created	label	description
6120	5675	2007-02-27 12:30:00.000	v1	
6121	5675	2007-02-28 03:45:05.000	v2	
6143	5678	2008-01-15 08:13:47.000	v1	

Note that:

- the `versionableid` is the `id` of the versionable node (which may not exist anymore, which means it's not a foreign key reference), and is common to a set of versions for the same node.

3.7.4 Proxies

Proxies are a Nuxeo feature, expressed as a node type holding only a reference to a frozen node and a convenience reference to the versionable node of that frozen node.

Proxies by themselves don't have additional content-related schema, but still have security, locking, etc. These facts are part of the node type inheritance, but the proxy node type table by itself is a normal node type table.

Table **`proxies`**:

id	targetid	versionableid
9944	6120	5675

The `targetid` is the id of a version node.

The `versionableid` is duplicated here for performance reasons, it could be retrieved from the target using a JOIN.

3.7.5 Security

The Nuxeo security model is based on the following:

- a single ACP is placed on a (document) node,
- the ACP contains an ordered list of named ACLs, each ACL being an ordered list of individual grants or denies of permissions,
- the security information on a node (materialized by the ACP) also contains local group information (which can emulate owners).

Table **`acls`**:

id	pos	name	grant	permission	user	group
5678	0	workflow	true	WriteProperties	cobrian	
5678	1	workflow	false	ReadProperties		Reviewer
5678	2	user	false	ReadProperties	kbauer	

This table is slightly denormalized (`name`s with identical values follow each other by `pos` ordering), but this is to minimize the number of JOIN to get all ACLs for a document. Also one cannot have a named ACL with an empty list of ACEs in it, but this is not a problem given the semantics of ACLs.

The `user` column is separated from the group column because they semantically belong to different namespaces. However for now in Nuxeo groups and users are all mixed in the `user` column, and the `group` column is kept empty.

3.7.6 Miscellaneous

The lifecycle information (lifecycle policy and lifecycle state) is stored in a dedicated table.

The dirty information (a flag that describes whether the document has been changed since its last versioning) is stored in the same table for convenience.

Table **`misc`**:

id	lifecyclepolicy	lifecyclestate	dirty
5670	default	draft	FALSE
5678	default	current	FALSE
9944	publishing	pending	FALSE

4 Future features

JCR and JCR 2 have a number of features that have to be taken into account in the design of the storage model, but are not necessary for the basic Nuxeo mapping.

4.1 Per-document facets

Nuxeo will in the future allow the addition of per-document facets, this will be managed through per-node instance mixins. They will be stored in an additional `mixintypes` table.

Table **`mixintypes`**:

id	mixintype
5678	withcomments
5678	withannotation

4.2 Optimized hierarchy

To improve hierarchical searches, for instance find all documents under a given parent, a separate table storing the total hierarchy will be used. This table can be maintained by the application or by stored procedures triggers.

Table **`ancestors`**:

id	ancestorid	depth
1234	1	1
5678	1234	1
5678	1	2

Note that the number of lines of this table is equal to the number of placeful documents multiplied by the average document depth.

The `depth` column stores the depth of a node relative to its ancestor. Only strict ancestors are stored, a node is not its own ancestor (and therefore `depth` is always at least 1).

The `ancestors` table gives a fast way to:

- check whether a node is under another node,
- get the set of all nodes under a given node,
- get inherited information about all nodes above a given node.

4.3 Same-name siblings

JCR allows an implementation to have same-name siblings. This is already allowed with the current design as the `name` of the children of a node is not a unique key for that node.

4.4 Shareable nodes

JCR 2 allows an implementation to have shareable nodes. This can be implemented at the storage level by relaxing the constraint that the `id` be the primary key in the `hierachy` table, and allowing it to appear several times with a different `parentid`, and all the non-hierarchy information of the `hierarchy` table (type, versioning, etc) into a separate `nodes` table.

4.5 Workspaces

JCR allows several workspaces, between which some clone and merge operations are possible. Two documents with the same id can appear in different workspaces, but all workspaces share the same version histories.

This is already taken into account by the current design, the workspace name is the repository name in the `repositories` table.

4.6 Residual properties

JCR allows node types to define residual properties, which are properties with arbitrary names. As this name is not defined in the property definition of the node type, it has to be specified in a dedicated column.

This model is also known as the entity-attribute-value model (EAV).

Each scalar and array type has its own table to hold residual properties.

Table **`residual_strings`**:

id	name	value
5678	somevalue	foo

For residual properties that are multi-valued, a variant on collection table is used:

Table **`residual_stringarrays`**:

id	name	pos	value
5678	somearray	0	foo
5678	somearray	1	bar

When a node is retrieved, the residual properties tables are checked only if the node type allows residual properties. An additional column in the main `hierarchy` table could hold a flag specifying if there are actually residuals stored, or even a bitmask with the specific residual types present.

4.7 Local groups

A table will hold local group information. The exact semantics of the local groups have to be determined.

Table **`localgroups`**:

id	localgroup	user
5678	Reviewer	cobrian
5678	Reviewer	tlennox
5678	Owner	jbauer

If groups of groups are needed, an additional `group` column can be used.

4.8 Version predecessors and successors

The JCR predecessors and successors are stored in a dedicated table. They describe a graph. In JCR 2 simple versioning, this table is implicit in the ordering of the versions by creation date.

Table **versiongraph**:

versionableid	id	successorid
5675	6120	6121
5675	6120	6123
5675	6143	6144

Note that:

- the tuple (versionableId, id) is unique (versionableId is a denormalized column, it can be deduced from the id),
- the tuple (id, successorId) is unique.

4.9 Versioned containers

JCR also records (with full versioning in JCR 2), when a node with versionable children is versioned, to what version history id each child was referring.

Table **versionchildren**:

parentid	id	pos	name
6120	5640	0	foo
6120	5641	1	bar

Notes:

- the id column refers to versionable node ids,
- the parentId column refers to version nodes,
- the tuple (parentId, id) is unique,
- the tuple (parentId, pos) is unique (excluding NULL pos values for),
- for versioned child nodes of unordered containers the pos will be NULL.

4.10 Version activities

JCR 2 specifies additional versioning features: activities and baselines.

An activity records a group of changes. The user can start and stop activities, and while an activity is in effect all versioning changes are associated to it. A version records which activity created it, and a checked out document records which activity checked it out. A checked in document does not have an activity.

Table **hierarchy** (continued):

id	...	activityid
5670		8010
5671		8010
5675		
5678		8011

A title can also be associated with an activity.

Table **activitytitles**:

activityid	title
8010	testing
8011	v2.1

4.11 Version configurations and baselines

A configuration is a subtree of documents rooted at a particular node, but that does not include any other configuration that may exist inside this tree. The configuration's state can be thought of as a special kind of history that doesn't store document versions but baselines.

A document is designated as the root of a configuration by using an additional configurationid column.

Table **hierarchy** (continued):

id	...	configurationid
5678		408

The configuration itself records its root, and will be associated with a baseline when the configuration is checked in. The baselines are special version objects that record the full state of a configuration. The state of a configuration consists of all the baseversionid of the nodes belonging to that configuration. A configuration can be restored from any baseline with the same root, which has the effect of restoring all the nodes to their recorded baseversionid.

Table **configurations**:

configurationid	rootid	baselineid
408	5678	950

Table **baselines**:

baselineid	created	label
950	2008-07-15 12:35:03.000	foo

Table **baselinestates**:

baselineid	id	baseversionid
950	5678	5671
950	5679	5711
950	5670	5712

The above storage is one of the simplest possible, but a more efficient storage of baseline states would involve intelligent storage of deltas between trees, otherwise the storage size is proportional to the tree size and not to the changes between configurations.