ALMA MATER STUDIORUM - UNIVERSITA' DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI SCIENZE

CORSO DI LAUREA IN SCIENZE E TECNOLOGIE INFORMATICHE

PROGETTAZIONE E REALIZZAZIONE DI UNA PIATTAFORMA DBAAS PER JACKRABBIT/JCR

Relazione finale in:

BASI DI DATI

Relatore

Dott.ssa Annalisa Franco

Presentata da

Parisi Luca

Sessione II

Anno Accademico 2013 - 2014

INDICE

INTRODUZIONE

Lo scopo di questa tesi è presentare un progetto per la realizzazione di una piattaforma DBAAS per il database Jackrabbit. Per piattaforma DBAAS si intende un sistema in grado di offrire ai clienti un database su richiesta (DataBase As A Service), e di amministrare automaticamente i database per conto dei clienti. I clienti, quindi, non si devono preoccupare di gestire i propri database come se fossero implementati *in-house,* è compito della piattaforma DBAAS rendere sempre i dati consistenti disponibili ai clienti, tramite opportune operazioni di backup, restore e scaling. Jackrabbit è un'implementazione open-source dello standard JCR, che definisce il modello astratto di un repository JCR. Il modello di dati definito nello standard JCR presenta le caratteristiche sia di un RDBMS che di un file system, oltre ad altre caratteristiche aggiuntive.

La struttura dati offerta da Jackrabbit è molto flessibile e viene definita dal contenuto man mano che viene aggiunto. Si differenzia molto dal modello relazionale, che definisce in anticipo la struttura e quindi la forma dei dati. Jackrabbit, ad oggi, viene utilizzato soprattutto dai CMS (Content Management System) in quanto il modello JCR si appresta molto bene a gestire il contenuto dei siti web.

Il progetto viene realizzato in un primo momento tramite un prototipo della piattaforma DBAAS, in cui vengono effettuati test intensivi per controllare ogni funzionalità della piattaforma.

Il prototipo viene sviluppato in un singolo host, nel quale però è possibile simulare tutte le operazioni base della piattaforma DBAAS, come se fosse il sistema finale. In un secondo momento, il progetto sarà realizzato nella sua versione finale, con tutto l'hardware necessario a garantire un servizio ottimale.

La tesi è formata da 6 capitoli, nei quali sono descritte le fasi di progettazione e implementazione del prototipo.

Nel primo capitolo viene spiegato nel dettaglio lo standard JCR, che definisce la struttura del repository astratto e le API Java che gestiscono tutto il repository e il suo contenuto. Capire il

modello di dati JCR è fondamentale per la comprensione delle diverse componenti del progetto.

Il secondo capitolo confronta il modello di dati JCR con il modello di dati relazionale, che attualmente è il modello di dati più usato e diffuso nel mondo. In questo capitolo vengono messe a confronto i due modelli di dati, e vengono proposte alcune osservazioni relative agli scenari applicativi in cui un modello di dati è preferibile all'altro (relazionale vs. JCR).

Il terzo capitolo descrive in dettaglio il software Apache Jackrabbit, che è un'implementazione completa dello standard JCR. Jackrabbit può essere visto come un'interfaccia tra l'applicazione e i dati, in quanto offre una visione astratta del contenuto, che è gestibile dall'applicazione tramite le API JCR. I dati veri e propri sono salvati nei persistence manager e nei datastore, che sono i componenti di più basso livello dell'architettura di Jackrabbit.

Nel quarto capitolo viene presentato il progetto della piattaforma DBAAS (secondo il formalismo ER) e vengono definite tutte le operazioni di base della piattaforma. La piattaforma viene controllata e gestita da un database relazionale contenente il suo stato attuale.

Nel quinto capitolo viene descritta l'architettura del prototipo e il suo ambiente di sviluppo.L'ambiente di test del prototipo è stato reso il più possibile simile al sistema finale, rendendo ad esempio tutti i componenti del prototipo accessibili da remoto. Ad esempio, il datastore è implementato in un disco di rete, e viene gestito dal prototipo (così come sarà gestito dal sistema finale) tramite il protocollo NFS (Network File System).

Nel sesto e ultimo capitolo vengono descritte tutte le operazioni base della piattaforma DBAAS, con le relative procedure che possono essere eseguite e testate sul prototipo.

CAPITOLO 1: LO STANDARD JCR

1.1 INTRODUZIONE

Lo Standard JCR (Java Content Repository) è un insieme di specifiche che definiscono un modello astratto di repository e le API java per l'immagazzinamento dei dati. Le specifiche sono state sviluppate sotto il "Java Community Process" (JCP) nelle JSR-170 (versione 1, rilasciata nel 2005 [1]) e JSR-283 (versione 2, rilasciata nel 2009 [2]).

All'interno di un sistema informatico, un repository JCR interessa il livello dei dati.

In particolare è stato progettato con la seguente idea: ogni oggetto è un contenuto (ad esempio articoli, blog, commenti, immagini, dati strutturati, codice, …).

La figura 1.1 mostra in quale strato del sistema informatico si colloca un repository JCR.

Il modello del repository JCR supporta alcune caratteristiche presenti sia nei database relazionali che nei file system, oltre ad altre caratteristiche aggiuntive. Le caratteristiche fondamentali di un repository JCR sono la lettura e la scrittura dei dati.

Come un database relazionale, può memorizzare dati strutturati, supporta le query e le transazioni. Come un file system, offre la possibilità di creare gerarchie, controllo degli accessi e locking. Oltre a queste caratteristiche, un repository JCR supporta anche i dati non strutturati e multivalore, la ricerca tramite "full-text", la gestione delle versioni e l'osservazione del contenuto tramite eventi.

La figura 1.2 mostra le caratteristiche di un repository JCR, in confronto alle caratteristiche di un database relazionale e di un file system.

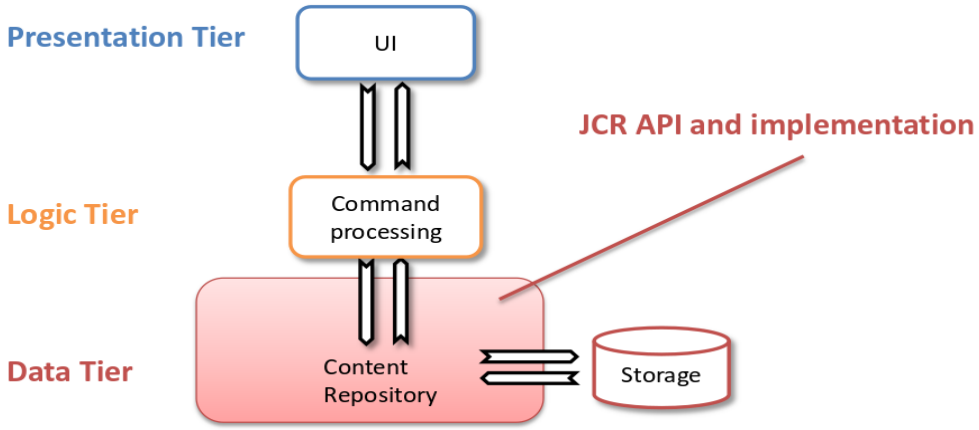

Figura 1.1: Il repository JCR fa parte del livello dei dati di un sistema. Le applicazioni possono accedere ai dati tramite l'implementazione delle API JCR, definite anch'esse nello standard JCR.

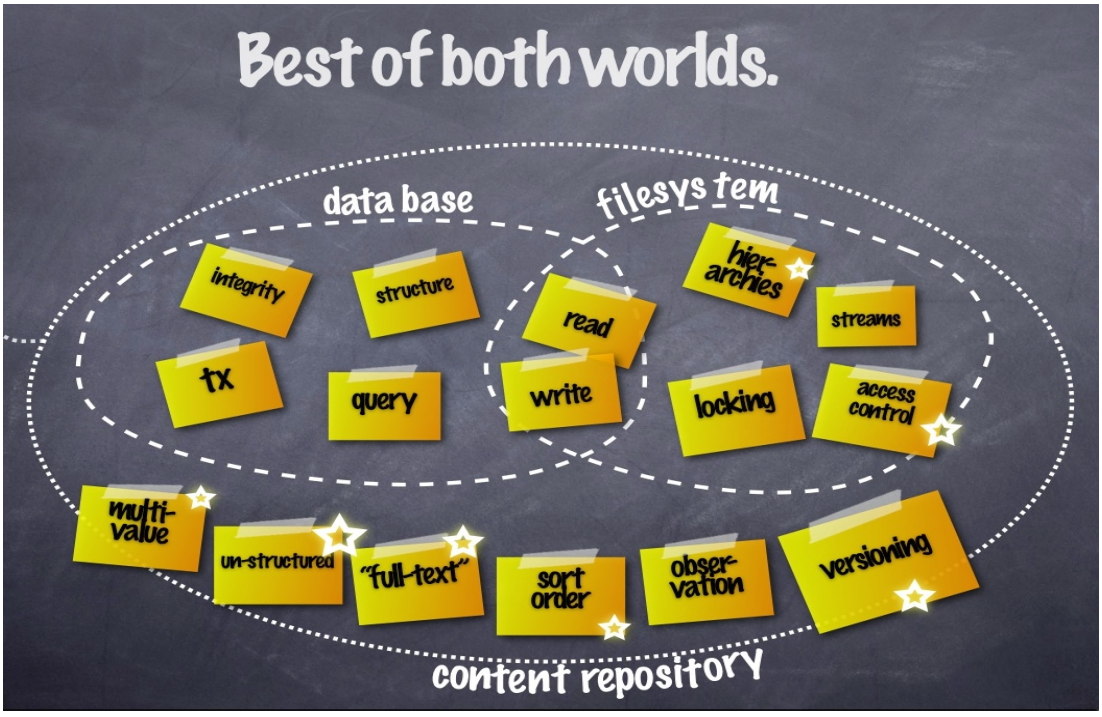

Figura 1.2: Un repository JCR è stato progettato per avere, oltre alle caratteristiche dei database relazionali e dei file system, anche altre funzionalità (ad esempio l'osservazione del contenuto tramite eventi)

1.2 MODELLO ASTRATTO DEL REPOSITORY

In questo paragrafo sono riportate le caratteristiche del modello di un repository JCR:

- Un Repository JCR è composto da uno o più workspace persistenti.
- Ogni workspace è composto da un grafo aciclico di items, dove gli archi rappresentano la relazione padre–figlio.
- Un item può essere sia un nodo che una proprietà.
- La posizione di un item nel grafo è definita dal percorso del nodo radice fino ad esso, come accade per i file system.
- Un nodo può avere zero o più figli.
- Ogni nodo ha un identificatore univoco all'interno del workspace.
- Una proprietà non può avere figli, ma può avere un valore singolo o più valori (come un array).
- Ogni proprietà deve comunque avere un valore, non esiste il valore nullo.
- Le proprietà sono il contenitore di tutti i dati del repository.
- Ogni proprietà ha un tipo, che può essere scelto fra: STRING, URI, BOOLEAN, LONG, DOUBLE, DECIMAL, BINARY, DATE, NAME, PATH, WEAKREFERENCE, REFERENCE.
- Le proprietà di tipo REFERENCE e WEAK REFERENCE contengono l'identificatore del nodo a cui fanno riferimento. Nelle proprietà di tipo WEAK REFERENCE non c'è l'obbligo di rispettare l'integrità referenziale, a differenza delle proprietà di tipo REFERENCE.

La figura 1.3 mostra il diagramma delle classi in UML del modello astratto del repository JCR:

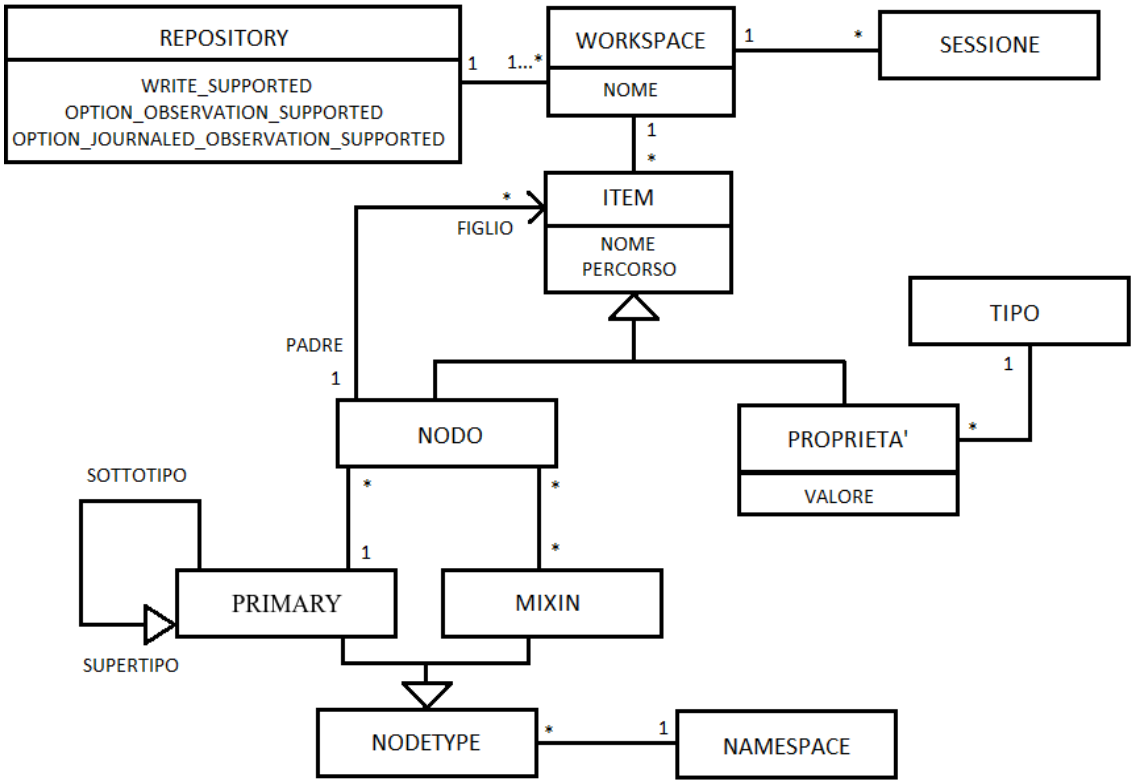

Figura 1.3: Rappresentazione del modello di un repository JCR, tramite il diagramma delle classi di UML.

La figura 1.4 mostra un esempio di contenuto e com'è organizzato all'interno di un repository JCR. Nell'esempio in figura, il repository R è formato da 3 workspace (w0, w1, w2). Il workspace w0 è formato da 4 nodi (A,B,C,E). I nodi A,B e C sono figli del nodo root (presente in ogni workspace), mentre il nodo E è figlio del nodo A.

Il nodo A ha una proprietà chiamata D, di tipo String, il cui valore è "Once upon a time...".

Il percorso del nodo root è '/'

Il percorso del nodo A è '/A' e il percorso della proprietà D è '/A/D'.

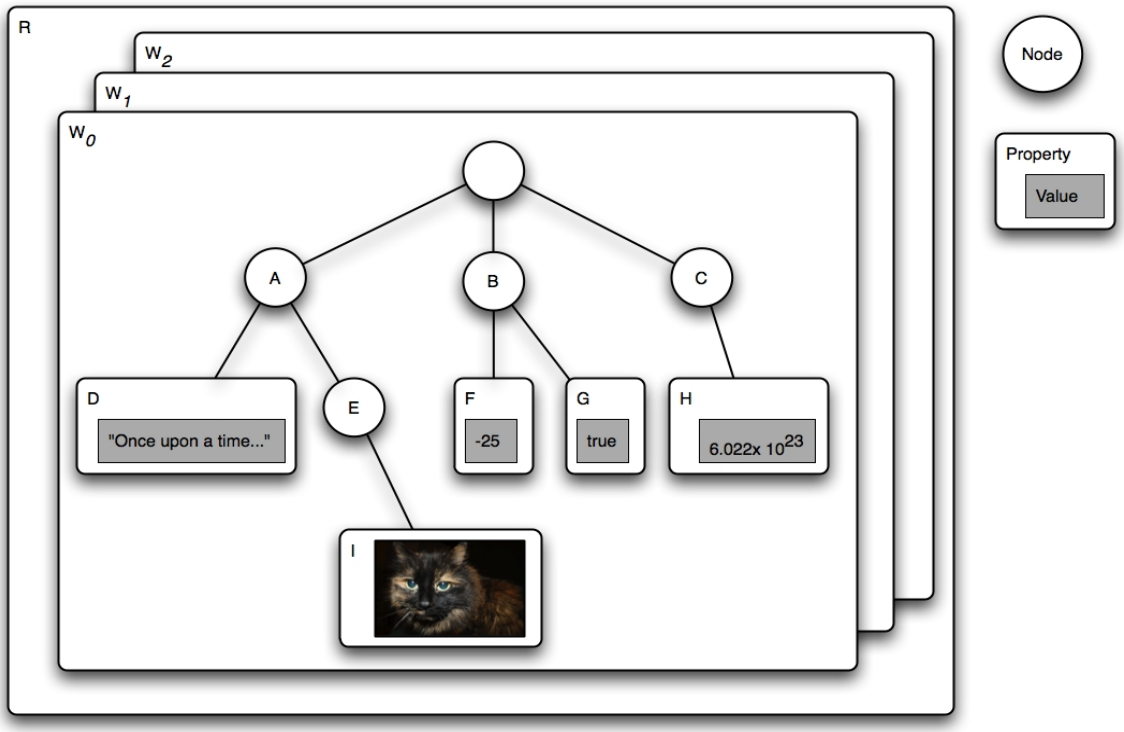

Figura 1.4: Esempio del contenuto di un repository JCR.

1.2.1 Namespace

Ogni repository ha un registro per i namespace, che contiene il mapping fra il prefisso e il namespace. Ogni namespace contiene la definizione di un insieme di nodetype.
Ad esempio, il namespace nt contiene la definizione dei nodetype nt:base, nt:unstructured, e altri. Ogni repository JCR deve contenere almeno questi namespace:

- 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
- xml = http://www.w3.org/XML/1998/namespace
- *(the empty string) = (the empty string)*

1.2.2 Nodetype

Un nodetype definisce i vincoli di un nodo, tra i quali i figli e le proprietà che devono essere obbligatoriamente presenti.
Il diagramma delle classi del modello del repository mostrato in figura 1.3 specifica che:

- Un nodetype può avere un supertipo, dal quale eredita la definizione;
- Un nodetype può essere primario o mixin;
- Ogni nodo ha un nodetype primario e zero o più mixin nodetype.

Il nodetype primario definisce la struttura di un nodo, mentre il mixin nodetype definisce vincoli aggiuntivi al nodo.
Il nodetype primario viene assegnato al nodo quando viene creato.
Un nodetype può essere dichiarato astratto, cioè non può essere direttamente assegnato ad un nodo, ma può essere il supertipo di altri nodetype.
Durante il ciclo di vita del nodo si possono aggiungere e rimuovere i mixin nodetype.
I nodetype definiscono com'è il contenuto di un nodo: non strutturato, semi-strutturato o strutturato.

Un nodetype può definire, per ogni item, i seguenti attributi:

- protetto (protected);
- autocreato (autocreated);
- obbligatorio (mandatory).

Se un item N è un nodo ed è definito protetto:

- non può essere rimosso;
- i nodi figli non si possono aggiungere, rimuovere o riordinare;
- le proprietà non si possono ne aggiungere ne rimuovere;

- i valori delle proprietà non si possono cambiare;
- il nodetype primario non può essere cambiato;
- i mixin nodetype non si possono né aggiungere né rimuovere.

Se un item P è una proprietà ed è definita protetta:

- non può essere rimossa;
- il valore di P non può cambiare.

Se un item è autocreato, viene creato automaticamente durante la creazione del suo nodo padre.
Se un item è obbligatorio, l'item deve esistere prima che il suo nodo padre sia salvato in modo persistente.

1.2.3 nt:base

Ogni repository deve avere la definizione del nodetype nt:base, che è il supertipo di tutti i nodetype.
È un nodetype astratto, ed è' l'unico nodetype primario senza supertipi.
nt:base definisce 2 proprietà: jcr:primaryType e jcr:mixinTypes:

- jcr:primaryType è una proprietà obbligatoria e autocreata di tipo NAME, che memorizza il nome del nodetype primario del nodo;
- jcr:mixinTypes è una proprietà protetta ma non obbligatoria, e può assumere valori multipli. E' presente solo quando il nodo ha uno o più mixin nodetype dichiarati, e per ognuno ne memorizza il nome.

Dato che nt:base è il supertipo di tutti i nodetype, ogni nodo di un repository JCR ha le proprietà jcr:primaryType e jcr:mixinTypes.

La figura 1.5 mostra un diagramma degli oggetti in stile UML che rappresenta un esempio di gerarchia di nodetype primari sottotipi di nt:base.
La terminologia usata per definire gli attributi fa parte della grammatica CND (Compact Namespace and NodeType Definition) [3], in cui:

- il '-' definisce una proprietà al nodetype primario corrente;
- il '+' definisce un nodetype primario figlio del nodetype primario corrente;
- il '*' indica che la definizione può essere multipla.

Nell'esempio in figura, sia il nodetype primario nt:unstructured che il nodetype primario nt:hierarchyNode contengono la definizione delle proprietà JCR:primaryType e JCR:mixinTypes, in quanto sono sottotipi di nt:base.
Quindi, se un nodo ha come nodetype primario nt_hierarchyNode, deve avere 3 proprietà definite prima di essere salvato in modo persistente nel workspace:

- JCR:primaryType;
- JCR:MixinTypes;
- JCR:Created.

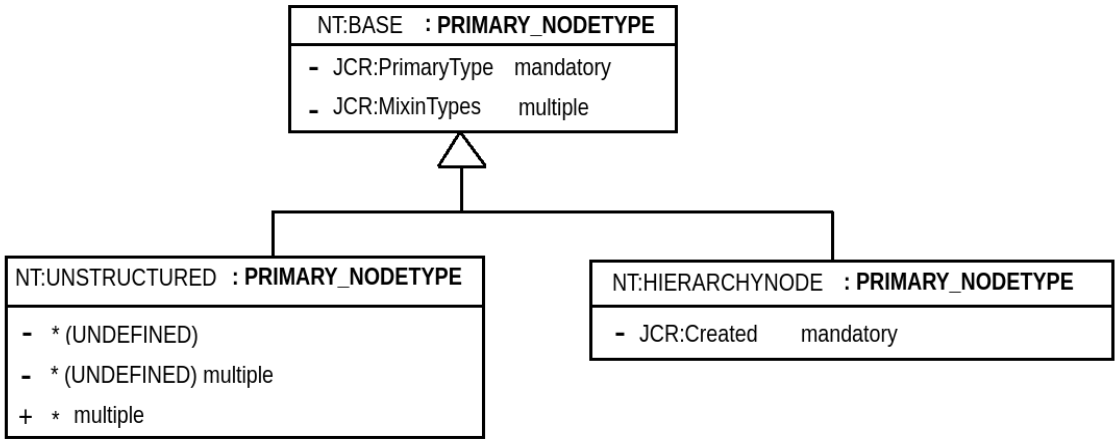

Figura 1.5: Esempio di alcuni sottotipi del nodetype primario nt:base, rappresentati mediante un diagramma degli oggetti in stile UML.

1.2.4 Contenuto non strutturato

Un repository può supportare il nodetype primario nt:unstructured.
Se il repository supporta i dati non strutturati, deve supportare anche il tipo di proprietà UNDEFINED.
Se un nodo ha come nodetype primario nt:unstructured e zero mixin nodetype assegnati, può essere il padre di qualsiasi proprietà e nodo, senza nessun vincolo.
In questo caso il contenuto del nodo è completamente non strutturato.

1.2.5 Contenuto semi-strutturato:

Se un nodo ha il nodetype primario nt:unstructured e uno o più mixin nodetype assegnati, può essere il padre di qualsiasi proprietà e nodo, ma deve rispettare i vincoli dei suoi mixin nodetype.
In questo caso il contenuto del nodo è semi-strutturato.

1.2.6 Contenuto strutturato:

Se un nodo ha il nodetype primario diverso da nt:unstructured, deve rispettare tutti i vincoli del nodetype primario, e non può essere il padre di nessuna proprietà e nodi aggiuntivi.
In questo caso il contenuto del nodo completamente è strutturato.

1.2.7 Nodi referenziabili:

Un repository può supportare nodi referenziabili.
Un nodo deve avere il mixin nodetype mix:referenceable per poter essere referenziato da una proprietà di tipo REFERENCE o WEAKREFERENCE, la quale ha come valore l'identificatore del nodo referenziato.
Il mixin nodetype mix:referenceable aggiunge al nodo referenziabile una nuova proprietà chiamata jcr:uuid, che è obbligatoria, autocreata e protetta.
Questa proprietà assegna al nodo referenziabile un identificatore univoco all'interno del workspace.

1.3 JAVA API

In seguito vengono descritte le API Java definite nello standard JCR.
Le api java si trovano nel package javax.jcr, e offrono tutti i metodi necessari per interagire col repository.

1.3.1 Connessione al repository

Per potersi connettere ad un repository JCR, un'applicazione deve ottenere per prima cosa un oggetto *Repository*. Le API Java mettono a disposizione molti metodi che restituiscono un oggetto *Repository,* uno dei quali è:

Repository JcrUtils.getRepository(“repository url”);

Per poter interagire col contenuto di un workspace, l'applicazione ha bisogno di una sessione connessa ad un workspace. L'oggetto *Repository* offre il metodo login(), che restituisce all'applicazione un oggetto *Session:*

Session Repository.login(Credentials credentials, String workspaceName)

L'applicazione può quindi richiamare il metodo login() passando come parametri le credenziali utente del repository e il nome del workspace, per ottenere una sessione collegata a quel workspace.

1.3.2 Lettura del contenuto

Ogni sessione, indipendentemente dal suo livello di autorizzazione, può accedere al repository in 3 modi diversi di lettura:

- Accesso diretto;
- Accesso tramite attraversamento;
- Accesso tramite query.

L'Accesso diretto a un item si ha quando si può accedere ad esso tramite il suo percorso assoluto, il suo percorso relativo o il suo identificatore.
Esempi di accesso diretto:

Node Session.getNode(String absPath)
Property Session.getProperty(String absPath)
Node Session.getNodeByIdentifier(String identifier)
Node Node.getNode(String relPath)
Property Node.getProperty(String relPath)

L'accesso tramite attraversamento a un item si ha quando si iterano tutti i figli di un nodo.

Esempi di accesso tramite attraversamento:

NodeIterator Node.getNodes()
PropertyIterator Node.getProperties()

Un repository può supportare l'accesso tramite query in 2 possibili linguaggi:

- JCR-SQL2, che esprime la query con un linguaggio simile all'SQL
- JCR-JQOM (JCR Java Query Object Model), che esprime la query tramite un albero di oggetti java

1.3.3 Esportazione del contenuto

Un repository JCR deve supportare l'esportazione dei contenuti in 2 formati XML: system view e document view.
L'esportazione tramite system view è completa e contiene tutto il contenuto del repository, e genera un file XML di grandi dimensioni.
L'esportazione tramite document view è più "human readable", ma ha una perdita di informazioni.
Per entrambi i tipi di esportazione è possibile scegliere il nodo sorgente, se esportare ricorsivamente tutti i nodi figli del nodo sorgente e se esportare anche i valori delle proprietà binarie.
Esempio di esportazione in formato system view:

```
void    Session.exportSystemView(String    absPath,
                                 OutputStream out,
                                 boolean      skipBinary,
                                 boolean noRecurse)
```

Esempio di esportazione in formato document view:

```
void    Session.exportDocumentView(String    absPath,
                                   OutputStream out,
                                   boolean      skipBinary,
                                   boolean noRecurse)
```

1.3.4 Scrittura del contenuto

Un repository può supportare la scrittura del contenuto.
Per determinare se un repository supporta la scrittura, si interroga il repository descriptor table. Se l'attributo Repository.WRITE_SUPPORTED è true, il repository supporta la scrittura del contenuto, altrimenti è soltanto leggibile.

1.3.5 Tipi di scrittura

Un metodo di scrittura JCR può essere fatto a livello di sessione o a livello di workspace.
Esempi di metodi Session-Write:

Node.addNode(),	(Aggiunge un nodo figlio al nodo corrente)
Node.setProperty(),	(Imposta una proprietà al nodo corrente)
Session.importXML().	(Importa il contenuto nel workspace tramite XML)

Esempi di metodi Workspace-Write:

Session.save()	(Salva le modifiche fatte da una sessione in modo persistente)
Workspace.importXML().	*(*Importa il contenuto nel workspace tramite XML*)*

Le scritture possono essere fatte dentro una transazione o in assenza di transazioni. Le modifiche fatte da una sessione sono salvate in uno spazio temporaneo associato alla sessione

corrente (transient Storage).
La scrittura nello spazio temporaneo non controlla i vincoli dei nodetype ad ogni operazione eseguita, quindi gli item possono anche essere non validi temporaneamente.
Una modifica che è salvata nello spazio temporaneo della sessione è nello stato “pending”.
Per salvare le modifiche di una sessione in modo persistente, si chiama il metodo Session.save(). Dopo il salvataggio, le modifiche entrano nello stato “dispatched”. Se durante il salvataggio uno o più item non rispettano i vincoli di struttura del loro nodetype primario, viene generata l'eccezione InvalidItemStateException.
Le modifiche fatte tramite workspace entrano immediatamente nello stato “dispatched”.
In assenza di transazioni, tutte le modifiche che sono nello stato “dispatched” vengono direttamente salvate in modo persistente, quindi entrano nello stato “persisted”.
Le modifiche di una transazione che sono nello stato “dispatched”, vengono salvate in uno spazio temporaneo dedicato alla transazione (transaction-context). Se le modifiche fanno parte di una transazione, entrano nello stato persisted dopo il commit della transazione.
Tutte le modifiche che sono nello stato “pending” e “dispatched” sono visibili solo dalla sessione che le ha generate, perché ancora non sono salvate nel workspace in modo persistente. Tutte le modifiche che sono nello stato “persisted” sono salvate in modo persistente nel workspace, e sono quindi visibili da tutte le altre sessioni.
Le figure 1.6 e 1.7 mostrano, rispettivamente, come avviene la scrittura in assenza di transazioni e in presenza di transazioni.
La figura 1.8 mostra gli stati in cui può essere una modifica fatta al contenuto di un workspace, tramite un diagramma degli stati di UML.

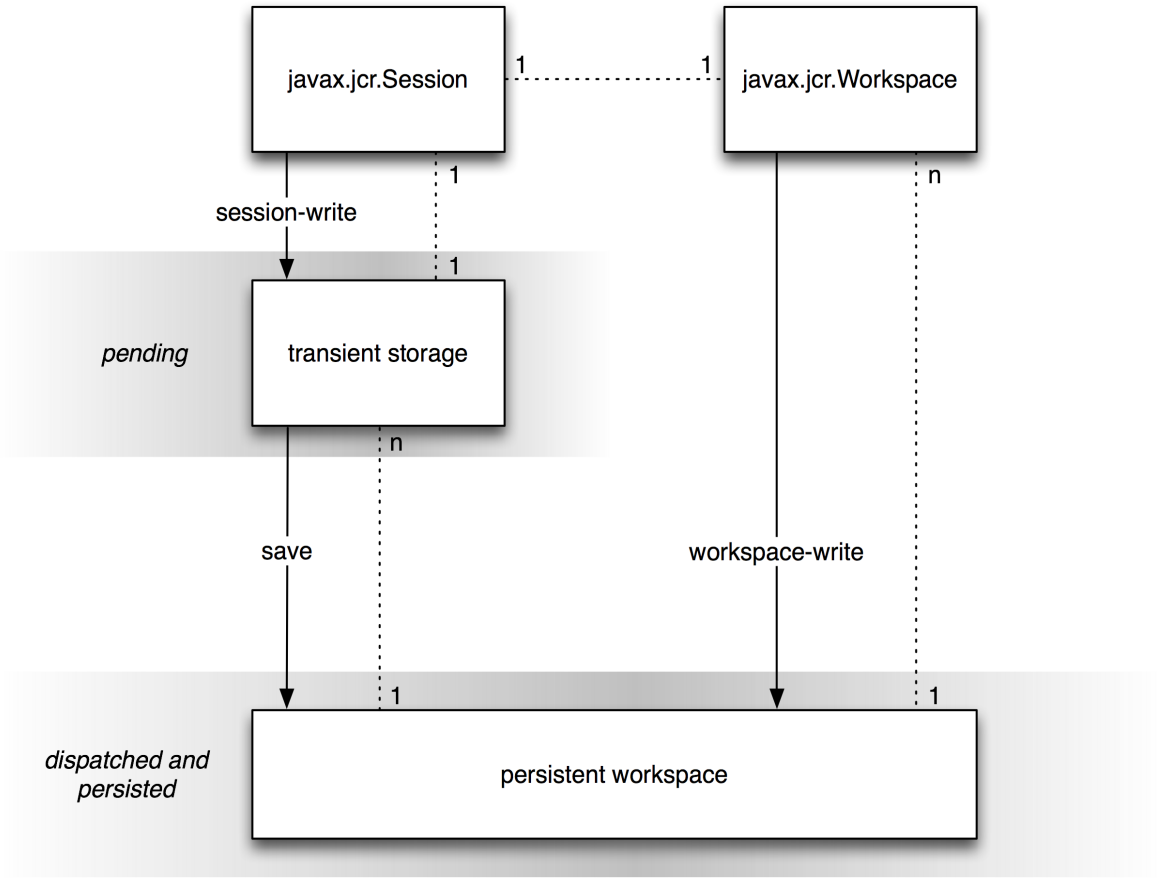

Figura 1.6: In assenza di transazioni, se una modifica entra nello stato DISPATCHED viene subito salvata nel workspace in modo persistente, entrando nello stato PERSISTED.

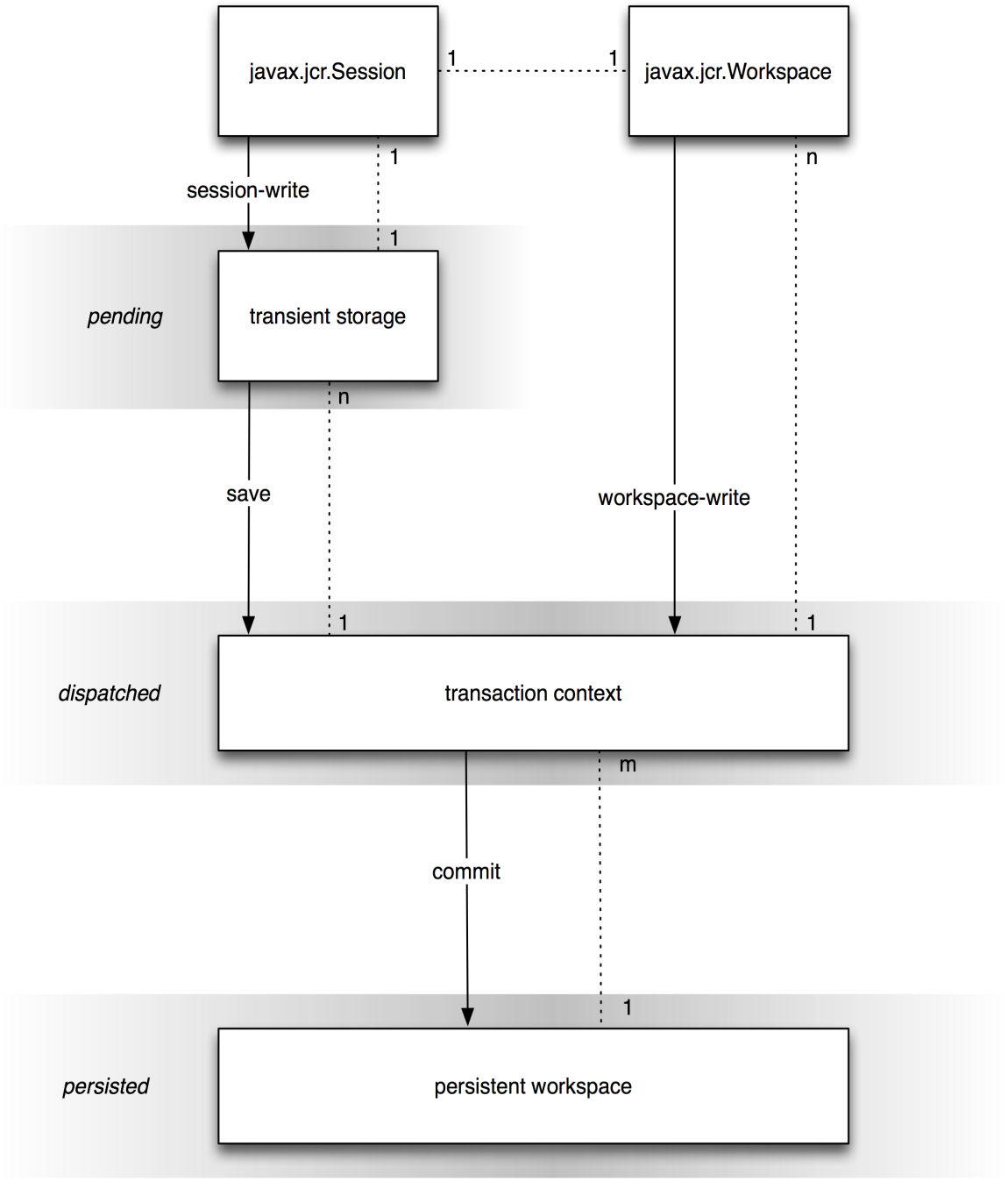

Figura 1.7: In presenza di transazioni, una modifica che entra nello stato DISPATCHED viene salvata in uno spazio temporaneo dedicato alla transazione.
Solo quando la transazione fa commit, la modifica viene salvata nel workspace in modo persistente.

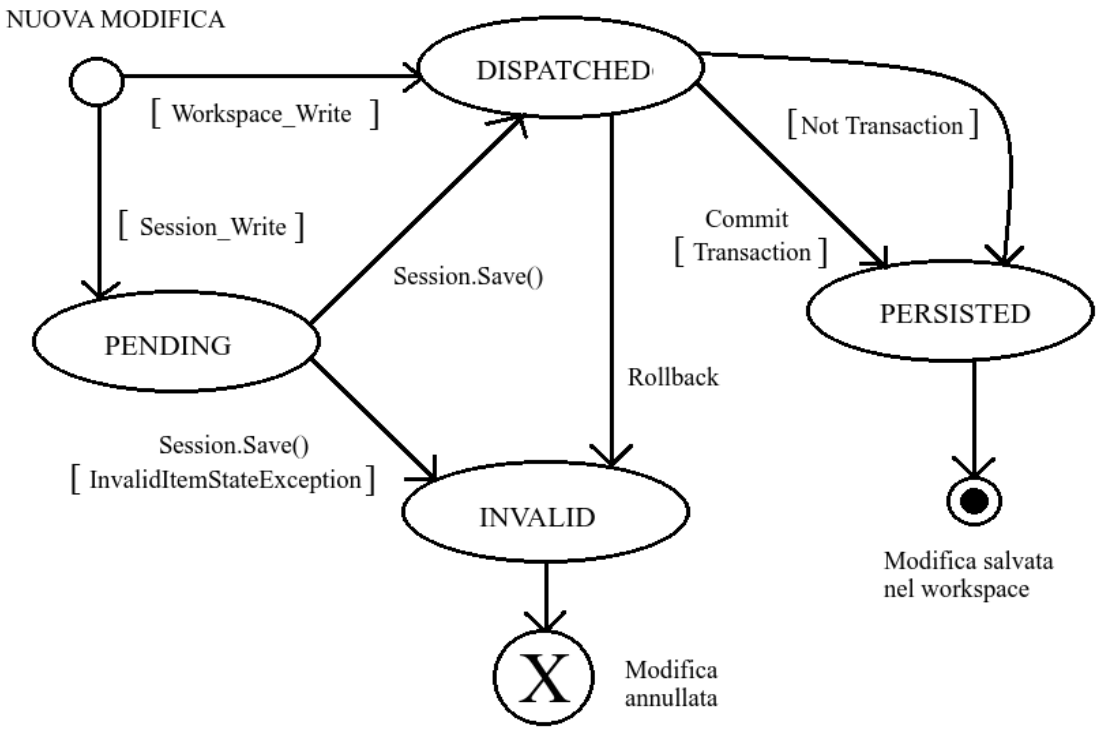

Figura 1.8: Diagramma degli stati UML di una modifica fatta da un'applicazione al workspace.
L'attributo Transaction, se true, indica che la modifica è stata fatta all'interno di una transazione.
Una modifica viene salvata nel workspace in modo persistence se e solo se raggiunge lo stato PERSISTED.

1.3.6 Importazione del contenuto

Un repository può supportare l'importazione del contenuto da XML.
Per determinare se un repository supporta l'importazione si interroga il repository descriptor table. Se l'attributo Repository.OPTION_XML_IMPORT_SUPPORTED è true, il repository supporta l'importazione del contenuto da XML. In questo caso, deve supportare l'importazione sia in formato system view che in formato document view.

L'importazione consiste in una scrittura del contenuto nel workspace, e può essere fatta sia a livello di Session che a livello di Workspace, con i seguenti metodi:

void Workspace.importXML(String parentAbsPath, InputStream in, int uuidBehavior)
void Session.importXML(String parentAbsPath, InputStream in, int uuidBehavior)

Il parametro uuidBehavior definisce come sono gestiti gli identificatori dei nodi durante l'importazione del contenuto, in quanto si potrebbero verificare collisioni tra i nodi importati e inodi preesistenti.
L'interfaccia javax.jcr.ImportUUIDBehavior definisce 4 opzioni:

- IMPORT_UUID_CREATE_NEW:
  Ad ogni nodo proveniente dall'importazione XML, il repository assegna un nuovo identificatore; in questo modo non si verificano mai collisioni di identificatori.

- IMPORT_UUID_COLLISION_REMOVE_EXISTING:
  Se un nodo proveniente dall'importazione ha lo stesso identificatore di un nodo esistente, il repository rimuove il nodo esistente (e tutto il suo sottografo) prima di fare l'importazione del nuovo nodo.

- IMPORT_UUID_COLLISION_REPLACE_EXISTING:
  Se un nodo proveniente dall'importazione ha lo stesso identificatore di un nodo esistente, il repository sostituisce il nodo esistente col nuovo nodo.

- IMPORT_UUID_COLLISION_THROW:
  Se un nodo proveniente dall'importazione ha lo stesso identificatore di un nodo esistente, il repository genera un'eccezione di tipo ItemExistsException

1.3.7 Osservazione del contenuto

Un repository può supportare l'osservazione del contenuto, che permette alle applicazioni di ricevere una notifica ogni volta che nel workspace si verificano modifiche persistenti (che entrano cioè nello stato "persisted").
Lo standard JCR definisce un modello a eventi generale e le API specifiche per effettuare l'osservazione del contenuto asincrona e tramite journal.
Per determinare se un repository supporta l'osservazione del contenuto, si interroga il repository descriptor table.
Se l'attributo Repository.OPTION_OBSERVATION_SUPPORTED è true, il repository supporta l'osservazione del contenuto asincrona.
Se l'attributo Repository.OPTION_JOURNALED_OBSERVATION_SUPPORTED è true, il repository supporta l'osservazione del contenuto tramite journal.
Una modifica persistente al workspace è rappresentata da un insieme di uno o più eventi.
Ogni evento riporta una semplice modifica alla struttura del workspace.
I 6 tipi di eventi standard sono:

- NODE_ADDED;
- NODE_MOVED;
- NODE_REMOVED;
- PROPERTY_ADDED;
- PROPERTY_REMOVED;
- PROPERTY_CHANGED.

Nel caso dell'osservazione del contenuto tramite journal, è presente anche il tipo di evento PERSIST, che viene generato dopo una chiamata al metodo *Session.save()*.
Ogni evento è definito dalle seguenti informazioni:

- Il tipo dell'evento;
- il percorso dell'item interessato;

- l'identificatore dell'item interessato;
- la data, espressa come un offset dall'epoca 1 gennaio, 1970 00:00:00.000 GMT (Gregorian). E' un valore in millisecondi, la cui granularità dipende dall'implementazione del repository:

1.3.8 Osservazione del contenuto asincrona

Se un'applicazione osserva il contenuto di un workspace in modo asincrono, riceve una notifica appena agli eventi si verificano.
L'osservazione del contenuto si dice asincrona in quanto l'operazione che genera l'evento non deve aspettare una risposta da parte dell'applicazione.
Per osservare il contenuto in modo asincrono, l'applicazione registra un oggetto di tipo *EventListener* sul workspace selezionato.
La registrazione dell'oggetto *EventListener* viene effettuata dall'oggetto *ObservationManager,* acquisito tramite l'oggetto *Workspace* col metodo:

ObservationManager Workspace.getObservationManager()

Per registrare l'oggetto *EventListener,* l'oggetto *ObservationManager* mette a disposizione il seguente metodo:

void ObservationManager.addEventListener(
EventListener listener,
int eventTypes,
String absPath,
boolean isDeep,
String[] uuid,
String[] nodeTypeName,
boolean noLocal)

I parametri di questo metodo filtrano gli eventi che l'applicazione desidera ricevere:

- *EventListener listener:*
  L'oggetto listener viene passato dall'applicazione e deve implementare il metodo
  void EventListener.onEvent(EventIteratorevents)
  come richiesto dall'interfaccia *EventListener*.
  Ogni volta che si verifica una modifica nel contenuto del workspace, viene chiamato il metodo *onEvent(),* al quale sono passati come parametri tutti gli eventi che si sono verificati.

- *int eventTypes:*
  Indica per quali tipi di eventi si vuole ricevere una notifica.
  Ad esempio, se al parametro viene assegnato il valore:
  eventTypes = Event.NODE_ADDED | Event.PROPERTY_ADDED;
  viene chiamato l'evento EventListener.onEvent solo quando viene aggiunto un nodo o una proprietà al workspace.

- *String absPath:*
  Indica il percorso del nodo del nodo sorgente di cui si vogliono ricevere gli eventi

- *boolean isDeep:*
  Se true vengono ricevuti gli eventi anche per il sottografo del nodo sorgente

- *String[] uuid:*
  Si ricevono gli eventi solo dei nodi che hanno l'identificatore nell'array uuid.
  Se è null, questo filtro non viene considerato.

- *String[] nodeTypeName:*
  Si ricevono gli eventi solo dei nodi che hanno il nodetype contenuto nell'array

nodeTypeName.
Se è null, questo filtro non viene considerato.

- *boolean noLocal:*
  Se true non vengono ricevuti gli eventi generati dalla stessa sessione con la quale è stato registrato l'event listener.

1.3.9 Osservazione del contenuto tramite journal

L'osservazione del contenuto tramite journal permette ad un applicazione di connettersi ad un workspace e avere un report di modifiche che si sono verificate in un determinato lasso di tempo. Il journal è un log in cui il repository scrive tutte le modifiche salvate in modo persistente in un workspace. Nel journal sono contenuti tutti gli eventi che si sono verificati dalla creazione del workspace, quindi nel tempo può diventare di grandi dimensioni.
Per leggere tutti gli eventi, l'applicazione deve prendere l'oggetto *EventJournal,* restituito dall'oggetto *ObservationManager,* tramite il seguente metodo:

EventJournal ObservationManager.getEventJournal(
int eventTypes,
String absPath,
boolean isDeep,
String[] uuid,
String[] nodeTypeName,
boolean noLocal).

I parametri di questo metodo sono gli stessi del metodo *ObservationManager.addEventListener* (vedi paragrafo 1.3.8).

L'oggetto *EventJournal* permette di scegliere da quale data leggere il log degli eventi, scartando gli eventi che si sono verificati prima, tramite il metodo:

void EventJournal.skipTo(Calendar date)

CAPITOLO 2: CONFRONTRO FRA JCR E RDBMS

2.1 MODELLI DI DATI

Un sistema informativo è costituito dall'insieme delle informazioni utilizzate, prodotte e trasformate da un'azienda durante l'esecuzione dei processi aziendali.
Un sistema informatico è la parte del sistema informativo che, tramite tecnologie informatiche, automatizza processi aziendali e può offrire un supporto alle decisioni al verificarsi di nuovi problemi.
Alla base di un sistema informatico risiedono i dati, che contengono tutte le informazioni della realtà per cui è stato progettato il sistema informatico.
Fino agli anni sessanta, tutti i dati di un sistema informatico venivano memorizzati in nastri magnetici, che consentivano la lettura e la scrittura dei dati solo tramite accesso sequenziale.
Negli anni sessanta, con la comparsa dei dischi magnetici ad accesso diretto, è cambiata l'interazione con i dati da parte dei sistemi informatici, i quali potevano leggere un dato in poco tempo, a differenza dei nastri magnetici che richiedevano operazioni di lettura giornaliere.
L'introduzione dei dischi ad accesso diretto nei sistemi informatici ha quindi permesso un'interattività sempre maggiore con i dati memorizzati, i quali venivano letti e scritti sempre con maggiore frequenza.
Dato che i sistemi informatici dovevano memorizzare una grande quantità di informazioni, a partire dagli anni sessanta sono stati progettati diversi modelli di dati, per organizzare i dati secondo una struttura definita. Storicamente, i primi modelli di dati progettati sono stati:

- Il modello gerarchico;
- Il modello reticolare;
- Il modello relazionale.

Tra questi, il modello di dati che ha avuto maggior successo è il modello relazionale, il quale memorizza i dati in una struttura ben definita. Recentemente sono stati progettati altri modelli più adatti alla gestione di diversi problemi, tra i quali il modello JCR e altri modelli facenti parte del movimento NOSQL (Not Only SQL).

2.1.1 Modello gerarchico

Il modello gerarchico prevede che i dati siano organizzati secondo una struttura ad albero, che riflette una gerarchia esistente tra le entità che appartengono al database (*vedi figura 2.1*).
I dati vengono rappresentati tramite relazioni 1:N, dove un padre può avere molti figli, ma un figlio può avere un solo padre. Storicamente, il modello gerarchico è stato il primo modello di database ad affermarsi sul mercato, introdotto da IBM col database IMS.
Questo modello di dati non ha avuto molto successo, perché sono pochi i problemi reali rappresentabili tramite un modello gerarchico puro. [4]

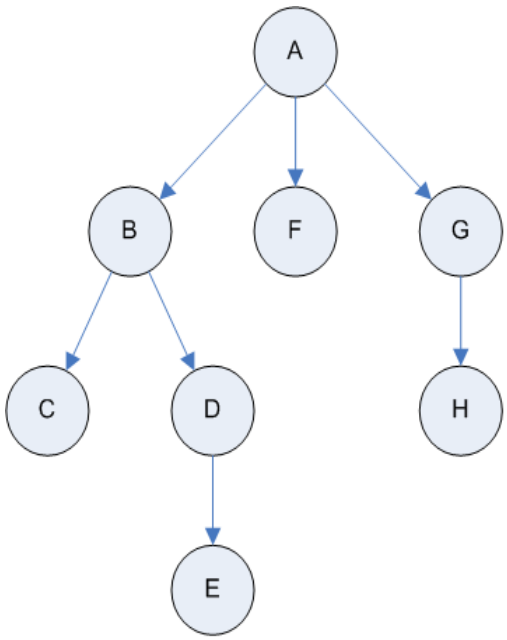

Figura 2.1: schema del modello di dati gerarchico.

2.1.2 Modello reticolare

Il modello reticolare prevede che i dati siano rappresentati tramite un grafo orientato (*vedi figura 2.2*). La logica del modello reticolare è basata su record e puntatori, consentendo ai record di collegarsi in qualsiasi direzione. Questo modello ha trovato implementazioni in diversi settori dell'informatica, quali ad esempio la gestione degli oggetti in memoria. [5]

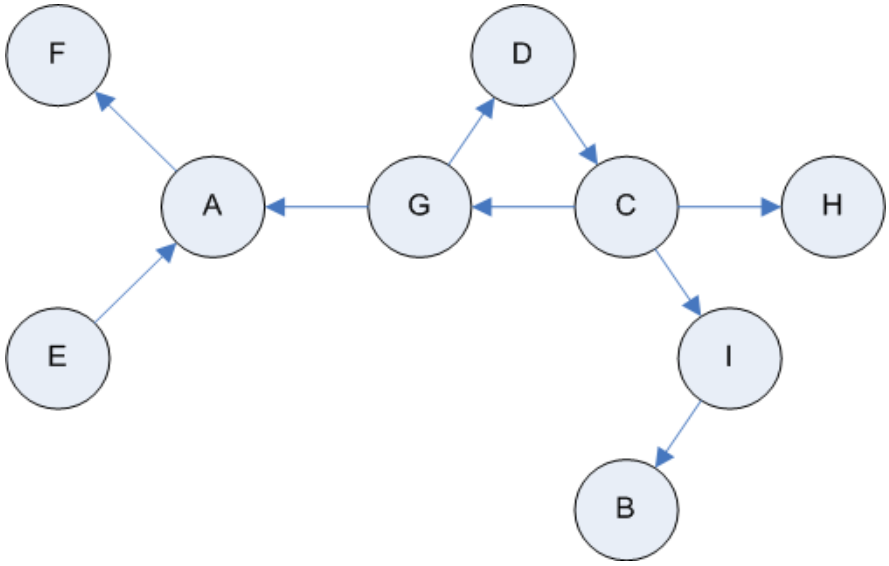

Figura 2.2: schema del modello di dati reticolare.

2.1.3 Modello relazionale

Il modello relazionale è basato sulla teoria degli insiemi, e definisce i concetti di relazione, dominio, tuple e attributi. È stato proposto da Codd negli anni '70, per semplificare la scrittura di interrogazioni sui database e per favorire l'indipendenza dei dati. [6]
È il modello di dati che ha avuto più successo, dovuto sia ai suoi fondamenti matematici che alla soluzione pratica che offriva per risolvere molti problemi incontrati negli anni settanta, ottanta e novanta. Inoltre, grazie al principio della normalizzazione, consentiva un risparmio dello spazio riservato all'immagazzinamento dei dati. [7]

Il modello relazionale prevede che la struttura sia completamente separata dai dati.
Questo porta vantaggi in fase di progettazione, in quanto il database viene progettato per far rispettare tutti i vincoli del problema. Tutti i record devono avere la stessa la struttura delle relazioni, altrimenti non possono essere memorizzati nel database.
Durante il ciclo di vita della base di dati, il problema potrebbe evolvere e portare ulteriori vincoli, che si ripercuotono nella modifica della struttura del database. La gestione della struttura può portare a diversi problemi e diventa più difficoltoso gestire i dati preesistenti.

2.1.4 Modello JCR

Il modello JCR eredita caratteristiche sia dal modello gerarchico che dal modello reticolare.
Come il modello gerarchico, è basato sulla relazione padre – figlio dei nodi.
Supera i vincoli del modello gerarchico perché permette ai nodi di fare riferimento ad altri nodi orizzontalmente (non hanno il ruolo di padre, ma sono raggiungibili grazie al riferimento a senso unico). [7]
In figura 2.3 viene mostrato lo schema del modello JCR.
La struttura del modello JCR si forma con l'aggiunta del contenuto nel database.
Tramite il nodetype primario e i mixin nodetype, è possibile specificare, per ogni nodo, la sua struttura (vedi paragrafo 1.2.2). Quindi, la struttura finale del contenuto di un repository JCR dipende dal contenuto del repository.
Il modello di dati JCR sta avendo un successo crescente nei sistemi di gestione dei contenuti (CMS), perché riflette la stessa struttura del contenuto di siti web.
La figura 2.4 mostra come un repository JCR sia adatto a memorizzare il contenuto di un sito web.

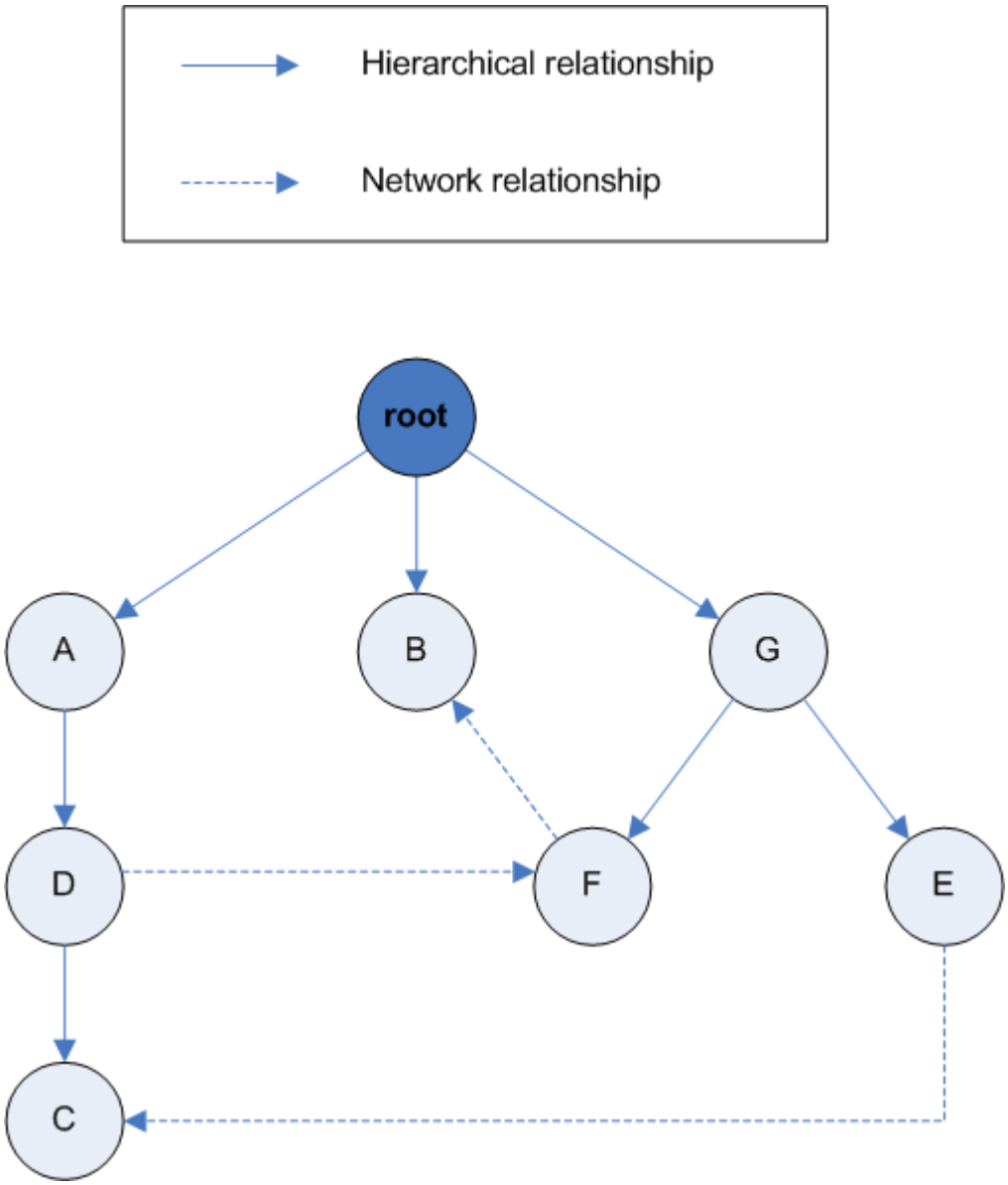

Figura 2.3: schema del modello di dati JCR. Gli archi continui esprimono una relazione gerarchica, mentre gli archi discontinui esprimono una relazione reticolare

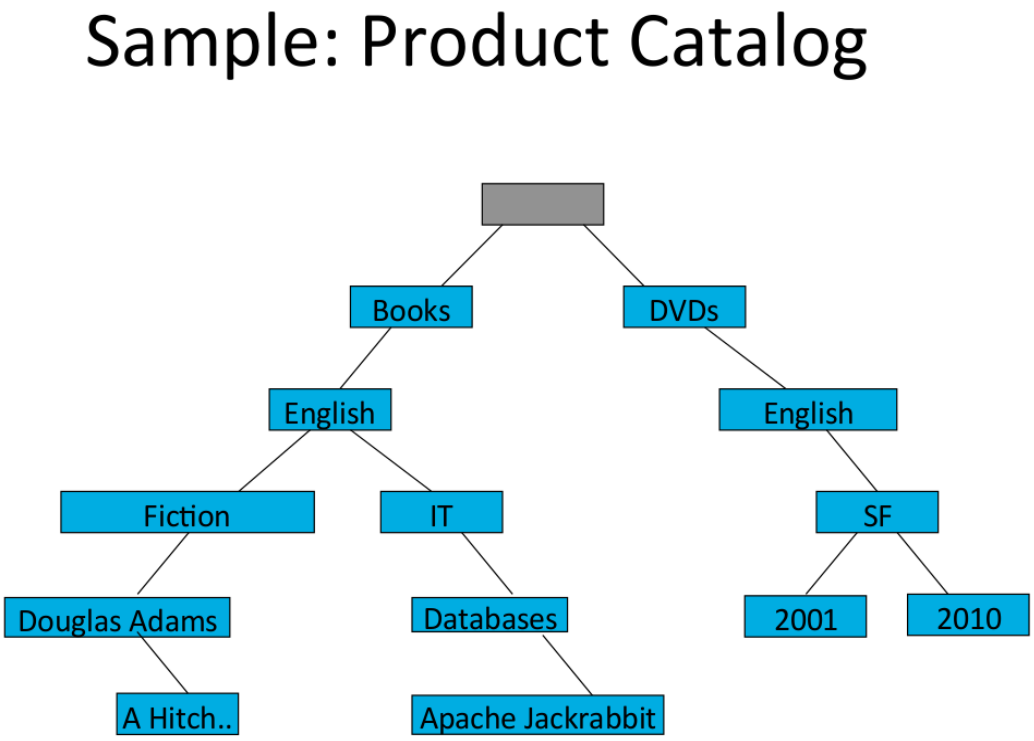

Figura 2.4: Esempio del contenuto di un sito web.

Il modello JCR in questo caso è adatto alla memorizzazione del contenuto.

2.2 RUOLI DI RESPONSABILITÀ

La base di dati di un sistema informatico viene controllata e gestita dal database administrator (DBA), che ha il pieno controllo del database e si occupa della sua amministrazione, manutenzione, aggiornamenti e sicurezza.
Le applicazioni del sistema informatico si interfacciano alla base di dati per poter offrire le funzionalità all'utente finale, con le quali può leggere il contenuto del database ed eventualmente modificarlo.
Per ogni base di dati si possono identificare 3 diversi ruoli, ognuno dei quali ha diverse responsabilità per ogni caratteristica del database:

- Database administrator (DBA);
- Programmatore delle applicazioni;
- Utente finale.

A differenza del modello usato per la base di dati, può cambiare il dominio di responsabilità per ogni ruolo del database.
Nel confronto fra il modello JCR e il modello relazionale, si possono considerare alcune caratteristiche di un database, e vedere, per ogni ruolo, come cambia il dominio di responsabilità in base al modello di dati scelto per il database.
Il confronto dei domini di responsabilità viene effettuato secondo le principali caratteristiche di un database: [7]

- Contenuto
  Chi è responsabile della lettura e scrittura dei dati nel database?

- Struttura
  Chi è responsabile della struttura del database?

- Integrità dei dati
  Chi è responsabile dell'integrità dei dati?
  (I dati sono integri all'interno di un database se, quando vengono inseriti, rispettano i vincoli definiti dalla struttura del database).

- Coerenza dei dati
  Chi è il responsabile della coerenza dei dati?
  La coerenza dei dati deve essere garantita anche in caso di guasti al database e di successivo ripristino.
  (Il database è in uno stato coerente se tutti i dati memorizzati rispettano i vincoli definiti dalla struttura del database).

2.2.1 Ruoli di responsabilità nel modello relazionale

Nel modello relazionale la struttura è ben definita e separata dai dati, e la gestione dei ruoli è semplice.
Il DBA è il proprietario del database e di tutte le strutture che permettono di memorizzare i record. Il suo dominio di responsabilità è massimo.
Il programmatore, in questo contesto, ha la possibilità di modificare parte della struttura per aggiornare i vincoli della base di dati, in seguito alle nuove richieste dell'utente o all'evolversi del problema.
Il dominio di responsabilità del programmatore riguarda il contenuto e parte della struttura.
L'utente può leggere e scrivere i dati nel database tramite l'applicazione, quindi il suo dominio di responsabilità riguarda solo il contenuto dei dati.
La figura 2.5 mostra la relazione fra ruoli e domini di responsabilità, nel caso di una base dati che implementa il modello relazionale.[7]

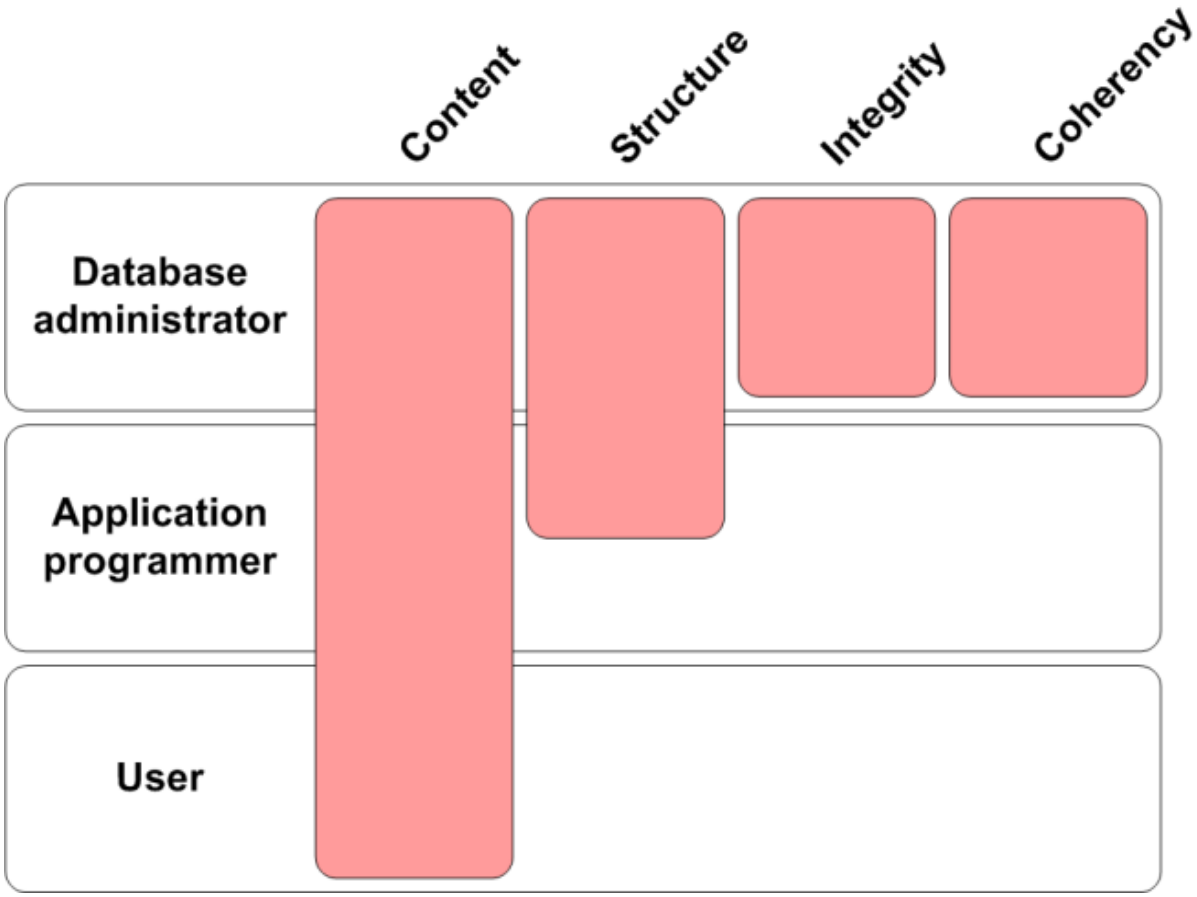

Figura 2.5: Grafico che mostra il dominio di responsabilità, per ciascun ruolo, di un database relazionale.

2.2.2 Ruoli di responsabilità nel modello JCR

Nel modello JCR la struttura segue il contenuto dei dati, quindi il dominio di responsabilità dei 3 ruoli cambia rispetto al modello relazionale.

Il DBA non è più il solo responsabile della struttura dei dati, che è a carico sia del programmatore che dell'utente, ma rimane comunque il solo responsabile dell'integrità e della coerenza dei dati. In questo contesto, tramite l'applicazione, l'utente finale può definire nuovi nodetype e inserire nel repository JCR un nuovo tipo di contenuto, andando a modificare la struttura del database.

La figura 2.6 mostra il dominio di responsabilità dei 3 ruoli nel caso di una base dati che *implementa il modello JCR. [7]*

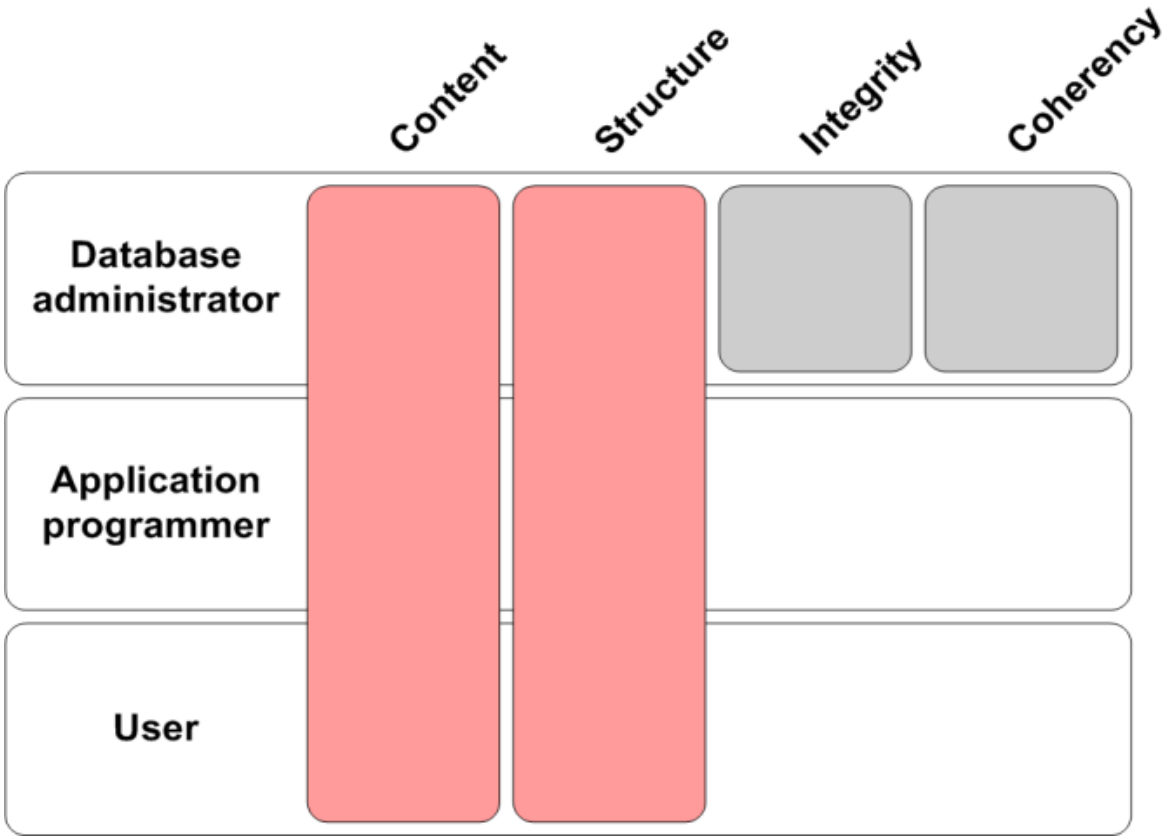

Figura 2.6: Grafico che mostra il dominio di responsabilità, per ciascun ruolo, di un repository JCR. A differenza del modello relazionale, la struttura del database diventa una responsabilità anche del programmatore e dell'utente.

2.3 CONFRONTO TRA I DUE MODELLI

È meglio avere una base di dati che implementa il modello relazionale o il modello JCR?
Il confronto fra i 2 modelli fa parte di un più ampio dibattito sviluppatosi anni fa sia in ambienti accademici che nel web: [7]

Deve essere la struttura a seguire i dati, o devono essere i dati a seguire la struttura?

La realtà, è che entrambi i modelli portano sia vantaggi che svantaggi, e bisogna scegliere il modello della base di dati che è più adatto alla gestione di un problema. Un possibile criterio per la scelta del modello, riguarda il livello di conoscenza di un problema. Se in anticipo tutti i vincoli del problema sono noti, è meglio scegliere il modello relazionale, che definisce la struttura della base dati. Se il problema invece non è conosciuto a fondo, potrebbe essere meglio il modello JCR, dove la struttura si forma in base al contenuto dei dati ed è più flessibile.
La seguente analogia aiuta a capire meglio quando usare i 2 modelli: [7]

Una casa ha una struttura ben definita, i muratori non possono costruire una casa senza un progetto ben fatto. Lo sviluppo di una città, d'altro canto, non segue un progetto fisso. Per una città non è possibile sapere in anticipo quante case saranno presenti, e quali zone saranno più abitate di altre. In questo caso, si lascia che la struttura della città emerga da sola, in seguito alla costruzione delle singole case.

Facendo riferimento all'analogia, se tutti i vincoli di un problema sono ben fissati, come lo è il progetto di una casa, allora il modello relazionale è la scelta più adatta. Se invece alcuni vincoli del problema non sono noti, o non si conosce a fondo il problema, può essere più adatto un repository JCR, strutturando i vincoli noti temporaneamente ma lasciando la possibilità all'applicazione e all'utente di definire nuove strutture (quindi nuovi nodetype).
In questo caso la struttura finale del database si sviluppa come una città, casa dopo casa, senza però sapere in anticipo la struttura finale.

Un altro criterio per la scelta del modello da utilizzare, può essere suggerito dalle seguenti domande: [7]

- Si conosce in anticipo la tipologia di utenti dell'applicazione?
- Il comportamento degli utenti è conosciuto?
- L'uso finale dell'applicazione è noto?
- Le entità possiedono una loro struttura naturale?

Le risposte a queste domande aiutano a capire meglio qual è il modello di dati che meglio si appresta a gestire il problema.

Per concludere:

- Nel modello relazionale i dati sono guidati da una struttura ben definita Non è possibile inserire un record che non soddisfa la struttura della relazione.

- Nel modello JCR i dati non richiedono una struttura preesistente per poter essere salvati. La struttura viene definita dal contenuto.

CAPITOLO 3: APACHE JACKRABBIT

Apache Jackrabbit è un'implementazione completa del modello del repository astratto definito nello standard JCR. Jackrabbit è open-source e sviluppato dall' Apache Software Foundation. Essendo un progetto open-source non esiste un supporto tecnico, quindi gli utenti che utilizzano jackrabbit possono riportare eventuali bug nella mailing list a loro dedicata (users@jackrabbit.apache.org). [8]

3.1 VERSIONI

Dal sito ufficiale di Apache Jackrabbit (http://jackrabbit.apache.org) si possono scaricare 2 versioni diverse di Jackrabbit, oltre a tutto il codice sorgente:

- standalone server
- web application

Le caratteristiche delle due versioni saranno descritte nei paragrafi seguenti.

3.1.1 Standalone server

Lo standalone server è un pacchetto jar eseguibile contentente al suo interno tutte le librerie java usate da jackrabbit. È il modo più semplice per iniziare ad usare Jackrabbit, in quanto è sufficiente eseguire il pacchetto jar per avere un repository JCR funzionante.
Per eseguire lo standalone server bisogna utilizzare il seguente comando (tramite una shell linux):

java -jar <percorso-standalone-server.jar> <parametri>

Se non viene specificato nessun parametro, il server cerca la cartella del repository dentro alla cartella ./jackrabbit (il percorso relativo dipende dalla working directory della shell), e un file di configurazione del repository, chiamato repository.xml. Se il server non trova né la cartella né il file di configurazione, li crea in automatico con le configurazioni di default. Se non viene specificato nessun parametro, di default lo standalone-server rende disponibile il repository nell'url:
http://<host-name>:8080/
tramite il quale è possibile vedere il contenuto del repository tramite un browser web.
Tramite i parametri è possibile cambiare il numero di porta di default (8080) e specificare il percorso della cartella del repository.

3.1.2 Web application

L'applicazione web di Jackrabbit è un archivio in formato war (web application archive) che è eseguibile da un qualsiasi software che implementa le specifiche Java Servlet [9] e JavaServer Pages [10], definite nelle JSR del Java Community Process. Ad esempio, può essere eseguito dal web server Apache Tomcat (vedi paragrafo 5.2.1).
Rispetto alla versione standalone è più gestibile e permette un maggiore controllo delle operazioni svolte da Jackrabbit. Al suo interno contiene le servlet necessarie per collegarsi ad un repository, definite nel file di configurazione web.xml. La servlet che l'applicazione web di Jackrabbit mette a disposizione per avviare un repository si chiama RepositoryStartupServlet.java. Dai file di configurazione dell'applicazione web di Jackrabbit è possibile specificare in quale percorso si trova la cartella del repository. Se la servlet non trova nessun repository nella cartella specificata, ne crea uno nuovo con le configurazioni di default.

3.1.3 Correzione bug web application

L'applicazione web di Jackrabbit scaricabile dal sito ufficiale di Apache presenta un bug: non è possibile importare il contenuto in formato XML tramite il metodo Session.importXML(), a causa di un errore nella lettura dei parametri. In particolare, il metodo Session.importXML() chiama al suo interno una libreria java che ha il compito di leggere il nome del workspace dall'url del repository. Questa libreria si chiama "jackrabbit-webdav", e per correggere il bug bisogna modificare una sua classe definita nel file "AbstractLocatorFactory.java".
Per risolvere il bug è sufficiente aggiungere sotto a queste righe di codice: [11]

```
if (pathPrefix != null && pathPrefix.length() > 0)
{
        if (!b.toString().endsWith(pathPrefix))
        {
                b.append(pathPrefix);
        }
        if (href.startsWith(pathPrefix))
        {
                href = href.substring(pathPrefix.length());
        }
```

le seguenti istruzioni:

```
/*Added new code start*/
if(!href.startsWith(pathPrefix)&& href.contains(pathPrefix))
{
        href = href.substring(href.indexOf(pathPrefix +
                                                pathPrefix.length());
}
/*Added new code end*/
```

Questo bug è stato risolto grazie all'utente "Shankar Pednekar", che riporta la soluzione per risolvere questo bug in un commento presente in questo "bug report":
https://issues.apache.org/jira/browse/JCR-3763

Nonostante sia stata trovata la soluzione a questo bug, il team di sviluppo di Apache non ha ancora corretto l'applicazione web di Jackrabbit scaricabile dal sito ufficiale.
Per compilare il codice sorgente e correggere quindi la libreria, è necessario avere installato Maven, un software di Apache che consente la compilazione del codice java senza scaricare tutte le dipendenze delle librerie manualmente. È necessario compilare il codice open-source di Jackrabbit con Maven perché viene distribuito come progetto Maven.
Dopo aver modificato il file "AbstractLocatorFactory.java", bisogna compilare la libreria facendo eseguire il seguente comando (la working directory della shell deve essere nella cartella della libreria):

mvn clean package

Al termine della compilazione, la nuova libreria in formato jar si trova nella cartella "target", creata da Maven durante la compilazione. La libreria da sostituire si trova dentro alla cartella "WEB-INF/lib" dell'archivio dell'applicazione web di Jackrabbit, ed è possibile sostituirla con la nuova libreria compilata tramite un gestore di archivi. Una volta risolto il bug precedente è possibile utilizzare il metodo Session.importXML(), per importare il contenuto in un workspace tramite xml.
Però c'è un altro bug: non è possibile importare nel repository un file xml contenente i valori delle proprietà binarie, a causa del parser xml usato di default da Jackrabbit.
Questi bug sono stati scoperti durante l'attività di tirocinio, che prevedeva di realizzare una applicazione java che facesse una migrazione "a caldo" dalla versione standalone all'applicazione web di Jackrabbit. Per aggirare il secondo bug, è stata sviluppata un'applicazione java che esporta tutto il contenuto, in formato system view, dal repository standalone di Jackrabbit senza i valori delle proprietà binarie. L'applicazione poi importa tutto il contenuto nel nuovo repository tramite il metodo Session.importXml().

Al termine della migrazione, l'applicazione trasferisce i valori delle proprietà binarie usando un altro metodo funzionante:

Dest_Node.setProperty(
"property name",
Source_Node.getProperty("property name").getValue(), PropertyType.BINARY);

Dove l'oggetto *Dest_Node* e *Source_Node* fanno riferimento allo stesso nodo nel workspace sorgente e destinazione.

3.2 CONFIGURAZIONE DI JACKRABBIT

Jackrabbit ha bisogno di 2 cose per poter istanziare un repository:

- Repository Home Directory
- File di configurazione del repository

La Repository Home Directory è il percorso della cartella contenente il repository e tutti i suoi file di configurazione. Il file di configurazione principale è “repository.xml”, che definisce come sono implementati tutti i componenti del repository. I componenti principali di un repository Jackrabbit sono il persistence manager e il datastore, che sono i contenitori effettivi dei dati.
La figura 3.1 mostra come Jackrabbit sia un interfaccia verso i dati, che possono essere salvati sia nel persistence manager che nel datastore, in modo del tutto trasparente all'applicazione.
L'applicazione può leggere e modificare i dati tramite le API definite nello standard JCR (vedi paragrafo 1.3).

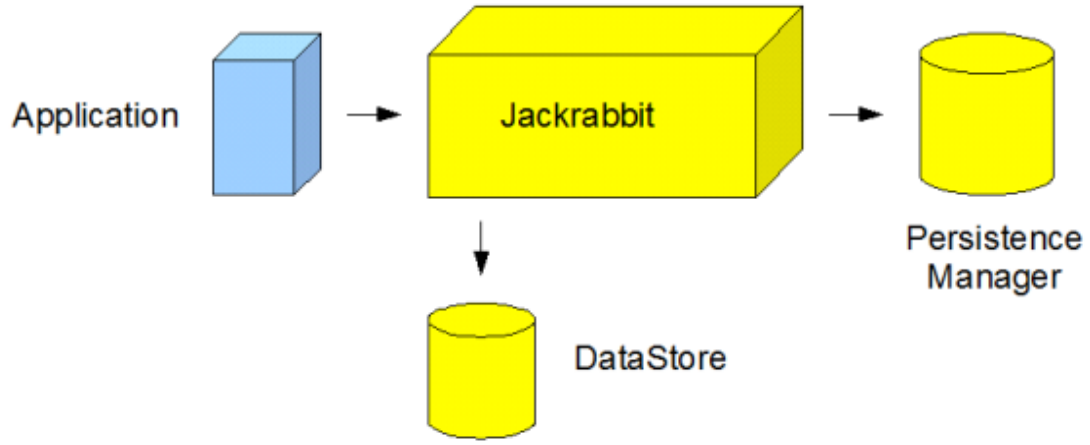

Figura 3.1: schema dell'architettura di Jackrabbit. Il contenuto del repository può essere salvato sia nel persistence manager che nel datastore, in modo del tutto trasparente all'applicazione.

La struttura del file “repository.xml” è la seguente:

<Repository>

<!-- Il file system virtuale usato dal repository per immagazzinare ad esempio i namespace e i nodetype -->
<FileSystem .../>

<!-- Sicurezza per l'autenticazione e le autorizzazioni -->
<Security .../>

<!-- Indica dove e come i workspace sono gestiti -->
<Workspaces .../>

<!-- Template della configurazione del workspace -->

```
<Workspace name="...">
  <FileSystem .../>
  <PersistenceManager .../>
  <SearchIndex .../>          <!-- optional -->
  <ISMLocking .../>           <!-- optional, available since 1.4-->
</Workspace>

<!-- Configurazione del contenitore delle versioni dei nodi -->
  <Versioning .../>

<!-- Configurazione degli indici di ricerca -->
  <SearchIndex .../>    <!-- optional -->

<!-- Configurazione dell'eventuale cluster -->
  <Cluster .../>        <!-- optional, available since 1.2 -->

<!-- Configurazione del datastore -->
  <DataStore .../>      <!-- optional, available since 1.4 →

</Repository>
```

3.3 PERSISTENCE MANAGER

Il persistence manager è uno dei componenti più importanti dell'architettura di Jackrabbit, in quanto contiene la struttura persistente di nodi e proprietà di ogni workspace. Per ogni workspace esiste un persistence manager dedicato.
L'affidabilità, le prestazioni e l'integrità dei dati del persistence manager sono fondamentali per la consistenza dei dati del repository. [12]
Per ogni workspace del repository esiste un file di configurazione chiamato workspace.xml, che definisce com'è implementato il suo persistence manager. Di default, il file

workspace.xml è una copia del template definito nel file repository.xml, però può essere modificato dopo la creazione del workspace.
Il persistence manager può essere implementato sia su file system che su database. I database nei quali Jackrabbit consente di implementare un persistence manager sono:

- Apache Derby;
- H2 Database Engine;
- MySQL;
- PostgreSQL;
- MS SQL Server;
- Oracle.

Nel file “workspace.xml”, bisogna specificare i seguenti parametri per permettere a Jackrabbit di connettersi al database:

- l'url del database;
- le credenziali utente;
- il nome dello schema;
- il prefisso delle tabelle.

Il prefisso delle tabelle serve per identificare le tabelle che appartengono ad un persistence manager, in quanto nello stesso database possono essere presenti più persistence manager.
In questo caso, il prefisso che imposta Jackrabbit di default è il nome del workspace, che è univoco all'interno del repository.

3.4 DATASTORE

Il datastore consente di salvare i valori delle proprietà binarie di grandi dimensioni all'esterno del persistence manager. È opzionale, ma se utilizzato può migliorare le prestazioni del repository. [13]
Ogni volta che viene aggiunta nel workspace una proprietà binaria, Jackrabbit controlla la sua dimensione (in byte). Se la dimensione è maggiore del parametro *minRecordLength* (definito nel file repository.xml), allora il dato viene salvato nel datastore, altrimenti nel persistence manager. Le caratteristiche principali di un datastore sono le seguenti:

- La scrittura e la lettura non sono bloccanti;
- Più repository possono usare lo stesso datastore;
- I dati sono immutabili:
  Quando i valori binari delle proprietà vengono modificati, non vengono modificati i relativi dati binari, ma viene invece aggiunto un dato nuovo.
  Jackrabbit mette a disposizione delle applicazioni un garbage collector, che quando viene richiamato fa la pulizia di tutti i file obsoleti del datastore;
- Il backup a caldo è supportato.

Il datastore può essere implementato in 2 modi:

- File datastore
- Database datastore

3.4.1 File datastore

Il file datastore salva ogni valore binario delle proprietà in un file, il cui nome è il codice hash del contenuto. In lettura, il file datastore non usa nessuna cache locale, quindi il contenuto del file viene letto direttamente dal file system all'occorrenza. In scrittura, il nuovo dato binario viene prima salvato in un file temporaneo, poi viene rinominato con il codice hash del

contenuto. La dimensione massima che può avere un file del datastore è limitata dal file system. Ad esempio, se il datastore è implementato su un file system FAT32, la dimensione massima di ogni file è 4Gb.

3.4.2 Database datastore

Il database datastore salva ogni valore binario delle proprietà in un database relazionale.
Tutti i dati sono salvati in una tabella, la cui chiave primaria è il codice hash del contenuto.
In lettura, il dato binario può essere prima copiato in un file temporaneo sul server o essere letto direttamente dalla tabella (dipende dal parametro *copyWhenReading* definito nella configurazione del datastore). In scrittura, il nuovo dato binario viene prima salvato nella tabella con una chiave primaria temporanea, poi, finita la scrittura, la chiave viene aggiornata con il valore del codice hash del contenuto.
Apache raccomanda di usare il database datastore solo se si hanno dei vincoli relazionali molto forti, altrimenti è meglio implementare il datastore su file system. Infatti le prestazioni migliori si hanno con il file datastore perchè le operazioni di lettura e scrittura interessano direttamente il file system, che è più veloce di un database.

3.5 CLUSTER

Un cluster di Jackrabbit è formato da tante istanze di Jackrabbit che sono configurate per accedere allo stesso repository in modo concorrente. [14] Ogni istanza di Jackrabbit ha la propria repository home directory, che contiene i file di configurazione del repository e gli indici di ricerca.
Quando un repository viene condiviso da tante istanze di Jackrabbit, mette a disposizione un journal, il quale contiene tutte le modifiche fatte al contenuto del repository. Ogni istanza di Jackrabbit legge il journal periodicamente e, se le altre istanze di Jackrabbit hanno modificato il contenuto del repository, aggiorna i suoi indici di ricerca. Per ogni istanza di Jackrabbit, il tempo di attesa tra una lettura e l'altra del journal è definito dal parametro *SyncDelay*, definito nella configurazione della singola istanza di Jackrabbit.

CAPITOLO 4: PROGETTO DELLA PIATTAFORMA DBAAS

4.1 SPECIFICHE DI PROGETTO

L'idea del progetto è di realizzare un sistema in cloud che mette a disposizione del cliente un repository Jackrabbit. È compito del sistema cloud aggiornare il software e mantenere sempre i dati consistenti e disponibili al cliente, tramite processi di backup, gestione dei guasti e scaling. Il processo di backup non deve interrompere le attività del repository, che rimane sempre raggiungibile dal cliente. Quando si verificano dei guasti ad una parte del sistema, la piattaforma deve ridirigere il traffico del repository in una sua copia di backup, in modo del tutto trasparente al cliente. Per scaling si intende che la piattaforma DBAAS deve potersi adattare al carico di lavoro di ogni repository, aggiungendo e rimuovendo risorse automaticamente.

Il sistema è formato da un sito di front-end, che permette al cliente di registrarsi, ottenere uno o più repository ed eventualmente configurarli. Ad ogni repository è associato un url, con il quale il cliente può connettersi, leggere e scrivere il contenuto, senza doversi preoccupare dell'amministrazione del repository.

Dal punto di vista del cliente, la piattaforma offre un servizio chiamato DBAAS (Database As A Service).

4.2 ANALISI DEI REQUISITI

La realizzazione del progetto è basata su alcuni requisiti di sistema:

- Ad ogni cliente possono essere assegnati 1 o più repository, ad ognuno dei quali si può collegare tramite le credenziali utente scelte.
- Per ogni cliente interessa sapere il nome, i contatti e tutti i repository in servizio.
- Ogni repository ha associato un database relazionale che contiene tutti i persistence manager e il journal.
- Ad ogni repository è associato un datastore, implementato su file system.
- Ogni repository è formato da un cluster di nodi indipendenti, configurati in modo opportuno per accedere in modo concorrente al contenuto del repository.
- Ogni repository è raggiungibile tramite un bilanciatore di carico, che distribuisce il carico di lavoro tra tutti i nodi del cluster. Il bilanciatore di carico fa anche da interfaccia fra il cliente e il repository, Questo consente al cliente di conoscere solamente l'host name del bilanciatore di carico (che rimane sempre fisso) per potersi connettere al repository.
- Per rendere il sistema scalabile al traffico dati, durante il ciclo di vita del repository possono essere aggiunti e rimossi nodi automaticamente.

4.3 PROGETTAZIONE CONCETTUALE

La piattaforma del DBAAS viene controllata e gestita da un database relazionale, che definisce tutti i requisiti e vincoli del sistema.

In figura 4.1 viene riportato lo schema concettuale del database, secondo il formalismo ER.

Nella tabella 4.2, 4.3 e 4.4 vengono riportate le descrizioni degli attributi, rispettivamente, delle entità REPOSITORY, CLUSTER_NODE e CLIENTE.

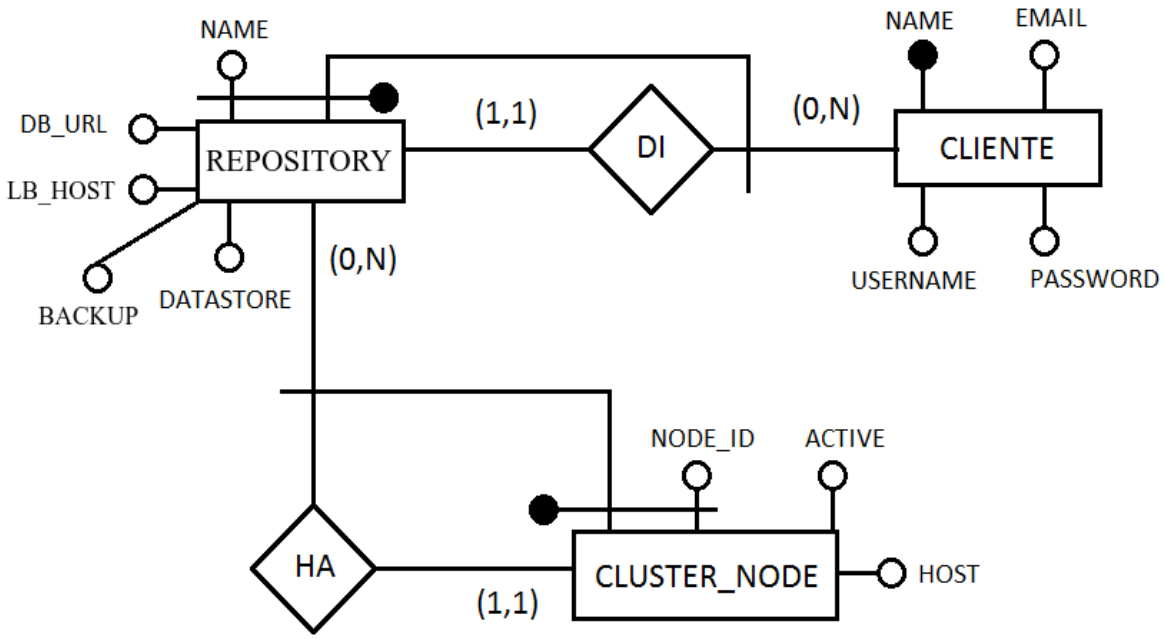

Figura 4.1: Schema ER del database di gestione della piattaforma DBAAS

Nome Attributo	Tipo	Primary Key	Descrizione
CLIENTE.Name	varchar	X	Attributo dell'entità CLIENTE
Name	varchar	X	Nome del repository scelto dal cliente tramite il sito di front-end
DB_Url	varchar		Url del server che contiene il database relazionale del repository
Datastore	varchar		Indirizzo remoto della cartella contenente il datastore
Backup	varchar		Indirizzo remoto della cartella contenente la copia di backup del repository
LB_Host	varchar		Host name pubblico del bilanciatore di carico

Tabella 4.2: Attributi dell'entità REPOSITORY

Nome Attributo	Tipo	Primary Key	Descrizione
CLIENTE.Name	varchar	X	Attributo dell'entità CLIENTE
REPOSITORY.Name	varchar	X	Attributo dell'entità REPOSITORY
Node_ID	integer	X	Identificatore univoco del nodo all'interno del cluster. Viene gestito tramite un contatore.
Host	varchar		Host name nel quale viene eseguito il nodo di Jackrabbit.
Active	boolean		Indica se il nodo di Jackrabbit è attivo, cioè se è raggiungibile dal bilanciatore di carico per prendere parte del carico di lavoro

Tabella 4.3: Attributi dell'entità CLUSTER_NODE

Nome Attributo	Tipo	Primary Key	Descrizione
Name	varchar	X	Nome del cliente
Email	varchar		Indirizzo email del cliente
Username	integer		Username del cliente, valida sia per accedere nella parte riservata del sito di front-end che per accedere al repository.
Password	varchar		Password del cliente, valida sia per accedere nella parte riservata del sito di front-end che per accedere al repository.

Tabella 4.4: Attributi dell'entità CLIENTE

4.4 PROGETTAZIONE LOGICA:

Facendo riferimento allo schema concettuale mostrato in figura 4.1, viene fatta la progettazione logica del database di gestione della piattaforma DBAAS:

REPOSITORY(Name, Cliente:CLIENTE, DB_Url, Datastore, Backup, LB_Host)
CLUSTER_NODE(Node_Id, Repository:REPOSITORY, Host, Active)
CLIENTE (Name, Email, Username, Password)

4.5 OPERAZIONI BASE

Il core della piattaforma DBAAS è formato da alcune operazioni base che vengono eseguite in automatico dal sistema:

- Creazione di un repository Jackrabbit
- Aggiunta di un nodo Jackrabbit al cluster
- Scaling
- Backup a caldo di un repository
- Restore di un repository

Quando capita un guasto ad un repository, il cliente rimane senza servizio finché non viene eseguita la procedura di restore. Le specifiche del progetto prevedono però che il contenuto del repository sia sempre raggiungibile dal cliente (vedi paragrafo 4.1). Per rispettare quindi le specifiche, la piattaforma del DBAAS prevede un sistema che subentri al posto del repository danneggiato, gestendo le richieste del cliente in modo del tutto trasparente ad esso.
Questo sistema è stato progettato col nome di Running Backup Repository.

4.6 RUNNING BACKUP REPOSITORY

Il progetto del DBAAS prevede che, per ogni repository attivo, esista un repository clone che ha il ruolo di “Running Backup Repository”.
Il Running Backup Repository porta 2 vantaggi alla piattaforma DBAAS:

- Durante il restore del repository guasto, il cliente non rimane senza servizio
- Il traffico dati viene spostato dal repository originale al Running Backup Repository in modo del tutto trasparente al cliente, che continua sempre a collegarsi nello stesso host-name del bilanciatore di carico.

Il Running Backup Repository non gestisce lo scaling, in quanto viene usato solo nei casi di emergenza. È formato da un singolo nodo di Jackrabbit attivo. Rimane sempre in esecuzione ma non è direttamente raggiungibile dal cliente. Il suo contenuto è sincronizzato con quello del repository originale, tramite un'applicazione che implementa l'osservazione del contenuto asincrona, definita nelle API java dello standard JCR (vedi paragrafo 1.3.8).
Il contenuto del Running Backup Repository viene salvato su un datastore e su un database differenti da quelli del repository originale, e viene gestito come un normale repository.
La figura 4.5 mostra il diagramma di attività UML della procedura di restore, quando è presente il Running Backup Repository. Quando il repository originale si guasta, tutto il carico di lavoro viene sostenuto dal Running Backup Repository, che garantisce la continuità del servizio al cliente. Quando il repository originale torna attivo, se nel frattempo il cliente ha modificato il contenuto dei dati, deve rieseguire a sua volta gli eventi che si sono verificati nel Running Backup Repository.
La figura 4.6 mostra l'architettura della piattaforma DBAAS di un repository e del relativo Running Backup Repository. Il bilanciatore di carico (Load Balancer) è l'interfaccia fra il cliente e il repository. Se il repository originale è attivo, il bilanciatore di carico ridirige tutto il carico di lavoro nei suoi nodi (nodo1 e nodo2). Se il repository originale è danneggiato, il bilanciatore di carico sposta tutto il carico di lavoro nell'unico nodo del running Backup Repository in modo del tutto trasparente al cliente.

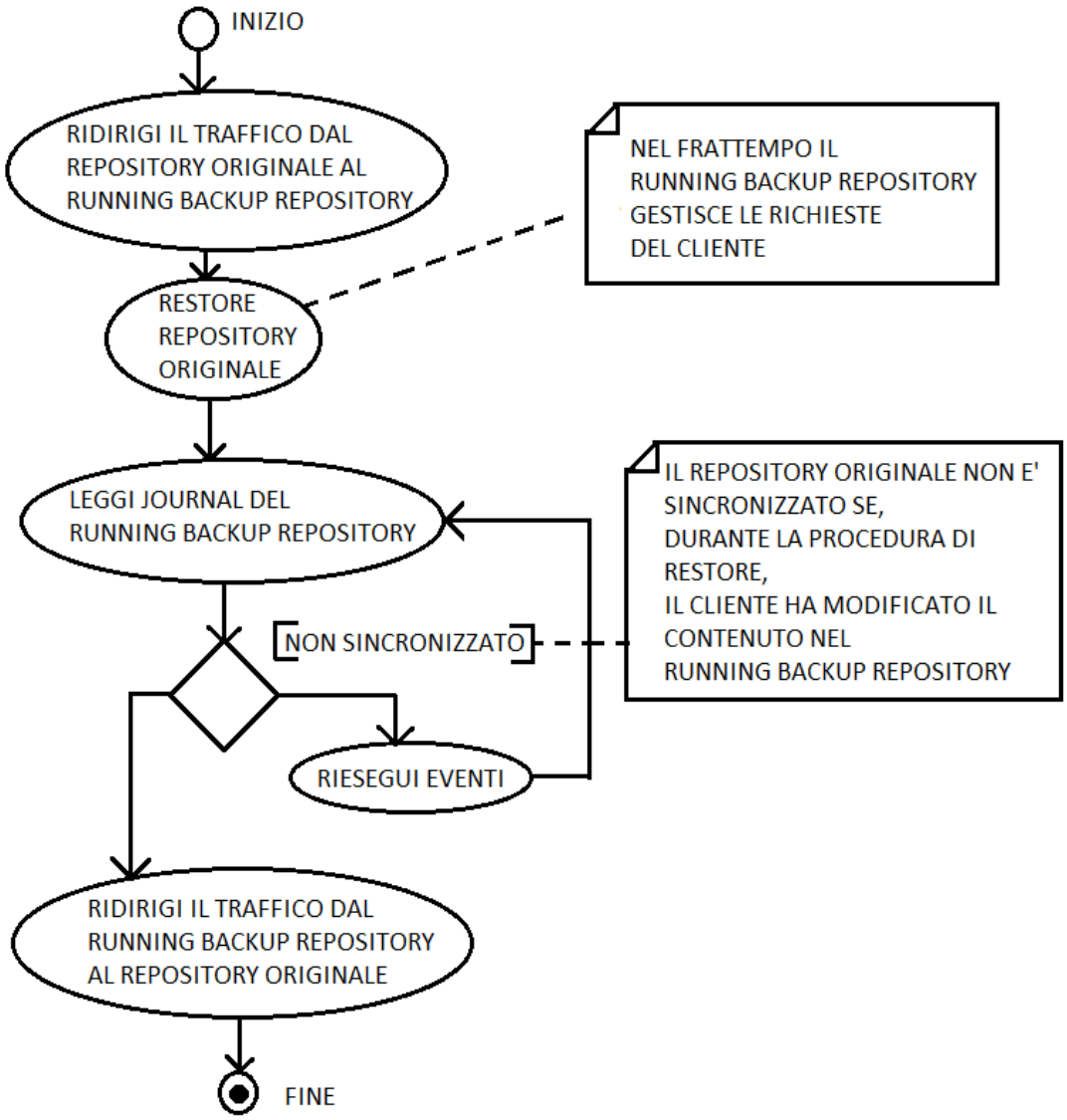

Figura 4.5: Diagramma di attività della procedura che esegue il restore.

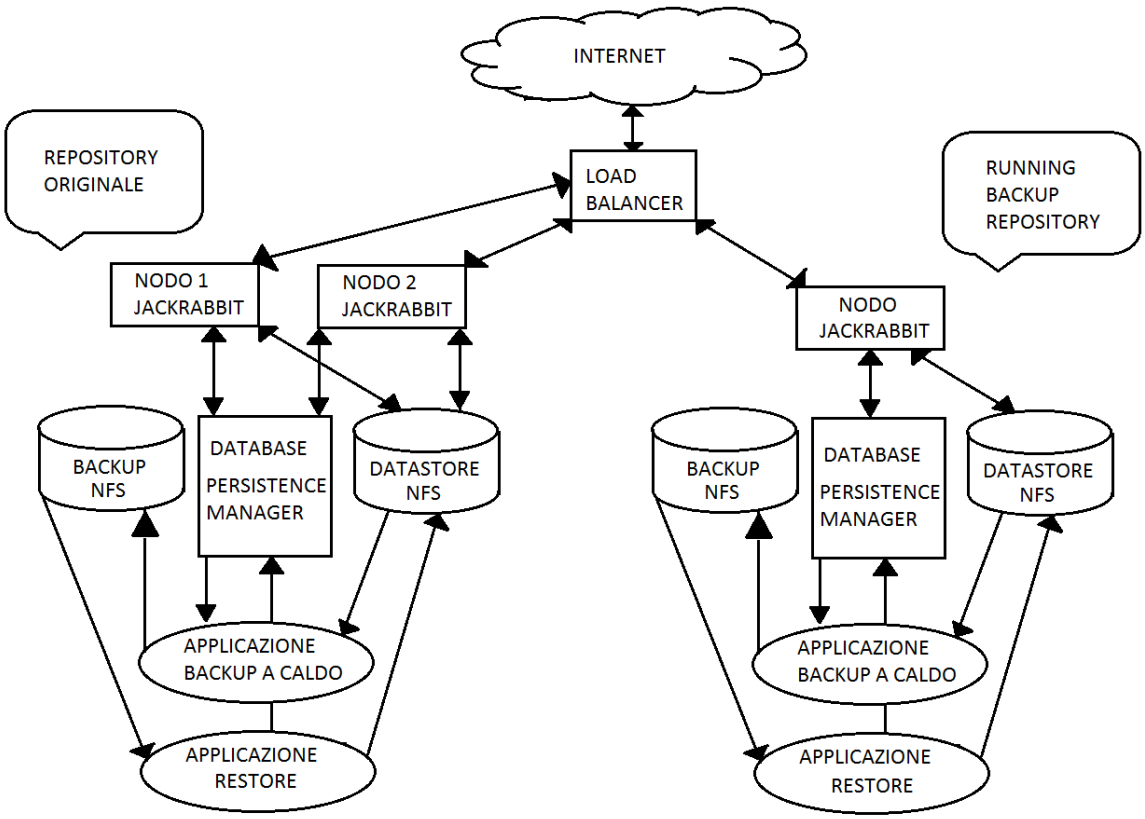

Figura 4.6: Architettura della piattaforma DBAAS di un repository e del relativo Running Backup Repository.

CAPITOLO 5: PROTOTIPO DELLA PIATTAFORMA DBAAS

5.1 INTRODUZIONE

Nel corso di questo lavoro di tesi è stato creato un prototipo del sistema nel quale sono implementate tutte le operazioni base della piattaforma e sul quale saranno effettuati test intensivi. Quando tutti i test avranno avuto successo, il progetto potrà essere implementato nella sua versione finale che fornirà il servizio al cliente.
Il prototipo è formato da un singolo host, che contiene tutti i repository di test e in cui viene simulato il servizio al cliente. Ad ogni operazione di base corrisponde un'applicazione java che esegue in automatico la relativa procedura.

5.2 AMBIENTE DI SVILUPPO

L'ambiente di sviluppo del prototipo è formato da un sistema operativo linux (ubuntu 14.04) e dal seguente software:

- Apache Jackrabbit 2.8 web application
- Apache Tomcat 8.0.12
- Java Runtime Environment 8 Oracle

5.2.1 Apache Tomcat

Apache Tomcat è un'implementazione open source delle specifiche Java Servlet e JavaServer Pages definite nelle JSR del Java Community Process. [15] Sia per le specifiche del Java Servlet che per le specifiche delle JavaServer Pages sono presenti diverse versioni, ognuna definita in una diversa JSR.
Diverse versioni di Tomcat implementano diverse specifiche.

La versione di Tomcat 8.0.12 implementa la versione 3.1 delle specifiche del Java Servlet e la versione 2.3 delle specifiche JavaServer Pages, definite rispettivamente nelle JSR-340 e JSR-245. [16]
Per installare Tomcat è sufficiente scaricare l'archivio dal sito ufficiale di Apache ed estrarlo in una qualsiasi cartella. L'archivio contiene al suo interno tutti i file necessari all'esecuzione di Tomcat. Il percorso della cartella viene memorizzato in una variabile di ambiente chiamata CATALINA_HOME, così ogni applicazione fa riferimento a quella variabile per trovare il percorso dell'installazione di Tomcat.
Per comprendere il funzionamento di Tomcat, è necessario capire il ruolo di ogni sua cartella[17]:

- /bin:
  contiene gli script per avviare e fermare un istanza di tomcat.
  Per avviare un'istanza di Tomcat si esegue lo script startup.sh, per fermarla si esegue lo script shutdown.sh

- /conf:
  contiene i file di configurazione per tomcat.
  I 2 file principali sono server.xml e web.xml.

- /lib:
  contiene le dipendenze java che richiedono le applicazioni web in esecuzione su tomcat

- /logs:
  contiene i file di log. Il file di log principale è "catalina.out".
  È presente anche un file di log per ogni giorno di attività, chiamato
  "nome-host"."aaaa-mm-gg".log

- /webapps:
  contiene tutte le applicazioni web che vengono eseguite da tomcat in formato war (web application archive).
  In questa cartella verrà eseguita l'applicazione web di Jackrabbit.

- /work:
  contiene tutte le servlet che sono state generate dai file jsp delle applicazioni web.

La cartella di installazione di Tomcat consente di avviare un'istanza di Tomcat che esegue l'applicazione web di Jackrabbit, simulando così un nodo di Jackrabbit in esecuzione nel prototipo. Per simulare un cluster di nodi di Jackrabbit è necessario quindi avere in esecuzione nel prototipo istanze multiple di tomcat, indipendenti tra di loro. Ogni istanza di tomcat ha la propria socket TCP, nella quale ascolta le richieste provenienti dall'esterno.
Si possono creare istanze multiple di Tomcat in 2 modi:

- Ogni istanza ha la propria installazione completa di Tomcat.
  In questo caso viene replicata la cartella di installazione per ogni istanza di Tomcat.
- Ogni istanza di Tomcat ha un'installazione parziale e condivide con le altre istanze i file binari (situati in *$CATALINA_HOME/bin*) e le librerie (*$CATALINA_HOME/lib*) dell'installazione completa.

Per lo sviluppo del prototipo è stata scelta la seconda opzione, che consente di risparmiare spazio disco.
I file binari di startup e shutdown identificano le diverse istanze di Tomcat grazie alla variabile d'ambiente CATALINA_BASE, che contiene il percorso della cartella d'installazione della corrente istanza di Tomcat.

Per poter essere eseguita, ogni istanza di tomcat richiede 4 porte di sistema libere, che vengono configurate nel file $CATALINA_BASE/conf/server.xml: [17]

- Connector port:
  È la porta TCP in cui tomcat rimane in attesa delle richieste http. Ogni istanza di Tomcat quindi ha lo stesso host name del prototipo, ma si differenzia dalle altre grazie alla Connector port univoca.

- Shutdown port:
  È la porta TCP nella quale Tomcat aspetta il comando di spegnimento. La richiesta di shutdown deve provenire da una connessione instanziata dal'host del prototipo, non può provenire da altri host.

- AJP (Apache Jserv Protocol) Connector Port:
  È la porta TCP usata dal Connector AJP per creare una socket e rimanere in attesa di connessioni entranti.
  Il connector AJP viene usato per integrare Tomcat con un web server Apache, per permettere al web server di gestire direttamente le richieste di contenuto statico delle applicazioni web, senza passare per Tomcat.

- Redirect Port:
  Viene usata internamente da tomcat per ridirigere il traffico dati.

Sia Tomcat che Jackrabbit, essendo scritti interamente in Java, hanno bisogno di un JRE (Java Runtime Environment) installato nel sistema, per poter essere eseguiti.

5.2.2 Java Runtime Environment

Il JRE (Java Runtime Environment) è un ambiente di esecuzione per applicazioni scritte in linguaggio java. Il JRE installato nel prototipo è la versione Java 8 sviluppata da Oracle.
Gli script di esecuzione di Tomcat devono conoscere il percorso della cartella in cui è installato il JRE, che viene letto dalla variabile d'ambiente JAVA_HOME.
Il percorso dell'installazione di default di Java 8 Oracle è: /usr/lib/jvm/java-8-oracle.

5.2.3 Apache Jackrabbit

Per poter eseguire l'applicazione web di Jackrabbit, Tomcat ha bisogno della libreria java che implementa le API dello standard JCR, che deve essere salvata dentro alla cartella $CATALINA_HOME/lib. La libreria si chiama jcr-2.0.jar ed è scaricabile gratuitamente dal sito:

http://mvnrepository.com/artifact/javax.jcr/jcr/2.0

Ogni istanza di Jackrabbit è indipendente dalle altre in esecuzione nel prototipo, questo consente di simulare tanti nodi di jackrabbit situati in host differenti.
Per poter impostare un cluster di nodi di Jackrabbit che accede in modo concorrente allo stesso repository, è necessario configurare, per ogni nodo, gli stessi persistence manager e lo stesso datastore, altrimenti i diversi nodi farebbero riferimento a repository diversi.

5.3 PERSISTENCE MANAGER

Il database relazionale scelto per implementare tutti i persistence manager del repository è MySql, in quanto è open source ed è perfetto per testare tutte le operazioni base del prototipo.
Nel progetto finale sarà poi possibile passare ad un database relazionale di qualità migliore, ad esempio Oracle.
Dato che l'applicazione web di Jackrabbit deve leggere e scrivere il contenuto dei persistence manager su MySql, ha bisogno della libreria che contiene i driver di connessione per i

database MySql, che deve essere presente nella cartella delle librerie di tomcat ($CATALINA_HOME/lib). La libreria si chiama mysql-connector-java.jar ed è scaricabile gratuitamente dal sito:

http://dev.mysql.com/downloads/connector/j/

5.4 DATASTORE

Il datastore di ogni repository viene implementato su file system.
Per rendere l'architettura del prototipo più simile a quella del sistema finale, il datastore viene implementato su un file system di rete, tramite il protocollo NFS. NFS significa Network File System e indica un filesystem distribuito, ovvero condiviso fra un certo numero di nodi di una rete [18]. Questo consente di avere lo stesso disco di rete condiviso da tutti i nodi del cluster, e ogni nodo può leggere e scrivere nel datastore in modo del tutto trasparente, come se fosse una cartella locale. Questa implementazione porta ad una simulazione più reale del prototipo, in quanto nel progetto finale il datastore sarà contenuto in un disco di rete accessibile da tutti i nodi del cluster.

5.5 BACKUP

Una operazione base della piattaforma DBAAS è il backup a caldo dei repository.
Ogni copia di backup del repository è sostanzialmente una copia sia dei persistence manager che del datastore. Nel prototipo, la copia di backup viene salvata nello stesso disco di rete del datastore, ma nel progetto finale sarà memorizzata in un disco differente.
Come per il datastore, le copie di backup vengono gestite tramite il protocollo NFS.

5.6 BILANCIATORE DI CARICO

Nel progetto della piattaforma DBAAS è presente anche un bilanciatore di carico per ogni repository, che si occupa di distribuire il carico di lavoro tra tutti i nodi attivi del cluster di Jackrabbit. Il bilanciatore di carico ha anche il ruolo di interfaccia fra il cliente e il repository, in quanto il cliente conosce solo l'host-name del bilanciatore di carico per potersi connettere al repository. Nel prototipo non è stato implementato il bilanciatore di carico perché non è necessario al fine di testare le operazioni di base del DBAAS.

5.7 TEST

Dopo che l'architettura del prototipo è stata definita e l'ambiente di sviluppo è stato impostato, si passa all'esecuzione dei test delle operazioni di base della piattaforma DBAAS (riportate nel paragrafo 4.5). Ogni operazione di base corrisponde ad un'applicazione scritta in java. L'implementazione delle operazioni di base nel prototipo della piattaforma DBAAS saranno presentate nel capitolo seguente.

CAPITOLO 6: OPERAZIONI DI BASE

6.1 CREAZIONE DI UN REPOSITORY JACKRABBIT

Quando il cliente richiede un nuovo repository dal sito di front-end, il sistema chiama la procedura per la creazione di tutti i componenti del nuovo repository (persistence manager, datastore remoto, backup remoto), che conterranno tutto il contenuto del repository.
Nel server MySql viene creato un nuovo database che conterrà i persistence manager e il journal del repository. Per il datastore e la copia di backup la procedura crea 2 nuove cartelle remote, associate al nuovo repository.
Dopo che sono stati definiti tutti i componenti, la procedura inserisce, nella tabella REPOSITORY del database di gestione della piattaforma DBAAS, il record che definisce tutti i componenti del nuovo repository.

6.2 AGGIUNTA DI UN NODO JACKRABBIT AL CLUSTER

Questa procedura viene chiamata dal sistema dopo che sono stati definiti tutti i componenti del nuovo repository, ma ancora non c'è nessun nodo di Jackrabbit configurato per usarli.
L'aggiunta di un nuovo nodo ad un repository richiede l'esecuzione delle seguenti operazioni:

- Creazione di una nuova istanza di Tomcat
- Configurazione della nuova istanza di Jackrabbit
- Avvio dell'istanza di Jackrabbit.

6.2.1 Creazione di una nuova istanza di Tomcat

Dato che nel prototipo ogni nodo di Jackrabbit viene eseguito su un'istanza di Tomcat, la procedura crea per prima cosa una nuova istanza di Tomcat.
Nell'host che implementa il prototipo del DBAAS vengono trovate 4 porte di sistema libere, che saranno poi associate rispettivamente alla Connector_Port, Shutdown_Port, AJP_Port e Redirect_Port della nuova istanza di Tomcat (vedi paragrafo 5.2.1).
A questo punto la procedura imposta la variabile d'ambiente $CATALINA_BASE con il percorso della cartella di configurazione della nuova istanza di Tomcat, poi avvia la nuova istanza eseguendo lo script *$CATALINA_HOME/bin/startup.sh.*

6.2.2 Configurazione della nuova istanza di Jackrabbit

Dopo che la nuova istanza di Tomcat è stata avviata, la procedura aggiunge l'archivio dell'applicazione web di Jackrabbit nella cartella *$CATALINA_BASE/webapps.*
Quando l'istanza di tomcat è in esecuzione, estrae in automatico tutti i nuovi archivi aggiunti alla sua cartella webapps. Dopo che l'archivio dell'applicazione web di Jackrabbit è stato estratto, la procedura ferma l'istanza di Tomcat eseguendo lo script *$CATALINA_HOME/bin/shutdown.sh,* altrimenti l'applicazione web di Jackrabbit sarebbe subito eseguita con le configurazioni di default, creando un nuovo repository vuoto.
Prima di essere avviata, la nuova istanza di Jackrabbit deve essere configurata per puntare al database dei persistence manager e al datastore del repository in cui si vuole aggiungere un nuovo nodo di Jackrabbit. Nel file repository.xml, la procedura assegna al nuovo nodo un identificatore univoco all'interno del cluster, e imposta i parametri dei persistence manager e del datastore.

I persistence manager vengono configurati con i seguenti parametri:

- *Url:*
  Contiene l'url del server MySql e del database contenente i persistence manager;

- *Credenziali utente:*
  Username e password dell'utente utlizzato da Jackrabbit per accedere al database MySql. L'utente deve avere i permessi di lettura e scrittura dell'intero database;

- *Prefisso delle tabelle:*
  Serve per distinguere le tabelle dei diversi persistence manager.
  Di default, Jackrabbit setta come prefisso delle tabelle dei persistence manager il nome del relativo workspace, che è univoco all'interno del repository.

Il datastore viene configurato con questi 2 parametri:

- *Path:*
  è il percorso locale della cartella che contiene il datastore.
  La procedura crea una cartella nel prototipo che, grazie al protocollo NFS, fa riferimento alla cartella remota condivisa da tutti i nodi del cluster, in modo del tutto trasparente a Jackrabbit;

- *minRecordLength:*
  è il numero minimo di byte che deve avere il valore di una proprietà binaria per venire salvata nel datastore e non nel persistence manager.

6.2.3 Avvio del repository

Ora il nodo di Jackrabbit è configurato ed è pronto per essere avviato ed essere aggiunto nella lista dei nodi attivi del cluster. Per avviare la nuova istanza di Jackrabbit bisogna riavviare l'istanza di tomcat, la quale esegue l'applicazione web di Jackrabbit automaticamente durante l'avvio.
Se il nodo di Jackrabbit che viene aggiunto al cluster è il primo nodo in assoluto, bisogna controllare se nel database MySql associato al repository sono state create in automatico le tabelle che gestiscono i persistence manager e il journal.
Per testare se lo specifico nodo di Jackrabbit riesce a collegarsi al repository correttamente, bisogna provare a connettersi direttamente al nodo di Jackrabbit tramite un browser web.
In caso di errori è possibile vedere il log dell'istanza di tomcat nel file catalina.out, che contiene il log di tutte le applicazioni web eseguite da Tomcat.
Se non ci sono errori, la procedura inserisce, nella tabella CLUSTER_NODE del database di gestione della piattaforma DBAAS, il record relativo al nuovo nodo aggiunto.

6.3 SCALING

Lo scaling consiste nell'adeguazione dinamica delle risorse hardware, usate dal sistema per rispondere al carico di lavoro di un repository. L'architettura di un cluster Jackrabbit consente di scalare il carico di lavoro tra tutti i nodi, tramite un bilanciatore di carico che gestisce le richieste del cliente. Tutti i nodi comunque continuano a condividere in modo concorrente lo stesso database del repository e lo stesso disco di rete del datastore, che diventano quindi il collo di bottiglia di tutto il traffico dati.
In questa tesi è stato considerato solo il problema dello scaling al livello dei nodi, non al livello della memorizzazione dei dati che potrà essere considerato in un lavoro futuro.
Per testare lo scaling nel prototipo, sono state sviluppate 2 procedure che, rispettivamente, aggiungono e rimuovono un nodo dal cluster mentre il repository è in esecuzione, come dettagliato nei paragrafi seguenti.

6.3.1 Aggiunta di un nuovo nodo

Se il carico di lavoro aumenta, si vuole che il nuovo nodo di Jackrabbit sia aggiunto al cluster in tempi brevi, in quanto dovrà assorbire parte del carico di lavoro. Se si usa la stessa procedura spiegata nel paragrafo 5.8, prima di essere operativo il nuovo nodo deve leggere tutto il contenuto del repository e costruire tutti gli indici di ricerca. Se il repository possiede una grande mole di dati, questa operazione può impiegare molto tempo e il sistema diventa poco reattivo alla scalabilità. Per evitare tutta la ricostruzione degli indici, la procedura che aggiunge il nuovo nodo al cluster esegue le seguenti operazioni: [19]

- Crea una nuova istanza di Tomcat
- Ferma un nodo attivo del cluster
- Legge il “revision number” del nodo fermato dal database del repository
- Copia l'intera cartella del repository del nodo fermato nella cartella del repository del nuovo nodo. Quest'operazione è molto veloce perché la cartella del repository contiene solo i file di configurazione e gli indici di ricerca.
- Riavvia il nodo fermato
- Modifica il file repository.xml del nuovo nodo, assegnandoli un nuovo Node_Id
- Aggiunge alla tabella JOURNAL_LOCAL_REVISIONS del database del repository un nuovo record, formato dal nuovo Node_Id e dal revision number letto in precedenza
- Avvia il nuovo nodo

Con questa procedura, si ha la certezza che il nodo nuovo ha gli indici consistenti perché sono stati copiati da un nodo preesistente ma non attivo, quindi durante la copia gli indici non hanno subito modifiche Quando il nuovo nodo viene in seguito avviato, l'istanza di Jackrabbit deve controllare solamente il Journal del repository e aggiornare i suoi indici di ricerca solo se nel frattempo gli altri nodi ne hanno modificato il contenuto.

Infine, nella tabella CLUSTER_NODE del database di gestione della piattaforma DBAAS, la procedura aggiunge il record relativo al nuovo nodo aggiunto.

6.3.2 Rimozione nodo

Quando il carico di lavoro diminuisce, il sistema risparmia le risorse hardware rimuovendo dal cluster tutti i nodi non più necessari. Per rimuovere da un cluster un nodo di Jackrabbit, è sufficiente disattivare il nodo (spegnendo l'istanza di Tomcat che lo esegue) e cancellare tutti i file di configurazione ad esso associati.
Nella tabella CLUSTER_NODE del database di gestione della piattaforma DBAAS, la procedura elimina il record relativo ad ogni nodo rimosso.

6.4 BACKUP A CALDO

La procedura di backup a caldo viene eseguita su ogni repository quotidianamente, in un orario in cui l'attività sui repository è molto limitata (tipicamente in orario notturno).
Per creare una copia di backup di un repository, è sufficiente replicare i dati del database MySql associato e del datastore, assicurandosi che durante la copia i dati non vengano modificati per garantire la consistenza dei dati copiati. Se durante il backup il database è interrogabile in sola lettura, allora anche il datastore è in sola lettura, perché il datastore può venire modificato solo dopo che una tabella dei persistence manager viene modificata.
La procedura di backup del repository, imposta pertanto il database in sola lettura, successivamente esporta i dati del database e del datastore, e infine rende il database di nuovo scrivibile. Se durante il processo di backup un'applicazione esegue una scrittura al contenuto del repository, rimane in attesa finché il backup non è completato, senza generare nessuna eccezione.
Ad esempio, se l'applicazione chiama il metodo Session.save() per salvare tutte le modifiche in modo persistente, rimane in attesa finché il metodo Session.save() non ha finito la sua esecuzione, cioè finché il database è in modalità read-only.

6.4.1 Backup a caldo del database del repository

Nel prototipo, il backup a caldo del database del repository viene fatto esportando il database in un file in formato SQL. Prima di esportare il database, la procedura effettua su tutte le tabelle prima un “flush” e poi un “lock”. [20]
Il “flush” delle tabelle è necessario per assicurarsi che tutto il contenuto del database sia scritto in modo persistente sul disco fisso e non nella cache (o nella RAM).
Il “lock” rende le tabelle accessibili solo in modalità read-only. Il lock verrà rilasciato dall’applicazione solo una volta terminata anche la copia dei file del datastore.
Il repository rimane così leggibile dal cliente durante l'operazione di backup, nel quale però non potrà né aggiungere né modificare il contenuto.
Per esportare il database la procedura esegue mysqldump per esportare il database MySql associato al repository. [20] Mysqldump è un software che esporta il database MySql in un file in formato SQL, che contiene le definizioni e i record di tutte le tabelle.

6.4.2 Backup a caldo del datastore

Nel prototipo, il datastore si basa su file system, quindi è possibile eseguire una qualsiasi tecnica di backup a caldo che il file system supporta.
Per copiare il contenuto del datastore, la procedura esegue rsync. Rsync è un software che permette di copiare solo le differenze binarie dei file, trasferendo solo i blocchi di byte modificati. [21]
Dato che nel datastore di Jackrabbit i file vengono solo aggiunti e non possono essere modificati (vedi paragrafo 3.5), rsync copia nella cartella remota di backup solo i file che sono stati aggiunti nel repository in giornata (la procedura di backup a caldo viene fatta eseguire giornalmente), impiegando molto meno tempo rispetto ad una copia completa del contenuto.

6.5 RESTORE

Quando si verificano dei guasti nel disco del datastore o nel server MySql, il contenuto del repository non è più raggiungibile da nessun nodo di Jackrabbit. La piattaforma DBAAS ha quindi una procedura che in automatico fa il restore dei dati danneggiati o non più raggiungibili.
Nel prototipo, la procedura di restore consiste nell'importare la copia di backup del database nel server MySql e ripristinare tutti i file del datastore, sempre eseguendo il software rsync.
Dopo che il contenuto del repository è stato ripristinato, la procedura riavvia tutti i nodi del cluster, i quali in automatico ricostruiscono tutti i loro indici di ricerca, leggendo l'intero contenuto del repository.

6.6 RUNNING BACKUP REPOSITORY

Nella piattaforma DBAAS, ogni volta che viene creato un nuovo repository, il sistema chiama anche la procedura che crea il Running Backup Repository associato. Il Running Backup Repository viene creato prima dell'avvio del repository originale, in questo modo entrambi i repository nascono senza contenuto, e la sincronizzazione del contenuto incomincia direttamente in modalità asincrona (vedi paragrafo 1.3.8 – Osservazione del contenuto asincrona).
Il Running Backup Repository ha una propria applicazione che osserva ogni workspace del repository al quale è associato. Ogni volta che l'applicazione riceve una notifica sulla modifica del contenuto di un workspace, riesegue gli stessi eventi nel workspace del Running Backup Repository. Leggendo un evento, l'applicazione può distinguere il tipo dell'evento e il percorso assoluto dell'item interessato; non può leggere però tutte le informazioni associate all'item. L'applicazione dovrà in seguito leggere l'item modificato direttamente dal workspace, per ottenere tutte le informazioni aggiuntive necessarie.
Ad esempio, se si verifica un evento di tipo NODE_ADDED, l'applicazione sa che è stato aggiunto un nuovo nodo nel workspace osservato, e conosce il percorso assoluto del nuovo nodo. Per leggere tutte le informazioni relative al nuovo nodo, comunque, l'applicazione deve leggere il nodo direttamente dal workspace osservato.

L'algoritmo implementato dall'applicazione per la riesecuzione degli eventi tramite l'osservazione del contenuto asincrona è riportato nel paragrafo 6.6.1.
In caso di restore, quando il repository originale deve sincronizzare il suo contenuto con quello del Running Backup Repository, viene eseguita una procedura che legge tutto lo storico delle modifiche effettuate tramite il journal del repository, secondo lo standard JCR definito nel paragrafo 1.3.9. L'algoritmo per la riesecuzione degli eventi tramite journal è più complicato rispetto all'algoritmo di riesecuzione degli eventi in modalità asincrona, perché in questo caso l'applicazione non ha la certezza di trovare l'item soggetto all'evento nel workspace osservato, in quanto un evento futuro (ancora da leggere nel journal) lo potrebbe avere cancellato o spostato all'interno del workspace. Per gestire questo problema, bisogna etichettare tutti i nodi che non sono più presenti nel repository che si vuole sincronizzare.
Ad esempio, se nel journal viene letto un evento di tipo NODE_ADDED, ma il nodo soggetto all'evento non è più presente nel workspace osservato, può essere per 2 motivi:

1) Un evento futuro cancella il nodo;
2) Un evento futuro sposta il nodo all'interno del workspace.

In questo caso, l'applicazione aggiunge il nodo nel workspace di destinazione assegnandoli l'etichetta di “Incompleto”. Quando poi verrà eseguito l'evento futuro, l'applicazione riesegue l'evento futuro sul nodo incompleto.
Alla fine della lettura del journal, per avere la certezza che tutto il contenuto sia sincronizzato correttamente, si controllano tutti i nodi incompleti del workspace di destinazione.
Se il nodo incompleto è presente nel repository che si sta osservando, lo si esporta (con tutto il suo sottografo) e lo si importa nel repository da sincronizzare.
Se il nodo incompleto non è presente, viene semplicemente rimosso.

6.6.1 Algoritmo per la riesecuzione degli eventi tramite osservazione del contenuto asincrona

Questo algoritmo viene eseguito dall'applicazione che sincronizza il contenuto del Running Backup Repository con il contenuto del repository associato. Quando all'applicazione viene notificata una modifica, riesegue gli stessi eventi che si sono verificati nel contenuto del repository osservato.

L'applicazione usa 2 sessioni, di cui una è collegata al workspace osservato e l'altra è collegata al workspace di destinazione. Tramite le sessioni, l'applicazione può leggere e scrivere il contenuto nei workspace.

Per ogni evento, l'algoritmo controlla di che tipo è, e di conseguenza riesegue le stesse azioni nel workspace di destinazione. Ad esempio, quando un nodo o una proprietà vengono rimossi, l'algoritmo elimina lo stesso nodo o la stessa proprietà anche nel workspace sincronizzato.

Se un nodo viene aggiunto, l'algoritmo esporta il nuovo nodo dal workspace osservato in formato system view e lo importa nel workspace di destinazione; in questo modo, se al nuovo nodo è stato assegnato un'identificatore (UUID), il nuovo nodo del workspace di destinazione viene creato con lo stesso identificatore.

Viene riportato in seguito lo pseudocodice che descrive l'algoritmo per la riesecuzione degli eventi tramite osservazione del contenuto asincrona:

```
/*
list è la lista di eventi che si sono verificati nel workspace del repository osservato, e che devono essere rieseguiti nel workspace del repository di destinazione
*/
Redo_Events(Events list)
{
        Session s;      //sessione del workspace osservato
        Session rbr;    //sessione del workspace di destinazione
```

```
        while(list.hasNext())
        {
                Event e = list.next();  //viene letto il prossimo evento da eseguire
                switch(e.getType)       //controlla il tipo dell'evento
                {
                case NODE_ADDED:
                        Esporta Nodo In Formato System View Dal Workspace Osservato;
                        Importa Nodo Nel Workspace Di Destinazione;
                case NODE_REMOVED:
                        Rimuovi Nodo Nel Workspace Di Destinazione;
                case NODE_MOVED:
                        Leggi Il Nuovo Percorso Assoluto Del Nodo;
                        Sposta Nodo Nel Workspace Di Destinazione;
                case PROPERTY_ADDED:
                        Aggiungi Proprietà Nel Workspace Di Destinazione;
                        Leggi Valore Della Proprietà Dal Workspace Osservato;
                        Scrivi Valore Nella Proprietà Del Workspace Di Destinazione;
                case PROPERT_CHANGED:
                        Leggi Valore Della Proprietà Dal Workspace Osservato;
                        Scrivi Valore Nella Proprietà Del Workspace Di Destinazione;
                case PROPERTY_REMOVED:
                        Rimuovi Proprietà Dal Workspace Di Destinazione;
                } //fine switch
        } //fine while
Salva Sessione Del Workspace Di Destinazione
} //fine algoritmo
```

CONCLUSIONI

Tramite questa tesi è stato raggiunto l'obiettivo primario del progetto: realizzare un prototipo funzionante della piattaforma DBAAS per Jackrabbit. Lo sviluppo di un primo prototipo, con costi molto ridotti, ha permesso di testare tutte le funzionalità di Jackrabbit e verificare che tutte le operazioni base funzionino correttamente.

Il contenuto di questa tesi può essere usato come base per realizzare il sistema finale, che può essere realizzato in modi diversi a seconda delle scelte che farà il progettista.

Ad esempio, è possibile comprare tutto l'hardware necessario e implementare la piattaforma DBAAS *in-house,* avendo così una gestione completa di tutte le macchine e i dischi utilizzati.

Oppure si può scegliere di affidare tutta la gestione hardware ad un servizio di hosting (ad esempio Amazon EC2). In questo contesto, il progettista deve solo impostare l'ambiente di sviluppo del server in cloud e configurare tutte le procedure che eseguono le operazioni di base della piattaforma DBAAS.

Come tutti i software open-source, Jackrabbit non ha supporto tecnico, è presente solo una mailing list dove tutti gli utenti di Jackrabbit possono riportare eventuali bug e aiutarsi a vicenda. Lo sviluppo del progetto ha richiesto molto tempo a causa della scarsa documentazione che è possibile trovare riguardo a Jackrabbit. Molti concetti sono stati studiati e testati direttamente sul prototipo, perché, al momento della scrittura di questa tesi, non esiste su internet un progetto simile.

Ad esempio, la progettazione del Running Backup Repository è stata fatta inizialmente conoscendo solo le API Java che permettono di osservare il contenuto di un repository.

Successivamente sono stati progettati e implementati gli algoritmi che consentono di fare la sincronizzazione del contenuto tramire la riesecuzione degli eventi.

Il progetto della piattaforma DBAAS può essere esteso per gestire anche il problema dell'accesso ai dati. Con l'architettura cluster di Jackrabbit, infatti, tutti i nodi si dividono il carico di lavoro a livello di rete, ma continuano a fare tutti riferimento agli stessi persistence manager e datastore, i quali diventano il collo di bottiglia di ogni repository.

In questa tesi il problema dell'accesso ai dati non è stato considerato perché è indipendente dalla progettazione di un cluster di Jackrabbit, e sarà affrontato durante la realizzazione del sistema finale.

BIBLIOGRAFIA

[1] JSR-283, Content Repository for Java 2.0:
https://jcp.org/en/jsr/detail?id=283, 14 novembre 2014

[2] JSR-170, Content Repository for Java:
https://jcp.org/en/jsr/detail?id=170, 14 novembre 2014

[3] Grammatica CND
http://jackrabbit.apache.org/node-type-notation.html, 14 novembre 2014

[4] Modello gerarchico
http://en.wikipedia.org/wiki/Hierarchical_database_model, 15 novembre 2014

[5] Modello reticolare
http://it.wikipedia.org/wiki/Modello_reticolare, 15 novembre 2014

[6] CODD, E. F. A Relational Model of Data for Large Shared Data Banks. San Jose, California : ACM, 1970.

[7] Confronto fra JCR e RDBMS:
http://dev.day.com/content/ddc/blog/2009/01/jcrrdbmsreport/_jcr_content/images/jcrrdbmsreport/jcr_rdbms_report_chapuis.pdf, 15 novembre 2014

[8] Apache Jackrabbit:
http://jackrabbit.apache.org/, 16 novembre 2014

[9] JSR-340, Specifiche Java Servlet 3.1:
https://jcp.org/en/jsr/detail?id=340, 17 novembre 2014

[10] JSR-245, Specifiche Java ServerPages 2.3:
https://jcp.org/en/jsr/detail?id=245, 17 novembre 2014

[11] Jackrabbit, Risoluzione bug web application:
https://issues.apache.org/jira/browse/JCR-3763, 16 novembre 2014

[12] Jackrabbit, Persistence Manager:
https://wiki.apache.org/jackrabbit/PersistenceManagerFAQ, 16 novembre 2014

[13] Jackrabbit, Datastore:
https://wiki.apache.org/jackrabbit/DataStore, 16 novembre 2014

[14] Jackrabbit, Cluster:
https://wiki.apache.org/jackrabbit/Clustering, 16 novembre 2014

[15] Apache Tomcat:
http://tomcat.apache.org, 17 novembre 2014

[16] Tomcat, versioni:
http://tomcat.apache.org/whichversion.html, 17 novembre 2014

[17] Tomcat, Struttura delle directory e Porte TCP:
http://crunchify.com/how-to-run-multiple-tomcat-instances-on-one-server/, 17 novembre 2014

[18] Protocollo NFS:
https://help.ubuntu.com/community/SettingUpNFSHowTo, 17 novembre 2014

[19] Jackrabbit, Aggiunta nodo al cluster durante l'esecuzione del repository: https://blog.liip.ch/archive/2011/05/10/add-new-instances-to-your-jackrabbit-cluster-the-non-time-consuming-way.html, 18 novembre 2014

[20] MySql , Backup a caldo: http://dev.mysql.com/doc/refman/5.0/en/backup-methods.html, 18 novembre 2014

[21] Rsync: http://it.wikipedia.org/wiki/Rsync, 18 novembre 2014