Università degli Studi di Padova
Facoltà di Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea

Progettazione e realizzazione di un sistema di gestione di esperimenti scientifici basato su architetture Spring-Hibernate

Relatore: Prof. Giorgio Maria Di Nunzio

Laureando: Marco Rampazzo

Anno Accademico 2011-2012

Sommario

Negli ultimi anni l'interesse per la classificazione automatica di testi in categorie è in forte espansione. I fattori principali che hanno contribuito a promuovere questa espansione sono la maggiore disponibilità di documenti in forma digitale e la conseguente necessità di definire una struttura per la loro organizzazione in modo da migliorarne la gestione.
L'approccio alla classificazione automatica si basa soprattutto su tecniche di *machine learning*: un processo induttivo che costruisce in modo automatico un classificatore di testi sulla base dell'analisi di un insieme di documenti preclassificati e sulle caratteristiche delle categorie di interesse.
Non esiste un metodo standard per la definizione di questi classificatori; per questo i ricercatori studiano e propongono nuove soluzione per la costruzione di questi strumenti. I risultati vengono quindi pubblicati in documenti scientifici che solitamente contengono la definizione della configurazione e le prestazioni che questa configurazione ha ottenuto negli esperimenti svolti.
Dai dati che solitamente vengono inseriti nelle pubblicazioni, non si hanno a disposizione tutte le informazioni per poter riprodurre gli esperimenti; ne si ha la possibilità di confrontare in modo diretto i risultati ottenuti.
Lo scopo di questo progetto è quindi quello di costruire uno strumento che permetta di storicizzare gli esperimenti e che dia la possibilità di confrontare i risultati ottenuti con le diverse configurazioni.
La realizzazione può essere concettualmente divisa in due parti: nella prima parte si andrà a definire come è stata progettata la base di dati che permette di memorizzare in dettaglio gli esperimenti ed i relativi risultati ottenuti, successivamente, si analizzerà come è stato strutturato ed implementato il progetto in Java che si basa sulle architetture Spring e Hibernate per la gestione ed interrogazione della base di dati.
Il documento termina con le considerazioni personali sull'attività svolta e delle

proposte per il miglioramento dei risultati ottenuti.

Indice

Capitolo 1

Introduzione

L'*information retrieval* (IR) è l'insieme delle tecniche utilizzate per il recupero mirato dell'informazione in formato elettronico.
Per "informazione" si intendono tutti i documenti, i metadati ed i file presenti all'interno di collezioni di dati.
La maggiore disponibilità di documenti in forma digitale e la necessità di organizzare i documenti in modo da facilitarne la ricerca, sono i fattori principali che spingono la ricerca nel campo dell'*information retrieval* alla costruzione di classificatori automatici di testi che permettano di associare i testi a delle categorie tematiche.
Lo scopo della classificazione automatica dei dati è di trasformare un insieme di elementi non classificati in una serie coerente di informazioni restituite sotto forma di classi o gruppi.
Attraverso tale classificazione si tende a sintetizzare le caratteristiche primarie di ciascun raggruppamento ed è anche possibile stabilire una tipologia all'interno del campione preso in esame.

Apprendimento automatico

L'apprendimento supervisionato è una tecnica di apprendimento automatico che mira a istruire un sistema informatico in modo da consentirgli di risolvere dei compiti in modo automatico.
Si definiscono i dati in ingresso come insieme I;
Si definisce l'insieme dei dati in uscita come insieme O.
Si definisce una funzione h che associa ad ogni dato in ingresso ($i \in I$) la sua

risposta corretta ($o \in O$).
Tutti gli algoritmi di apprendimento supervisionato partono dal presupposto che se forniamo all'algoritmo un numero adeguato di esempi l'algoritmo sarà in grado di creare una funzione h' che approssimerà la funzione h. Se l'approssimazione di h risulterà adeguata quando proporremo ad h' dei dati in ingresso mai analizzati la funzione dovrebbe essere in grado di fornire delle risposte in uscita molto simili a quelle fornite da h e quindi il più possibile corrette.
Si può facilmente intuire che il buon funzionamento di questi algoritmi dipende in modo significativo dai dati in ingresso: se si forniscono pochi ingressi l'algoritmo potrebbe non aver abbastanza esperienza, mentre molti dati in ingresso potrebbero rendere eccessivamente lento l'algoritmo, dato che la funzione h' generata dagli ingressi potrebbe essere molto complicata.
Questi algoritmi inoltre sono molto sensibili al rumore, anche pochi dati errati potrebbero rendere l'intero sistema non affidabile e condurlo a decisioni errate.

Esperimenti

Per valutare se lo strumento creato come classificatore funzioni nel modo corretto, vengono fatti degli esperimenti.
Gli strumenti che stanno alla base per questi esperimenti sono:

- *training set*: che consiste nell'insieme di esempi utilizzati per addestrare un sistema supervisionato. Tipicamente, ogni esempio in un training set è costituito da un oggetto di input e da una risposta o classificazione associata all'input.

- *test set*: che consiste nell'insieme di esempi utilizzati per valutare le prestazioni di un sistema.

L'apprendimento di un classificatore supervisionato è tipicamente effettuato a partire dall'insieme di addestramento, il training set: il classificatore cerca delle relazioni empiriche tra i dati dell'insieme di addestramento in modo da riconoscere le caratteristiche che distinguono gli elementi dell'insieme. Successivamente, per verificare se le relazioni apprese dal classificatore sono realmente generali, si valuta il classificatore sul test set, tipicamente disgiunto dall'insieme di addestramento.

Valutazione

La valutazione sperimentale di un classificatore misura la sua efficacia, cioè la capacità del sistema a classificare nel modo corretto i documenti.
L'efficacia è di solito misurata in termini di precisione e richiamo.
La *precisione* (in inglese *precision*) è il rapporto di documenti pertinenti fra quelli recuperati.

$$P = \frac{numero\ di\ documenti\ pertinenti\ recuperati}{numero\ di\ documenti\ recuperati}$$

Il *richiamo*, (in inglese *recall*) è il rapporto fra il numero di documenti rilevanti recuperati e il numero di tutti i documenti rilevanti disponibili nella collezione considerata.

$$R = \frac{numero\ di\ documenti\ rilevanti\ recuperati}{numero\ di\ documenti\ rilevanti}$$

Un esperimento è quindi un test che viene fatto per un classificatore di testi, al quale viene prima insegnato a riconoscere le categorie attraverso il training set e successivamente vengono valutate le prestazioni che ottiene cercando di classificare i documenti presenti nel test set.

Confronto tra esperimenti

Ad oggi non esiste nessuno strumento conosciuto che permetta di storicizzare i risultati di esperimenti, e che permetta di definire uno standard che consenta di poter sia ripetere l'esperimento con la stessa configurazione, sia confrontare i risultati ottenuti.
Nasce quindi l'idea di definire e di implementare una base di dati che permetta di catalogare gli esperimenti di classificazione dei testi in modo da facilitare l'analisi ed il confronto tra gli esperimenti.
Questa base di dati darà anche la possibilità di riprodurre gli esperimenti che essa descrive, in quanto i dati a disposizioni dovrebbero offrire una visione completa della configurazione adottata per l'esperimento.

1.1 Strumenti utilizzati

Di seguito introduciamo brevemente gli strumenti che sono stati utilizzati per la realizzazione di questo progetto di tesi.

Progettazione concettuale e logica

Microsoft Visio 2010 è un software per la creazione di grafici e diagrammi, sviluppato da Microsoft per i sistemi operativi Windows.
Questo software è stato utilizzato per la realizzazione degli schemi della progettazione concettuale e logica della base di dati.

Progettazione fisica

PostgreSQL è un potente database relazionale open source.
Ha più di 15 anni di sviluppo attivo e un'architettura ampiamente collaudata che ha guadagnato una solida reputazione di affidabilità e integrità dei dati.
Questo software nella versione 9.1 è stato utilizzato per la creazione della base di dati, ed è stato scelto perché, oltre ad essere open source, funziona su tutti i principali sistemi operativi.

Sistema

Eclipse è un ambiente di sviluppo integrato multi-linguaggio e multi-piattaforma.
È stata utilizzata la versione "Indigo" del software per la realizzazione del progetto Java che permette di gestire la base di dati.

Hibernate è una piattaforma middleware open source per lo sviluppo di applicazioni Java che fornisce un servizio di Object-relational mapping, ovvero che gestisce la rappresentazione e il mantenimento su database relazionale di un sistema di oggetti Java.
Lo scopo principale di Hibernate è quello di fornire un mapping delle classi Java in tabelle di un database relazionale. Sulla base di questo mapping Hibernate gestisce il salvataggio degli oggetti di tali classi su database. Si occupa inoltre del reperimento degli oggetti da database, producendo ed eseguendo automaticamente le query SQL necessarie al recupero delle informazioni e la successiva reistanziazione dell'oggetto precedentemente mappato su database.

L'obiettivo di Hibernate è quello di esonerare lo sviluppatore dall'intero lavoro relativo alla persistenza dei dati. Esso si adatta al processo di sviluppo del programmatore, sia se si parte da zero sia se si parte da un database già esistente. Hibernate genera le chiamate SQL e solleva lo sviluppatore dal lavoro di recupero manuale dei dati e dalla loro conversione, mantenendo l'applicazione portabile in tutti i database SQL.

Spring è un framework open source per lo sviluppo di applicazioni su piattaforma Java.
È stato largamente riconosciuto all'interno della comunità Java quale valida alternativa al modello basato su Enterprise JavaBeans (EJB). Rispetto a quest'ultimo, il framework Spring lascia una maggiore libertà al programmatore fornendo allo stesso tempo un'ampia e ben documentata gamma di soluzioni semplici adatte alle problematiche più comuni.

Le librerie di Hibernate (nella versione 3.0) e di Spring (nella versione 2.5.6), sono state utilizzate per la gestione del database attraverso il progetto Java creato con Eclipse.

Capitolo 2

Progettazione concettuale

Dall'analisi della realtà che si vuole descrivere si è arrivati ad identificare gli aspetti caratterizzanti nell'esperimento, nella collezione e nei risultati.
L'esperimento viene compiuto con un classificatore. Il classificatore prende in input il training set (una parte della collezione di dati) per imparare a riconoscere come classificare i documenti, successivamente prova a classificare i documenti contenuti nella seconda parte della collezione (il test set). Sugli esiti della classificazione dei documenti vengono quindi prodotti degli indici che permettono di valutare le prestazioni del classificatore.

Definiti i requisiti per la realizzazione della base di dati si passa alla progettazione concettuale.
Lo scopo della progettazione concettuale è quello di rappresentare le specifiche della realtà di interesse in modo da fornire una descrizione completa di ciò che si vuole rappresentare.
Il prodotto di questa fase viene chiamato modello Entirà-Relazione (ER) o schema concettuale e fa riferimento ad un modello concettuale dei dati.
L'analisi verrà affrontata partendo dall'introduzione delle entità, per passare poi alle relazioni che le coinvolgono.

Modello ER

Il modello ER è un modello per la rappresentazione concettuale dei dati ad un alto livello di astrazione.
Per configurare al meglio la base di dati per descrivere in dettaglio gli esperimenti

di indicizzazione, si ritiene fondamentale utilizzare delle entità che definiscono l'esperimento, la collezione dei dati utilizzata, ed i risultati ottenuti.

2.1 Entità

Definiamo le Entità necessarie per la progettazione logica della base di dati.

Experiment
L'entità EXPERIMENT è l'entità "cardine" di tutto il database: gli esperimenti sono i fatti che si vogliono memorizzare e gestire.
Per questa entità si definiscono gli attributi: *id*, necessario per identificare l'esperimento e *name*, ossia il nome dell'esperimento.

Document
L'entità DOCUMENT rappresenta l'oggetto documento inteso come un testo; è uno degli elementi fondamentali per ogni esperimento in quanto il documento fa parte del training set e del test set descritto in precedenza.
Per questa entità vengono definiti gli attributi: *id*, per identificare univocamente un documento, *idoncollection*, usato per definire l'identificazione all'interno della collezione, *title*, che identifica il titolo del documento, ed infine *content*, nel quale si descrive brevemente il contenuto del documento.

Collection
L'entità COLLECTION descrive la collezione di documenti utilizzata nell'esperimento.
Per questa entità vengono definiti gli attributi: *id*, per identificare univocamente la collezione, *name*, il nome della collezione e *description*, che conterrà una descrizione della collezione.

Category
L'entità CATEGORY conterrà le informazioni relative alle categorie utilizzate per le classificazione dei documenti usati negli esperimenti.
Per questa entità vengono definiti gli attributi: *id*, per identificare univocamente una categoria, *name* il nome della categoria, *description*, che conterrà una descrizione sul contenuto ed il tipo di categoria.

Split

L'entità Split descrivere il modo in cui i documenti presenti in una collezione vengono utilizzati: se come documenti di allenamento (training) o come documenti di verifica (test).
Per questa entità si definiscono gli attributi: *id*, necessario per identificare lo split e *description*, che conterrà una descrizione dello split.

Indexing

L'entità Indexing descrive la configurazione adottata dall'esperimento per classificare i documenti della collezione.
Per questa entità vengono definiti: *id*, per identificare univocamente l'indicizzazione e altri attributi che fanno riferimento alla possibile configurazione di Lucene[1] che descrivono le scelte dell'indicizzazione: *asciiFoldingFilter*, che indica se viene usato il filtro per i caratteri ascii, *lengthFilter*, che indica se considerare per l'indicizzazione tutto il testo o solo una sua parte, *lowerCaseFilter*, che indica se si considerano differenti i caratteri maiuscoli dai minuscoli, *stopFilter*, se si utilizza un filtro per alcune parole, *stopwords*, che contiene l'elenco delle stopword.

Metric

L'entità Metric andrà a definire i risultati ottenuti dall'esperimento per le categorie utilizzate e alcuni dei risultati ottenuti dall'esperimento nel suo insieme.
Per questa entità vengono definiti gli attributi: *id*, per identificare univocamente la metrica e *description*, che conterrà una descrizione sul significato della metrica.

Statistic

L'entità Statistic descrive i risultati ottenuti dall'esperimento nel suo complesso per quegli indicatori di performance che non vengono considerati una metrica.
Per questa entità viene definito l'attributo *id* che caratterizza e identifica la statistica descrittiva.

[1] Lucene è una libreria di Java concepita per realizzare applicazioni che necessitano di funzionalità di indicizzazione e ricerca full text.
sito: http://lucene.apache.org

2.2 Associazioni

Si andrà ad analizzare in dettaglio le relazioni che verranno utilizzate nello schema ER.

Adopt

Le entità coinvolte in questa relazione sono Experiment e Indexing.
Questa relazione descrive il fatto che ogni esperimento adotta una sola, indicizzazione, cioè una scelta dei parametri da utilizzare per la classificazione dei documenti; mentre la stessa indicizzazione potrebbe essere usata in più esperimenti.

Figura 2.1: Adopt

Use

Le entità coinvolte in questa relazione sono Experiment e Collection.
Questa relazione descrive il fatto che ogni esperimento utilizza una sola collezione di documenti, ma la stessa collezione di documenti può essere utilizzata in più esperimenti.

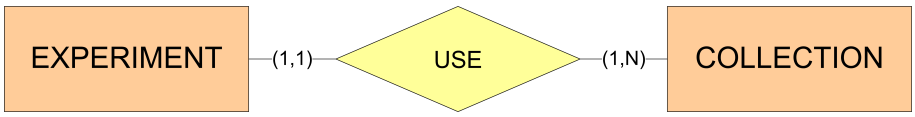

Figura 2.2: Use

Usesplit

Le entità coinvolte in questa relazione sono Experiment e Split.
Questa relazione descrive il fatto che in ogni esperimento viene utilizzato un solo

split per la collezione di documenti.
Lo split indica quali sono le categorie utilizzate nella classificazione, e come un documento della collezione viene usato, ossia indica se il documento viene utilizzato nel training set o nel test set dell'esperimento.

Figura 2.3: Usesplit

Outcome

Le entità coinvolte in questa relazione sono Experiment, Category e Document.
Questa relazione descrive il fatto che ad ogni documento utilizzato nell'esperimento, viene associata una categoria che è l'esito della classificazione di quel documento.
La cardinalità della relazione indica che ogni documento dell'esperimento è associato ad una categoria, ma ci possono essere delle categorie alle quali non è associato nessun documento.
Alla relazione outcome vengono associati due attributi: *rank* e *value* che forniscono delle indicazioni sulla bontà dei risultati ottenuti.

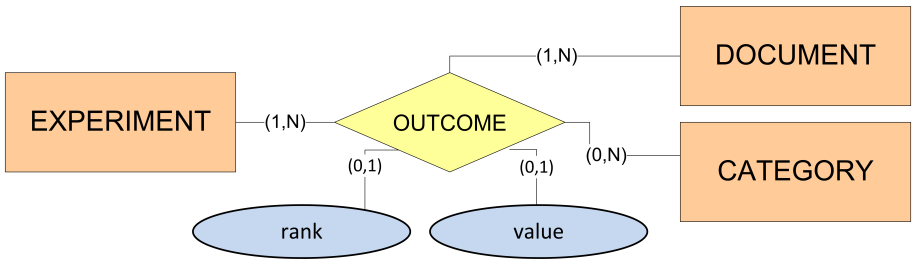

Figura 2.4: Outcome

GetMetricCategory

Le entità coinvolte in questa relazione sono Experiment, Metric e Category.
Questa relazione descrive il fatto che ad ogni esperimento vengono associate una

o più metriche. Le metriche descrivono il comportamento nell'esperimento di una particolare categoria.
La cardinalità della relazione indica anche che ci possono essere delle categorie alle quali non è associata alcuna metrica; un esperimento è associato almeno ad una coppia categoria-metrica; non tutte le metriche sono associate ad un esperimento e categoria.
Per la relazione viene definito l'attributo *value* che indica il valore della metrica ottenuta nell'esperimento dalla categoria.

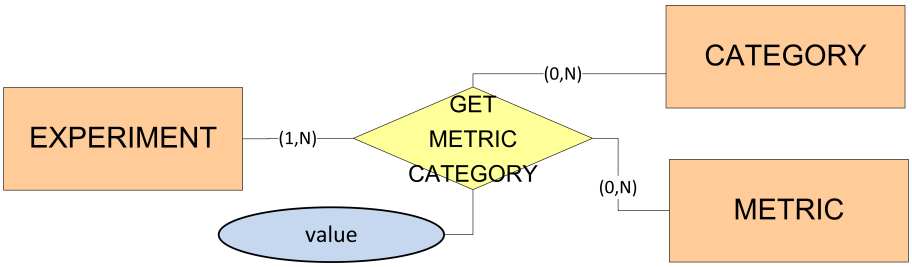

Figura 2.5: GetMetricCategory

GetStatistic

Le entità coinvolte in questa relazione sono Experiment, Metric e DescriptiveStatistics.
Questa relazione descrive il fatto che ad ogni esperimento vengono associate una o più metriche. Le metriche, in questo caso, danno una indicazione globale dell'esito dell'esperimento.
Alla relazione viene associato l'attributo *value* che indica il valore della metrica ottenuto per quella categoria.
La cardinalità della relazione indica che ogni statistica descrittiva è in relazione con un esperimento ed una metrica. Un esperimento ha più metriche e statistiche descrittive, ma non tutte le metriche sono associate ad una coppia esperimento-statistica descrittiva.

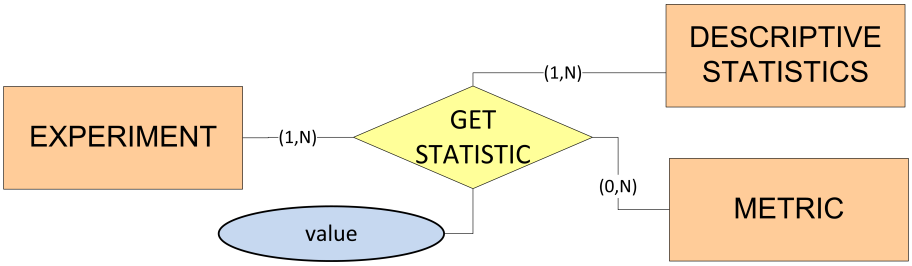

Figura 2.6: GetStatistic

GetMetric

Le entità coinvolte in questa relazione sono Experiment e Metric.
Questa relazione descrive il fatto che esistono metriche che si riferiscono all'esperimento e che non fanno parte delle statistiche descrittive.
Alla relazione viene associato l'attributo *value* che esprime il valore relativo alla metrica ottenuto dall'esperimento.

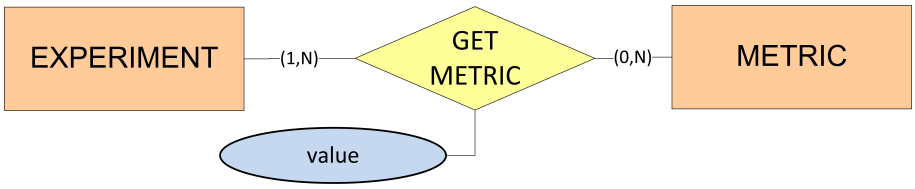

Figura 2.7: GetMetric

Indexes

Le entità coinvolte in questa relazione sono Indexing e Collection.
Questa relazione descrive il fatto che una configurazione di indicizzazione può essere utilizzata da più collezioni ed allo stesso tempo che la stessa collezione può utilizzare indicizzazioni diverse.
Alla relazione viene associato l'attributo *indexFile* che conterrà il nome del file di configurazione utilizzato in Lucene.

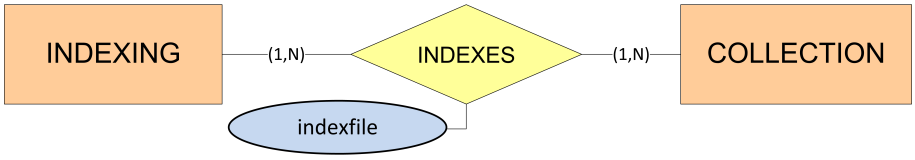

Figura 2.8: Indexes

DivideCollection

Le entità coinvolte in questa relazione sono Collection e Category.
Questa relazione descrive il fatto che ogni collezione è formata da documenti appartenenti a diverse categorie mentre una categoria viene associata unicamente ad una collezione.

Figura 2.9: DivideCollection

Contains

Le entità coinvolte in questa relazione sono Category e Document.
Questa relazione descrive il fatto che un documento fa parte di almeno una categoria ed allo stesso tempo una categoria contiene almeno un documento.

Figura 2.10: Contains

Divide

Le entità coinvolte in questa relazione sono Category, Split e Document.
Questa relazione descrive il fatto che uno split divide i documenti nelle relative categorie.

L'attributo della relazione *type* indica se il documento, appartenente alla categoria associata, fa parte del training set o del test set.
La cardinalità della relazione indica che uno split contiene almeno una coppia documento-categoria; non tutte le categorie sono utilizzate dallo split e queste categorie potrebbero non contenere documenti.

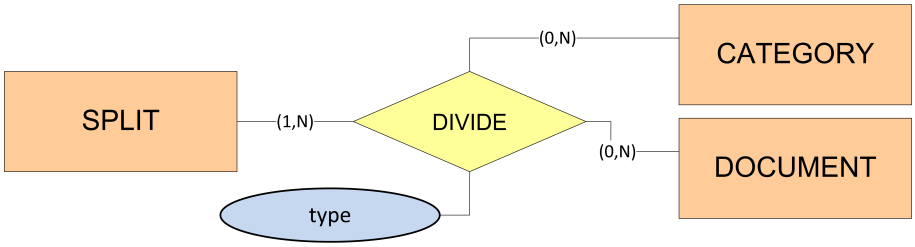

Figura 2.11: Divide

2.3 Schema E-R Complessivo

Nella pagina seguente lo schema E-R che riunisce tutte le relazioni espresse in precedenza.

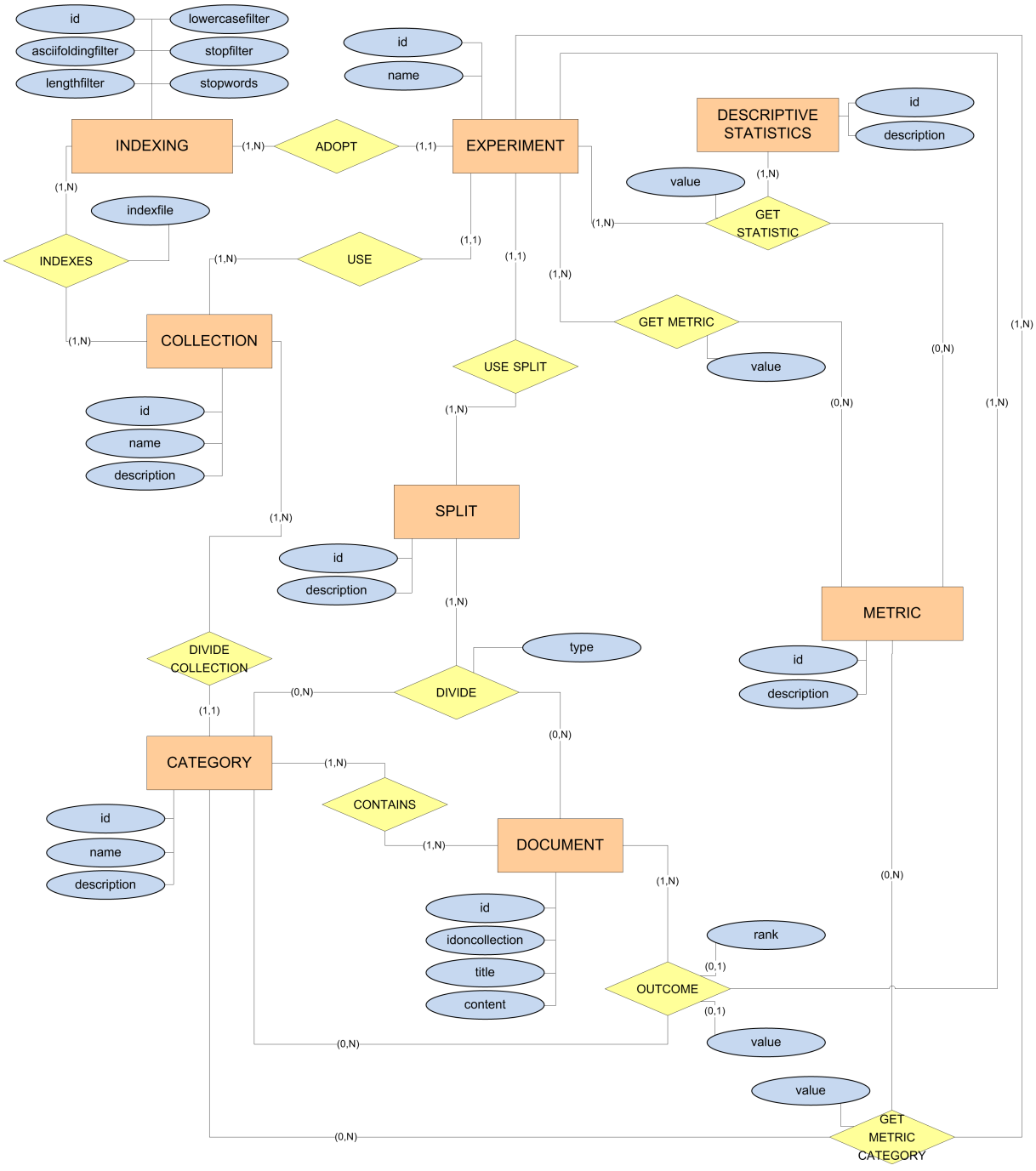

Figura 2.12: Schema E-R

2.3.1 Tabella riassuntiva

Di seguito la tabella riassuntiva delle entità e delle relazioni definite:

Entità	**Attributi**	**Attributi chiave**
Category	name, description	id
Collection	name, description	id
DescriptiveStatistics	description	id
Document	idonCollection, title, content	id
Experiment	name	id
Indexing	asciiFoldingFilter, lengthFilter, lowerCaseFilter, stopFilter, stopwords	id
Metric	description	id
Split	description	id

Tabella 2.1: Tabella riassuntiva della progettazione logica

2.3.2 Regole di derivazione

Una derivazione è un concetto che può essere ottenuto attraverso un'inferenza o un calcolo aritmetico da altri concetti dello schema.
Nello schema ER definito precedentemente, l'attributo *value* delle relazioni Getmetric, Getmetriccategory e Getstatistic è il risultato di calcoli aritmetici sul risultato della classificazione dei testi.
Per calcolare queste formule si va a considerare come tutti i documenti utilizzati nel test vengono classificati: queste operazioni richiedono molto tempo per essere calcolate, quindi si preferisce avere disponibile direttamente il risultato finale per un accesso immediato all'informazione.

Capitolo 3

Progettazione logica

La progettazione logica consiste nella traduzione dello schema E-R precedentemente prodotto nel modello di rappresentazione dei dati adottato dal Database Management System.
Le regole base per la conversione sono le seguenti:

- Per ogni entità si definisce una tabella con lo stesso nome aventi per attributi gli stessi attributi e per chiave l'identificatore;

- Per ogni associazione si definisce una tabella con lo stesso nome avente per attributi gli stessi attributi e per chiave gli identificatori delle entità coinvolte (legate con vincolo di integrità referenziale);

- Per ogni regola di vincolo si definisce un opportuno vincolo di integrità;

Per la migliore comprensione si ricorda che la descrizione di una tabella sarà composta nel seguente modo:
NOME (**attributo1**, **attributo2**, attributo3...)
dove:
NOME indica il nome della tabella;
attributo1 : indica il campo chiave (se più attributi sono sottolineati c'è una chiave composta);
attributo2 : essendo in grassetto, indica una campo obbligatorio;
attributo3 : indica un attributo non obbligatorio.
Graficamente ogni tabella viene rappresentata nel seguente modo:

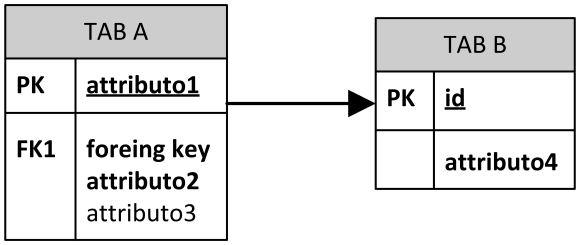

Figura 3.1: Esempio tabella schema logico

3.1 Conversione delle relazioni

Si andrà ad analizzare in dettaglio ogni conversione per le relazioni definite precedentemente nello modello E-R.

Adopt

Per la relazione ADOPT vista precedentemente nella Fig. 2.1 vengono tradotte le seguenti tabelle:

EXPERIMENT (**id**, **id_indexing**, name)
INDEXING (**id**, **asciiFilter**, **legthFilter**, **lowercaseFilter**, **stopwordFilter**, stopwords)

Con vincoli di integrità referenziale tra gli identificatori: *Experiment.id_indexing* e *Indexing.id*.

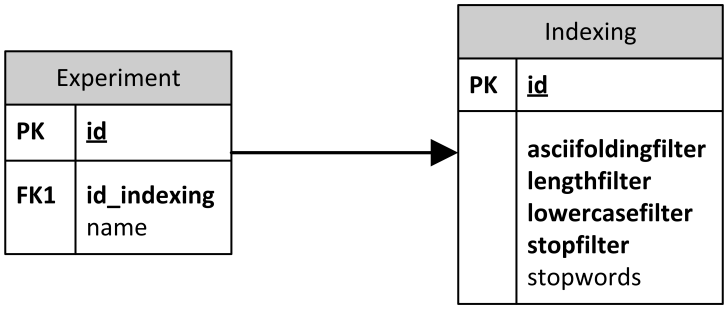

Figura 3.2: Schema logico: Adopt

Use

Per la relazione *Use* vista precedentemente nella Fig. 2.2 vengono tradotte le seguenti tabelle:

EXPERIMENT (**id**, **id_indexing**, **id_collection**, name)
COLLECTION (**id**, **name**, **description**)
Con vincoli di integrità referenziale tra gli identificatori: *Experiment.id_collection* e *Collection.id*.

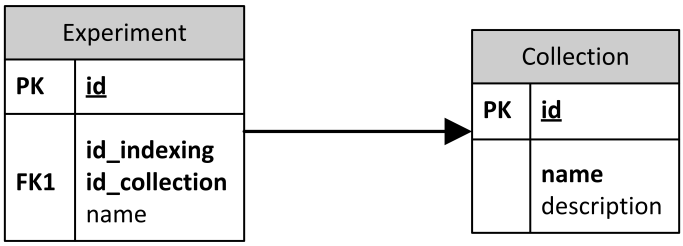

Figura 3.3: Schema logico: Use

Usesplit

Per la relazione *Usesplit* vista precedentemente nella Fig. 2.3 vengono tradotte le seguenti tabelle:

EXPERIMENT (**id**, **id_indexing**, **id_collection**, name)
SPLIT (**id**, description)
USESPLIT (**id_experiment**, **id_split**)

Con vincoli di integrità referenziale tra gli identificatori: *id_experiment* e *id_split*.
Questa relazione può essere migliorata eliminando la tabella *USESPLIT* ed inserendo nella tabella *EXPERIMENT* un nuovo campo con l'identificatore della collezione associata.
Le tabelle diventerebbero quindi:

EXPERIMENT (**id**, **id_indexing**, **id_collection**, **id_split**, name)
SPLIT (**id**, description)

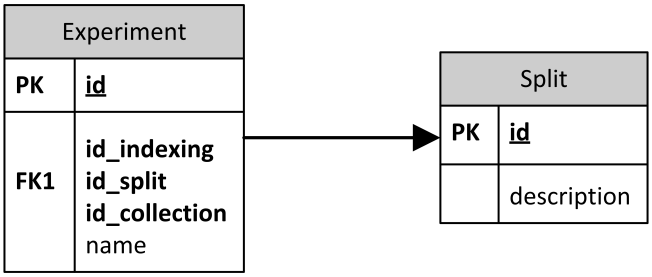

Figura 3.4: Schema logico: Usesplit

Outcome

Per la relazione *Outcome* vista precedentemente nella Fig. 2.4 vengono tradotte le seguenti tabelle:

EXPERIMENT (**id**, **id_indexing**, **id_collection**, **id_split**, name)
DOCUMENT (**id**, idonCollection, title, content)
CATEGORY (**id**, **name**, description)
OUTCOME (**id_experiment**, **id_document**, **id_category**, rank, value)

Con vincoli di integrità referenziale tra gli identificatori: *id_experiment*, *id_document* e *id_category*.

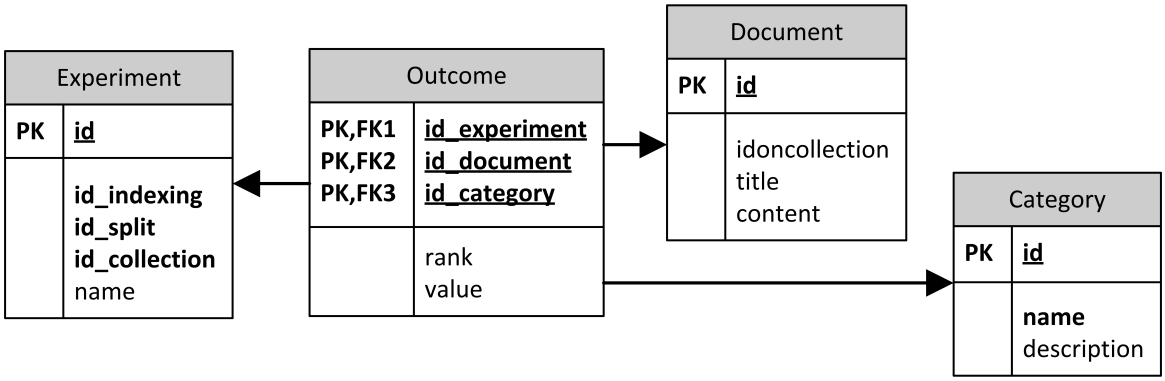

Figura 3.5: Schema logico: Outcome

GetMetricCategory

Per la relazione *GetMetricCategory* vista precedentemente nella Fig. 2.5 vengono tradotte le seguenti tabelle:

EXPERIMENT (**id**, **id_indexing**, **id_collection**, **id_split**, name)
CATEGORY (**id**, **name**, description)
METRIC (**id**, description)
GETMETRICCATEGORY (**id_experiment**, **id_category**, **id_metric**, value)

Con vincoli di integrità referenziale tra gli identificatori: *id_experiment*, *id_category* e *id_metric*.

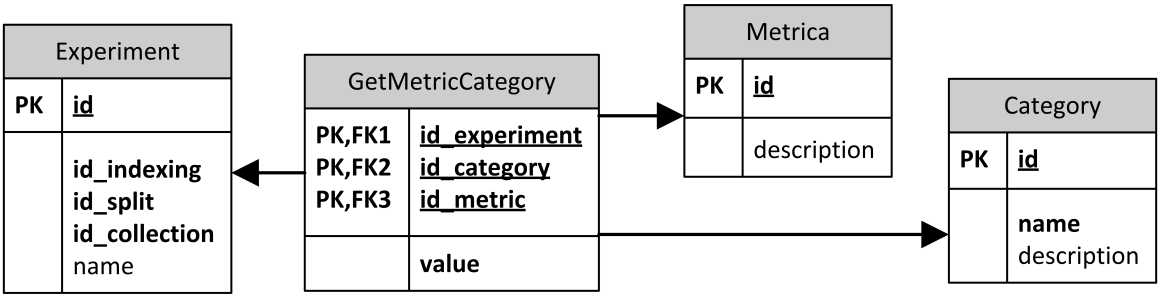

Figura 3.6: Schema logico: GetMetricCategory

GetStatistic

Per la relazione *GetStatistic* vista precedentemente nella Fig. 2.6 vengono tradotte le seguenti tabelle:

EXPERIMENT (**id**, **id_indexing**, **id_collection**, **id_split**, name)
DESCRIPTIVESTATISTICS (**id**, description)
METRIC (**id**, description)
GETSTATISTIC (**id_experiment**, **id_statistic**, **id_metrica**, value)

Con vincoli di integrità referenziale tra gli identificatori: *id_experiment*, *id_statistic* e *id_metric*.

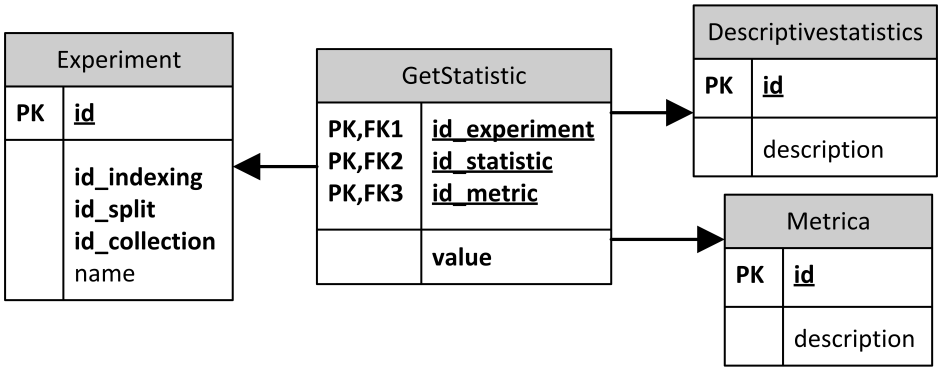

Figura 3.7: Schema logico: GetStatistic

GetMetric

Per la relazione *GetMetric* vista precedentemente nella Fig. 2.7 vengono tradotte le seguenti tabelle:

EXPERIMENT (**id**, **id_indexing**, **id_collection**, **id_split**, name)
METRIC (**id**, description)
GETMETRIC (**id_experiment**, **id_metric**, value)

Con vincoli di integrità referenziale tra gli identificatori: *id_experiment* e *id_metric*.

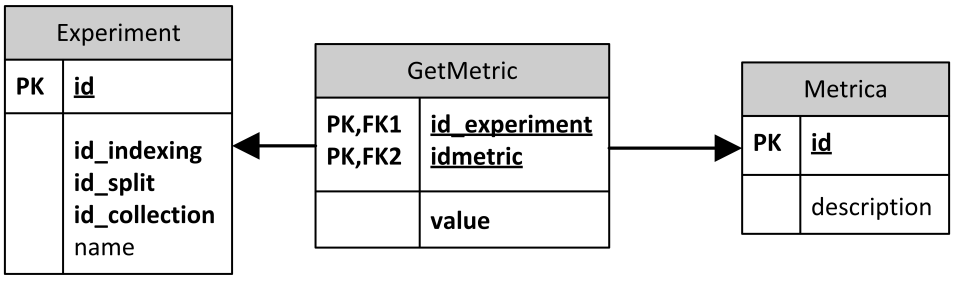

Figura 3.8: Schema logico: GetMetric

Indexes

Per la relazione *Indexes* vista precedentemente nella Fig. 2.8 vengono tradotte le seguenti tabelle:

INDEXING (**id**, **asciiFilter**, **legthFilter**, **lowercaseFilter**, **stopwordFil-**

ter, stopwords)
COLLECTION (**id**, **name**, description)
INDEXES (**id_indexing**, **id_collection**, indexFile)

Con vincoli di integrità referenziale tra gli identificatori: *id_indexing* e *id_collection*.

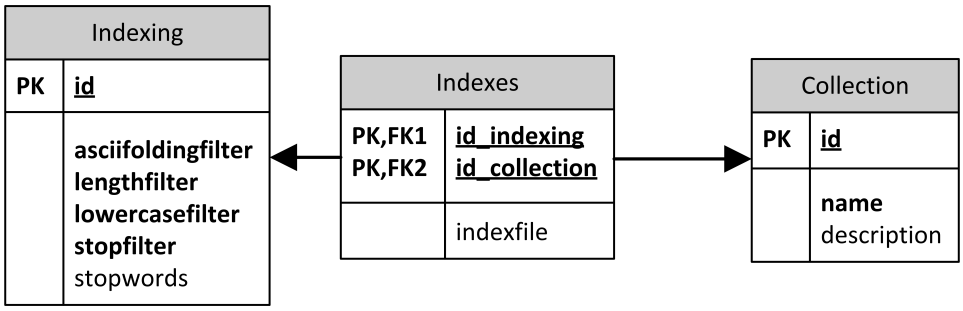

Figura 3.9: Schema logico: Indexes

DivideCollection

Per la relazione *DivideCollection* vista precedentemente nella Fig. 2.9 vengono tradotte le seguenti tabelle:

COLLECTION (**id**, **name**, description)
CATEGORY (**id**, **name**, description)
DIVIDE (**id_collection**, **id_category**)

Con vincoli di integrità referenziale tra gli identificatori: *id_collection* e *id_category*.
Questa relazione può essere migliorata eliminando la tabella *DIVIDE* ed inserendo nella tabella *CATEGORY* un nuovo campo con l'identificatore della collezione associata.
Le tabelle diventerebbero quindi:

COLLECTION (**id**, **name**, description)
CATEGORY (**id**, **name**, description, **id_collection**)

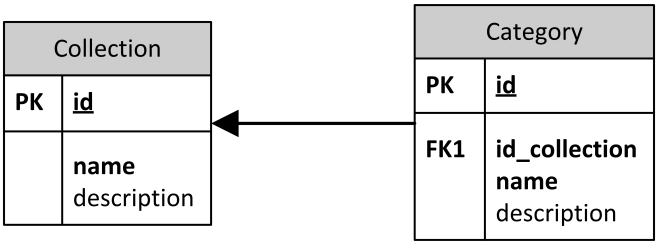

Figura 3.10: Schema logico: DivideCollection

Contains

Per la relazione *Contains* vista precedentemente nella Fig. 2.10 vengono tradotte le seguenti tabelle:

CATEGORY (**id**, **name**, description)
DOCUMENT (**id**, idonCollection, title, content)
CONTAINS (**id_category**, **id_document**)

Con vincoli di integrità referenziale tra gli identificatori: *id_category* e *id_document*.

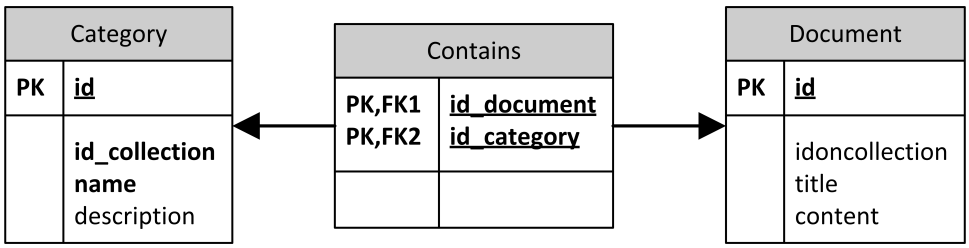

Figura 3.11: Schema logico: Contains

Divide

Per la relazione *Divide* vista precedentemente nella Fig. 2.11 vengono tradotte le seguenti tabelle:

CATEGORY (**id**, **name**, description)
SPLIT (**id**, description)
DOCUMENT (**id**, idonCollection, title, content)
DIVIDE (**id_category**, **id_document**, **id_split**)

Con vincoli di integrità referenziale tra gli identificatori: *id_category*, *id_split* e *id_document*.

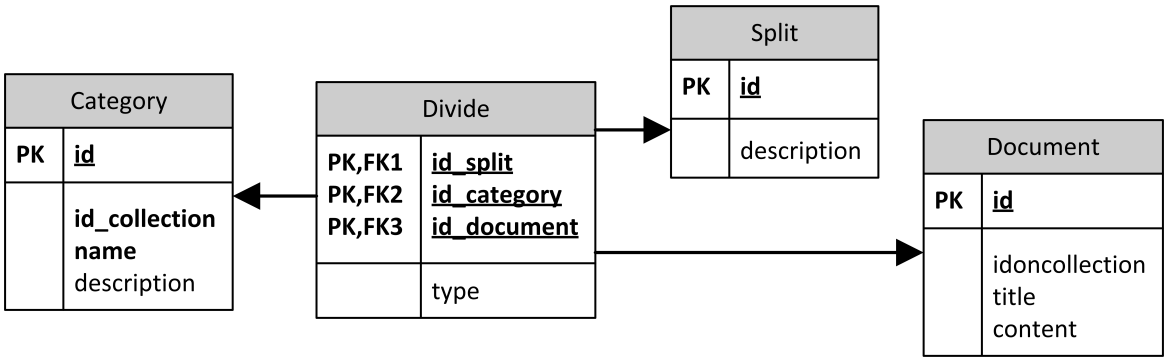

Figura 3.12: Schema logico: Divide

3.2 Schema Logico Complessivo

Nella pagina seguente lo schema logico complessivo, prodotto dall'unione di tutte le relazione descritte in precedenza.

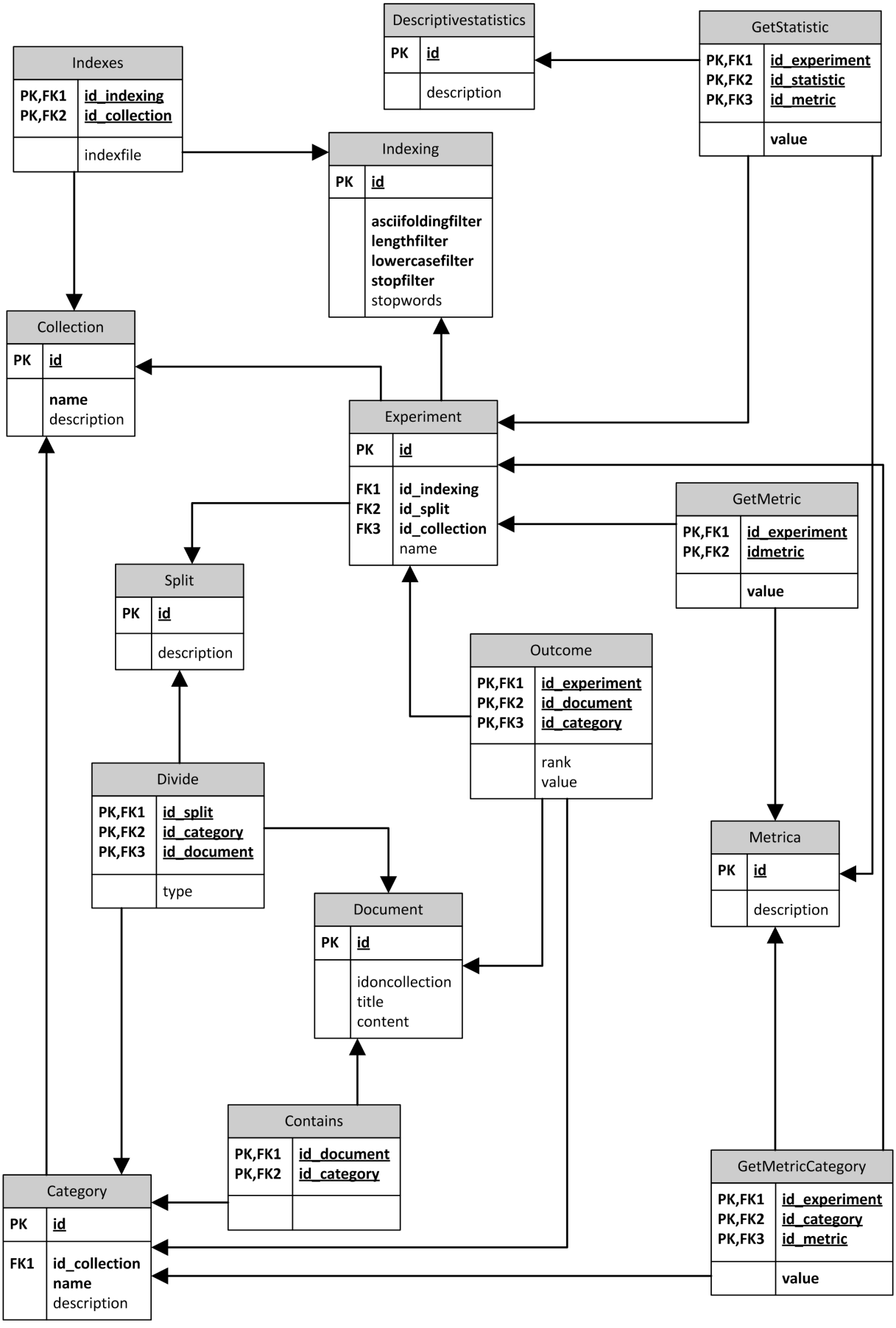

Figura 3.13: Schema Logico

Capitolo 4

Progettazione fisica

La terza ed ultima fase di progettazione di una base di dati è quella fisica, nella quale si completa lo schema logico prodotto nelle fasi precedenti con la specifica dei parametri fisici di memorizzazione dei dati.
Ogni forma di memorizzazione dipende soprattutto dall'ambiente che si è scelto di utilizzare. Per l'implementazione di questo progetto si è scelto di utilizzare PostgresSQL.
In questa fase vengono generati gli script che andranno a creare il database.

Definizione tabella *Category*

```
CREATE TABLE category (
  id integer NOT NULL,
  id_collection integer NOT NULL,
  name character varying(255) NOT NULL,
  description character varying(255),
  CONSTRAINT category_pkey PRIMARY KEY (id),
  CONSTRAINT fk_categorycollection FOREIGN KEY (id_collection)
    REFERENCES collection (id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION
)
```

Definizione tabella *Collection*

```
CREATE TABLE collection (
```

```
  id integer NOT NULL,
  name character varying(255) NOT NULL,
  description character varying(255),
  CONSTRAINT collection_pkey PRIMARY KEY (id)
)
```

Definizione tabella *Contains*

```
CREATE TABLE contains (
  id\_category integer NOT NULL,
  id\_document bigint NOT NULL,
  CONSTRAINT contains_pkey PRIMARY KEY (id_document, id_category),
  CONSTRAINT fk_containsdocument FOREIGN KEY (id_document)
    REFERENCES document (id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT fk_containscategory FOREIGN KEY (id_category)
    REFERENCES category (id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION
)
```

Definizione tabella *Descriptivestatistics*

```
CREATE TABLE descriptivestatistics (
  id character varying(255) NOT NULL,
  description character varying(255),
  CONSTRAINT descriptivestatistics_pkey PRIMARY KEY (id)
)
```

Definizione tabella *Divide*

```
CREATE TABLE divide (
  id_split integer NOT NULL,
  id_category integer NOT NULL,
```

```
  id_document bigint NOT NULL,
  type character varying(255),
  CONSTRAINT divide_pkey PRIMARY KEY (id_split, id_category, ↩
      id_document),
  CONSTRAINT fk_dividedocument FOREIGN KEY (id_document)
    REFERENCES document (id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT fk_dividesplit FOREIGN KEY (id_split)
    REFERENCES split (id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT fk_dividecategory FOREIGN KEY (id_category)
    REFERENCES category (id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION
)
```

Definizione tabella *Document*

```
CREATE TABLE document (
  id bigint NOT NULL,
  idoncollection character varying(255),
  title character varying(255),
  content character varying(255),
  CONSTRAINT document_pkey PRIMARY KEY (id)
)
```

Definizione tabella *Experiment*

```
CREATE TABLE experiment (
  id integer NOT NULL,
  id_collection integer NOT NULL,
  id_indexing integer NOT NULL,
  id_split integer NOT NULL,
  name character varying(255),
  CONSTRAINT experiment_pkey PRIMARY KEY (id),
```

```
  CONSTRAINT fk_experimentindexing FOREIGN KEY (id_indexing)
    REFERENCES indexing (id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION,
  COSTRAINT fk_experimentsplit FOREIGN KEY (id_split)
    REFERENCES split (id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT fk_experimentcollection FOREIGN KEY (id_collection)
    REFERENCES collection (id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION
)
```

Definizione tabella *Getmetric*

```
CREATE TABLE getmetric (
  id_experiment integer NOT NULL,
  id_metric character varying(255) NOT NULL,
  value double precision NOT NULL,
  CONSTRAINT getmetric_pkey PRIMARY KEY (id_experiment, id_metric),
  CONSTRAINT fk_getmetricmetric FOREIGN KEY (id_metric)
    REFERENCES metric (id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT fk_getmetricexperiment FOREIGN KEY (id_experiment)
    REFERENCES experiment (id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION
)
```

Definizione tabella *Getmetriccategory*

```
CREATE TABLE getmetriccategory (
  id_experiment integer NOT NULL,
  id_category integer NOT NULL,
  id_metric character varying(255) NOT NULL,
  value double precision NOT NULL,
```

```
  CONSTRAINT getmetriccategory_pkey PRIMARY KEY (id_experiment, ↩
      id_category, id_metric),
  CONSTRAINT fk_getmetriccatmetric FOREIGN KEY (id_metric)
    REFERENCES metric (id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT fk_getmetriccatexperiment FOREIGN KEY (id_experiment)
    REFERENCES experiment (id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT fk_getmetriccatcategory FOREIGN KEY (id_category)
    REFERENCES category (id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION
```

Definizione tabella *Getstatistic*

```
CREATE TABLE getstatistic (
  id_experiment integer NOT NULL,
  id_statistic character varying(255) NOT NULL,
  id_metric character varying(255) NOT NULL,
  value double precision NOT NULL,
  CONSTRAINT getstatistic_pkey PRIMARY KEY (id_experiment, ↩
      id_statistic, id_metric),
  CONSTRAINT fk_getstatisticmetric FOREIGN KEY (id_metric)
    REFERENCES metric (id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT fk_getstatisticstatistic FOREIGN KEY (id_statistic)
    REFERENCES descriptivestatistics (id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT fk_getstatisticexperiment FOREIGN KEY (id_experiment)
    REFERENCES experiment (id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION
)
```

Definizione tabella *Indexes*

```
CREATE TABLE indexes (
  id_collection integer NOT NULL,
  id_indexing integer NOT NULL,
  indexfile character varying(255),
  CONSTRAINT indexes_pkey PRIMARY KEY (id_collection, id_indexing),
  CONSTRAINT fk_indexesindexing FOREIGN KEY (id_indexing)
    REFERENCES indexing (id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT fk_indexescollection FOREIGN KEY (id_collection)
    REFERENCES collection (id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION
)
```

Definizione tabella *Indexing*

```
CREATE TABLE indexing (
  id integer NOT NULL,
  asciifoldingfilter boolean NOT NULL,
  lengthfilter boolean NOT NULL,
  lowercasefilter boolean NOT NULL,
  stopfilter boolean NOT NULL,
  stopwords character varying(255),
  CONSTRAINT indexing_pkey PRIMARY KEY (id)
)
```

Definizione tabella *Metric*

```
CREATE TABLE metric (
  id character varying(255) NOT NULL,
  description character varying(255),
  CONSTRAINT metric_pkey PRIMARY KEY (id)
)
```

Definizione tabella *Outcome*

```
CREATE TABLE outcome (
  id_experiment integer NOT NULL,
  id_document bigint NOT NULL,
  id_category integer NOT NULL,
  rank character varying(255),
  value double precision,
  CONSTRAINT outcome_pkey PRIMARY KEY (id_experiment, ↩
      id_document, id_category),
  CONSTRAINT fk_outcomedocument FOREIGN KEY (id_document)
    REFERENCES document (id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT fk_outcomeexperiment FOREIGN KEY (id_experiment)
    REFERENCES experiment (id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT fk_outcomecategory FOREIGN KEY (id_category)
    REFERENCES category (id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION
)
```

Definizione tabella *Split*

```
CREATE TABLE split (
  id integer NOT NULL,
  description character varying(255),
  CONSTRAINT split_pkey PRIMARY KEY (id)
)
```

Capitolo 5

Architettura Spring-Hibernate

Dopo aver progettato la base di dati, si è pensato di costruire ed implementare un progetto Java che, utilizzando le architetture Spring e Hibernate permettesse di gestire ed interrogare il database attraverso l'invocazione di metodi su oggetti Java.
Hibernate[1] è uno dei più noti framework Java per l'Object Relational Mapping[2] che semplifica e automatizza la gestione della corrispondenza tra il modello ad oggetti delle applicazioni e quello relazionale dei database ai quali queste ultime comunemente si interfacciano.
La caratteristica principale di Hibernate è quella di fornire un mapping delle classi Java in tabelle di un database relazionale; sulla base di questo mapping Hibernate gestisce il salvataggio degli oggetti di tali classi sul database. Si occupa inoltre del reperimento degli oggetti dal database, producendo ed eseguendo automaticamente le query SQL necessarie al recupero delle informazioni e la successiva reistanziazione dell'oggetto precedentemente reperito dal database.
Hibenate si pone quindi tra il modello ad oggetti che si ricava dalla logica della nostra applicazione e l'insieme delle relazioni.
Questo modulo nasconde le problematiche legate alla gestione delle connessioni, delle transazioni e delle eccezioni che altrimenti con i costrutti JDBC andrebbero gestite manualmente.
Per comprendere a fondo le potenzialità di Spring, invece, bisogna introdurre i

[1] http://www.hibernate.org

[2] Un ORM, acronimo di Object Relation Mapping, è una tecnologia che nasce per semplificare la persistenza di oggetti in un database relazionale, generando automaticamente il codice SQL necessario.

concetti di *Inversion of Control (IoC)* e *Dependency Injection (DI)*.
L'*Inversion of Control* è un principio architetturale basato sul concetto di invertire il controllo del flusso di sistema (Control Flow) rispetto alla programmazione tradizionale. Questo principio è molto utilizzato nei framework e ne rappresenta una delle caratteristiche basilari che distingue i framework dalle API. Nella programmazione tradizionale la logica di tale flusso è definita esplicitamente dallo sviluppatore, che si occupa tra le altre cose di tutte le operazioni di creazione, inizializzazione ed invocazione dei metodi degli oggetti. L'*Inversion of Control* invece inverte il controllo del flusso facendo in modo che non sia più lo sviluppatore a doversi preoccupare di questi aspetti ma sarà il framework che se ne farà carico reagendo ad un qualche "stimolo" esterno.
Il termine *Dependency Injection (DI)* viene usato per riferirsi ad una specifica implementazione dell'*Inversion of Control* rivolta ad invertire il processo di risoluzione delle dipendenze, facendo in modo che queste vengano iniettate dall'esterno. L'idea alla base della Dependency Injection è quella di avere un componente esterno che si occupi della creazione degli oggetti e delle loro relative dipendenze e di assemblarle mediante l'utilizzo dell'injection. In particolare esistono tre forme di injection:

- *Constructor Injection*, dove la dipendenza viene iniettata tramite l'argomento del costruttore;
- *Setter Injection*, dove la dipendenza viene iniettata attraverso un metodo "set";
- *Interface Injection* che si basa sul mapping tra interfaccia e relativa implementazione (non utilizzato in Spring);

Spring quindi fornisce un livello di astrazione per l'accesso ai dati mediante il framework Hibernate.
Per funzionare correttamente Spring ed Hibernate necessitano della definizione di JavaBean per la gestione degli oggetti che descrivono la struttura della tabella di rifermento e le relazioni con altre tabelle.
Questi JavaBean sono classi usate per incapsulare più oggetti in un singolo oggetto (il bean), così da poter utilizzare il bean come unico oggetto per la comunicazione invece degli oggetti individuali.
Le convenzioni richieste per una classe JavaBeans sono:

- La classe deve avere un costruttore senza argomenti;
- Le sue proprietà devono essere accessibili usando getProperty, setProperty, isProperty (usato per i booleani al posto di get) e altri metodi (così detti metodi accessori) seguendo una convenzione standard per i nomi;
- La classe dovrebbe essere serializzabile (capace di salvare e ripristinare il suo stato in modo persistente);
- Non dovrebbe contenere alcun metodo richiesto per la gestione degli eventi;

Infine, per accedere ai dati si utilizzano i Data Access Object (DAO)[3] ossia oggetti che forniscono un'interfaccia astratta per un database o un meccanismo di persistenza, mettendo a disposizione alcune operazioni specifiche senza dipendere dai dettagli del database.
IL DAO fornisce una mappatura dalle chiamate di applicazioni per lo strato di persistenza.
Questo isolamento separa quanto concerne le esigenze applicative per l'accesso ai dati, in termini di specifici oggetti e tipi di dati utilizzati (l'interfaccia pubblica del DAO vera e propria), da come queste esigenze possono essere soddisfatte con uno specifico DBMS, lo schema del database, ecc (l'attuazione della DAO).

5.1 File di configurazione

Per permettere al progetto di gestire correttamente il database bisogna creare dei file xml sia per il riconoscimento e la connessione del database, sia per definire la struttura delle singole tabelle.
I file di configurazione necessari sono:

- hibernate.cfg.xml;
- beans.xml;
- file xml di mapping delle tabelle del DB;

[3]Maggiori dettagli si possono trovare nella documentazione ufficiale http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

5.1.1 Configurazione di Hibernate

Il file *hibernate.cfg.xml* è il file di configurazione utilizzato da Hibernate.
Prendiamo come esempio il file utilizzato nel progetto per analizzare la struttura:

```
<hibernate−configuration>
  <session−factory>
    [...]
  </session−factory>
</hibernate−configuration>
```

Tra i tag <session−factory> vengono inseriti i parametri di configurazione per la connessione.
Per prima cosa si va a definire con che tipo di database si dovrà interfacciare Hibernate.
Andremo quindi ad inserire i parametri relativi al linguaggio SQL ed al driver da utilizzare, l'URL dove trovare l'istanza del database e username e password per accedere all'istanza.
Successivamente c'è la possibilità di definire vari parametri per configurare il funzionamento di Hibernate, come ad esempio la proprietà di creare/ricreare il database ad ogni esecuzione del progetto, seguendo le specifiche dei file xml di configurazione delle tabelle, usare una classe particolare per il sistema di cache per la comunicazione col database o definire il tipo di sessione da utilizzare.
Dopo aver definito tutti i parametri per la configurazione della connessione, si andranno ad inserire i riferimenti alla struttura delle entità presenti nel database e il JavaBean corrispondente.
Per fare questo vengono usati i file xml di mappatura, per i quali saranno presenti dei riferimenti all'interno del file *hibernate.cfg.xml*, per essere riconosciuti da Hibernate.
Di seguito un esempio di codice da inserire per mappare le tabelle che utilizzeremo nel database:

```
<mapping resource="db/table/Category.hbm.xml"/>
```

Il file hibernate.cfg.xml finale sarà quindi:

```
<hibernate−configuration>
  <session−factory>
```

```
<!-- hibernate dialect -->
<property name="hibernate.dialect">
    org.hibernate.dialect.PostgreSQLDialect</property>
<property name="hibernate.connection.driver_class">
    org.postgresql.Driver</property>
<property name="hibernate.connection.url">
    jdbc:postgresql://localhost:5432/experiment</property>
<property name="hibernate.connection.username">
    hibernate</property>
<property name="hibernate.connection.password">
    hibernate</property>

[...]

<!-- mapping to xml table description -->
<mapping resource="db/table/Category.hbm.xml"/>
[...]

  </session-factory>
</hibernate-configuration>
```

5.1.2 Configurazione di Spring

Il file *beans.xml* è il file di configurazione utilizzato da Spring.
Al suo interno vengono definiti i vari Bean da utilizzare per la comunicazione con il database.
Andiamo ad analizzare le parti che compongono il file xml.
Come per Hibernate anche Spring ha bisogno del Bean di configurazione della connessione; in questo caso identificato dal Bean dataSource. Questo Bean contiene le definizioni del driver per la comunicazione, l'URL dell'istanza e username e password per accedere al database.
Successivamente si andrà a definire il Bean sessionFactory che si occuperà di creare e gestire la sessione.
Al suo interno troviamo più proprietà: il datasource da utilizzare, la mappatura delle risorse ed infine le proprietà relative ad Hibernate.

Il prossimo Bean che verrà definito descrive l'HibernateTemplate ossia l'oggetto che viene utilizzato per gestire la sessione all'interno delle classi Java.
Infine, nel file xml si andranno ad inserire i Bean con la definizione degli oggetti che identificano i DAO, ossia le classi che gestiscono l'accesso ai dati.
Il file di configurazione finale sarà il seguente:

```
<bean id="dataSource" class="org.springframework.jdbc.
        datasource.DriverManagerDataSource">
  <property name="driverClassName"
    value="org.postgresql.Driver" />
  <property name="url"
    value="jdbc:postgresql://localhost:5432/experiment"/>
  <property name="username" value="hibernate" />
  <property name="password" value="hibernate" />
</bean>

<bean id="sessionFactory" class="org.springframework.
        orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="mappingResources">
   <list>
    <value>db/table/Category.hbm.xml</value>
    [...]
   </list>
  </property>
  <property name="hibernateProperties">
  <props>
    <prop key="hibernate.dialect">
        org.hibernate.dialect.PostgreSQLDialect</prop>
    <prop key="hibernate.show_sql">true</prop>
    <prop key="hibernate.hbm2ddl.auto">update</prop>
  </props>
  </property>
</bean>
```

```
<bean id="hibernateTemplate" class=
    "org.springframework.orm.hibernate3.HibernateTemplate">
  <property name="sessionFactory" ref="sessionFactory" />
</bean>

<bean name="categoryDAO" class=
    "datastore.hibernate.CategoryHibernateDAO">
  <property name="hibernateTemplate" ref="hibernateTemplate"/>
</bean>
[...]
```

5.2 Mapping

Hibernate ha bisogno di sapere come caricare e memorizzare gli oggetti della classe persistente.
Il file di mapping della classe descrive la struttura fisica della tabella nel database ed il tipo di associazione con i dati persistenti nel JavaBeans associato.
Per prima cosa viene definita la classe Java che verrà usata come oggetto persistente della tabella, ed i riferimenti alla tabella. Successivamente si andrà a definire per ogni variabile contenuta nella classe a quale campo della tabella corrisponde.
Nell'ordine viene definito:

- id, l'identificatore della tabella;
- attributi presenti nella tabella;
- eventuali associazioni tra tabelle;

Nei prossimi paragrafi sarà analizzato più in dettaglio la composizione di questi tre aspetti.

Mapping identificatori

L'elemento <id> definisce il mapping e le proprietà del campo chiave.
Esistono due tipi di identificatori: quelli semplici, nei quali si usa un unico campo come chiave, e quelli composti, nei quali la chiave viene identificata come una

combinazione di più campi.
Le classi mappate devono dichiarare il campo della chiave primaria della tabella del database.
Nel caso in cui si utilizzi una chiave composta, la classe utilizzata per definire le chiave, sarà un nuovo oggetto che descrive la struttura della chiave.
Questa nuova classe che definisce la chiave deve sovrascrivere i metodi equals() e hashCode() per consentire di comparare gli identificatori; inoltre, la classe deve implementare java.io.Serializable.

Mapping attributi

Il mapping degli attributi avviene attraverso il tag <property> all'interno della definizione della classe nel file.
Il mapping associa un attributo della tabella, alla relativa variabile del Bean.

Mapping delle associazioni

Le associazioni vengono trattate in modo diverso a seconda del tipo di relazione che si vuole definire. Esistono quindi configurazioni diverse tra loro per le definizioni delle associazioni one-to-many, many-to-one e many-to-many.

Vantaggi mapping delle associazioni

La corretta definizione del mapping per le associazioni, permette ad Hibernate non solo di riconoscere correttamente la struttura del database, ma permette anche la gestione dei record nel database nell'ordine corretto per rispettare i vincoli d'integrità.
Per rendere più chiara l'affermazione usiamo un esempio.
Se per esempio viene creato un oggetto di tipo Category, e a questo oggetto viene valorizzata la variabile collection, Hibernate controllerà se esiste un record nella tabella Collection con *id* presente nella variabile dell'oggetto Category; in caso quella collezione non sia stata definita, il record per la collezione verrà creato e successivamente verrà inserito nella tabella Category il record che fa riferimento alla collezione appena creata. Se invece all'oggetto Category viene valorizzata la variabile documents con una lista di documenti, nel momento in cui si chiederà ad Hibernate di inserire fisicamente nel database l'oggetto Category, il framework

si occuperà di inserire nella tabella Category il record specificato dalle variabili, verificare se nella tabella Document sono presenti record con gli *id* presenti nel Set<Document> dell'oggetto Category, se non presenti inserirà nella tabella i record seguendo la mappatura dell'oggetto Document, infine andrà ad inserire nella tabella Contains i riferimenti della relazione many-to-many. In questo modo rispetterà tutti i vincoli di integrità.

5.3 Esempio di definizione

Per approfondire i temi trattati in questo capitolo, si propone come esempio la definizione e mappatura dell'entità Category.

5.3.1 Definizione del javabean Category

È importante capire come una entità fisica, e le relative relazioni, vengano tradotte in oggetti definiti da classi Java.
Iniziamo l'analisi della definizione dei JavaBeans prendendo come esempio la classe Category.

```
public class Category {
  //variabili
  private int id;
  private Collection collection;
  private String name;
  private String description;
  private Set<Outcome> outcomes = new HashSet<Outcome>(0);
  private Set<Getmetriccategory> getmetriccategories
          = new HashSet<Getmetriccategory>(0);
  private Set<Document> documents = new HashSet<Document>(0);
  private Set<Divide> divides = new HashSet<Divide>(0);

  //esempio di getter e setter
  public int getId(){
    return this.id;
  }
```

```
    public void setId(int id) {
      this.id = id;
    }
    public Set<Document> getDocuments() {
      return this.documents;
    }
    public void setDocuments(Set<Document> documents) {
      this.documents = documents;
    }
    [...]
  }
```

Descriviamo le parti che compongono la classe Category: per prima cosa troviamo la parte relativa alla definizione delle variabili che compongono un oggetto di tipo Category. Le variabili definite avranno lo stesso nome degli attributi presenti nella tabella a livello fisico nel database. A queste vengono aggiunte delle variabili di tipo Set<object> che sono utilizzate per relazioni della tabella.
Ad esempio nell'oggetto documents, definito come un Set<Document> troveremo, se esistenti, i documenti collegati alla categoria considerata.
Nella seconda parte del codice della classe, troviamo tutti metodi per l'accesso ai dati attraverso i metodi *getter* e *setter*.

5.3.2 Mapping Category

Il mapping della tabella Category è il seguente:

```
<hibernate-mapping>
 <class name="db.table.Category" table="category"
        schema="public">
  <id name="id" type="int">
   <column name="id" />
   <generator class="assigned" />
  </id>
 <many-to-one name="collection" class="db.table.Collection"
     fetch="select" cascade="save-update">
```

```
 <column name="id_collection" not-null="true" />
</many-to-one>
<property name="name" type="string">
 <column name="name" not-null="true" />
</property>
<property name="description" type="string">
 <column name="description" />
</property>
<set name="outcomes" table="outcome" inverse="true"
     lazy="true" fetch="select" cascade="save-update">
 <key>
  <column name="id_category" not-null="true" />
 </key>
 <one-to-many class="db.table.Outcome" />
</set>
<set name="getmetriccategories" table="getmetriccategory"
     inverse="true" lazy="true" fetch="select"
     cascade="save-update">
 <key>
  <column name="id_category" not-null="true" />
 </key>
 <one-to-many class="db.table.Getmetriccategory" />
</set>
<set name="documents" table="contains" inverse="false"
     lazy="true" fetch="select" cascade="save-update">
 <key>
  <column name="id_category" not-null="true" />
 </key>
 <many-to-many entity-name="db.table.Document">
 <column name="id_document" not-null="true" />
 </many-to-many>
</set>
<set name="divides" table="divide" inverse="true"
     lazy="true" fetch="select" cascade="all">
 <key>
```

```
    <column name="id_category" not-null="true" />
   </key>
   <one-to-many class="db.table.Divide" />
  </set>
 </class>
 </hibernate-mapping>
```

Innanzi tutto viene definita la classe Java che verrà usata come oggetto persistente della tabella, ed i riferimenti della tabella:
<class name="db.table.Category"table="category"schema="public">.
Successivamente si definisce per ogni variabile contenuta nella classe a quale campo della tabella corrisponde.
La prima associazione viene fatta per l'identificatore, tra l'attributo *id* della tabella e la variabile *id* della classe.
Successivamente viene mappato ogni campo nella relativa variabile di riferimento all'interno del JavaBean.
Nell'ultima parte del file vengono mappati i Set per le associazioni definite per la tabella.

5.3.3 DAO

Per l'accesso ai dati viene definita la classe CategoryDAO come segue:

```
public interface CategoryDAO {
  void create(Category c)
   throws DataAccessException, PropertyValueException;
  Category readCategory(int id)
   throws ObjectNotFoundException, DataAccessException;
  List<Category> listCategories() throws DataAccessException;
  void delete(Category c) throws DataAccessException;
  Set<Document> readDocuments(Category c)
   throws ObjectNotFoundException, DataAccessException;
  Set<Outcome> readOutcomes(Category c)
   throws ObjectNotFoundException, DataAccessException;
  Set<Getmetriccategory> readGetmetriccategories(Category c)
   throws ObjectNotFoundException, DataAccessException;
```

```
    Set<Divide> readDivides(Category c)
     throws ObjectNotFoundException, DataAccessException;
}
```

Nel codice della classe categoryDAO troviamo la definizione dei metodi per l'accesso ai dati:

- create(c) permette la creazione di un nuovo record nella tabella Category;
- readCategory(**int** id), ritorna il record della categoria con l'id richiesto;
- listCategories(), ritorna tutte le categorie presenti nella tabella;
- readDocuments(Category c), ritorna i documenti collegati alla categoria passata come parametro;
- readOutcomes(Category c), ritorna gli esiti della categoria;
- readGetmetriccategories(Category c), ritorna gli oggetti di tipo *Getmetriccategory* collegati alla categoria;
- readDivides(Category c), ritorna gli oggetti di tipo *Divide* collegati alla categoria.

Capitolo 6

Organizzazione del progetto

Il progetto Java è stato sviluppato utilizzando l'IDE Eclipse.
Si è pensato di dividere il progetto in package che contenessero ognuno una particolare tipologia di classi Java.
I package sono:

- db.table
- datastore
- datastore.hibernate
- datastore.exception
- datastore.impl
- util
- test

Nei prossimi paragrafi daremo una definizione di ogni package e analizzeremo le classi che lo compongono.

6.1 db.table

Il primo package costruito è stato db.table. Al suo interno troviamo:

- le definizioni dei JavaBeans;
- i file xml di configurazione delle mappatura;

Per quanto riguarda i file xml di mappatura, ne troveremo uno per ogni tabella mappata.
Per quanto riguarda i JavaBeans, nel package non troveremo solamente un bean per ogni tabella, ma per le tabelle che hanno la chiave composta da più campi, si è dovuto creare un ulteriore bean, in modo da poter gestire l'id come un unico oggetto.
Si è scelto di nominare queste classi-chiave seguendo la regola che alla classe-chiave venisse assegnato lo stesso nome della classe della tabella, con l'aggiunta di un "Id" finale.

6.2 datastore

Il package datastore contiene al suo interno le interfacce dei DAO.
Alle interfacce definite per ogni tabella, si è aggiunta la classe Datastore.java.
Questa interfaccia va ad estendere tutte le interfacce DAO definite; in questo modo, con un oggetto Datastore, si potrà accedere a tutte le entità del database.

6.3 datastore.hibernate

Nel package datastore.hibernate sono presenti le implementazioni delle interfacce DAO presenti nel package datastore.
Si è scelto di nominare le classi nel formato: NometabellaHibernateDAO.java.
Queste classi contengono nel nome la parola Hibernate perchè i metodi che le costituiscono sono stati implementati usando la struttura di Hibernate per l'interfaccimento con il database.

6.4 datastore.exception

Questo package contiene la classe di eccezione per la miglior politica di gestione degli errori nelle operazioni sul database.
La classe definita è ObjectNotFoundException; questa eccezione viene lanciata nel caso in cui non si ritrovi un elemento.

6.5 datastore.impl

Il package datastore.impl contiene al suo interno la classe DbmsDatastore.java: l'implementazione dell'interfaccia Datastore definita i precedenza.
In questa classe vengono sviluppati tutti i metodi di interfacciamento al database utilizzando i DAO creati in precedenza.

6.6 util

Nel package util si trovano delle classi di supporto usate per la generazione di file di log per monitorare il funzionamento del progetto.
Le classi definite sono LogExecution e Output. La prima viene utilizzata dal Datastore per tenere traccia delle operazioni fatte nel database, la seconda viene utilizzata dalle classi di test per tracciare le operazioni svolte ed i risultati ottenuti.

6.7 test

Nel package test si trovano le classi utilizzate per i test. Si è pensato di dividere i test in due categorie: i test di caricamento ed i test di modifica ed interrogazione.

6.7.1 Test di caricamento

Il test di caricamento è composto dalla classe Test_CaricamentoCompleto.java che va ad alimentare la base di dati con un esperimento di esempio.
Questa classe opera seguendo i seguenti passi:

1. Creare un file di monitoraggio delle operazioni;
2. Inserire nel file di monitoraggio il tempo di inizio dell'elaborazione;
3. Creare gli oggetti che andranno successivamente caricati;
4. Creare l'oggetto DbmsDatastore per l'interfacciamento con il database;
5. Richiamare i metodi create(oggetti) con gli oggetti creati in precedenza per il caricamento nel database;

6. Chiudere il file di monitoraggio inserendo il tempo di fine elaborazione e dando quindi una stima del tempo necessario per l'elaborazione.

Nel file di monitoraggio vengono inserite le righe di commento di ogni operazione, in modo da monitorare le fasi dell'elaborazione. Oltre a questo file si dispone del file di log creato con log4j che tiene traccia in modo più completo di tutte le operazioni compiute.
L'utilità di questo test non è stata solo quella di verificare che i metodi di caricamento funzionassero correttamente, ma è stata anche quella di avere una classe che permettesse di creare una situazione "standard" per l'esecuzione dei test successivi.

6.7.2 Test di modifica ed interrogazione

I test di modifica ed interrogazione sono stati divisi per entità.
Sono state quindi create tante classi di test quante sono le classi DAO definite.
La nomenclatura di ogni classe segue il formato: TestSpring_NomeTabella.java.
Le operazioni di test di ogni classe sono:

1. Creare un file di monitoraggio delle operazioni;
2. Inserire nel file di monitoraggio il tempo di inizio dell'elaborazione;
3. Creare gli eventuali oggetti necessari per i test successivi; compreso l'oggetto DbmsDatastore per l'interfacciamento con il database;
4. Richiamare tutti i metodi definiti nel DAO; eseguendo prima quelli di interrogazione e successivamente la cancellazione;
5. Ogni test inseriva nel file di monitoraggio il "titolo" de test e l'esito ("OK" oppure "ERRORE" e una breve descrizione dell'errore); in questo modo si poteva verificare con facilità i metodi che funzionavano in modo inatteso;
6. Chiudere il file di monitoraggio inserendo il tempo di fine elaborazione e dando quindi una stima del tempo necessario per l'elaborazione.

Anche in questo caso, oltre al file di monitoraggio, si dispone del file di log creato con la libreria log4j che tiene traccia in modo più completo di tutte le operazioni compiute.
Nei test eseguiti si è partiti sempre da una situazione "base", ottenuta eseguendo il test di caricamento completo.

Capitolo 7

Conclusioni

L'obiettivo di questo elaborato era quello di costruire uno strumento che permettesse di gestire in modo semplice la storicizzazione ed il confronto degli esperimenti di classificazione automatica di testi.
Per realizzare questo obiettivo si è da prima fatta l'analisi dei requisiti per comprendere quali sono le informazioni necessarie che caratterizzano un esperimento. Dallo studio del problema si è poi passati alla progettazione dello schema concettuale, successivamente tradotto nello schema logico che ha portato alla realizzazione fisica della base di dati che dà la possibilità di memorizzare tutte le informazioni caratterizzanti di un esperimento. Dopo la progettazione sono state fatte delle prove di caricamento nella base di dati di un esperimento di esempio, utilizzando comandi SQL, per verificarne la corretta progettazione. In questa fase sono stati verificati i vincoli di precedenza sul popolamento delle entità dovuto ai vincoli relazionali.
Terminata questa prima fase di progettazione del database, ci si è concentrati sul progetto Java che, attraverso i costrutti messi a disposizione da Hibernate e Spring fosse in grado di gestire la base di dati. Si è dovuto quindi comprendere come configurare correttamente i due framework per utilizzarli nel modo migliore per gli obiettivi che ci si era prefissati. All'inizio della fase di progettazione si è scelto di considerare un "caso base" dal quale partire per sviluppare poi tutto il resto del progetto: sono state quindi scelte le entità Category e Document e la relazione many-to-many tra le due entità.
Questo caso è di particolare interesse, in quanto una relazione di questo tipo viene gestita in modo diverso da Hibernate: la tabella "intermedia" della relazione (Contains) non viene definita come Bean all'interno del progetto Java, e non viene

quindi gestita in modo analogo alle altre tabelle, ma viene aggiornata in modo automatico da Hibernate. Era quindi fondamentale capire come riuscire ad accedere ai documenti a partire da una categoria, e viceversa.
L'idea di base era quella di rendere il più veloce possibile l'interrogazione del database configurando, nel file xml di mappatura delle tabelle, fetch="select" e lazy="true"; in questo modo ad ogni interrogazione si sarebbero scaricati dal database solo i record della tabella e non i dati relativi agli oggetti collegati; le variabili di tipo Set<oggetto> non sarebbero quindi state valorizzate nel momento in cui si accedeva ad una categoria; per accedere ai dati collegati si doveva eseguire una nuova operazione di selezione nel database. Questa configurazione può portare ad avere problemi di persistenza. I problemi di persistenza sono dovuti al tentativo di accedere ad un dato non disponibile fisicamente. Ad esempio, nel momento in cui si prova ad accedere in modo diretto alla variabile Set<Document> a partire da un oggetto Category, viene lanciata una eccezione del tipo: javax.persistence.PersistenceException. Questa eccezione è dovuta al fatto che l'oggetto Set<Document> non viene valorizzato nel momento in cui si interroga il database per la richiesta di una categoria. Per risolvere questi i problemi di persistenza si è dovuto gestire l'accesso ai dati collegati in modo accurato eseguendo all'interno della stessa sessione di collegamento con il database, prima la richiesta dell'oggetto "principale" (nell'esempio, la categoria), poi la richiesta degli oggetti collegati. In questo modo la variabile Set<Document> viene correttamente valorizzata (resa persistente) e quindi è utilizzabile all'interno del progetto Java.
Terminata la fase di studio dei framework, si è passati ad estendere i risultati ottenuti nel caso analizzato, creando le classi del progetto necessarie per gestire tutta la base di dati.
Il lavoro si è poi concluso con la fase di test, nella quale si è verificato il corretto funzionamento di tutti i metodi costruiti per la gestione di tutte le entità della base di dati.

Lo strumento creato permette non solo di confrontare i risultati ottenuti negli esperimenti di classificazione con le diverse configurazioni, ma dà la possibilità di analizzare tutti gli aspetti di un esperimento partendo da punti di osservazione diversi. Ad esempio si può analizzare a quali collezioni, categorie e insiemi di training set lo stesso documento viene associato in esperimenti diversi e, a partire da questo risultato, andare a stabilire i parametri degli esperimenti che hanno

ottenuto delle prestazioni migliori.
Ad oggi questo confronto non è immediatamente attuabile in quanto gli articoli scientifici che descrivono gli esperimenti forniscono solo indici prestazionali che spesso rappresentano una media pesata delle performance ottenute.
La scelta dei framework Hibernate e Spring, e la struttura data al progetto, permetteranno di utilizzare i metodi di persistenza messi a disposizione per interrogare la base di dati attraverso una interfaccia grafica.
Si deve inoltre considerare che lo strumento è stato pensato per essere utilizzato per la ricerca nel campo della classificazione automatica di testi, ma questo progetto è stato realizzato in maniera tale da poter essere esteso per essere utilizzato anche da altre aree di ricerca nel campo dell'information retrieval.

Bibliografia

[1] **Atzeni, Ceri, Paraboschi, Torlone;** *Basi Di Dati (Modelli e Linguaggi di Interrogazione)*; McGraw Hill, 2003.

[2] **Bauer, King;** *Java peristence with Hibernate*; Manning, 2007.

[3] **Bauer, King;** *Hibernate in Action*; Manning, 2005.

[4] **Fisher, Murphy;** *Spring persistence with Hibernate*; Apress, 2010.

[5] **Sebastiani;** *Machine Learning in Automated Text Categorization*; ACM Computer Surveys, 2002.

[6] **Hibernate:** *http://www.hibernate.org/.*

[7] **Spring:** *http://www.springsource.org/.*

[8] **Postgres:** *http://www.postgresql.org.*

[9] **Eclipse:** *http://www.eclipse.org/.*

[10] **Wikipedia:** *http://www.wikipedia.org.*

[11] **Lucene:** *http://lucene.apache.org.*

[12] **Supporto programmazione:** *http://www.html.it.*

[13] **Archivio on-line di collezioni:** *http://archive.ics.uci.edu/ml.*

Elenco delle figure