INDICE

Capitolo 1. Considerazioni introduttive **pag. 1**

Capitolo 2. Aspetti della traduzione multilingua di testo **pag. 4**

2.1. Il ciclo di traduzione di un testo pag. 4

2.2. La fase di pre-traduzione pag. 6

2.3. Il livello di ricerca per approximate match pag. 9

2.3.1. Analisi della metodologia applicata attualmente pag. 9

Capitolo 3. Introduzione al problema del reperimento di informazioni su testi e stato dell'arte **pag. 13**

3.1. Data Retrieval (DR) e Information Retrieval (IR) pag. 13

3.2. Information Retrieval Systems pag. 15

3.2.1. I metodi di indicizzazione pag. 18

3.2.1.1. Inverted index pag. 18

3.2.1.2. Signature index pag. 21

3.2.1.3. Concept index pag. 22

3.2.2. Prestazioni di un I.R.S. pag. 23

3.2.2.1. Parametri per valutare l'efficacia: precision e recall pag. 24

3.2.2.1.1. La curva richiamo-precisione pag. 25

3.2.2.2. Valutazione dell'efficienza pag. 28

3.2.3. Un metodo per "eseguire" un'interrogazione pag. 29

Capitolo 4. L'ambiente Oracle 8i – *inter*Media **pag. 33**

4.1. La creazione dell'indice pag. 34

4.1.1. La classe Datastore pag. 36

4.1.2. La classe Filter pag. 38

4.1.3. La classe Section_group pag. 39

4.1.4. La classe Lexer pag. 39
4.1.5. La classe Stoplist pag. 40
4.1.6. La classe Wordlist pag. 40
4.2. Metodi e tipi di ricerca pag. 41
4.2.1. Interrogazioni "Direct Match" pag. 41
4.2.1.1. Ricerche per frasi pag. 42
4.2.2. Interrogazioni "Indirect Match" pag. 43
4.2.2.1. Wildcard Match pag. 44
4.2.2.2. Mistyping pag. 45
4.2.2.3. Derivazione o radicalizzazione di termini pag. 45
4.2.2.4. Gestione delle relazioni tra termini pag. 46
4.2.3. Interrogazioni composte pag. 46
4.2.3.1. Operatori Booleani pag. 47
4.2.3.2. Interrogazioni in linguaggio naturale pag. 47
4.2.3.3. Interrogazioni strutturate pag. 47
4.2.3.4. Ricerca all'interno di sezioni nei documenti pag.48
4.3. La gestione delle istruzioni INSERT, UPDATE e DELETE pag. 48
4.4. L'ottimizzazione pag. 50

Capitolo 5. Analisi di fattibilità di possibili soluzioni per la ricerca di approximate match testuali pag. 52

5.1. La struttura della banca dati testuale pag. 52
5.2. Utilizzo delle funzionalità di Oracle 8i *inter*Media pag. 53
5.2.1. Considerazioni sull'applicazione della ricerca "about" per risolvere il problema pag. 54
5.3. Analisi di un procedimento ad hoc per la ricerca in modalità approximate match di frasi pag. 55

Capitolo 6. Progettazione del procedimento scelto per la ricerca di approximate match testuali **pag. 61**

6.1. Il programma di normalizzazione pag. 61

6.1.1. Che cosa è un algoritmo di stemming pag. 61

6.1.2. Un modello di stemmer presente in letteratura pag. 63

6.1.3. Analisi ad oggetti del processo di normalizzazione pag. 64

6.1.3.1. Modello statico pag. 65

6.1.3.1.1. La classe Regola pag. 65

6.1.3.1.2. La classe Stemming pag. 67

6.1.3.1.3. La classe RicercaInDbOracle pag. 69

6.1.3.1.4. La classe Filtri pag. 71

6.1.3.1.5. Strutture delle tabelle utilizzate per la normalizzazione pag. 72

6.1.3.2. Modello dinamico pag. 73

6.1.3.3. Modello funzionale pag. 74

6.2. Il procedimento di indicizzazione pag. 78

6.3. Progettazione della modalità per classificare i risultati della ricerca pag. 80

Capitolo 7. Test e conclusioni **pag. 85**

7.1. Confronto tra I risultati ottenuti con il vecchio ed il nuovo algoritmo pag. 85

7.2. Valutazione d'efficienza e punti critici pag. 92

7.3. Possibilità di sviluppi futuri pag. 96

Appendice A. Listati Java dell'implementazione del progetto di ricerca di similarità tra frasi **pag. 98**

Appendice B. Introduzione al linguaggio Java e al suo utilizzo connesso ad una base di dati Oracle **pag. 115**

B.1. Caratteristiche del linguaggio Java pag. 115

B.2. Presentazione del linguaggio Java pag. 117
B.2.1. Le classi pag. 117
B.2.2. Variabili istanza pag. 117
B.2.3. Variabili di classe pag. 118
B.2.4. Metodi istanza pag. 118
B.2.5. Metodi di classe pag. 118
B.2.6. Metodi costruttori pag. 119
B.2.7. Metodi conclusivi pag. 120
B.2.8. Variabili locali pag. 120
B.2.9. Costanti pag. 120
B.2.10. Le interfacce pag. 121
B.2.11. I package pag. 121
B.2.12. Alcune parole chiave pag. 122
B.3. La classe "Vector" pag. 123
B.4. Modificatori di accesso pag. 125
B.4.1. Public pag. 125
B.4.2. Protected pag. 125
B.4.3. Private pag. 126
B.4.4. Synchronized pag. 126
B.4.5. Native pag. 126
B.5. Garbage Collector pag. 127
B.5.1. Garbage detection pag. 127
B.5.2. Tecniche di deframmentazione pag. 129
B.6. Gestione delle eccezioni pag. 129
B.6.1. Blocchi catch multipli pag. 130
B.6.2. La clausola finally pag. 131
B.7. Integrazione tra Java e ORACLE 8i pag. 132
B.7.1. JDBC pag. 132
B.7.2. SQLJ pag. 133

B.7.2.1. Le primitive SQLJ pag. 133
B.7.2.2. Gli oggetti "iterator" pag. 134

Riferimenti bibliografici pag. 137

Capitolo 1

CONSIDERAZIONI INTRODUTTIVE

La traduzione multilingua di testi come, per esempio, manuali tecnici, software o libri è l'attività primaria della Logos S.p.A.
Il lavoro svolto per elaborare la tesi ha avuto lo scopo di apportare un miglioramento al sistema di traduzione adottato presso questa importante azienda. In particolare, questo sistema comprende una fase completamente automatizzata, la pre-traduzione, che ha la finalità di eseguire una prima analisi di un documento da tradurre e di produrre, per quanto possibile, una traduzione computerizzata. Questa operazione esegue un'analisi delle frasi da tradurre basandosi su una banca dati testuale contenente una memoria storica di tutte le traduzioni precedenti. La frase da tradurre viene quindi ricercata all'interno di questo archivio per verificare se, eventualmente, sia stata già tradotta in passato. Si può intuire quanto possa essere interessante reperire anche quelle frasi che, pur se non uguali, sono simili in significato a quella da tradurre.
Il progetto analizzato ha proprio l'obiettivo di implementare un algoritmo di ricerca che porti al reperimento, all'interno di una banca dati testuale, delle frasi che hanno un più forte legame di similarità con una frase ricevuta in ingresso.

Nel capitolo 2 si descrivono il ciclo di operazioni messo a punto in Logos per eseguire le traduzioni multilingue di testi e, più dettagliatamente, la fase di pre-traduzione. Si esegue, inoltre, una prima breve analisi degli aspetti da migliorare nella pre-traduzione; partendo da queste considerazioni iniziali verrà indirizzato il lavoro svolto.

Nel capitolo 3 si descrivono gli aspetti principali della disciplina dell'Information Retrieval. Ad essi, infatti, si possono ricondurre molti aspetti della ricerca di similarità tra frasi (come l'utilizzo di particolari strutture di indici e sistemi di ricerca).

Nel capitolo 4 si analizzano le potenzialità di un prodotto commerciale, Oracle 8i *inter*Media, che abbiamo utilizzato per costruire un indice ed eseguire ricerche su un insieme di dati non strutturato come è una banca dati testuale.

Nel capitolo 5 si esegue un'analisi di alcune possibili soluzioni per implementare un processo valido per la ricerca di similarità tra frasi in lingua inglese. Si valuta la fattibilità di ciascuna di esse e si sceglie la strategia ritenuta più idonea per ottenere risultati soddisfacenti.

Nel capitolo 6 si eseguono l'analisi dettagliata e la progettazione della soluzione scelta. Per rendere comprensibile il prototipo realizzato che implementa l'algoritmo progettato si è pensato di darne una rappresentazione semi-formale secondo le specifiche e la sintassi del modello OMT (Object Modeling Technique).

Nel capitolo 7 si illustra dapprima il confronto in efficacia tra il nuovo procedimento di ricerca e il vecchio. Successivamente si valutano le prestazioni dal punto di vista dell'efficienza del nuovo progetto e si cerca di fornire una valutazione obiettiva circa l'opportunità di utilizzo reale dell'algoritmo all'interno del contesto aziendale. Nelle appendici A e B si riportano, rispettivamente, i listati Java contenenti i metodi descritti nel modello costruito nel capitolo 6 e un'introduzione al linguaggio di

programmazione Java e agli strumenti che permettono di utilizzare questo linguaggio in connessione ad una base di dati Oracle.

Capitolo 2

ASPETTI DELLA TRADUZIONE MULTILINGUA DI TESTO

Nel presente capitolo si descriverà l'attività di traduzione di testo presso la Logos S.p.A.. Entreremo poi più in dettaglio nella fase di pre-traduzione: valuteremo e descriveremo le problematiche che, proprio all'interno di questa fase, sono state affrontate durante il lavoro svolto per elaborare la presente tesi.

2.1 Il ciclo di traduzione di un testo

L'attività di traduzione (ved. Fig. 2.1) inizia con un primo contatto tra la figura interna del Production Manager[1] e l'azienda cliente che richiede la traduzione. In generale la maggioranza delle traduzioni effettuate riguarda manuali tecnici e localizzazioni di software.

Una volta che il cliente ha accettato le condizioni di contratto ed ha fornito la documentazione da tradurre, si provvede al filtraggio del documento stesso. Questa fase di filtraggio prende in esame il formato del materiale fornito dal cliente (RTF, Frame Maker, Page Maker, XML, HTML, Quark_Xpress) e provvede alla sua trasposizione in testo puro con codici di posizione[2].

---

[1] Il Production Manager è la persona incaricata di gestire interamente un processo di traduzione.

[2] All'interno di questi codici vengono sintetizzate le formattazioni interne caratteristiche di ogni formato. La loro presenza insieme al testo puro è necessaria per riportare il documento, una volta tradotto, nello stesso formato dell'originale. Questi codici sono racchiusi tra angolari (< e >).

Una volta che il documento è stato filtrato viene scomposto in singole frasi attraverso un procedimento di segmentazione.

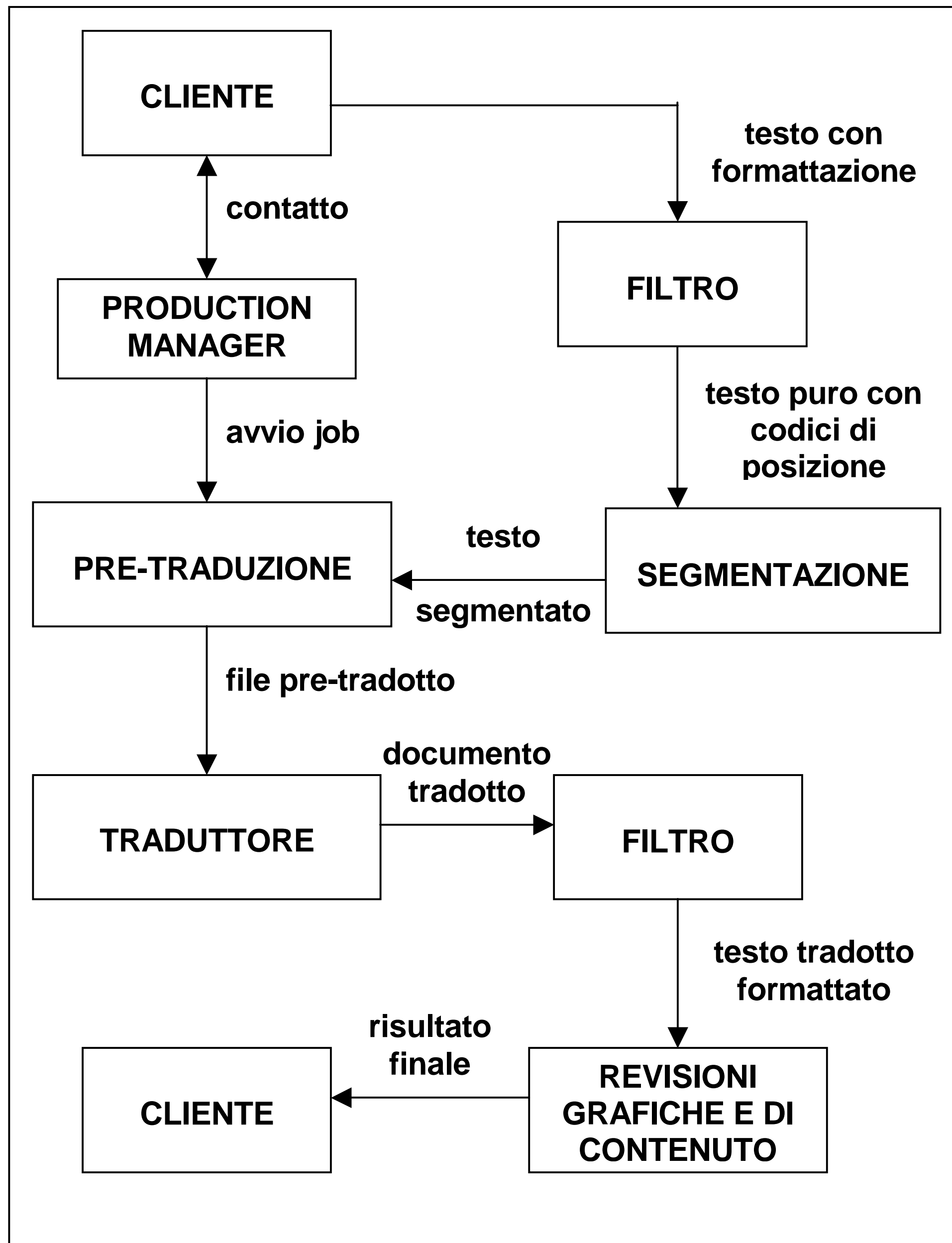


Fig. 2.1. Le fasi del ciclo della traduzione

A questo punto il Production Manager richiede l'apertura di un Job di pre-traduzione. Il Production Manager dovrà specificare diversi parametri. Tra di essi, quelli che interessano la traduzione sono:

- **Codice Cliente**: è l'identificativo del cliente che richiede la traduzione;
- **Codice Settore**: indica l'argomento trattato dal documento tradotto;
- **Lingua Source**: indica la lingua di partenza della traduzione;
- **Lingua Target**: indica la lingua in cui il documento deve essere tradotto.

Questo è l'inizio della fase completamente automatizzata di pre-traduzione. Essa verrà descritta in modo più dettagliato nel prossimo paragrafo.

L'esecuzione, in base ai parametri forniti dal Production Manager, della pre-traduzione del testo proveniente dalla fase di filtraggio, produce come risultato un file che contiene una prima versione tradotta del documento. Questo file contiene anche alcune indicazioni per la persona che si dovrà occupare di terminare l'operazione di traduzione. Queste indicazioni riguardano la modalità con cui è stata effettuata la traduzione automatica.

Il traduttore prenderà quindi in esame il risultato della pre-traduzione e, una volta sistemate le eventuali imperfezioni, restituirà l'elaborato finale.

Attraverso un'altra operazione di filtraggio (speculare alla precedente) viene ricostruito il formato del documento originariamente fornito dal cliente (RTF, Frame Maker, ecc.).

A questo punto dopo una verifica grafica sul risultato del filtraggio e una revisione sul risultato della traduzione (con eventuale coivolgimento in feedback del traduttore) viene consegnato al cliente il documento tradotto.

2.2 La fase di pre-traduzione

La fase di pre-traduzione, nell'ambito di tutto il ciclo presentato nel paragrafo precedente, è quella che riveste maggior importanza.

In questa fase la struttura di appoggio è una base di dati contenente la tabella TERMS. Questa tabella contiene la memoria storica di tutte le frasi che sono state tradotte in passato per tutti i clienti. Le tuple di questa tabella sono strutturate in questo modo:

- Codice identificativo della frase;
- Codice identificativo del cliente per il quale la frase è stata tradotta;
- Codice del settore a cui si riferisce la frase;
- frase già tradotta in precedenza (una colonna per ogni lingua conosciuta).

Considerando come parametri d'ingresso i dati forniti dal Production Manager ed il testo segmentato in frasi si avvia una serie di interrogazioni sulla tabella TERMS. Queste ricerche possono suddividersi in due livelli: **exact match** e **approximate match**.

Il livello **exact match** si suddivide a sua volta in tre sottolivelli.

Al primo di questi sottolivelli si effettua una ricerca esatta della frase da tradurre nelle tuple della tabella aventi **codice del cliente e codice di settore** uguali a quelli contenuti nella richiesta effettuata dal Production Manager (ved. Fig. 2.2). Se la frase cercata è presente nelle tuple considerate viene reperita la versione corrispondente alla lingua target impostata. La frase recuperata a questo livello viene classificata come exact match di colore verde. È questo il colore, infatti, con cui verrà visualizzata nel risultato della pre-traduzione. Un exact match verde si può già considerare praticamente una traduzione corretta.

Qualora non andasse a buon fine la ricerca effettuata nel primo sottolivello si procede con una ricerca esatta all'interno dell'insieme delle frasi tradotte che fanno capo al **codice del cliente** considerato (senza vincolare il codice di settore). Un risultato positivo che esce da questa interrogazione viene classificato come exact match di colore blu. Un exact match blu comporterà, da parte del traduttore, un'attenzione maggiore rispetto all'exact match verde.

Un ultimo sottolivello di ricerca per exact match contempla un'interrogazione all'interno dell'insieme delle frasi che fanno capo al **codice del settore** considerato (non si vincola, in questo caso, il codice del cliente). Il risultato di questa ricerca viene classificato come exact match di colore giallo. Le frasi pre-tradotte di colore giallo dovranno essere quindi valutate con attenzione ancora maggiore da parte del traduttore.

La ricerca a livello **approximate match** viene eseguita qualora non vengano

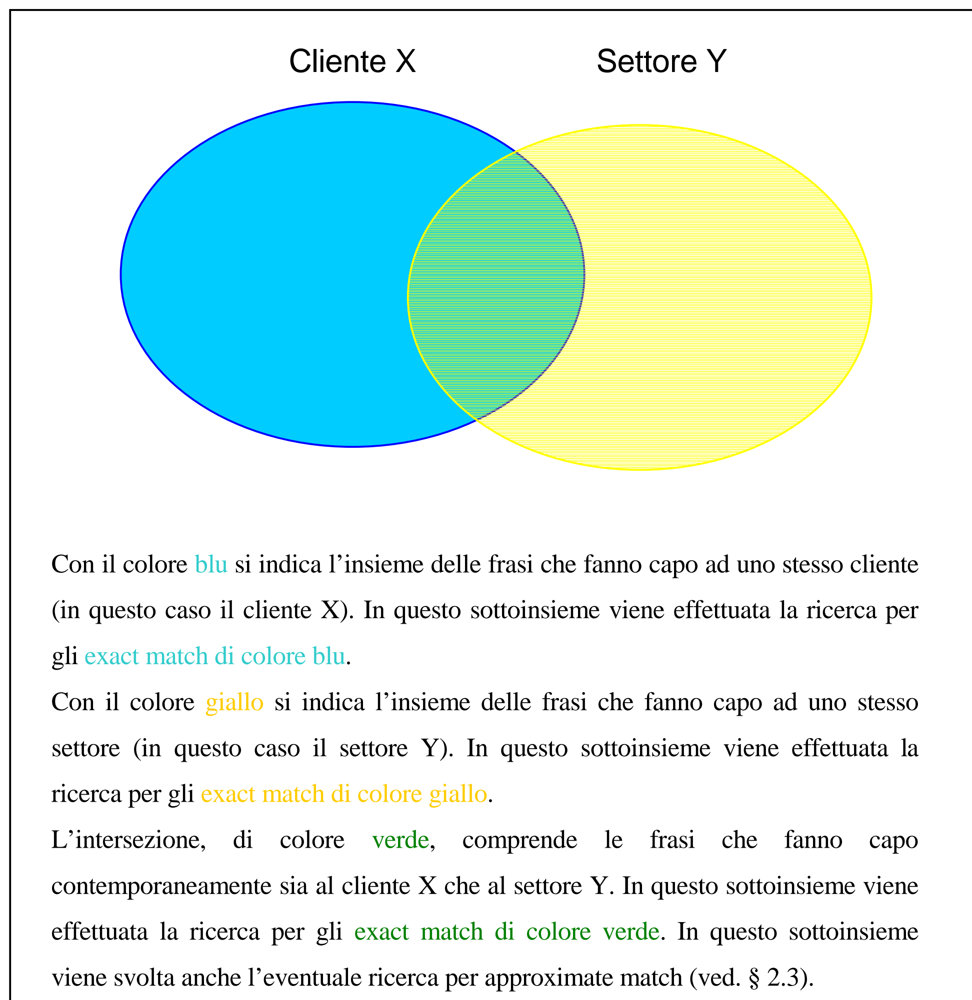


Fig. 2.2. Insiemi e livelli di exact match

restituiti risultati da nessuno dei tre livelli di ricerca exact match.

Poiché la ricerca di approximate match è stato l'argomento analizzato per lo svolgimento della tesi sarà analizzato in dettaglio nel prossimo paragrafo.

2.3 Il livello di ricerca per approximate match

La ricerca approximate match cerca di reperire, all'interno della tabella TERMS, quelle frasi che possono avere un significato simile a quella da tradurre. In particolare la ricerca di queste frasi viene operata nell'insieme in cui viene eseguita la ricerca per exact match di colore verde (ved. Fig. 2.2).

2.3.1 Analisi della metodologia applicata attualmente

Attualmente la ricerca di approximate match viene eseguita nel modo empirico che descriviamo ora cercando di evidenziare i punti che possono apparire

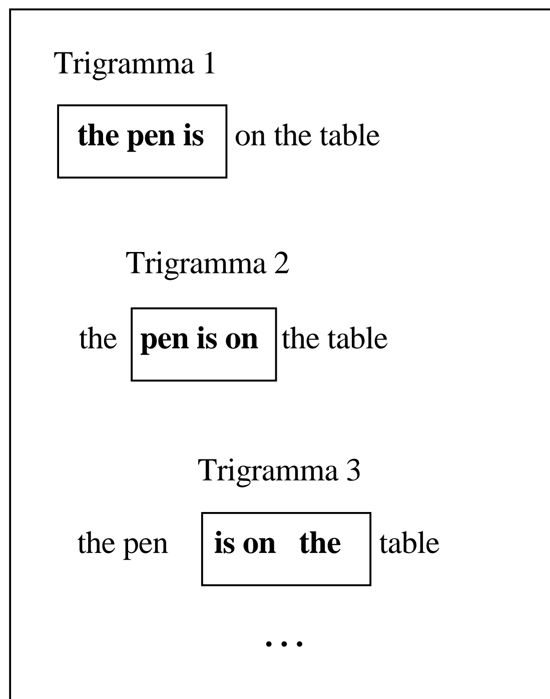


Fig. 2.3. Generazione dei trigrammi

migliorabili.

Occorre dire, prima di tutto, che la ricerca approximate fa riferimento ad una tabella, WORDS, presente sulla base di dati per la pre-traduzione. Le tuple di questa tabella contengono i seguenti campi:

- trigramma (ved. Fig. 2.3) presente nella frase della tabella TERMS (una riga per ogni lingua);

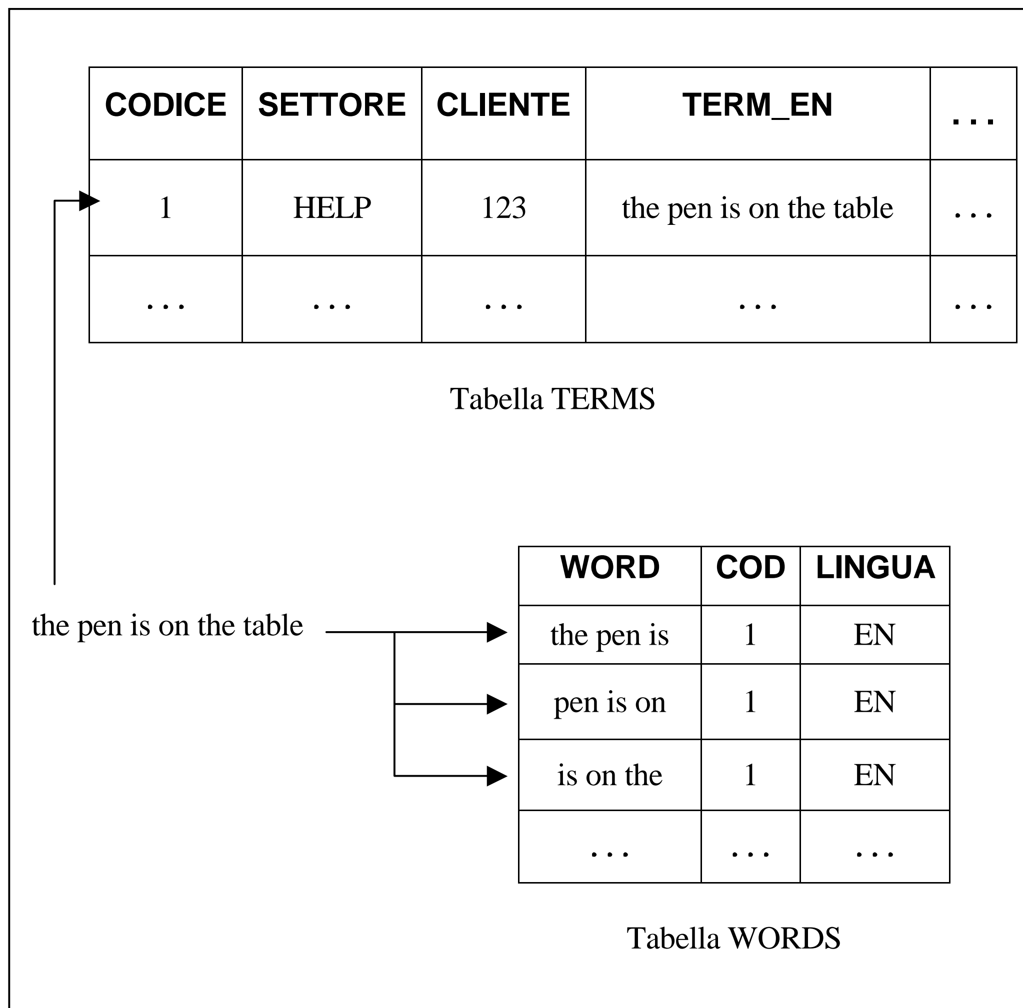


Fig. 2.4. Inserimento dei trigrammi nella tabella WORDS

- codice identificativo della frase di TERMS che contiene il trigramma;
- lingua corrispondente al trigramma.

Ogni volta che viene inserita una tupla nella tabella TERMS, vengono generati i trigrammi che compongono ogni frase in ogni lingua presente nella tupla stessa (ved. Fig. 2.3). Ogni singolo trigramma viene inserito in ogni singola tupla della tabella WORDS (ved. Fig. 2.4).

Vediamo ora come viene eseguita la ricerca per approximate match.

La frase da tradurre viene scomposta in trigrammi come in fig. 2.3. Per ogni trigramma generato viene svolta un'operazione di join tra la tabella WORDS e la tabella TERMS basata sul campo codice. In linguaggio SQL l'istruzione per svolgere questa operazione sarà:

```
select FRASE_IN_LINGUA_TARGET
from WORDS a, TERMS b
where a.WORD = 'trigramma della frase da tradurre'
and a.COD = b.CODICE
and b.CLIENTE = 'cliente'
and b.SETTORE = 'settore';
```

Per ogni frase reperita attraverso questa operazione viene calcolato un valore[3] proporzionale alla sua somiglianza con la frase oggetto di traduzione.

Le frasi che risultano dal join vengono quindi ordinate in base a questo valore. Le frasi a cui corrispondono i 10 valori più elevati verranno inserite nel risultato della pre-traduzione come suggerimenti per il traduttore.

L'analisi svolta sul procedimento di ricerca per approximate match ha messo in risalto alcuni aspetti che possono essere modificati e migliorati.

In primo luogo si è notata la complessità del procedimento con cui si esegue il reperimento delle frasi (presenti nella tabella TERMS) che contengono i

[3] Questo valore viene determinato semplicemente calcolando il numero di parole uguali tra la frase reperita e la frase da tradurre.

trigrammi generati partendo dalla frase da tradurre. La ricerca di ogni trigramma può essere vista come un'operazione di reperimento di testo all'interno di una banca dati testuale. Partendo da questo presupposto pensiamo che la valutazione dell'utilizzo di un sistema che permetta di eseguire facilmente operazioni di questo tipo possa essere utile (indipendentemente dal fatto che si mantenga o meno la modalità di ricerca per trigrammi). Nel capitolo 3 vedremo di descrivere gli aspetti principali della disciplina dell'*Information Retrieval* nell'ambito della quale vengono studiati sistemi per il reperimento di testo. Nel capitolo 4 verranno valutate le funzionalità di un pacchetto commerciale (Oracle 8i *inter*Media Text) che si presta ad essere utilizzato per svolgere ricerche di testo.

Un secondo aspetto da migliorare riguarda la modalità con la quale viene gestita la valutazione della somiglianza in significato tra le frasi. Questo aspetto appare, infatti, trattato in maniera un po' troppo semplicistica. Un'analisi di questo problema e una sua possibile soluzione viene esposta nei capitoli 5 e 6.

C a p i t o l o 3

INTRODUZIONE AL PROBLEMA DEL REPERIMENTO DI INFORMAZIONI SU TESTI E STATO DELL'ARTE

Dopo una prima analisi generale dei concetti riguardanti il reperimento di informazioni su testi, verranno descritte le strutture che supportano questa disciplina.

3.1 Data Retrieval (DR) e Information Retrieval (IR)

Con *Information Retrieval* (Reperimento di Informazioni) vogliamo indicare quella disciplina che si occupa di studiare, progettare e realizzare sistemi informatici, denominati *Information Retrieval Systems* (Sistemi per il Reperimento delle Informazioni), che consentono la memorizzazione, la manutenzione e il reperimento di dati all'interno di insiemi non strutturati [1]. Un esempio di applicazione delle metodologie dell'Information Retrieval può essere la ricerca di termini all'interno di una enciclopedia digitale.
Con il termine *Data Retrieval* vogliamo, invece, definire la disciplina che si occupa di gestire i dati attraverso i *Data Base Management Systems* (DBMS) [1]. I DBMS sono sistemi che gestiscono informazioni di tipo strutturato cioè dati per i quali si riescono a definire delle relazioni, delle entità, degli attributi fra la struttura generale e il particolare dato memorizzato. Un classico campo di applicazione dei DBMS è quello della *office automation* il cui obiettivo

principale è quello di manipolare dati in tabelle strutturate o creare reports aziendali.

Nella tabella 3.1 sono poste a confronto proprietà caratterizzanti di Data Retrieval e Information Retrieval per meglio capire la differenza tra le due metodologie.

	Data Retrieval	**Information Retrieval**
Matching	Exact match	Partial match, best match
Inference	Deduction	Induction
Model	Deterministic	Probabilistic
Query Language	Artificial	Natural
Query Specification	Complete	Incomplete
Items wanted	Matching	Relevant

Tabella 3.1 Data Retrieval e Information Retrieval

Nel Data Retrieval normalmente si ricerca una corrispondenza esatta cioè si vuole valutare se un elemento è presente o meno in un archivio. Nell'Information Retrieval l'interesse principale è di trovare quegli elementi che, più o meno parzialmente, corrispondono alla richiesta. All'interno di questi saranno poi selezionati quelli che più sono pertinenti all'oggetto di ricerca.

L'inferenza usata nel Data Retrieval è di tipo semplicemente deduttivo: "aRb e bRc allora aRc". Nell'Information Retrieval le relazioni sono specificate con un grado di certezza o incertezza. Questo porta automaticamente a descrivere il Data Retrieval come approccio deterministico e l'Information Retrieval come approccio probabilistico. Il linguaggio di interrogazione per il DR sarà di

tipo strutturato (come SQL) mentre per l'IR ci si avvicina al linguaggio naturale. Nel DR l'interrogazione generalmente specifica in modo completo che cosa si vuole, nell'IR questo è inevitabilmente impossibile. Quest'ultima differenza nasce dal fatto che nell'IR si ricercano documenti che siano pertinenti (*relevant*) in contenuto con la richiesta, piuttosto che eseguire una ricerca esatta.

Per concludere si può dire che, per la sua natura, l'Information Retrieval si presta ad essere trattato attraverso approcci che spaziano dal calcolo delle probabilità alla fuzzy logic [2].

3.2 Information Retrieval Systems

Lo studio di sistemi e di metodologie per il recupero dell'informazione non è cosa nuova. Già negli anni '50 – '60 si analizzavano degli strumenti che tramite l'utilizzo del calcolatore permettessero di effettuare ricerche su grosse banche di dati testuali. Tuttavia è in questi ultimi anni che gli "***Information Retrieval Systems***" (I.R.S.) hanno subito grosse evoluzioni e sono oggetto di molti studi da parte dei ricercatori [5]. Tutto questo è sicuramente legato al crescente utilizzo delle reti di computer, in particolare di Internet che, se da un lato ha accresciuto enormemente la quantità d'informazione digitale disponibile per gli utenti, dall'altro ha evidenziato l'importanza di disporre di sistemi efficaci per il recupero dell'informazione.

Un altro fattore che ha riportato in auge metodologie come quella dell'Information Retrieval è legato allo sviluppo delle nuove memorie di massa che possono contenere grandi quantità di byte e che quindi, così come per Internet, offrono all'utente molte informazioni che comunque vanno ricercate con qualche metodo che sia efficace ed efficiente. Questa metodologia si differenzia, come detto, dai DBMS in quanto non prevede un'organizzazione dei dati ed in generale dell'informazione che deve essere

memorizzata e recuperata. Le tecniche di Information Retrieval possono essere applicate a qualsiasi file di testo, comunque esso sia stato organizzato. Ciò conferisce ai sistemi che utilizzano tale metodo di recupero una enorme flessibilità. L'obiettivo che l'Information Retrieval si propone di raggiungere è quello di ritrovare in una qualsiasi banca dati testuale, come può essere un'intera enciclopedia digitale, la parte di testo che contiene la parola o la frase inserita come termine di ricerca. Appare evidente come tale metodologia sia molto più potente dei DBMS che lavorano solo su dati precedentemente strutturati, e quindi non applicabili ad una vasta classe di dati come possono essere le banche d'informazione testuali che contengono articoli, news, pubblicazioni, enciclopedie digitali ed in generale collezioni di documenti.

Naturalmente la disciplina dell'Information Retrieval è molto complessa e si avvale di una serie di processi e metodi che permettono di ottenere gli obiettivi prefissati. In generale un sistema che permette all'utente di recuperare del testo da una collezione di documenti si compone di:

- **Un indice che permette di effettuare il recupero dell'informazione;**
- **Un sistema che permette di effettuare la ricerca;**

In particolare, la creazione dell'indice (**indicizzazione**) rappresenta un punto cruciale verso il quale si stanno concentrando le ricerche.

L'indicizzazione è quel processo che permette di rappresentare in maniera sintetica il documento o la collezione di documenti che si intendono memorizzare e recuperare. Tramite l'indicizzazione è possibile definire simbolicamente i collegamenti fra i termini "rappresentativi" del documento e la posizione (**locazione**) dei suddetti termini all'interno del testo o dell'insieme di documenti, cioè dove quel particolare termine è memorizzato. Noto il termine (o i termini) da recuperare, l'indice consente di ottenere la

locazione corrispondente nel testo se il termine è presente nei documenti. Una volta definito l'indice, il processo di recupero dell'informazione consta dei seguenti passi (ved. Fig. 3.1):

- Interrogazione dell'utente (inserimento del testo da recuperare);
- Apertura dell'indice e ricerca del testo inserito tramite l'interrogazione;
- Risposta all'interrogazione e recupero testo;
- Visualizzazione.

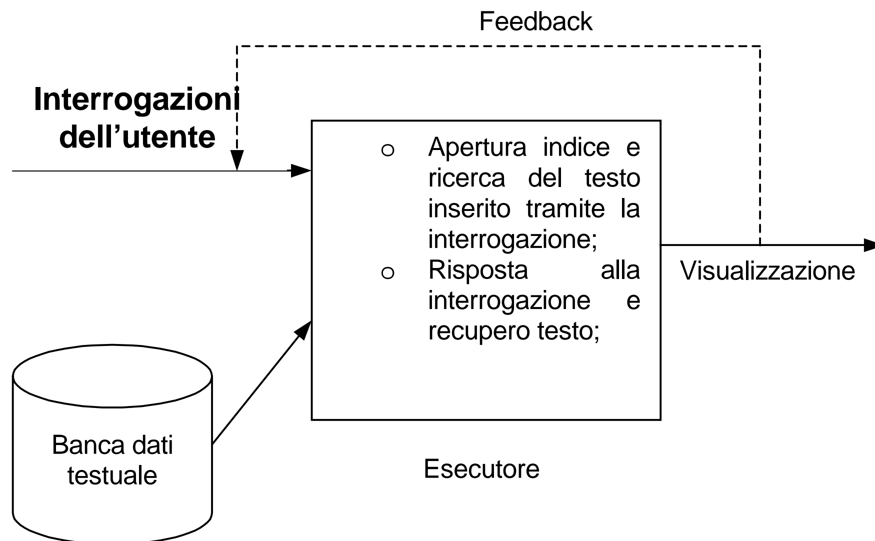


Fig. 3.1. Schema del processo di recupero di informazioni

3.2.1 I Metodi d'Indicizzazione

Nel corso degli anni sono stati individuati diversi metodi d'indicizzazione, ognuno dei quali applicabile a diverse tipologie d'informazioni [1,5]. Questi metodi sono:

- *Inverted Index;*
- *Signature Index;*
- *Concept Index;*

3.2.1.1 Inverted Index

L'inverted index è sicuramente il metodo di indicizzazione più utilizzato ed è applicabile a qualsiasi file di testo. Questo metodo è basato sull'analisi lessicale del testo al fine di ottenere una sequenza di singoli termini o token da indicizzare. Il testo viene suddiviso in singole parole che vengono analizzate e filtrate in modo tale da eliminare i **termini più comuni** presenti nel documento. Questa operazione può essere effettuata adottando diverse tecniche. Una prima modalità consiste nell'analizzare la **frequenza** di ogni termine nel documento. A partire da una tabella del tipo:

Parola	**Frequenza**
Termine 1	Frequenza 1
. . .	. . .
Termine i	Frequenza i
. . .	. . .
Termine n	Frequenza n

che associa ogni parola al rispettivo numero di occorrenze nel testo, viene individuata una **soglia di frequenza** al di sopra della quale si devono eliminare le parole. Solitamente la soglia è del 25% - 30%: una parola viene eliminata dall'indice se è ripetuta più del 25% di volte sul totale delle parole presenti nel documento. In questa maniera si crea una lista di termini (estratti dal documento principale) ripulita da tutte quelle parole comuni come possono essere gli articoli, gli avverbi, le congiunzioni, ecc.

Un altro metodo per effettuare la "scrematura" dalle *noise words*[4] consiste nell'utilizzare quella che viene definita *Stop List*: si tratta di una lista che contiene tutte quelle parole che non si vogliono indicizzare perché o troppo comuni o di scarso significato ai fini del recupero.

Dopo aver compiuto queste operazioni si otterrà una lista di parole ripulita dalle parole comuni e dalle parole comunque volutamente eliminate tramite la stop list; nell'elenco finale saranno presenti solo i termini rappresentativi del documento ed utili ai fini della ricerca.

A questo punto, in teoria, il metodo inverted index passa alla eventuale determinazione delle *stems*, cioè all'individuazione di tutti quei termini che hanno la stessa radice e che, tramite questa, possono essere recuperati (ad esempio veloce - velocemente); in questa maniera si riesce a diminuire ulteriormente il numero di parole che l'indice dovrà trattare. Bisogna comunque sottolineare come quest'ultima operazione (*stemming*) in pratica non venga quasi mai effettuata durante la fase di indicizzazione in quanto piuttosto complessa.

Come si avrà modo di illustrare nel capitolo 5, l'analisi di questo problema, finalizzata alla stesura e all'implementazione di un algoritmo che permetta di

---

[4] Letteralmente "parole rumorose". Sono termini che non aggiungono nulla al significato di un testo.

eseguire lo stemming dei termini, è una fase importante dell'attività svolta presso la Logos.

Dopo aver provveduto all'ordinamento alfabetico dei termini presenti nella struttura, l'ultimo passo che permette di creare l'indice è quello che associa alle parole, selezionate tramite le precedenti fasi, la **locazione**. In questa fase si stabilisce la **relazione (parola, locazione),** che fa corrispondere ad ogni parola la lista di documenti che la contengono. L'inverted index assumerà quindi una configurazione definitiva del tipo:

Parola	**Frequenza**	**Locazione**
Animale	Frequenza 1	Documento1.txt, Prova2.txt, Documento5.txt
. . .	. . .	. . .
Cane	Frequenza i	Prova1.txt, Documento1.txt
. . .	. . .	. . .
Uomo	Frequenza n	Prova2.txt

Una volta che è stato creato l'indice, il sistema di recupero (I.R.S.) si occuperà di:

- Ricevere in input l'interrogazione dell'utente;
- Aprire l'indice;
- Stabilire la corrispondenza Termine Interrogazione - Termine Indice;
- Determinare la locazione dei termini che soddisfano l'interrogazione.

Naturalmente il sistema deve essere completato da una serie di interfacce che consentono all'utente di porre l'interrogazione al sistema e di visualizzarne i risultati.

Come è stato già detto in precedenza l'inverted index è il metodo d'indicizzazione maggiormente utilizzato fra i sistemi di recupero a pieno testo. Ciò è dovuto non solo alle soddisfacenti prestazioni che sono state riscontrate negli I.R.S. che utilizzano questo tipo di struttura, ma soprattutto alla possibilità di poter automatizzare l'intero processo d'indicizzazione. L'automazione è il maggior vantaggio che l'inverted index offre rispetto agli altri metodi d'indicizzazione.

3.2.1.2 Signature Index

Un altro metodo utilizzato per creare gli indici è il Signature index. Questo metodo è molto simile all'Inverted Index. Infatti utilizzare il signature index significa andare ad applicare il metodo inverted solo sulle prime n parole (solitamente $n=40$) presenti nel testo e non su tutto il documento come solitamente fa l'inverted index.

La scelta di questo metodo tende evidentemente a creare indici di modeste dimensioni, facilmente consultabili e molto veloci nel recuperare le informazioni. D'altra parte all'alta velocità di recupero si contrappone una bassa precisione legata proprio alla scarsa quantità di termini indicizzati con i quali soddisfare l'interrogazione dell'utente. Ecco perché il signature index viene utilizzato per creare sistemi di recupero che lavorano su banche di dati contenenti articoli, pubblicazioni, ed in generale tutte quelle informazioni nelle quali è presente un abstract che riassume le parti essenziali del testo vero e proprio. Nell'abstract presumibilmente sono contenute tutte le parole "significative" che permettono il recupero del documento.

3.2.1.3 Concept Index

L'ultimo metodo d'indicizzazione è il Concept Index che si basa sulla conoscenza di concetti chiave dai quali andare a costruire una rete di relazioni fra i termini del documento indicizzato (ved. Fig. 3.2). La navigazione di questa rete permetterà, per gradi, di andare ad individuare il termine e quindi il documento da recuperare.

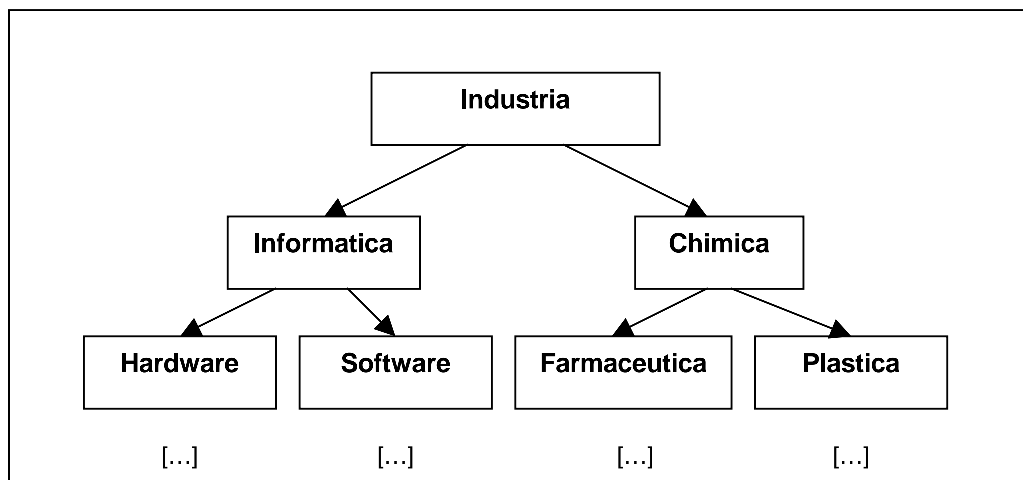


Fig. 3.2. Esempio di una rete di interazioni fra termini

Questo metodo anche se in alcuni casi è molto efficace, specie per particolari classi di documenti che si riferiscono ad una stessa terminologia, non è molto utilizzato in quanto è complesso da implementare. Inoltre i sistemi che utilizzano il concept index dipendono molto dal modo in cui vengono espressi i concetti che sono alla base della rete relazionale.

Per migliorare l'efficacia del metodo intervengono nel processo di recupero ed indicizzazione vocabolari, thesauri e sinonimi che accrescono la base di conoscenza (i concetti) affinché possano essere soddisfatte anche quelle query non direttamente riconducibili all'indice creato. I limiti del metodo

comunque rimangono, specie quando si ha a che fare con testi di carattere generale per i quali l'utilizzo dei thesauri, ecc. non è sufficiente.
Va comunque sottolineato il pregio di questo metodo che permette di ottenere come risultato della ricerca oltre che il testo direttamente collegato all'interrogazione, anche altri testi che possono essere equivalenti o "vicini" all'oggetto dell'interrogazione fatta. L'utente nel peggiore dei casi avrà in risposta un serie di documenti che comunque sono attinenti all'argomento d'interesse.
Il concept index è tuttora in via di sviluppo grazie all'utilizzo delle reti neurali, all'intelligenza artificiale ed alle tecniche di lavoro collaborativo che stanno diventando sempre più diffuse tramite Internet e l'utilizzo delle reti di computer.

3.2.2 Prestazioni di un I.R.S.

Le prestazioni degli Information Retrieval Systems vengono solitamente misurate secondo l'efficienza e l'efficacia nel recupero delle informazioni.
Prima di tutto dobbiamo definire cosa s'intende per efficienza ed efficacia di un sistema. Diremo allora che un sistema è efficace se riesce a soddisfare gli obiettivi per i quali è stato progettato e quindi se riesce a recuperare il testo desiderato dall'utente. Lo stesso sistema è efficiente quando raggiunge gli obiettivi seguendo le migliori strategie, ottimizzando le risorse, lavorando nella maniera migliore (ad esempio ottimizzando la velocità di recupero o la memoria macchina occupata).
Per valutare l'efficienza e l'efficacia degli I.R.S. dobbiamo necessariamente introdurre alcuni parametri attraverso i quali andremo a valutare le prestazioni del sistema. Solo tramite questi elementi ed il loro confronto potremo pervenire a delle valide stime.

3.2.2.1 Parametri per valutare l'efficacia: precision e recall

Per valutare l'efficacia di un sistema di Information Retrieval è necessario avere a disposizione una collezione bibliografica, un insieme di richieste da soddisfare e un insieme di giudizi di pertinenza (quali documenti sono pertinenti a ciascuna richiesta).

Supponiamo, quindi, di poter definire, per ogni documento in una data collezione, se il documento stesso è pertinente o meno con riferimento ad una data richiesta.

La misura dell'efficacia cerca di stabilire la similarità fra l'output del sistema di Information Retrieval, costituito di solito da un insieme di documenti ordinati per pertinenza, e quello dei giudizi di rilevanza noti a priori. La grandezza fondamentale, che sta alla base della maggior parte delle misure proposte, è quella di "documento recuperato pertinente"; in altri termini ogni ragionevole misura di valutazione deve essere proporzionale al numero di documenti recuperati dal sistema che sono pertinenti.

Le due variabili di riferimento che sono state definite per misurare l'efficacia di un sistema sono precisione (precision) [1] e richiamo (recall) [1], ed entrambe sono proporzionali al numero di documenti pertinenti recuperati; quello che cambia è il peso assegnato a ciascun documento addizionale pertinente. Più precisamente, la precisione è data dal rapporto fra il numero di documenti recuperati pertinenti e il numero di documenti recuperati. Il richiamo è il rapporto fra il numero di documenti recuperati pertinenti e il numero di documenti pertinenti. La precisione, pertanto, misura l'abilità del sistema di recuperare *solo* documenti pertinenti; il richiamo misura l'abilità del sistema di recuperare *tutti* i documenti pertinenti. Sperimentalmente si è visto che queste due variabili, comprese fra 0 e 1, crescono in modo inverso. Per bassi valori del richiamo si hanno buoni valori di precisione; se si vuole aumentare il richiamo si ha di solito una diminuzione della precisione.

Avendo definito questi concetti, quello che serve è un meccanismo basato su di essi per esprimere sinteticamente le prestazioni di un sistema.

3.2.2.1.1 La curva richiamo-precisione

Il metodo più diffuso per rappresentare e misurare l'efficacia utilizza una curva richiamo-precisione [1], che unisce in un grafico una serie di punti rappresentativi di coppie di valori richiamo-precisione. Per calcolare questi punti si procede nel modo seguente. Per ogni richiesta, si considera l'insieme ordinato dei documenti recuperati dal sistema e si sceglie come variabile indipendente il richiamo. Per ogni valore di richiamo compreso nell'intervallo (0 – 1) si calcola il valore di precisione associato. In pratica, occorre

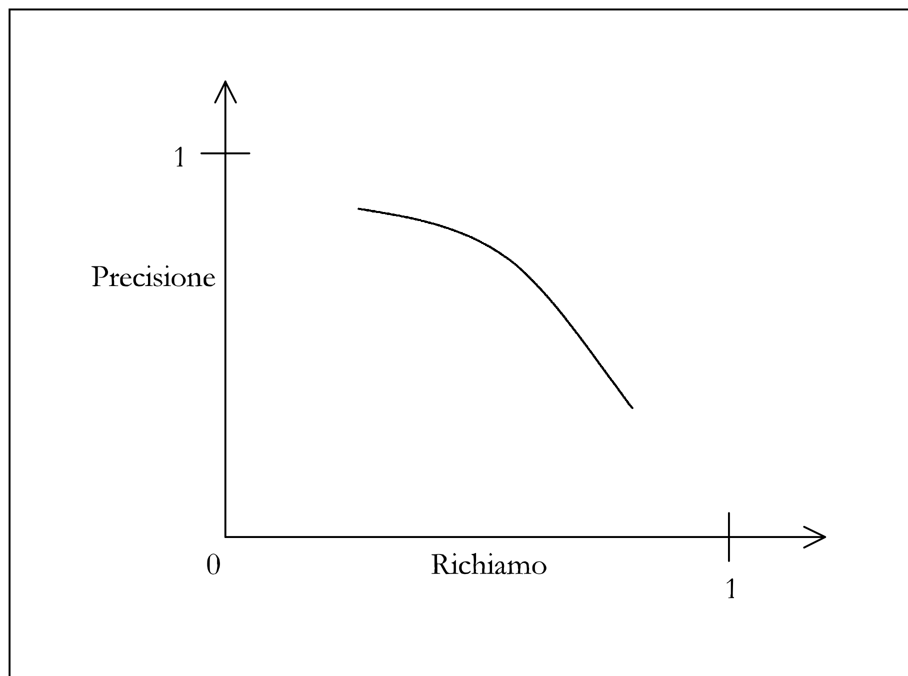
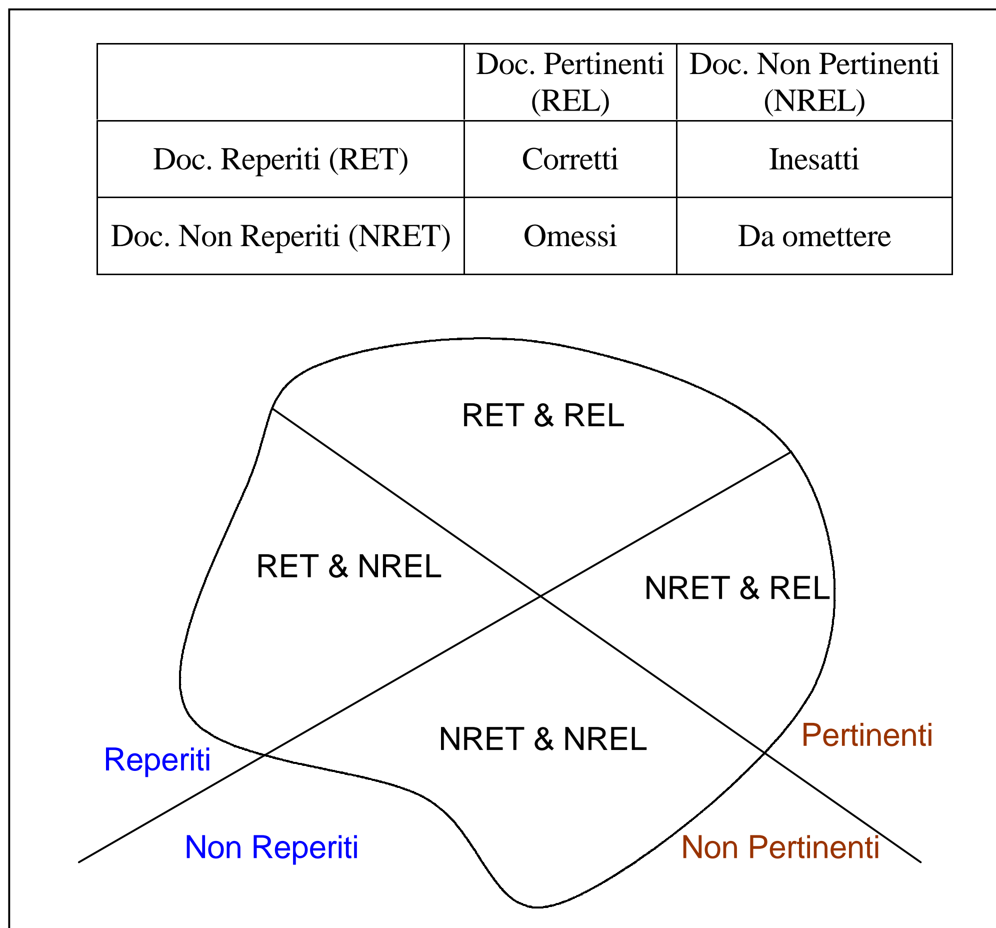


Fig. 3.3. Modello di curva Precisione-Richiamo (convessa)

determinare per ogni punto il sottoinsieme di documenti recuperati sufficiente a garantirsi i livelli di richiamo desiderati e poi calcolare il valore di precisione

relativamente ad esso. Tracciando la precisione in funzione del richiamo, quello che tipicamente si ottiene è una classica curva monotona
discendente concava o convessa (ved. Fig. 3.3).
Una volta costruita la curva precisione-richiamo relativa a ciascuna richiesta, il passo successivo è di fondere i risultati in una curva di prestazioni complessiva, facendo la media dei valori di precisione in ciascun punto relativamente all'insieme di richieste. A questo punto si possono confrontare le



	Doc. Pertinenti (REL)	Doc. Non Pertinenti (NREL)
Doc. Reperiti (RET)	Corretti	Inesatti
Doc. Non Reperiti (NRET)	Omessi	Da omettere

Fig. 3.4. Schema di suddivisione di una collezione di documenti

prestazioni di sistemi differenti sovrapponendo le relative curve richiamo-

precisione e adottando metodi statistici per valutare se le differenze riscontrate siano significative o meno [9].
Per capire meglio schematizziamo i concetti. L'insieme dei documenti si può partizionare in 4 sottoinsiemi (ved. Fig. 3.4), dove la classificazione di un documento considera se lo stesso è pertinente o meno all'interrogazione e se è stato reperito o meno dall'I.R.S.. I documenti nell'insieme RET & NREL, pur se soddisfano l'interrogazione, non sono di rilievo ai fini della ricerca.; ogni I.R.S. dovrebbe cercare di ridurre al minimo la cardinalità di questo insieme. Questi documenti sono anche detti *false drops* o *false alarms* o *false hits*.
I documenti nell'insieme NRET & REL, viceversa, sono quelli che non sono reperiti ma che dovrebbero esserlo. Ovviamente anche la cardinalità di questo insieme dovrebbe essere ridotta al minimo.
Un sistema ideale sarebbe, quindi, quello che rende vuoti questi due insiemi.
In base allo schema precedente possiamo definire le formule per il calcolo di richiamo e precisione:

richiamo[5] $R = \frac{\#RET \,\&\, REL}{\#REL}$

precisione $P = \frac{\#RET \,\&\, REL}{\#RET}$.

Poiché è estremamente difficile anche solo avvicinarsi alla costruzione di un I.R.S. ideale (R = 1, P = 1) spesso capita, purtroppo, che in risposta all'interrogazione si abbia un vasto insieme di documenti che, pur contenendo il termine oggetto dell'interrogazione, non sono d'interesse per l'utente.
Per quanto riguarda le modalità di indicizzazione, questo fenomeno è riscontrabile soprattutto nei sistemi che utilizzano l'inverted index poiché in

[5] Il simbolo # vuole indicare la cardinalità dell'insieme.

questo caso si fa riferimento solo a dati oggettivi come ad esempio la frequenza dei termini o le relazioni parola-locazione, mentre non si fa riferimento a conoscenze che possono essere acquisite dall'esterno del sistema, come legami logici fra termini e concetti, parole appartenenti a specifici vocabolari, ecc.; queste conoscenze sono, invece, alla base del concept index e permettono di determinare se il documento recuperato è pertinente all'interrogazione posta dall'utente, cioè se esiste una relazione logica fra il documento recuperato e la query.

Occorre comunque precisare che, attraverso tecniche basate sul calcolo delle probabilità, è possibile migliorare notevolmente la prestazione dal punto di vista della pertinenza (*relevance*) dei documenti recuperati [3] anche utilizzando una struttura inverted index.

In conclusione, sicuramente il concept index è la metodologia più indicata per ottenere livelli soddisfacenti di efficacia, ma questa metodologia è fortemente limitata dalle difficoltà di implementazione che vengono riscontrate nel momento in cui si deve creare la base di conoscenza. Va comunque detto che le prestazioni dei sistemi attualmente utilizzati crescono enormemente quando l'utente conosce bene l'argomento da recuperare o la base di testi dalla quale attingere le informazioni; ciò comporta il più delle volte richieste molto specifiche che spesso il sistema riesce a soddisfare pienamente.

3.2.2.2 Valutazione dell'efficienza

L'efficienza dei sistemi di Information Retrieval può essere misurata tramite una serie di parametri che mettono in evidenza come il sistema raggiunge gli obiettivi. Per i metodi d'indicizzazione un fattore determinante è la velocità di recupero che è strettamente legata alla dimensione dell'indice da trattare. Infatti minore è la dimensione dell'indice maggiore è la velocità di recupero

poiché il sistema dovrà compiere meno iterazioni per verificare se il termine di ricerca è contenuto nella struttura.

Ottime prestazioni per quanto riguarda il tempo di risposta sono state riscontrate nei sistemi che utilizzano gli inverted index. Questi metodi infatti, anche tramite algoritmi di compressione, riescono a creare degli indici di dimensioni modeste rispetto al testo da indicizzare. Altri fattori determinanti sono la **facilità** e la **velocità** con le quali si riesce a creare o a ricostruire l'indice. Anche in questo caso gli inverted index, che sono completamente automatizzati, raggiungono ottime prestazioni, mentre per il concept index bisogna intervenire manualmente.

Come vedremo nel capitolo 5 le esigenze riscontrate nell'attività di pre-traduzione in Logos porteranno a preferire l'utilizzo di un inverted index come struttura d'appoggio per le ricerche di tipo approximate match testuale.

Possiamo quindi concludere che in generale i sistemi che utilizzano il concept index sono più efficaci, mentre i sistemi che utilizzano gli inverted index sono più efficienti.

3.2.3 Un metodo per "eseguire" un'interrogazione

Vedremo ora come può essere implementata la fase di ricerca vera e propria da parte dell'esecutore di un I.R.S. (ved. Fig. 3.1). Analizzeremo soltanto il contesto dell'inverted index in quanto quello più largamente utilizzato.

Il metodo con cui l'esecutore cerca di esaudire una richiesta segue i seguenti passi:

1. *parsing* della query in termini ed operatori logici (se presenti);
2. ricerca nell'indice dei termini e recupero, dalle liste invertite, delle liste di documenti;

3. esecuzione degli algoritmi di operazioni su liste (AND, OR, NOT) e "*ranking*" (valutazione dell'importanza) dei documenti che rispondono ai requisiti della query;
4. ordinamento dei risultati in base al loro "*rank*".

Queste quattro fasi possono essere progettate indipendentemente l'una dall'altra. L'unico vincolo è, ovviamente, quello di rispettare le dipendenze funzionali che ci sono tra ognuna di loro.

Tralasceremo, ora, la prima fase, quella di *parsing*, per soffermarci maggiormente sulle altre.

La fase di ricerca nell'indice dipende dall'organizzazione seguita da questa struttura dati. Utilizzando un array di parole ordinate alfabeticamente il tipo di ricerca più appropriato, per esempio, è la *ricerca binaria* che ha una complessità $O(\log Nt)$ dove Nt è il numero di parole presenti nell'indice.

La terza fase consiste nell'esecuzione delle operazioni contenute nell'interrogazione, ad esempio per restituire il risultato di una ricerca avente come argomenti **`cane and animale`** sarà necessario effettuare l'*intersezione* tra le liste riferite a *cane* e quelle riferite ad *animale*. Analogo discorso vale anche per l'OR e per il NOT. Con le liste invertite ordinate secondo un campo identificativo dei documenti `doc_id`, l'esecuzione di algoritmi di intersezione, unione, differenze avviene in tempi proporzionali alla somma delle lunghezze delle liste. Considerando, per esempio, l'intersezione di due liste lunghe, rispettivamente, $L1$ e $L2$, il tempo di intersezione sarà $O(L1+L2)$. Varie ottimizzazioni di questi algoritmi sono presenti in letteratura [4].

Durante la fase di calcolo degli operatori, solitamente, un esecutore di interrogazioni calcola il valore dell'importanza (*Rank*) dei documenti contenuti nella lista dei risultati sfruttando le informazioni contenute nell'indice. Vari approcci a questa determinazione di importanza possono

essere valutati (probabilistico, vettoriale, booleano) e per queso si rimanda alla vasta letteratura disponibile sull'argomento [1,6,7,8].

L'ultima fase dell'algoritmo di ricerca consiste nell'*ordinamento* della lista dei risultati secondo il *rank*.

Gli algoritmi di sorting convenzionali, richiedono un numero di confronti proporzionali a *Wlog(W)* (*W* è il numero di elementi da ordinare). Se *W=1.000.000* il numero di confronti necessario ad ordinare l'intera lista sarà di circa 20 milioni, che si traduce in diversi secondi di attesa su di un computer di media potenza.

Per questo motivo generalmente, mostrando solo i primi R risultati più significativi, si fa in modo che non sia necessario ordinare tutti i documenti trovati.

Il tipico algoritmo utilizzato, quindi, non è uno di *sorting* ma uno di *selezione* e sfrutta una struttura dati definita *heap* [4] (ved. Fig. 3.4).

Gli inserimenti in un *heap* hanno un tempo *medio* di esecuzione

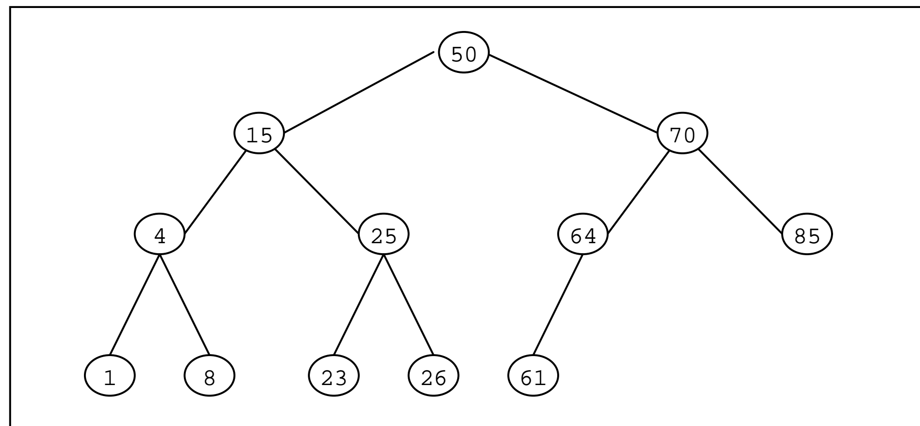


Fig. 3.4. Un esempio di struttura dati "heap"

proporzionale al numero di elementi da inserire *O(n)* (nel caso pessimo il tempo è *O(nlog(n))* come nel caso del *sorting)*; considerando l'esempio

precedente il numero di confronti adesso scende a 1.000.000 con notevole risparmio di tempo.

C a p i t o l o   4

L'AMBIENTE ORACLE 8i INTERMEDIA

In questo capitolo vedremo le caratteristiche del pacchetto Oracle 8i – *inter*Media affrontando dapprima una breve descrizione delle sue funzionalità e, successivamente, convogliando l'attenzione su *inter*Media Text. Circa questo strumento saranno descritti la modalità di indicizzazione dei testi e i modi in cui vengono gestite le interrogazioni tipiche dell Information Retrieval.

Questa parte di tesi sarà volutamente sintetica nel contenuto per non appesantire eccessivamente il lettore con dati tecnici. È, comunque, necessario dare una descrizione di un importante strumento di lavoro analizzato nello svolgere il lavoro di tesi.

Oracle *inter*Media è uno strumento in grado di gestire contenuto multimediale. Esso permette ad Oracle di gestire testi, documenti, audio, video in maniera integrata con le altre informazioni strutturate presenti sulla base di dati.

*inter*Media Text è una estensione di Oracle 8i che permette l'indicizzazione di documenti di testo e l'esecuzione di interrogazioni (come, per esempio, la ricerca dei documenti che contengono una certa parola) utilizzando semplicemente i costrutti di SQL standard. Si ha quindi una integrazione tra la gestione dei testi e le più tradizionali operazioni svolte sulle basi di dati relazionali. i*nter*Media Text consente, in altre parole, di estendere ad un

DBMS Oracle tutte le operazioni svolte tipicamente da un **Information Retrieval System** (ved. § 3.2).

4.1 La creazione dell'indice

Il primo passo nell'utilizzo di *inter*Media è la creazione dell'indice sul testo.
Occorre dire, prima di tutto, che, se le normali interrogazioni in un DBMS sono più lente senza la presenza di un indice (*b-tree index*), le interrogazioni eseguite attraverso la primitiva `contains` (ved. § 4.2.2), caratteristica peculiare di *inter*Media Text, non sono possibili senza aver prima creato un indice sulla colonna contenente il testo sul quale operare le ricerche.
Il metodo di indicizzazione utilizzato da *inter*Media è quello dell'inverted index (ved. § 3.2.1.1). Questo metodo è basato sulla costruzione di una struttura che associa ad ogni parola i documenti che la contengono e la posizione all'interno dei quali si trova la parola considerata.
Un tale indice viene creato attraverso il comando

```
CREATE INDEX my_index on my_table ( my_column )
  INDEXTYPE IS ctxsys.context
  PARAMETERS
    ( 'datastore      my_datastore
       filter         my_filter
       section group  my_section_group
       lexer          my_lexer
       wordlist       my_wordlist
       stoplist       my_stoplist
       storage        my_storage'
    );
```

Oracle 8i riconosce il tipo di struttura da creare ed inizia l'indicizzazione del testo. All'interno della clausola `PARAMETERS` viene inserita una lista di parametri che permettono di costruire un indice che sia adatto alle nostre esigenze e alla tipologia di documenti che abbiamo. Questi parametri verranno analizzati più in dettaglio nell'apposita sezione ma, per meglio capire in che

momento dell'indicizzazione essi interverranno, vediamo una descrizione sommaria dell'operazione.

L'indice è creato attraverso una sequenza di passi, eseguiti sequenzialmente, denominata *Indexing Pipeline* (ved. Fig. 4.1). Diamo ora una breve descrizione del significato di ogni passo:

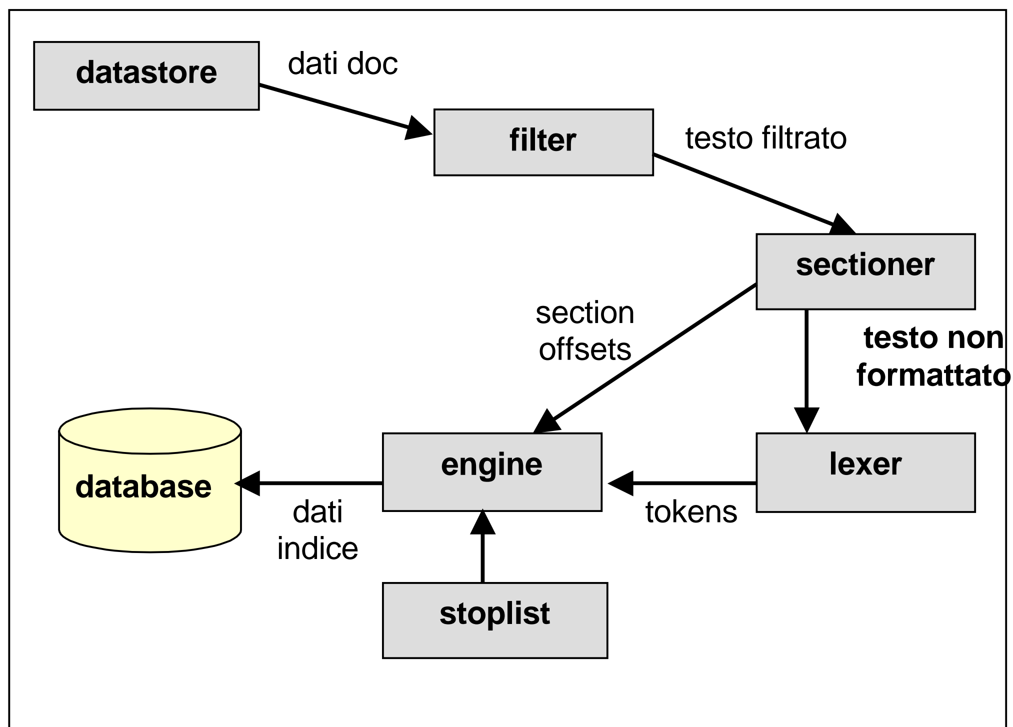


Fig. 4.1. La "Indexing Pipeline" per la creazione di un indice

- *datastore*: in questo stadio della pipeline viene eseguito un ciclo di lettura sulla colonna indicizzata e sono prodotti come output i dati del documento. Questi dati possono essere testo vero e proprio oppure, per esempio, puntatori che, a loro volta, permetteranno di accedere a testo;

- *filter*: vengono assunti in ingresso i dati provenienti dal datastore i quali vengono “tradotti” in testo leggibile.
- *sectioner*: converte in testo senza formattazioni (*plain text*) l’output del filter. Questa conversione include la ricerca di etichette che delimitano particolari sezioni nel testo e la loro rimozione. Il sectioner darà come output anche gli eventuali *section offsets* (ved. Fig. 4.1) che verranno utilizzati per la ricognizione delle sezioni presenti sul documento.
- *lexer*: viene analizzato il testo proveniente dal sectioner e viene separato in termini (*tokens*). La segmentazione viene eseguita tenendo presenti le regole di delimitazione delle parole nei vari linguaggi gestiti (nelle lingue asiatiche, per esempio, la segmentazione è piuttosto complessa).

Al termine di questa sequenza di passi viene di fatto costruito l’indice invertito (inverted index). Per compiere questa operazione vengono presi in esame i termini provenienti dal lexer, i section offsets provenienti dal sectioner e la lista di parole considerate di scarso significato che non si intende inserire nell’indice (*stoplist*).

Ogni passo della pipeline è influenzato dal modo in cui sono stati impostati i parametri elencati nell’istruzione `create index` vista in precedenza. Nei prossimi paragrafi descriveremo più in dettaglio come sia possibile impostare questi parametri (che, secondo la terminologia fornita da Oracle, chiameremo classi) e il significato delle loro impostazioni.

4.1.1 La classe Datastore

Attraverso questo parametro viene indicato il modo di interpretare i dati estratti dalla colonna di tabella su cui viene costruito l’indice. Il risultato di questa lettura verrà poi trasmesso, come accennato prima, alla classe Filter (ved. § 4.2.1.2). La classe Datastore ha la possibilità di gestire cinque oggetti per specificare come il nostro testo è stato memorizzato.

DIRECT_DATASTORE

Questo è il caso più semplice nel senso che si assume il fatto che il contenuto dei documenti di testo sia stato immagazzinato proprio nella colonna indicizzata (un documento per riga). Questo valore è assunto come default, pertanto nell'istruzione `create index` può essere omesso.

FILE_DATASTORE

In questo caso i dati letti sulla colonna indicizzata saranno considerati nomi di files. Verranno perciò aperti questi files attraverso il file system locale; il loro contenuto verrà restituito come contenuto dei documenti di testo.

URL_DATASTORE

Si utilizza per testo memorizzato in files su World Wide Web (acceduti mediante protocollo *http* o *ftp*) oppure in files nel file system locale (acceduti attraverso il protocollo *file*).

DETAIL_DATASTORE

Questo oggetto si utilizza qualora il testo sia memorizzato direttamente nella base di dati ma in sottotabelle (*detail tables*) le quali saranno referenziate, attraverso una chiave d'accesso, da una tabella principale (*master table*).

USER_DATASTORE

Serve per poter utilizzare procedure create dall'utente che manipolano i dati presenti in colonne della tabella. Per esempio un utente potrebbe unire al testo, attraverso appunto una procedura, autore e data per poter indicizzare, insieme al contenuto del documento, anche queste due informazioni aggiuntive. (Va sottolineato il fatto che *inter*Media non permette di definire indici composti).

4.1.2 La classe Filter

Attraverso la classe filter viene specificato come il testo venga filtrato per l'indicizzazione. Viene data la possibilità di indicizzare testo formattato da word processor, testo non formattato (*plain text*), HTML e XML.

Per i testi formattati, Oracle utilizza i filtri per costruirne versioni temporanee, in formato plain text o HTML. Vengono poi indicizzate le parole derivanti da queste versioni temporanee. La classe filter viene gestita attraverso l'utilizzo di uno dei seguenti objects:

CHARSET_FILTER

Permette di convertire documenti aventi un set di caratteri diverso da quello gestito dalla nostra base di dati

INSO_FILTER

Vengono utilizzati strumenti di filtraggio forniti dalla *Inso Corporation*. Questi strumenti permettono di gestire la maggior parte dei formati esistenti per i documenti (come, per esempio quelli di Microsoft Word per Windows e di Acrobat/PDF).

NULL_FILTER

Si usa quando il documento da indicizzare è in formato *plain text* o HTML e non sono necessarie operazioni di filtraggio.

USER_FILTER

Permette di inserire un filtro costruito dall'utente. Questo filtro consiste in un programma che, invocato dall'*indexing engine*, si occuperà di effettuare le operazioni di filtraggio di ogni documento da indicizzare. Si potrà inserire, per esempio, uno *script* che converta interamente un testo in lettere maiuscole.

4.1.3 La classe Section_group

Questa classe permette di individuare e gestire sezioni in un documento per poter poi eseguire interrogazioni attraverso la clausola `within` (ved. § 4.2.2). Questo passo di indicizzazione fornisce due output (ved. Fig. 4.1): *plain text*, scorporato da eventuali etichette, verso il lexer e *section offsets*, per riconoscere le sezioni presenti, verso l'*indexing engine*. Per poter creare un section group occorre specificare uno dei seguenti tipi di oggetti:

NULL_SECTION_GROUP

È il valore di default. Non vengono considerate sezioni sul documento.

BASIC_SECTION_GROUP

Viene utilizzato qualora si siano inserite nel documento sezioni cotrassegnate da etichette come <A> e </A>.

HTML_SECTION_GROUP

Serve per indicizzare documenti HTML. Verranno riconosciute tutte le sezioni normalmente definite su un documento in questo formato.

XML_SECTION_GROUP

È equivalente al caso precedente ma per documenti XML.

NEWS_SECTION_GROUP

Questo tipo di gruppo manipola documenti aventi sezioni tipiche dei messaggi di newsgroup.

4.1.4 La classe Lexer

La classe lexer viene utilizzata per specificare il linguaggio del testo da indicizzare. Questa scelta determinerà il modo in cui il testo stesso viene diviso in termini (*tokens*). Per gestire questa preferenza si sceglierà uno dei seguenti oggetti:

BASIC_LEXER

Utilizzato per estrarre i termini da testi scritti in linguaggi la cui codifica viene eseguita utilizzando un byte per carattere. (Come nel caso, per esempio, della lingua inglese e della maggior parte dei linguaggi occidentali).

CHINESE_VGRAM_LEXER

JAPANESE_VGRAM_LEXER

KOREAN_LEXER

Utilizzati per identificare termini in testi rispettivamente in cinese, giapponese e coreano.

4.1.5 La classe Stoplist

La stoplist è una lista di parole che, per vari motivi, non hanno una
rilevanza tale da essere inserite nell'indice del nostro testo. Un esempio di parole che fanno spesso parte di stoplists sono senz'altro gli articoli, le preposizioni, le congiunzioni. Nella maggior parte dei casi non vale la pena utilizzare spazio per queste parole.

La classe stoplist detiene quindi una lista di queste parole, le cosiddette *stopwords* (o *noise words*). L'*indexing engine* consulta la stoplist ed esegue un filtraggio in modo tale da non far rientrare nell'indice nessuna delle *stopwords*.

*inter*Media fornisce stoplists standard per molti linguaggi ma, naturalmente, è possibile crearne una propria secondo le esigenze.

4.1.6 La classe Wordlist

La classe Wordlist, in realtà, non ha nessun effetto sull'indicizzazione. Essa permette di impostare i parametri che serviranno per eseguire ricerche complesse sul testo.

L'unico oggetto fornito per questa classe è la BASIC_WORDLIST i cui parametri da impostare sono:

stemmer	specifica in quale linguaggio eseguire l'eventuale *stemming* delle parole (ved. § 3.2.1.1);
fuzzy_match	specifica le routines da eseguire per trovare parole "somiglianti"; serve per risolvere, per esempio, casi di errori di battitura (*mistyping*);
fuzzy_score	indica "quanto somiglianti" devono essere le parole rintracciate rispetto all'oggetto della ricerca;
fuzzy_numresults	serve per limitare il risultato della ricerca a un certo numero delle parole corrispondenti.

4.2 Metodi e tipi di ricerca

Presentiamo ora i differenti scenari di ricerca che si presentano, in generale, nell'ambito dell'Information Retrieval. Illustreremo, per ciascuno di essi, le modalità che Oracle 8i *inter*Media utilizza nei diversi tipi di interrogazione.

4.2.1 Interrogazioni "Direct Match"

Nello scenario più semplice l'utente conosce il termine esatto che egli sta cercando (ad esempio *Computers*). Il processore di ricerca (ved. § 3.2.3) deve semplicemente cercare le corrispondenze tra questo termine e ogni parola in ogni documento nella banca dati testuale e restituire quei documenti che contengono almeno una corrispondenza. Poiché questo può essere un procedimento molto lento quando sono presenti molti documenti da analizzare, il processore analizza preventivamente i documenti costruendo un *inverted index* (Ved. 3.2.1.1). Come abbiamo detto, questo indice contiene tutti i termini significativi presenti nei documenti con l'indicazione, per ogni termine, del documento nel quale si trova. Una semplice

interrogazione, quindi, può essere gestita scorrendo l'indice alla ricerca dei documenti pertinenti.

Per svolgere ricerche di questo tipo *inter*Media fornisce la primitiva `contains` che viene inserita in istruzioni SQL come nel seguente esempio:

```
select text
from my_table
where contains (text, 'cane') > 0;
```

Questa ricerca restituirà il contenuto dei documenti referenziati dalla tabella `my_table` che contengono la parola "cane". È da sottolineare il fatto che la primitiva `contains` restituisce come risultato un valore proporzionale alla pertinenza del documento reperito rispetto alla interrogazione. Per questo motivo si pone il vincolo "`> 0`".

4.2.1.1 Ricerche per frasi

In alcuni casi, il termine che l'utente sta cercando non è una singola parola ma una frase (o comunque un insieme di parole come, ad esempio, *computer science*). Una ricerca per frase richiede forzatamente che l'inverted index contenga informazioni sulla posizione di ogni singola parola all'interno dei documenti. Questo consente al processore di reperire i documenti che contengono la sequenza di termini cercata. L'indice costruito da i*nter*Media Text contiene informazioni circa la posizione dei termini proprio per consentire l'esecuzione di ricerche per frasi sempre attraverso la primitiva `contains`.

In alcuni casi può capitare di eseguire ricerche per più parole che però non sono contigue. Per esempio l'utente può essere interessato a reperire i documenti che contengano "Camillo Cavour", "Camillo Benso Conte di Cavour" oppure "Cavour, Camillo". *inter*Media Text permette, attraverso l'operatore `near`, di ricercare una combinazione delle parole "Camillo" e

“Cavour” in modo tale che esse siano vicine l’una all’altra ma non necessariamente contigue. Il livello di “vicinanza” tra le parole viene impostato dall’utente. Vediamo un esempio:

```
select text
from my_table
where contains (text, near((cane, gatto),10))) > 0;
```

in questo caso saranno restituiti quei documenti che contengono le parole “cane” e “gatto” intervallate, al massimo, da altri dieci termini.

In alcuni linguaggi la distinzione tra parole e frasi a volte può sfumare. Per esempio, il Tedesco e l’Olandese hanno parole composte che spesso devono essere divise per permettere di eseguire ricerche. i*inter*Media contiene un “decompositore” per far fronte a casi come questi.

Nei linguaggi orientali (in particolare Cinese, Giapponese e Coreano) non ci sono separatori di parola nei documenti. i*inter*Media, attraverso la classe Lexer (ved. § 4.2.1.4) suddivide questi tipi di documenti in termini che si prestino all’indicizzazione.

4.2.2 Interrogazioni “Indirect Match”

Le tecniche di ricerca analizzate nel precedente paragrafo sono spesso insufficienti per la maggior parte dei casi in cui si voglia reperire testo da banche dati. L’utente può non conoscere il termine esatto da ricercare, il termine voluto può comparire in più forme o coniugazioni all’interno dei documenti o l’utente può volere ricercare un’argomento, non una particolare parola. In tutti queste situazioni l’esecutore deve fornire gli strumenti necessari per una corretta ricerca.

Esistono due approcci di base per eseguire interrogazioni “Indirect Match”: *term expansion*, in cui un termine è espanso in un insieme di termini (ad esempio un verbo viene espanso in tutte le sue coniugazioni), e *term*

normalization, in cui più termini vengono fatti "collassare" in un singolo termine (ad esempio tutte le coniugazioni di un verbo vengono portate all'infinito).

Tecniche di *term expansion* e *term normalization* possono essere applicate sia durante il processo di indicizzazione che durante il processo di interrogazione. Un termine dell'indice può essere "espanso" in un insieme di termini, i quali vengono posti poi tutti nell'indice, oppure può essere "normalizzato" in modo tale che diversi termini equivalenti siano ricondotti alla stessa voce nella struttura indice. Nello stesso modo, un termine dell'interrogazione può essere espanso, per poi ricercare tutti i termini prodotti dall'espansione, o normalizzato, per ricondurlo a una forma equivalente, prima di effettuare la scansione dell'indice.

L'esecutore deve quindi possedere la capacità, a seconda dei casi, di eseguire espansioni o normalizzazioni sia in fase di creazione di indici che in fase di risposta ad una interrogazione.

La scelta tra i due approcci è spesso basata su fattori che incrementano le prestazioni del sistema di reperimento come le dimensioni dell'indice risultante o il tempo impiegato nella espansione o normalizzazione dei termini.

I diversi casi che ora vedremo comporteranno l'utilizzo di tecniche di espansione o normalizzazione.

4.2.2.1 Wildcard Match

Spesso si presentano situazioni in cui è necessario effettuare ricerche utilizzando solo una parte di parola. È questo il caso in cui si utilizzano le *wildcard*, quegli indicatori che sostituiscono una qualsiasi sequenza di caratteri (ad esempio "*" in MS-DOS). Deve essere riscontrata una

corrispondenza parziale con i termini dell'indice. La primitiva `contains` supporta questa *wildcard search* con l'operatore "%". Vediamo un esempio:

```
select text
from my_table
where contains (text, 'cane%') > 0;
```

restituirà "cane" ma anche, per esempio, "canestro".

4.2.2.2 Mistyping

Un termine può apparire in forme differenti anche a causa di errori o della scarsa qualità dei documenti (come i risultati di un OCR). In casi come questo è importante che il processore sia in grado di eseguire la normalizzazione dei termini errati basandosi su regole note. Questo è anche ciò che *inter*Media fa.

4.2.2.3 Derivazione o radicalizzazione di termini

Le parole nel linguaggio naturale hanno molte forme elaborate con le quali appaiono nei documenti. Per esempio "destroys" e "destroying" sono un insieme di forme elaborate della stessa radice "destroy". I termini di una interrogazione possono essere espansi (*derivazione*) in tutte le loro forme elaborate utilizzando regole morfologiche che tengono conto delle regole grammaticali del linguaggio naturale (ciò può essere eseguito, per esempio, adottando un procedimento di *stemming* (ved. § 3.2.1.1)). I termini espansi saranno allora confrontati con ogni altra forma elaborata presente nell'indice. Agendo in maniera opposta, le forme elaborate dei termini di un'interrogazione possono essere normalizzate (*radicalizzazione*). Viene così prodotta la radice delle parole che sarà utilizzata nella ricerca all'interno dell'indice (le cui voci saranno, a loro volta, radici dei termini dei documenti.

*inter*Media gestisce sia l'espansione che la normalizzazione attraverso la creazione di un concept index (*theme index*) che viene abilitata per mezzo dell'attributo *index_themes* nella classe Lexer. I dati per creare la rete di interazioni del concept index vengono estratti da una *knowledge base* interna[6] contenente circa mezzo milione di termini [10]. L'operatore che permette di eseguire interrogazioni sfruttando questa struttura ausiliaria è `about`. Questa funzionalità di *inter*Media verrà meglio analizzata e descritta nel capitolo 5.

4.2.2.4 Gestione delle relazioni tra termini

I legami tra i termini (sinonimi, relazioni gerarchiche, ecc.) in genere sono gestiti attraverso l'introduzione di relazioni in thesauri[7] creati dall'utente.
*inter*Media permette l'inserimento e l'utilizzo di strutture thesaurus ma, spesso, ciò non è necessario dato il grande numero di relazioni presenti nella knowledge base interna.

4.2.3 Interrogazioni composte

Spesso gli utenti vogliono combinare termini multipli in interrogazioni composte per esprimere in maniera più precisa ciò che essi intendono ricercare. Per esempio può essere necessario introdurre espressioni come "cane but not pastore tedesco" oppure "gatto and felino". *inter*Media permette di eseguire interrogazioni di questo tipo attraverso la primitiva

---

[6] La knowledge base interna è organizzata in maniera gerarchica secondo sei argomenti principali: business and economics, science and technology, geography, government and military, social environment, abstract ideas and concepts.

[7] Thesauri è il plurale di thesaurus. Un thesaurus è un documento (in genere un file di testo) che, come un dizionario o un'enciclopedia, unisce termini legati da significato (sinonimi e contrari) o da relazioni di appartenenza (ad esempio "cane appartiene a mammiferi"). La differenza principale tra un thesaurus e la knowledge base interna sta nella diversa disposizione dei concetti all'interno del file che li contiene [11].

`contains`. Ogni espressione in una interrogazione composta può essere di uno qualsiasi dei tipi di interrogazioni viste nei paragrafi precedenti.

4.2.3.1 Operatori Booleani

Gli operatori booleani (AND, OR, NOT) possono essere utilizzati per costruire un'espressione logica all'interno di un'interrogazione. *inter*Media permette di costruire espressioni di questo tipo all'interno della primitiva `contains`. Vediamo un esempio:

```
select text
from my_table
where contains (text, 'cane and gatto') > 0;
```

4.2.3.2 Interrogazioni in linguaggio naturale

Il modo più immediato per impostare un'interrogazione è farlo attraverso il linguaggio naturale. i*nter*Media gestisce questo tipo di interrogazioni per mezzo dell'operatore `about`. Attraverso questo operatore *inter*Media cerca di estrarre i termini dall'interrogazione per poi ricercarli nell'indice. Se è presente un theme index i termini dell'interrogazione vengono espansi per mezzo di esso e della knowledge base interna. Se non è presente un theme index ogni parola viene elaborata attraverso un algoritmo di stemming. I risultati di questa elaborazione vengono poi introdotti in una "query espansa" che conterrà i termini da ricercare nell'indice.

Occorre precisare che tutte queste operazioni sono trasparenti all'utente il quale inserisce semplicemente un'interrogazione in linguaggio naturale.

4.2.3.3 Interrogazioni strutturate

Spesso le applicazioni richiedono che le ricerche testuali siano combinate con altri criteri di ricerca all'interno di basi di dati. Poiché la tipica primitiva

`contains` che consente le ricerche testuali è integrata in istruzioni in SQL standard possiamo dire che le interrogazioni strutturate sono ben supportate da *inter*Media.

4.2.3.4 Ricerca all'interno di sezioni nei documenti

Questo tipo di ricerca considera la struttura interna dei documenti. Per esempio l'utente può voler ricercare un termine all'interno della *title section* di un documento in formato HTML oppure un nome nella *address section* di un messaggio di posta elettronica. i*nter*Media permette la ricerca in particolari sezioni di documenti per mezzo dell'operatore `within`. Vediamo un esempio:

```
select text
from my_table
where contains(text,'isle within title_section')>0;
```

4.3 La gestione delle istruzioni INSERT, UPDATE e DELETE

Le istruzioni di DML (Data Manipulation Language) a cui facciamo riferimento riguardano l'inserimento, la modifica o la cancellazione di documenti indicizzati. In particolare, la loro esecuzione viene gestita in questo modo:

- INSERT: il riferimento al documento inserito viene dislocato in una coda, chiamata dr$pending, per essere valutato successivamente per l'indicizzazione. Le interrogazioni eseguite prima che venga presa in considerazione la coda, non reperiranno il contenuto del nuovo documento inserito.
- UPDATE: il contenuto del documento modificato viene invalidato immediatamente e il riferimento al documento stesso viene inserito nella coda dr$pending per nuovamente analizzato per l'indicizzazione. Le

interrogazioni eseguite prima che avvenga la nuova indicizzazione non reperiranno né il nuovo contenuto né il vecchio.

- DELETE: il contenuto del documento cancellato viene invalidato immediatamente (non viene fisicamente cancellato ma il riferimento nell'indice al documento oggetto di delete, viene marcato come non più valido; in questo modo viene segnalato alle interrogazioni di rimuovere da qualsiasi risultato i documenti marcati). Non sono necessarie altre operazioni.

Possiamo dire che *inter*Media gestisce l'eliminazione di dati in maniera sincrona e l'aggiunta in maniera asincrona. Vediamo un paio di esempi: se eseguiamo l'istruzione

```
DELETE from table where contains(text,'delword')>0;
```

una successiva interrogazione

```
SELECT * from table where contains(text, 'delword')>0;
```

non restituirà alcuna tupla. L'effetto dell'istruzione di cancellazione è immediato (sincrono).

Se, invece, eseguiamo l'istruzione

```
INSERT into table values(1, 'inword');
```

una successiva interrogazione

```
SELECT * from table where contains(text, 'inword')>0;
```

non restituirà anche in questo caso alcuna tupla. Per fare in modo che venga inserita nell'indice invertito la lista dei documenti contenenti "inword" e, di conseguenza vedere soddisfatta l'interrogazione, occorre imporre al sistema di analizzare il contenuto della coda dr$pending.

*inter*Media mette a disposizione due metodi per eseguire l'aggiornamento dell'indice e renderlo consistente rispetto agli inserimenti o alle modifiche di documenti operati sulla banca dati testuale: *sync* e *background*.

Il metodo sync consiste nell'esecuzione dell'istruzione

```
alter index myindex rebuild online parameters('sync');
```

La specifica `online` è importante in quanto senza di essa l'esecuzione di interrogazioni non sarebbe possibile durante l'operazione sync.
Il metodo background prevede l'avvio di un processo attraverso il comando

```
ctxsrv –user ctxsys/
```

Una volta avviato, il processo analizza periodicamente la coda dr$pending ed esegue automaticamente in background le operazioni di aggiornamento dell'indice.
In ognuno dei due metodi, comunque, l'aggiornamento dell'indice viene effettuato aggiungendo una lista invertita anche per i termini già presenti. A lungo andare questa frammentazione dell'indice (possono esistere più liste per un termine) può portare a cali di prestazioni. Per questo motivo è opportuno eseguire un'ottimizzazione.

4.4 L'ottimizzazione

L'operazione di ottimizzazione riguarda, oltre alla frammentazione accennata in precedenza, l'eliminazione fisica dei riferimenti invalidati risultato delle operazioni di delete.
La deframmentazione dell'indice consiste nell'accorpamento delle liste invertite che appartengono ad un singolo riferimento nell'indice. Se, cioè, dopo una serie di operazioni di `update` o `insert` ci troviamo nella situazione seguente:

```
Word1 doc1 doc2
...
Word1 doc8
```

la deframmentazione produrrà la nuova lista per il termine `Word1`

```
Word1 doc1 doc2 doc8.
```

L'eliminazione fisica dei riferimenti ai documenti cancellati comporta un'operazione di *garbage collection.* Attraverso di essa, per ogni termine

contenuto nei documenti cancellati, viene eliminata dalla corrispondente lista invertita l'occorrenza legata al documento non più valido.

Le operazioni di deframmentazione e di garbage collection riducono il numero di righe nell'indice rendendone più veloce la scansione.

Per eseguire queste operazioni *inter*Media fornisce due modalità di ottimizzazione: FAST e FULL.

L'ottimizzazione FAST prevede la risoluzione soltanto del problema della frammentazione e viene impostata attraverso l'istruzione

```
alter index myindex rebuild online parameters('optimize
fast');
```

L'ottimizzazione FULL esegue sia la deframmentazione che la garbage collection. Viene impostata con l'istruzione

```
alter index myindex rebuild online parameters('optimize
full');
```

Essendo un'operazione molto dispendiosa dal punto di vista delle risorse utilizzate e del tempo di esecuzione, l'ottimizzazione FULL, a differenza della FAST, non deve essere eseguita in una sola volta. Attraverso il parametro `maxtime`, con l'istruzione

```
. . . parameters('optimize full maxtime 5');
```

si può indicare che l'esecuzione abbia durata di cinque minuti. La successiva occasione in cui verrà avviata, l'ottimizzazione riprenderà dal punto in cui si è interrotta.

Capitolo 5

ANALISI DI FATTIBILITÀ DI POSSIBILI SOLUZIONI PER LA RICERCA DI APPROXIMATE MATCH TESTUALI

Riprendendo il discorso avviato nel capitolo 2, in questo capitolo analizziamo possibili alternative individuate per eseguire le ricerche con la metodologia "approximate match". L'analisi verrà svolta per il sistema di pre-traduzione che considera come lingua di partenza l'inglese. Questo tipo di traduzioni, infatti, è il più importante in quanto copre circa l'80% del totale delle traduzioni eseguite in Logos.

5.1 La struttura della banca dati testuale

Prima di affrontare qualsiasi tipo di analisi riprendiamo la descrizione della composizione della tabella TERMS (ved. § 2.2) utilizzata come banca dati testuale nella fase di pre-traduzione. La struttura di questa tabella è costituita dai seguenti campi:

- Codice identificativo della frase;
- Codice identificativo del cliente per il quale la frase è stata tradotta;
- Codice del settore a cui si riferisce la frase;
- Frase già tradotta in precedenza (una colonna per ogni lingua conosciuta).

Nella trattazione che segue considereremo un prototipo di tabella di questo tipo che chiameremo, comunque, TERMS. Chiameremo **Term_en** la colonna che conterrà la frase in inglese.

5.2 Utilizzo delle funzionalità di Oracle 8i – *inter*Media

Come accennato nel capitolo 4, *inter*Media Text offre la possibilità di

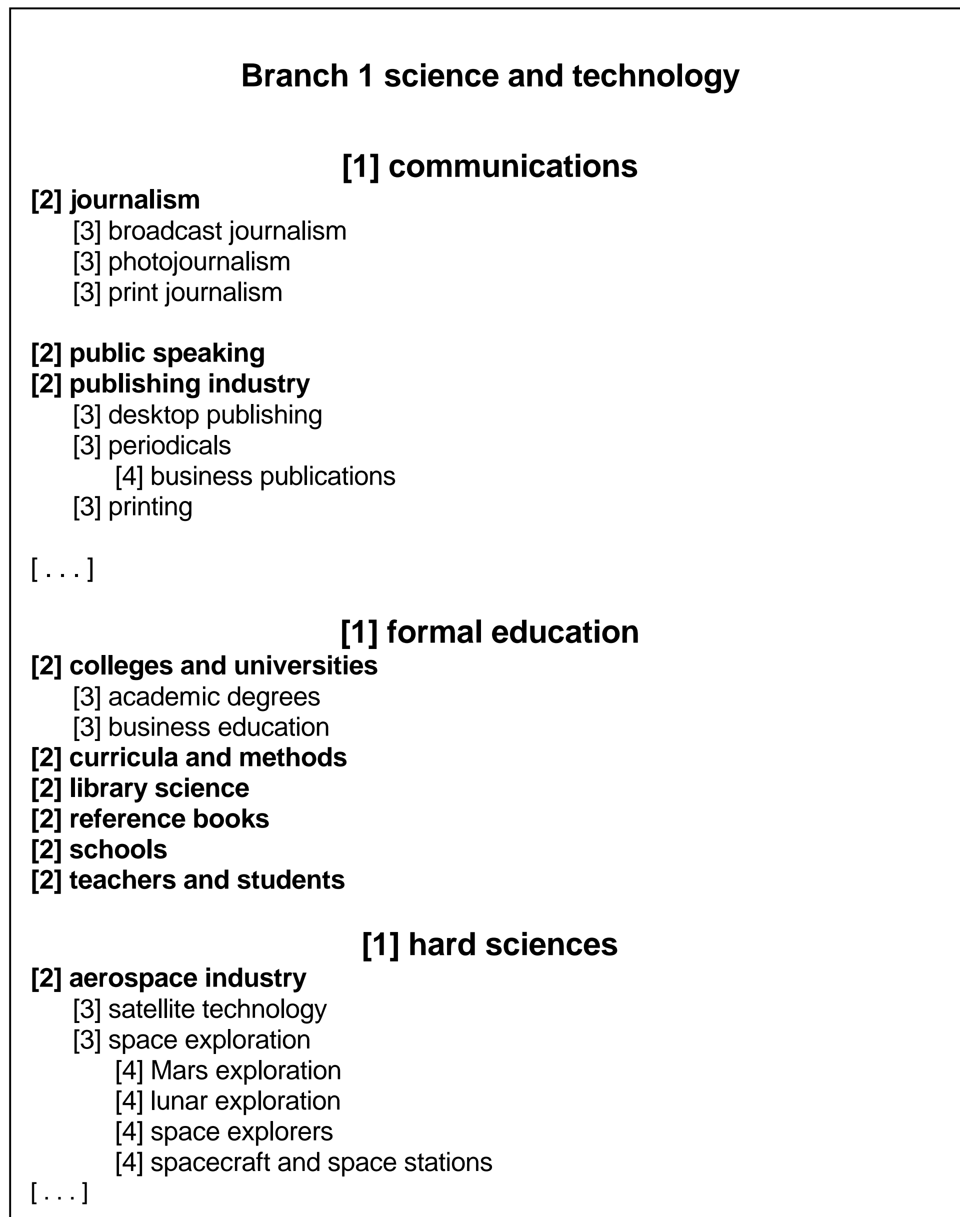


Fig. 5.1. Estratto della knowledge base interna di interMedia

costruire, con l'ausilio di una consistente knowledge base interna (ved. Fig. 5.1), un indice di tipo concept index (ved. § 3.2.1.3). Utilizzando questo tipo di indicizzazione della banca dati testuale, è possibile eseguire ricerche *concept-based* attraverso l'operatore `about`. Questo operatore esegue l'estrazione degli argomenti (themes) dal testo contenuto nell'interrogazione per poi cercarne le corrispondenze negli argomenti (themes) presenti nell'indice. In questo modo viene eseguita l'espansione dell'interrogazione [10].

5.2.1 Considerazioni sull'applicazione della ricerca "*about*" per risolvere il problema

Viste le potenzialità del prodotto *inter*Media, descritte nel capitolo 4, la prima ipotesi presa in esame per risolvere il problema di approximate match testuale è stata quella di cercare di sfruttare esclusivamente questo prodotto (si abbandonerebbe la generazione e la ricerca dei trigrammi descritta nel paragrafo 2.3.1). In particolare abbiamo cercato di capire se la creazione di un indice di tipo *concept* sulla nostra banca dati testuale e l'utilizzo dell'operatore `about` fossero soluzioni appropriate per il nostro problema. Visto che l'obiettivo della ricerca svolta è quello di reperire frasi simili per significato, il primo pensiero è stato quello di eseguire interrogazioni del tipo descritto nel paragrafo 4.2.3.2.

Già dai primi test, però, ci siamo accorti di come questa strada non fosse percorribile. Vediamo ora di spiegarne il motivo.

L'applicazione di tecniche di Information Retrieval alla fase di pre-traduzione deve comunque tenere conto delle finalità e del contesto in cui essa viene eseguita. La traduzione di testi non prevede solo una ricerca di documenti che possono essere legati, in qualche modo, dal fatto che trattano argomenti analoghi. Nella creazione del concept index, in base alla knowledge base, vengono generati themes troppo generali rispetto alle nostre necessità. Per

questo motivo eseguendo una ricerca "*about*" con *inter*Media l'espansione dell'interrogazione diventa eccessiva e comunque non appropriata rispetto alle esigenze che si hanno quando si sta eseguendo una traduzione. Per capire meglio vediamo il seguente esempio (ved. Fig. 5.2). Qualora si stia ricercando la parola "cats" si vuole che nel risultato dell'interrogazione compaiano frasi contenenti le parole "cats" e "cat". Questo poi permetterà di risalire a frasi già presenti nella base di dati, quindi già tradotte, che contengono le parole cercate. Una ricerca attraverso l'operatore `about` all'interno dei themes generati da *inter*Media potrebbe portare come risultati, per esempio, frasi contenenti "tiger" e "tigers". Questo accade perché le parole cat e tiger sono entrambe catalogate all'interno della categoria "felines" (felini). Se per un utente che esegue ricerche su una banca dati come un'enciclopedia digitale questo può essere un fattore positivo, per un traduttore evidentemente non lo è. Per eseguire la traduzione di "cat" (gatto) è inutile e dannoso reperire le frasi che contengono "tiger" (tigre).

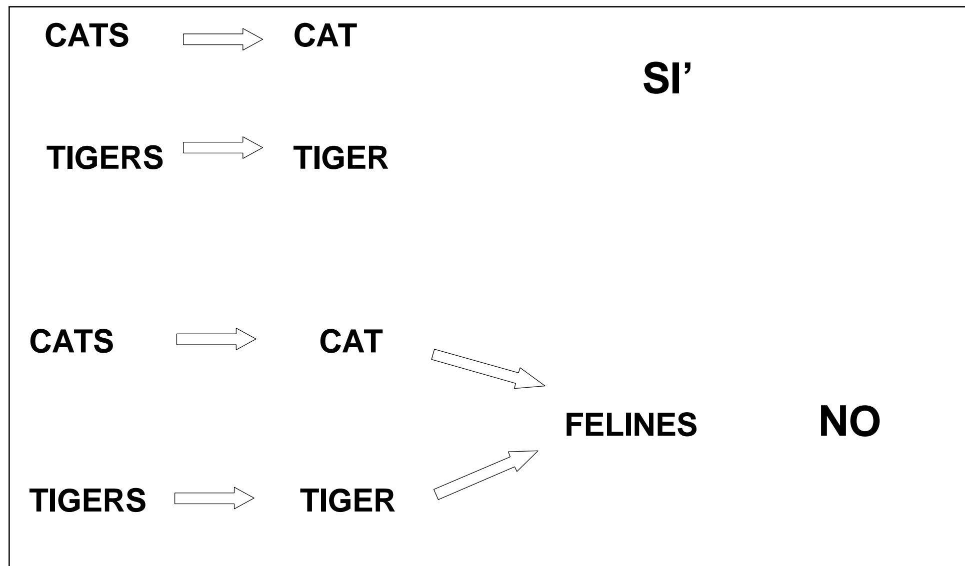


Fig. 5.2. Modalità di normalizzazione di termini

5.3 Analisi di un procedimento ad hoc per la ricerca in modalità approximate match di frasi

Il procedimento analizzato ora si basa sul concetto che due frasi simili dal punto di vista della traduzione presentano lo stesso “scheletro”. Per scheletro di una frase si intende il risultato di un’operazione di ***normalizzazione*** eseguita sulla frase stessa. Questa normalizzazione prevede l’eliminazione delle *noise words* (ved. 3.2.1.1), della punteggiatura, di eventuali codici (tag, parentesi, ecc.) e la ricerca della radice di ogni parola che compone la frase (quest’ultima operazione riguarda, ovviamente, le parole diverse dalle noise words). Secondo questo modo di procedere più frasi diverse potranno avere lo stesso scheletro. Il confronto, allora, nella ricerca di frasi simili, verrà eseguito non sulle frasi originali ma sui rispettivi scheletri. Questo ragionamento ci porta ad affermare che se due frasi hanno lo stesso scheletro è molto probabile che esse possano avere un significato molto simile. Vediamo un esempio:

frase 1: *this is a ball*

frase 2: *these are two balls*

Le due frasi, dal punto di vista della pre-traduzione, sono identiche. Il risultato di una normalizzazione sarà infatti, sia per la frase 1 che per la 2, il seguente: *be ball*.

Nell'operazione svolta sono state eliminate le noise words "*this*", "*a*", "*these*", "*two*" e sono stati ricondotti all'infinito i predicati verbali "*is*" e "*are*".

Possiamo scomporre il procedimento di ricerca di approximate match in due fasi principali.

La prima fase non fa parte effettivamente del processo di ricerca ma consiste nella preparazione della banca dati testuale su cui eseguire le ricerche. Viene applicato un algoritmo di normalizzazione ad ogni frase inglese presente sulla tabella TERMS (ved. Fig. 5.3). Si è pensato di aggiungere un campo ad ogni riga della tabella. In questo spazio vengono inseriti, riga per riga, i risultati della normalizzazione in modo tale da possedere, su una stessa struttura, sia i dati originali che i rispettivi dati normalizzati.

Una volta completata questa prima normalizzazione si provvede a creare una ***struttura indice*** sulla colonna contenente le frasi normalizzate. La struttura creata sarà un inverted index e sarà costruita per mezzo di *inter*Media Text.

Esaurita questa prima fase si avrà pronta la banca dati testuale su cui ci si baserà per cercare di soddisfare la ricerca per approximate match.

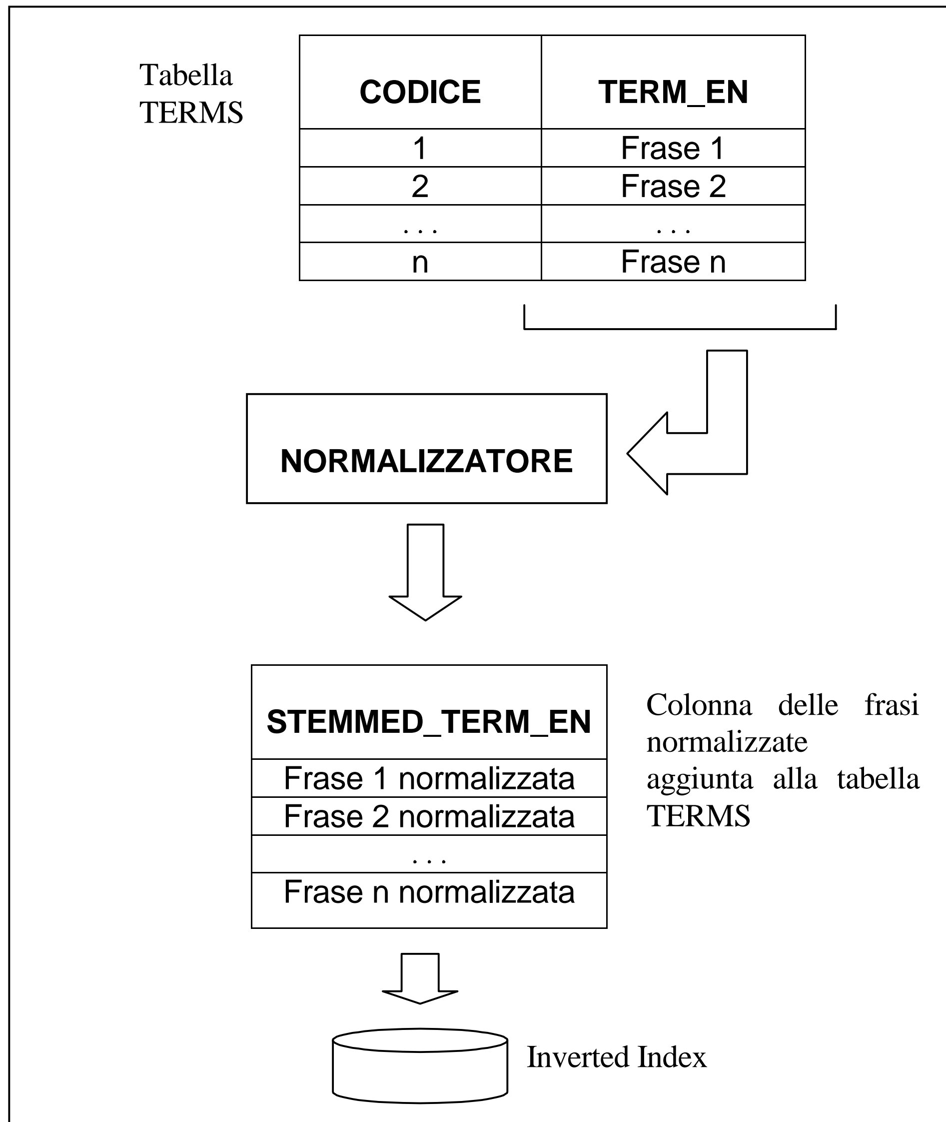


Fig. 5.3. Prima fase: preparazione della colonna normalizzata

Vediamo ora la seconda fase. Al momento dell'interrogazione, la frase introdotta dall'utente verrà presa in esame dallo stesso algoritmo di normalizzazione descritto prima e ne verrà costruita una versione normalizzata temporanea. Questa versione verrà utilizzata per le ricerche nell'indice.

A questo punto si introduce il procedimento attraverso il quale viene manipolata la versione temporanea per effettuare la ricerca e il relativo ranking

dei risultati. Si è pensato di riprendere la scomposizione in trigrammi presentata nel capitolo 2 e di applicarla alla versione temporanea dell'interrogazione. Gli N-grammi, ricordiamo, sono sequenze di N parole consecutive all'interno della frase. Essi sono generati facendo "scorrere" sulla frase una finestra che contenga N parole (ved. Fig. 5.4).

Ogni N-gramma sarà ricercato sulla colonna delle frasi "scheletro" della tabella TERMS (STEMMED_TERM_EN).

A questo punto si deve assegnare ad ogni frase risultato della ricerca un valore proporzionale alla vicinanza in significato con la frase oggetto di traduzione.

Un modo per implementare questo tipo di valutazione sarà descritto nel prossimo capitolo insieme alla progettazione concreta dei componenti del procedimento.

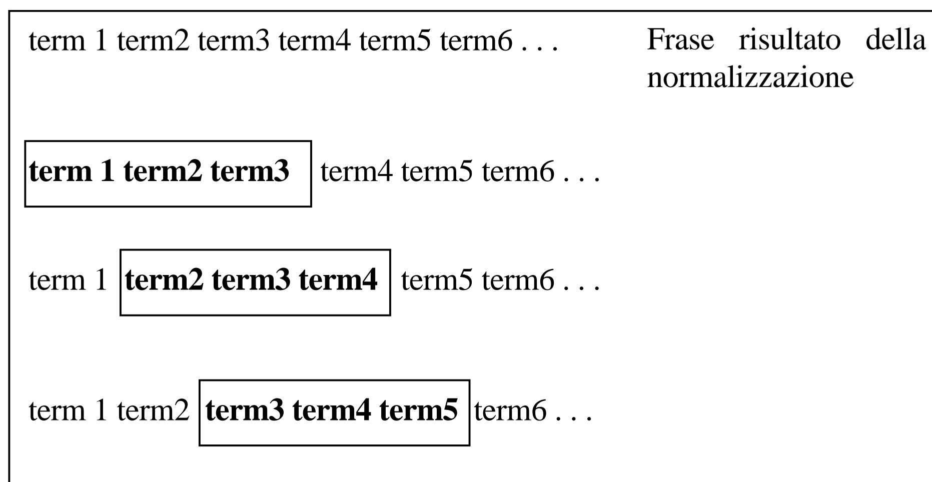


Fig. 5.4. Generazione degli N-Grammi (in questo caso trigrammi)

A questo punto si deve sviluppare un metodo per eseguire un ranking dei risultati della ricerca per approximate match.

Si è pensato di svolgere, per ogni frase del documento da tradurre, le operazioni seguenti:

1. sequenza di interrogazioni aventi per oggetto ogni singolo trigramma. Si ricavano così, per ogni trigramma, gli insiemi degli dentificativi delle frasi che lo contengono (chiamiamo questi insiemi $Q_1, \dots , Q_n$);
2. ognuno di questi insiemi risultato viene inserito in una operazione di unione con gli altri insiemi ottenuti rispettivamente per ogni trigramma. Si ottiene un insieme che chiamiamo Q;
3. da questo insieme Q risultato dell'unione, si estraggono gli identificativi delle frasi che hanno numerosità maggiore. Per una maggiore comprensione, questa operazione di ranking si può ricondurre ad una istruzione SQL di questo tipo:

```
select CODICE, COUNT(*)
from Q
group by CODICE
order by 2
stop after 10;
```

In questo modo avremo, come risultato, le dieci frasi che contengono il maggior numero di trigrammi che compongono la frase da tradurre. Secondo le impostazioni date inizialmente al problema avremo reperito le frasi il cui significato maggiormente si avvicina a quello della frase da tradurre.

Per fornire al traduttore un risultato utile e comprensibile in termini di vicinanza di significato tra le frasi si deve impostare una funzione che traduca questa vicinanza in un fattore numerico (ad esempio un valore compreso nell'intervallo [0,1]). Il traduttore, infatti, non considererà significativi i valori sotto una certa soglia.

In altre parole, si determinerà un valore che indichi in che misura una certa frase presente all'interno della banca dati testuale faccia parte del risultato della ricerca approssimata. Questo valore non potrà essere quindi un classico

valore booleano che indichi la appartenenza o la non appartenenza ad un insieme; la misura in questione si presta, invece, ad essere valutata facendo uso dei concetti e degli strumenti tipici della logica Fuzzy[8][12,13].

Nella *Fuzzy Logic*, infatti, la funzione che si vuole implementare è definita in questo modo: dato un insieme **I** e un suo sottoinsieme **W**, può essere definita una funzione $\mathbf{f_w: I \rightarrow [0,1]}$ tale che per ogni $\mathbf{e \in I, f_w(e) \in [0,1]}$ e rappresenta il grado di appartenenza di **e** a **W**. Il valore ZERO indica il completo NON-MEMBERSHIP, il valore UNO indica completo MEMBERSHIP e i valori nel mezzo sono usati per rappresentare valori intermedi con intermedi gradi di MEMBERSHIP. L'insieme I è detto UNIVERSE OF DISCOURSE la mappa ottenuta è detta MEMBERSHIP FUNCTION [12].

Il grado con cui la frase: "***x è in W***" è vera è determinato dal valore assunto da $\mathbf{f_w(x)}$.

---

[8] La fuzzy logic é un modo alternativo d'intendere la realtà: non esistono soltanto il vero o il falso ma vi é anche una verità di mezzo, una verità "sfumata" (traduzione letterale di fuzzy). Più precisamente ci si basa sul ragionamento con insiemi "fuzzy" o con insiemi di regole fuzzy. Questo significato risale alla prima ricerca sugli insiemi fuzzy negli anni 60 e 70 a opera di Lotfi Zadeh dell'UCB. Zadeh scelse l'aggettivo "fuzzy" nel suo saggio del 1965 intitolato appunto "Fuzzy Sets" [14]. Altri sinonimi: logica grigia, nebulosa o continua. Questa definizione viene data in contrapposizione ai cosiddetti insiemi "crisp" (netto, preciso) sui quali si basa la tradizionale "*logica binaria*".

Capitolo 6

PROGETTAZIONE DEL PROCEDIMENTO SCELTO PER LA RICERCA DI APPROXIMATE MATCH TESTUALI

In questo capitolo verrà implementato il procedimento descritto nel paragrafo 5.3. Si descriverà prima l'analisi e la stesura di un programma che esegue la normalizzazione di frasi in lingua inglese. Successivamente vedremo il procedimento di indicizzazione della colonna aggiunta alla tabella TERMS contenente le frasi già tradotte. Infine verrà presentato un metodo di classificazione dei risultati della ricerca.

6.1 Il programma di normalizzazione

Il programma per la normalizzazione ha come fulcro un algoritmo che esegue lo stemming dei termini di una frase generica in lingua inglese. Questo programma restituisce una frase "depurata" dalle noise words, da caratteri di punteggiatura e codici. Ogni parola significativa della frase normalizzata sarà quindi il risultato dell'applicazione dell'algoritmo di stemming.

6.1.1 Che cosa è un algoritmo di stemming

Un algoritmo di stemming (detto anche stemmer) è un processo che esegue la normalizzazione linguistica di parole. Questo significa che, attraverso di esso, le varianti di una stessa parola sono ridotte alla radice comune. Si avrà, per

esempio, la riduzione di nomi plurali ai rispettivi singolari o la determinazione dell'infinito delle forme coniugate dei verbi.

Per la maggior parte dei linguaggi occidentali, e questo vale anche per la lingua inglese, le parole tendono a rimanere costanti all'inizio e a variare nella parte finale. La rimozione, o la sostituzione, di queste parti finali (o suffissi) è proprio ciò che si definisce "stemming".

Nel corso degli anni molte strategie per la rimozione di suffissi sono state studiate [17 – 22]. La scelta di una tecnica dipende dall'ambito di utilizzo nel quale questa operazione si svolge. Solitamente, comunque, un programma di rimozione dei suffissi (stemmer) contiene una lista di suffissi analizzati e, per ognuno di essi, il criterio in base al quale esso può essere rimosso o sostituito.

Come accennato, l'operazione di stemming deve tenere conto dell'ambito in cui essa viene svolta; in particolare si può vedere come componente di processi più complessi, svolgendo funzioni come le seguenti:

- ausilio nell'espansione di una interrogazione. Esso, utilizzato in modo inverso alla maniera canonica, fornisce, per arrichire l'interrogazione, quei termini che hanno la stessa radice di quelli immessi dall'utente. Da questo punto di vista si può assimilare ad una semplice funzione di thesaurus [15];
- ausilio nella normalizzazione di una interrogazione poiché permette l'indicizzazione di più termini attraverso la radice comune.

Dal punto di vista della valutazione di prestazioni di un I.R.S. (ved. § 3.2.2) possiamo dire che l'operazione di stemming permette l'incremento del recall (richiamo) in quanto riduce le parole ad una comune radice morfologica. D'altra parte, però, lo stemming tende a ridurre la precision (precisione) qualora non si ponga attenzione al rischio di ridurre ad una radice comune termini che non sono correlati in significato.

6.1.2 Un modello di stemmer presente in letteratura

Come abbiamo detto, diversi algoritmi sono stati pubblicati. A noi, in questa sede, interessano soltanto i procedimenti che trattano i termini in lingua inglese. In particolare descriveremo brevemente l'algoritmo più conosciuto: il Porter [16]. Il *Porter Stemmer* analizza e rimuove circa 60 suffissi. L'approccio usato si basa sull'applicazione di più passi consecutivi. Ognuno di essi valuta la rimozione di suffissi.

Senza entrare nei particolari, il difetto principale che abbiamo riscontrato riguarda il fatto che non viene posta attenzione sul significato dei risultati restituiti. L'obiettivo di questo algoritmo non è quello di restituire una vera radice linguistica, piuttosto quello di incrementare le prestazioni in velocità. Comunque, l'applicazione di regole molto generiche e l'assenza di un controllo sul significato, può portare ad errori (normalizzazioni errate che non dovrebbero essere fatte o omissioni di normalizzazioni). Esempi di questi tipi di errori sono riportati nella tabella 6.1.

Normalizzazioni errate	**Omissioni**
Organization/Organ	European/Europe
Doing/Doe	Matrices/Matrix
Generalization/Generic	Urgency/Urgent
Numerical/Numerous	Create/Creation
Policy/Police	Decompose/Decomposition
University/Universe	Useful/Usefully
Addition/Additive	Route/Routed
Execute/Executive	Search/Searcher
Past/Paste	Explain/Explanation
Special/Specialized	Resolve/Resolution
Arm/Army	Machine/Machinery
Head/Heading	Triangle/Triangular

Tab. 6.1. Esempi di errori commessi dal Porter Stemmer

6.1.3 Analisi ad oggetti del processo di normalizzazione

Poiché per noi il significato dei risultati dell'operazione di stemming è importante realizzeremo appositamente un programma che ponga attenzione anche al risultato della normalizzazione. Per ottenere questo si svilupperà, all'interno del procedimento di normalizzazione, un algoritmo di stemming che sia "dictionary-assisted". Lo stemmer che vogliamo realizzare svolgerà essenzialmente due fasi: la prima contempla la vera e propria rimozione o sostituizione di suffissi (ad esempio *implementation› implement, application› apply)*, nella seconda, invece, viene verificata, attraverso un dizionario[9] che implementeremo con una tabella nella nostra base di dati (ved. § 6.1.3.1.5), l'effettiva esistenza del nuovo termine creato (riprendendo l'esempio precedente si verifica se il termine *implement* è presente sul dizionario).

Individuiamo, prima di tutto, le categorie di parole che possono venire manipolate dall'algoritmo. Ogni categoria verrà gestita in maniera diversa.

1. **Noise words**: i termini come articoli, preposizioni, numeri devono essere eliminati.
2. **Verbi coniugati**: i verbi, come detto nel capitolo precedente devono essere ricondotti al rispettivo infinito.
3. **Eccezioni**: sono le parole di cui si vuole evitare la normalizzazione (ad esempio nomi propri di persona che terminano con suffissi che possono essere erroneamente rimossi o sostituiti).
4. **Sostantivi Plurali**: i plurali saranno gestiti attraverso regole riconducibili a ciò che viene definito *inflectional stemming*[10].

---

[9] Il dizionario utilizzato è stato preparato dall'Interociter Bulletin di Dallas ed è stato scaricato dalla rete Internet al sito http://www.umich.edu/~archive/linguistics/texts/lexica.

[10] Da *inflectional morphology*. Quest'area della linguistica descrive quei cambiamenti che subisce un termine dopo l'applicazione di regole di sintassi che lo rendono plurale o in forma possessiva (genitivo

5. **Avverbi e parole derivate**: queste parole vengono trattate da regole di *derivational stemming*[11].

Nei prossimi paragrafi si descrive, con approccio ad oggetti, la progettazione del sistema che si occupa della normalizzazione delle frasi. Per la schematizzazione dei concetti sono stati utilizzati i formalismi propri dello strumento OMT (Object Modeling Technique) [23]. Si sono costruiti, pertanto, il modello statico, il modello dinamico ed il modello funzionale.

6.1.3.1 Modello statico

Il modello realizzato contiene le classi seguenti:

- **Regola**: classe creata per definire la struttura di ogni regola applicabile a seconda del suffisso della parola analizzata.
- **Stemming**: classe che gestisce lo stemming dei termini (inflectional o derivational).
- **RicercaInDbOracle**: classe attraverso la quale vengono eseguite la connessione e le interrogazioni delle tabelle che abbiamo creato per gestire i predicati verbali, le noise words, le eccezioni.
- **Filtri**: classe che contiene i metodi per eliminare i caratteri (come, per esempio, la punteggiatura) che non ha senso considerare nell'operazione di normalizzazione.

6.1.3.1.1 La classe Regola

Ogni istanza di questa classe definisce la modalità con cui viene gestito il suffisso **suffix**.

---

sassone). Queste regole non mutano il significato grammaticale di un termine (un sostantivo, al plurale, rimane un sostantivo).

[11] Da *derivational morphology*. A differenza dell'area precedente, questa descrive quelle regole morfologiche che, applicate ad una parola, ne cambiano il significato grammaticale (ad esempio in inglese abbiamo la radice *friend* (amico) e la derivazione *friend**ship*** (amicizia)).

Descriviamo gli attributi:

- **suffix**: è una stringa che contiene il suffisso gestito dalla istanza relativa;

Regola
String **suffix;** int **order;** int **level;** String **suffix1;** String **suffix2;**
Regola(String, int, int, String, String**); //**costruttore String **getSuffix( );** int **getOrder( );** int **getLevel( );** String **getSuffix1( );** String **getSuffix2( );**

Fig. 6.1. Struttura statica della classe Regola

- **order**: numero intero che stabilisce un ordine di importanza della regola;
- **level**: numero intero attraverso il quale si stabilisce in che modo viene trattato il suffisso della parola trattata;
- **suffix1, suffix2**: stringhe che contengono possibili suffissi da inserire al posto di **suffix** nella parola trattata dalla regola;

I valori degli attributi delle istanze della classe Regola vengono gestiti attraverso il metodo **applicaRegole** della classe Stemming (ved. § 6.1.3.1.2).

I metodi di questa classe svolgono funzioni abbastanza intuitive e servono per estrarre i valori degli attributi di ogni singola istanza di Regola.

6.1.3.1.2 La classe Stemming

Si è pensato di creare una generalizzazione che lega la classe astratta

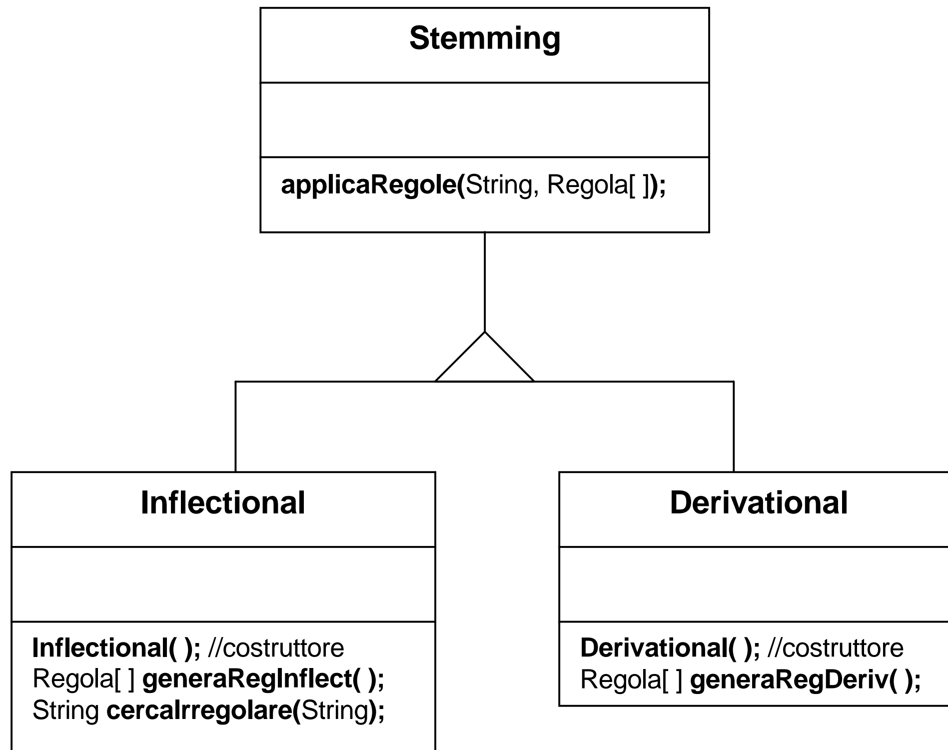


Fig. 6.2. Generalizzazione della classe astratta Stemming

Stemming a due sottoclassi che gestiscono rispettivamente le operazioni di **inflectional** e **derivational** stemming. Questa generalizzazione si rappresenta graficamente con il diagramma riportato in Fig. 6.2.

Descriviamo ora il metodo della classe astratta **Stemming**:

- **applicaRegole**(String, Regola[ ]): questo metodo ha come parametri di ingresso la **parola** da tradurre e il vettore **regole** di istanze di Regola creato da generaRegInflect o da generaRegDeriv (ved. sottoclassi); gli elementi del vettore contengono pertanto i suffissi significativi da

considerare. Viene confrontata la parte finale della parola con i valori di suffix di ogni istanza di Regola contenuta nel vettore. Qualora venga individuata una regola che contiene il suffisso corrispondente alla parte finale della parola, vengono estratti i relativi valori degli attributi e in base ad essi viene operata la rimozione o sostituzione del suffisso.

In particolare, in base al valore assunto dall'attributo level della classe Regola, viene svolta una delle seguenti operazioni:

level = 1 il suffisso della parola deve essere semplicemente rimosso e non sostituito con altro. Vengono restituiti due risultati: il primo viene generato rimuovendo l'ultimo carattere della parola, il secondo viene generato rimuovendo gli ultimi due caratteri;

level = 2 il suffisso della parola deve essere rimosso e sostituito con la stringa contenuta in suffix1;

level = 3 vengono restituiti due risultati: per produrre il primo viene rimosso il suffisso e sostituito con la stringa contenuta in suffix1; per il secondo viene solo rimosso il suffisso;

level = 4 anche in questo caso vengono restituiti due risultati: il primo viene prodotto sostituendo il suffisso con la stringa suffix1; il secondo viene prodotto sostituendo il suffisso con la stringa suffix2;

level = 5 questo valore di level indica che una parola che ha parte finale uguale a suffix deve essere lasciata invariata.

Analizziamo ora la sottoclasse **Inflectional**. Essa contiene i seguenti metodi:

- **Inflectional( )**: è il costruttore.
- **generaRegInflect( )**: questo metodo crea il vettore di istanze della classe Regola da utilizzare per valutare se la parola può essere soggetta a inflectional stemming.

- **CercaIrregolare**(String): questo metodo valuta se la parola avuta come parametro di ingresso è un plurale irregolare. In caso affermativo restituisce il relativo singolare.

La classe **Derivational** contiene i seguenti metodi:

- **Derivational**( ): è il costruttore.
- **generaRegDeriv**( ): questo metodo crea il vettore di istanze della classe Regola da utilizzare per valutare se la parola può essere soggetta a derivational stemming.

6.1.3.1.3 La classe RicercaInDbOracle

Questa classe, come detto, contiene i metodi per aprire la connessione e per

RicercaInDbOracle
RicercaInDbOracle( ); //costruttore String **cercaVerbo(**String**);** String **cercaInDizionario(**String**);** String **cercaInStoplist(**String**);** String **cercaInEccezioni(**String**);** String **cercaInAbbreviazioni(**String**);** void **chiudiConnessione( );** void **inserisci(**String, int**);**

Fig. 6.3. Struttura statica della classe RicercaInDbOracle

eseguire le interrogazioni sul database Oracle che contiene le tabelle dei verbi, delle noise words, delle eccezioni, delle abbreviazioni e del dizionario. Gli strumenti per mezzo dei quali vengono svolte queste operazioni sono i protocolli JDBC e l'interfaccia SQLJ; essi sono descritti in appendice B (§ B.8). La struttura statica della classe RicercaInDbOracle è riportata in Fig. 6.3.

Analizziamo i metodi della classe:

- **RicercaInDbOracle(**): è il costruttore. Questo metodo apre una connessione con il database Oracle.
- **cercaVerbo(**String): questo metodo verifica, attraverso una interrogazione sulla tabella VERBI_EN contenente i verbi inglesi, se la parola avuta come parametro in ingresso è un predicato verbale. Se ciò è vero, la parola verrà sostituita con il rispettivo infinito.
- **cercaInDizionario(**String): questo metodo verifica, attraverso la tabella WORDENGLISH, se il risultato dell'operazione di stemming, avuto come parametro in ingresso, è una parola inglese esistente.
- **cercaInStoplist(**String): questo metodo verifica, attraverso la tabella STOPLIST, se la parola avuta come parametro in ingresso è una noise word e deve essere eliminata.
- **cercaInEccezioni(**String): questo metodo verifica, attraverso la tabella ECCEZIONI, se la parola avuta come parametro in ingresso è un'eccezione e non deve essere normalizzata.
- **cercaInAbbreviazioni(**String): questo metodo verifica, attraverso la tabella ABBREV_EN, se la parola inserita come parametro di ingresso è una forma abbreviata di un verbo (it's, he's, he'll, ecc.). In caso affermativo essa viene sostituita con l'infinito del verbo stesso.
- **chiudiConnessione(**): questo metodo chiude la connessione al database Oracle aperta attraverso il metodo costruttore RicercaDbOracle( ).
- **inserisci(**String, int): questo metodo inserisce la frase normalizzata (stringa in ingresso) nella riga corrispondente al codice (int) avuto in ingresso. Viene così riempita la colonna STEMMED_TERM_EN aggiunta alla tabella TERMS che contiene le frasi normalizzate (ved. Fig. 5.3).

Le strutture delle tabelle utilizzate dai metodi di questa classe sono descritte nel paragrafo 6.1.3.1.5.

6.1.3.1.4 La classe Filtri

Come accennato, la classe Filtri contiene i metodi statici che manipolano una frase per renderla analizzabile dall'algoritmo di normalizzazione. La struttura statica della classe Filtri è riportata in Fig. 6.4.

Filtri
String **filtraCodice(**String**);** boolean **alfanumerica(**String**);** String **pulisci(**String**);** StringTokenizer **segmenta(**String**);**

Fig. 6.4. Struttura statica della classe Filtri

I metodi della classe sono:

- **filtraCodice**(String): questo metodo elimina dalla frase avuta come parametro in ingresso i codici di posizione (ved. § 2.1). Restituisce la frase "pulita" da questi codici.
- **alfanumerica**(String): questo metodo prende in esame una parola e determina se contiene al suo interno caratteri numerici. Restituisce un valore booleano che indica se la parola analizzata è alfanumerica e deve pertanto essere eliminata in quanto poco significativa per l'indicizzazione.
- **pulisci**(String): questo metodo elimina la punteggiatura dalla frase da normalizzare. Restituisce la frase filtrata.
- **segmenta**(String): questo metodo applica la segmentazione in termini (tokens) della frase che riceve come parametro di ingresso. Come carattere

delimitatore dei tokens si utilizza lo spazio. Restituisce la frase suddivisa in tokens.

6.1.3.1.5 Strutture delle tabelle utilizzate per la normalizzazione

Tabella **VERBI_EN**

Contiene, per ogni verbo, infinito, simple present (terza persona singolare), simple past, past continous, present continous.

- ID: valore intero identificativo del verbo
- VERB: forma verbale
- IDC: valore intero che specifica che forma corrisponde all'attributo VERB. I valori possibili sono: "0" per l'infinito, "1" per il simple present, "2" per il simple past, "3" per il past continous, "4" per il present continous.

Tabella **WORDENGLISH**

Contiene una raccolta di parole inglesi che utilizziamo come dizionario per verificare se un termine risultato dalla normalizzazione esiste.

- WORD: campo che contiene la parola.

Tabella **STOPLIST**

Contiene le parole che sono da considerare di scarso significato.

- WORD: noise word.

Tabella **ECCEZIONI**

Contiene i termini che non si vogliono normalizzare.

- WORD: parola da non normalizzare

Tabella **ABBREV_EN**

Contiene le forme abbreviate dei verbi (ad esempio, it's, he's, he'll) che devono essere ricondotte all'infinito del relativo verbo.

- ESPRESSIONE: forma abbreviata del predicato verbale.
- VERB: infinito nel quale il contenuto di ESPRESSIONE deve essere trasposto.

6.1.3.2 Modello dinamico

L'unica classe che si presta ad essere modellata dinamicamente è la classe REGOLA. Gli eventi e gli stati da descrivere con il modello dinamico riguardano la creazione del vettore di oggetti REGOLA utilizzato per valutare che tipo di operazione di stemming deve essere eseguita (ved. metodo

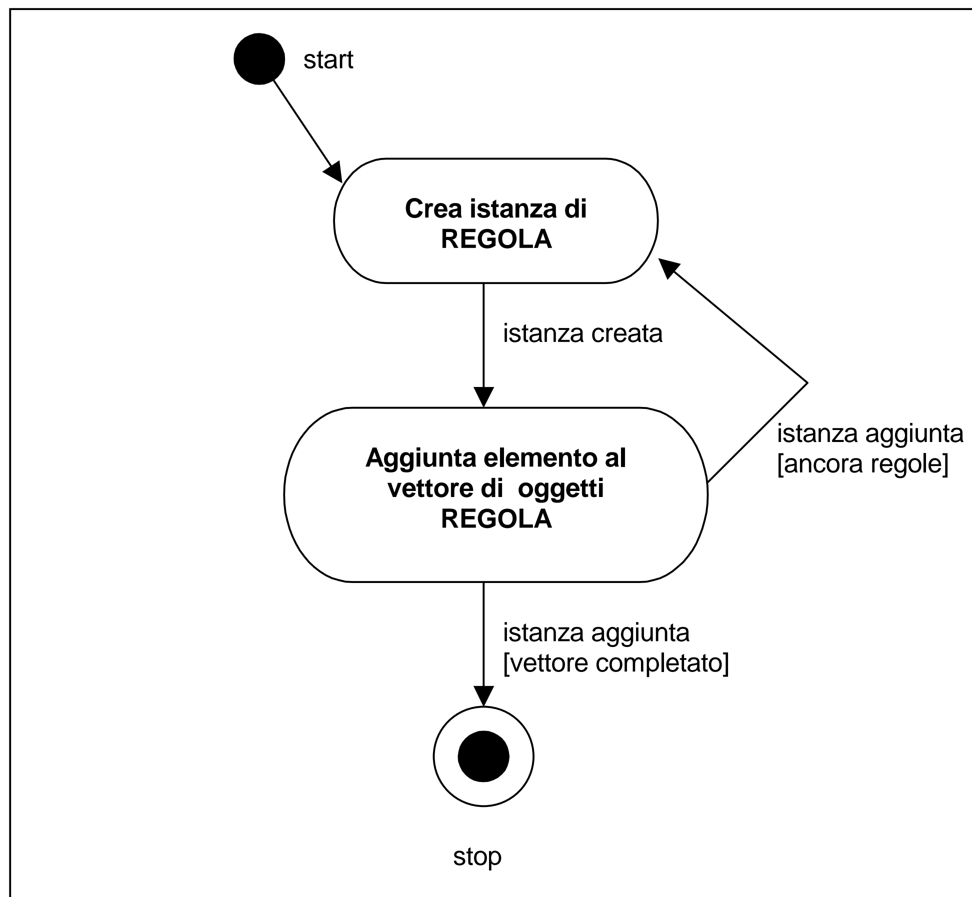


Fig. 6.5. Comportamento dinamico della classe Regola

applicaRegole in § 6.1.3.1.2). Il diagramma che descrive il comportamento dinamico della classe REGOLA è riportato in Fig. 6.5.

6.1.3.3 Il modello funzionale

Le funzioni svolte dal sistema di normalizzazione sono due:

- Normalizzazione della frase contenuta nel campo TERM_EN della tabella TERMS e conseguente inserimento del risultato nel campo STEMMED_TERM_EN (Fig. 6.6).

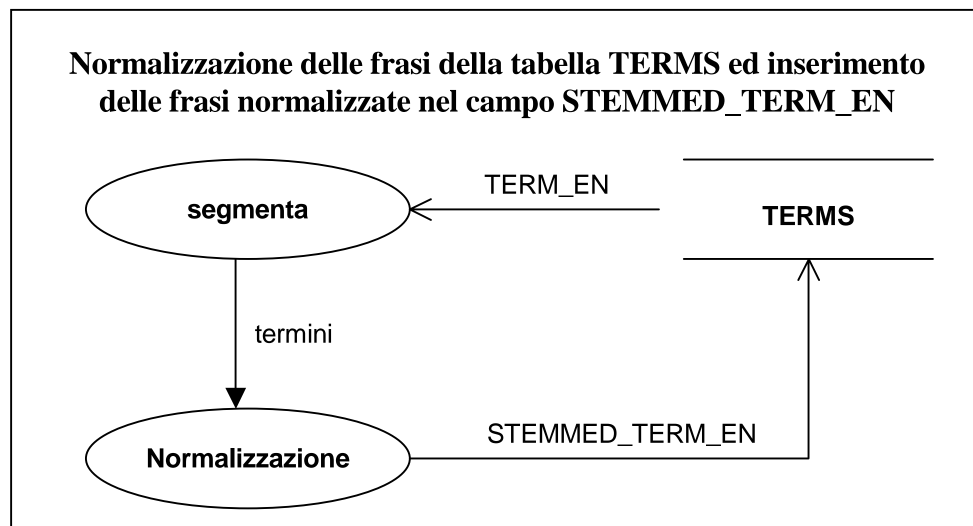


Fig. 6.6. Modello funzionale (livello 0)

- Normalizzazione della frase estratta dal documento da tradurre e creazione della versione normalizzata temporanea che sarà utilizzata per generare i trigrammi da ricercare nella colonna STEMMED_TERM_EN (Fig. 6.7).

Le principali differenze riguardano le strutture dalle quali sono estratte le frasi da elaborare e la gestione dei risultati dell'operazione di normalizzazione. In entrambi i casi, comunque, la logica di funzionamento del processo di normalizzazione è la medesima. I diagrammi che compongono il modello funzionale sono riportati in Fig. 6.6 ed in Fig. 6.7.

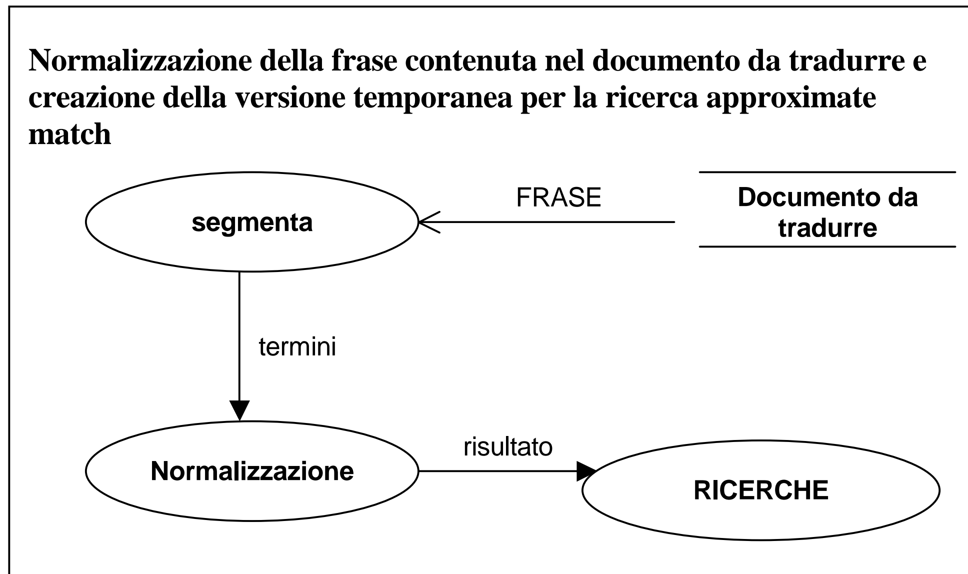


Fig. 6.7. Modello funzionale (livello 0)

In Fig. 6.7 abbiamo indicato, con il processo RICERCHE, il fatto che la versione temporanea della frase normalizzata viene utilizzata per eseguire la generazione dei trigrammi e la relativa ricerca nella colonna STEMMED_TERM_EN della tabella TERMS (ved. § 6.3).

In entrambi i diagrammi il processo "Normalizzazione" si espande ad un livello superiore che riportiamo in Fig. 6.8. A sua volta il processo "Normalizzazione" contiene due processi che vengono espansi al livello 2 e che sono riportati in Fig. 6.9 (RicercaInDatabase) e in Fig. 6.10 (Stemmer).

Il termine estratto dalla frase viene dapprima preso in esame dal metodo **pulisci** della classe Filtri che elimina i caratteri di punteggiatura. A questo punto inizia il ciclo di controlli sul termine stesso per determinarne la natura e per determinare quale eventuale operazione effettuare su di esso. La descrizione dei metodi utilizzati nel modello funzionale è stata data nel paragrafo 6.1.3.1. La relativa codifica in linguaggio Java viene, invece, riportata nell'appendice A.

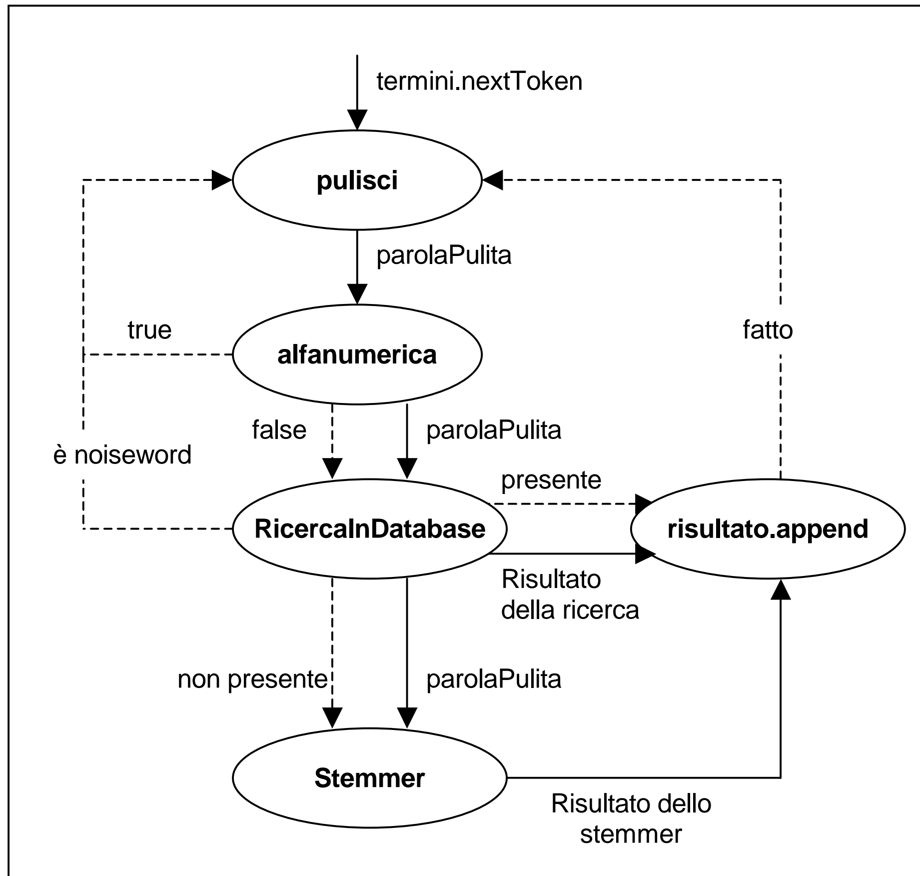


Fig. 6.8. Esplosione del processo Normalizzazione (livello 1)

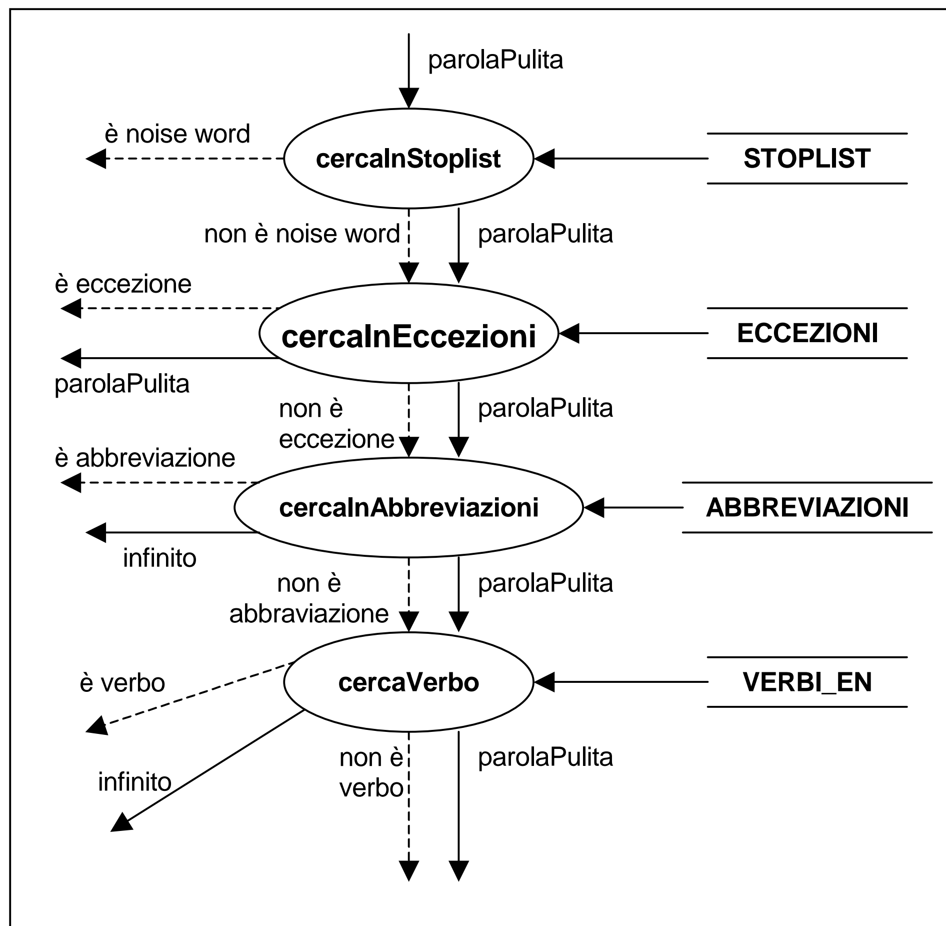


Fig. 6.9 Esplosione del processo RicercaInDatabase (livello 2)

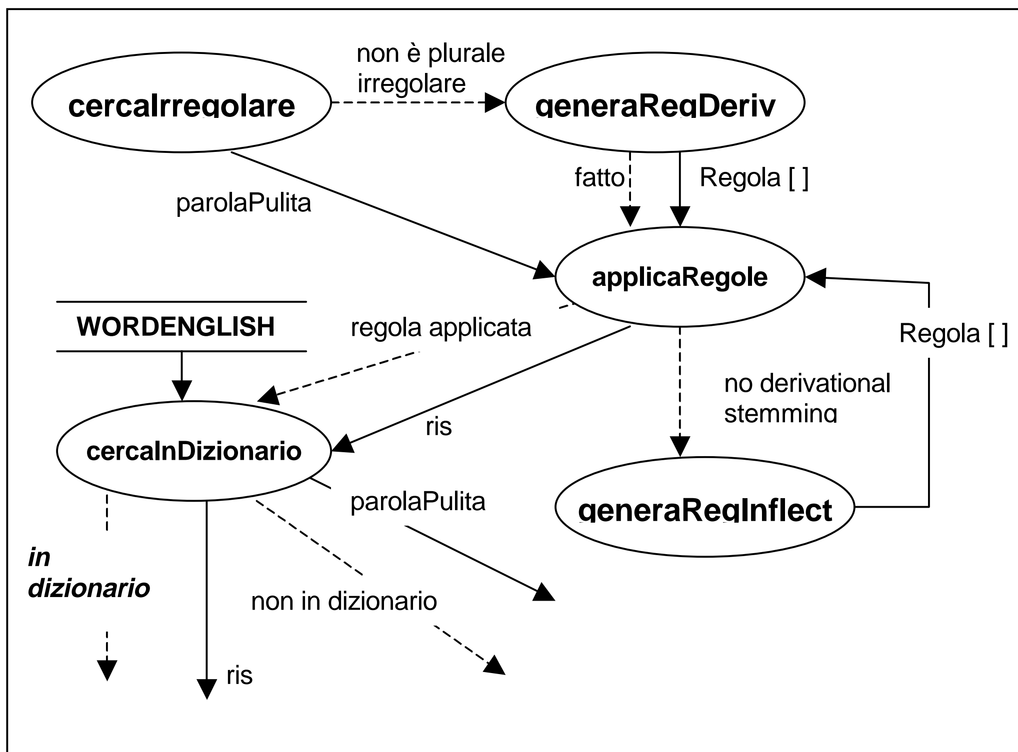


Fig. 6.10. Esplosione del processo Stemmer (livello 2)

6.2 Il procedimento di indicizzazione

Il procedimento di indicizzazione fa riferimento alla colonna STEMMED_TERM_EN della tabella TERMS. Questa indicizzazione viene eseguita attraverso Oracle 8i *inter*Media, utilizzando pertanto il comando `CREATE INDEX` (ved. § 4.1).

Il comando sarà:

```
CREATE INDEX idx_stemmed_term_en
ON terms(stemmed_term_en)
INDEXTYPE IS CTXSYS.CONTEXT
PARAMETERS('WORDLIST stem_null
            STOPLIST ctxsys.empty_stoplist');
```

Per quanto riguarda i parametri non specificati, si assumono validi quelli impostati per default dall'indexing engine di *inter*Media.

Per la classe DATASTORE (ved. § 4.1.1) il valore di default è DIRECT_DATASTORE e per le nostre esigenze è adeguato in quanto i documenti che vogliamo indicizzare (le frasi normalizzate) sono contenuti nella colonna sulla quale viene svolta l'operazione.

Per la classe FILTER (ved. § 4.1.2) il valore di default è NULL_FILTER che si applica quando il formato dei documenti è puro testo (come nel nostro caso).

Per la classe SECTION_GROUP (ved. § 4.1.3) il valore di default è NULL_SECTION_GROUP che permette di non valutare la presenza di sezioni sui documenti.

Per la classe LEXER (ved. § 4.1.4) il valore di default è BASIC_LEXER utilizza come delimitatori di tokens i caratteri di punteggiatura e gli spazi. Nel nostro caso i tokens delle frasi normalizzate sono delimitati da uno spazio ciascuno.

Per la classe STOPLIST (ved. § 4.1.5) abbiamo impostato una lista vuota in quanto abbiamo voluto gestire il controllo e l'eliminazione delle noise words attraverso il programma di normalizzazione.

Per quanto riguarda la classe WORDLIST (ved. § 4.1.6), come abbiamo detto essa non entra pienamente nell'indicizzazione. Le funzionalità che essa fornisce sono state sviluppate in modo più appropriato (rispetto alle esigenze riscontrate nell'ambito della traduzione) attraverso il procedimento implementato nel lavoro di tesi. Anche per questa classe, come per la precedente, è stata posta una limitazione nelle funzioni disabilitando lo *stemmer* interno di *inter*Media; non vogliamo, cioè, che, in fase di indicizzazione, *inter*Media intervenga con un'ulteriore normalizzazione. Questa impostazione è stata eseguita con le specifiche:

```
ctx_ddl.create_preference('stem_null','basic_wordlist');
```

per istanziare un oggetto BASIC_WORDLIST e

```
ctx_ddl.set_attribute('stem_null','stemmer','null');
```

per disabilitare la funzione di stemming dell'oggetto "stem_null".

6.3 Progettazione della modalità per eseguire le ricerche e classificarne i risultati

Nel paragrafo 5.3 abbiamo descritto in maniera superficiale la logica di una modalità di classificazione dei risultati della ricerca di approximate match.

In questo paragrafo sviluppiamo il progetto del processo RICERCHE lasciato sospeso in 6.1.3.3 (ved. Fig. 6.7).

Utilizzando il linguaggio di programmazione Java e le primitive SQLJ (ved. § B.7.2) per il collegamento al data base Oracle si pensa che una buona implementazione del procedimento descritto possa essere la seguente.

1. Generazione dei trigrammi che compongono la frase da tradurre.
2. Costruzione dell'istruzione SQL di interrogazione avente come termini di ricerca i trigrammi generati al passo precedente. L'istruzione (che sarà una SELECT) avrà una struttura del tipo visto nel paragrafo 5.3. Più precisamente:

```
SELECT codice, count(*)
FROM (SELECT codice
      FROM terms
      WHERE cod_cliente=codice introdotto
      AND cod_settore=codice introdotto
      AND CONTAINS(stemmed_term_en,'trigramma')>0
      UNION ALL
      SELECT codice . . . (una istruzione SELECT per ogni
      trigramma )
      . . .)
```

```
GROUP BY codice
ORDER BY 2;
```

Rispetto al modello sviluppato nel paragrafo 5.3, il risultato dell'unione delle interrogazioni interne è l'insieme che abbiamo indicato con Q.

3. Esecuzione dell'interrogazione impostata al passo precedente. Viene così creato un insieme di codici ordinato in base al rispettivo valore di COUNT Questo valore indica quanti trigrammi contiene la frase identificata dal relativo codice.
4. Si considerano gli identificativi corrispondenti ai dieci valori più alti dei contatori.
5. Attraverso una funzione si assegna un valore nell'intervallo [0,1] ad ogni identificativo di frase. Questa funzione, che si può assimilare ad una membership function (ved. § 5.3), riceverà in ingresso il valore del COUNT e, in base ad esso, calcolerà il grado di appartenenza all'insieme risultato della ricerca approximate match. Questo valore viene inserito nel documento ottenuto come risultato dell'operazione di pre-traduzione (ved. § 2.3). Indichiamo con "codice" l'identificativo della frase recuperata e con "count" il valore del relativo contatore. La funzione di membership **m** relativa ad ogni frase presente nel risultato del passo 4 può essere calcolata come rapporto tra il valore count relativo alla frase contrassegnata da codice e il numero **N** di trigrammi che compongono la frase da tradurre:

   $$\mathbf{m(codice)} = \frac{count(codice)}{N}\ .$$

La sequenza descritta si può tradurre schematicamente attraverso i formalismi OMT nella nuova classe (che chiamiamo Ricerche) il cui modello statico è riportato in Fig. 6.11.

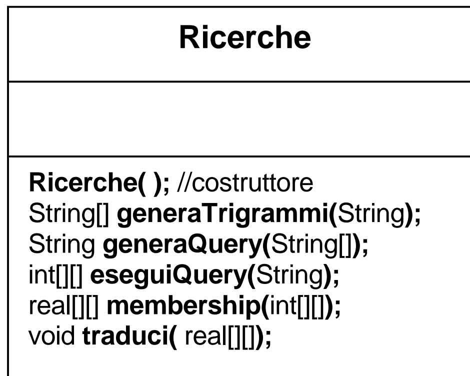


Fig. 6.11. Struttura statica della classe Ricerche

I metodi della classe sono:

- **Ricerche**( ): è il costruttore.
- **generaTrigrammi**(String frase): questo metodo restituisce un vettore di stringhe composto dai trigrammi generati partendo dalla stringa frase avuta in ingresso.
- **generaQuery**(String [] trigrammi): questo metodo prende in esame i trigrammi generati attraverso il metodo generaTrigrammi e costruisce una stringa contenente il testo dell'interrogazione descritta al passo 2.
- **eseguiQuery**(String query): questo metodo apre una connessione alla base di dati ed esegue l'interrogazione avuta in ingresso.
  Il metodo restituisce, in una matrice, l'insieme risultato dell'interrogazione descritto al passo 3.
- **membership**(int [] [] matrice): questo metodo si occupa del calcolo della membership function **m** relativa ad ogni codice contenuto nella prima colonna della matrice avuta in ingresso. Il calcolo avverrà come descritto nel passo 5 ed il valore count sarà prelevato dalla seconda colonna della matrice. Restituirà una matrice (di numeri reali) avente, per ogni riga, il codice della frase e il relativo valore di membership.

- **traduci**(real [] [] matrice): questo metodo esegue un'interrogazione sulla tabella TERMS per ognuno dei valori contenuti nella prima colonna della matrice avuta in ingresso aventi un valore di membership (seconda colonna) non inferiore a 0.4. Verrà estratto il contenuto della colonna di TERMS che contiene la frase nella lingua target della traduzione (ved. § 2.1).

  In SQL avremo, per i valori di membership maggiori o uguali a 0.4,

  ```
  select frase_in_lingua_target
  from terms
  where codice=:matrice[i][0].
  ```

  Ogni frase reperita verrà visualizzata insieme al relativo valore di membership contenuto nella seconda colonna della matrice avuta in ingresso.

Dal punto di vista funzionale, il sistema per classificare i risultati della ricerca viene esposto in Fig. 6.12.

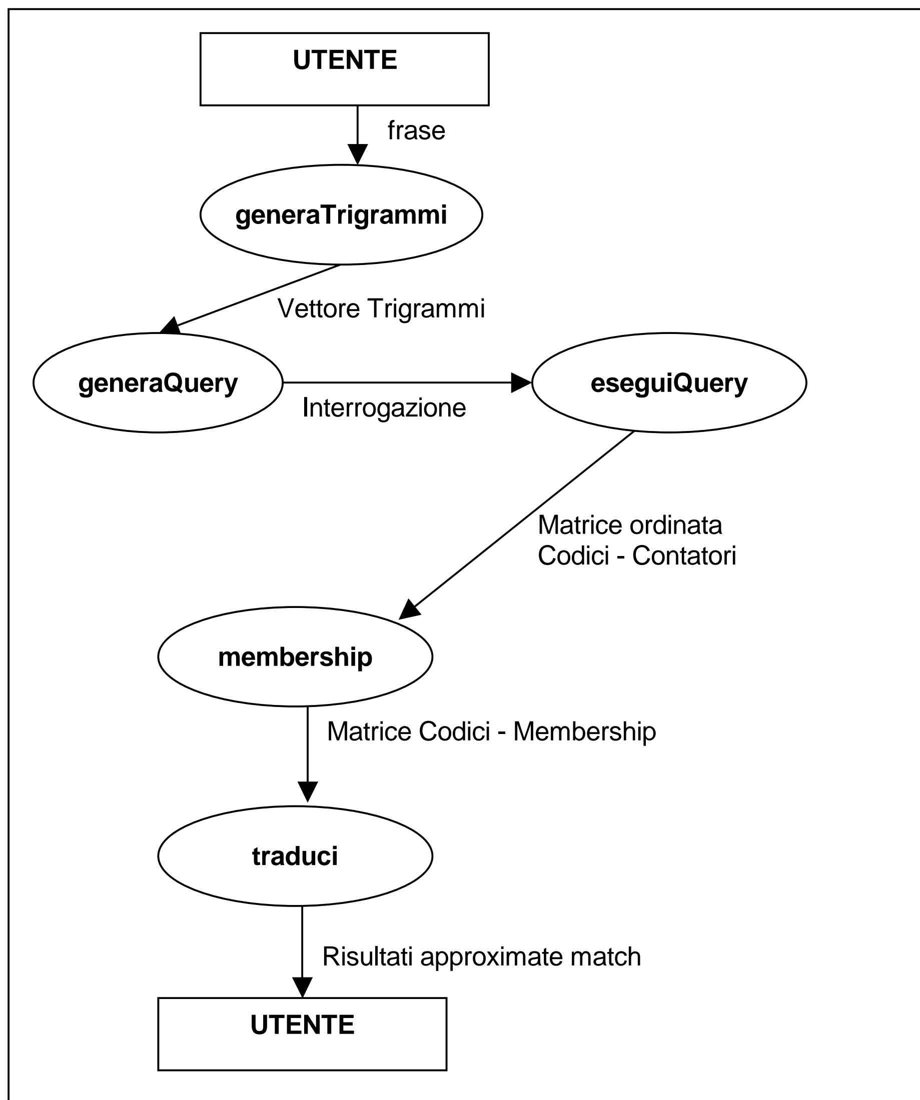


Fig. 6.12. Diagramma funzionale della classe Ricerche

Capitolo 7

TEST E CONCLUSIONI

In questo capitolo si descrivono alcune prove eseguite per valutare le prestazioni del nuovo metodo di ricerca di similarità tra frasi. Si descriveranno i miglioramenti apportati verificandone l'opportunità dal punto di vista dei "costi" che l'implementazione del nuovo procedimento comunque comporta.

7.1 Confronti tra i risultati ottenuti con il vecchio ed il nuovo algoritmo

Per eseguire le prove si utilizzano i risultati di operazioni di pre-traduzione già eseguite in precedenza presso la Logos; in particolare, i documenti pre-tradotti presi come riferimento sono quelli relativi al cliente (Corel) a cui corrisponde il prototipo di tabella TERMS descritto nel paragrafo 5.1. Applichiamo quindi il nuovo procedimento alle frasi per le quali è risultata necessaria la ricerca a livello approximate match (ved. § 2.2).

Di seguito riportiamo i risultati di alcuni test per illustrare il tipo di confronti effettuati. Per ognuno di essi si mostrano il risultato della normalizzazione applicata alla frase da tradurre e le frasi reperite tramite il procedimento nuovo con il relativo valore di membership function calcolato come indicato nel paragrafo 6.3 (per una comparazione più generale, si è rimosso il vincolo per cui si visualizzano soltanto le frasi che hanno un valore di membership non inferiore a 0.4). Di seguito si riporta la frase che era stata inserita nel risultato di pre-traduzione avente il valore di classificazione (ved. § 2.3.1) più elevato

tra quelle reperite. Occorre dire che il confronto tra i risultati prodotti dai due metodi di ricerca dovrebbe essere eseguito comparando le frasi reperite, non i valori numerici che indicano quanto i risultati stessi sono legati in significato alla frase da tradurre; questo vale perché questi due valori sono calcolati, come si può vedere dai paragrafi 2.3.1 e 6.3, in modo differente. L'analisi statistica effettuata (ved. Fig. 7.1) è stata eseguita riportando alla stessa unità di misura le due grandezze. Si è introdotta nel codice del prototipo una funzione che determina, per la frase reperita con valore di membership più elevato, il rapporto tra le parole uguali a quelle della frase cercata e il totale delle parole che compongono quest'ultima. Dai test seguenti si può notare come questa modifica renda ancora maggiore la differenza tra i valori di similarità calcolati con i due metodi posti a confronto.

Test 1

Frase da tradurre

You can also find updates and technical information, which was not available at press time, in the Release Notes.

Frase normalizzata

can find update technic information be avail press time release note

Frase recuperata con l'algoritmo nuovo

You can also find updates and technical information in the Release Notes that were not available at press time

Frase recuperata normalizzata (colonna stemmed_term_en della tabella TERMS)

can find update technic information release note be avail press time

Valore di membership 0.56

Valore ricalcolato per statistica 0.89

Frase recuperata con l'algoritmo vecchio

If you purchased an additional language module, you can also use Language to specify the language conventions for text in your documents and to specify a language to use with Grammatik, Hyphenation, Document Information, Spell Check, and Thesaurus

Valore di similarità 0.19

Test 2

Frase da tradurre

If you have a question about the features and functions of Corel applications or operating systems, look in the user guide, consult the online Help for the product you are using, or review the manuals in the Corel Reference Center (available in WordPerfect Office 2000, Paradox 9, and Corel WordPerfect Suite 8).

Frase normalizzata

have question feature function corel applicate operate system look use guide consult online help product be use review manual corel referent center avail wordperfect office paradox corel wordperfect suite

Frase recuperata con l'algoritmo nuovo

If you have a question about the features and functions of Corel applications, look in the user guide, consult the online Help for the application you are using, or review the manuals in the Corel Reference Center (available in WordPerfect Office 2000, Paradox 9, and Corel WordPerfect Suite 8)

Frase recuperata normalizzata (colonna stemmed_term_en della tabella TERMS)

have question feature function corel applicate look use guide consult online help applicate be use review manual corel referent center avail wordperfect office paradox corel wordperfect suite

Valore di membership 0.74

Valore ricalcolato per statistica 0.92

Frase recuperata con l'algoritmo vecchio

If you have more than one archived version of a file and you want to view an older version of a file, you must save a copy of it to your hard drive so that you can open it in its native application

Valore di similarità 0.19

Test 3

Frase da tradurre

The Corel product you are using is supported by the Corel Client Services team which is committed to provide quality customer service and support that is easy to access and convenient to use, while fostering one-to-one customer relationships.

Frase normalizzata

corel product be use be support corel client service team be commit provide quality custom service support be easy access convenient use foster one-to-one custom relation

Frase recuperata con l'algoritmo nuovo

The Corel application you are using is supported by the Corel Client Services team which is committed to provide quality customer service and support that is easy to access and convenient to use, while fostering one-to-one customer relationships

Frase recuperata normalizzata (colonna stemmed_term_en della tabella TERMS)

corel applicate be use be support corel client service team be commit provide quality custom service support be easy access convenient use foster one-to-one custom relation

Valore di membership 0.83

Valore ricalcolato per statistica 0.97

Frase recuperata con l'algoritmo vecchio

The default settings archive files in a single location, which is specified in the Corel Versions dialog box; however, you can also archive files in the same location as the original file that you are using as a basis for your archive

Valore di similarità 0.26

Test 4

Frase da tradurre

online <\{bmc emdash.bmp\}> on the Corel Web site

Frase normalizzata

online corel web site

Frasi recuperate con l'algoritmo nuovo

Downloads Masters from the Corel Web site

In order to view your show you must have Show It! <{\ul >plug-in.<}{\v glos_plug-in><}> You can download Show It! from the Corel Web site at www.corel.com

In order to view your show you must have Show It! plug-in. You can download Show It! from the Corel Web site at www.corel.com

Help, Corel Web Site

Help, Corel Web Site

Accessing information from the Corel Web site

Accessing information from the Corel Web site

You can access the Corel Web site directly from Corel Presentations

You can use the Corel Web site to get information about projects and templates, printing, fonts, and macros

To access the Corel Web site

Valore di membership Tutte le frasi recuperate hanno valore 0.5.

Valore ricalcolato per statistica 0.5

Frase recuperata con l'algoritmo vecchio

To download files from the Corel Web site

Valore di similarità 0.67

Si nota come in questo caso il fatto che il vecchio procedimento non eliminasse le noise words porta ad un valore di similarità più elevato.

Test 5

Frase da tradurre

For more information about registering your Corel product, see <{\b\uldb >www.corel.com/support/register.<}{\v!jumphtml('http://www.corel.com/support/register')>

Frase normalizzata

information register corel product see www.corel.com/support/register

Frase recuperata con l'algoritmo nuovo

-

Frase recuperata normalizzata (colonna stemmed_term_en della tabella TERMS)

-

Valore di membership -

Frase recuperata con l'algoritmo vecchio

For information contact http://www.corel.com/internet/barista.htm

Valore di similarità 0.22

Anche in questo caso il risultato viene influenzato dalla considerazione data alle noise words dal vecchio metodo.

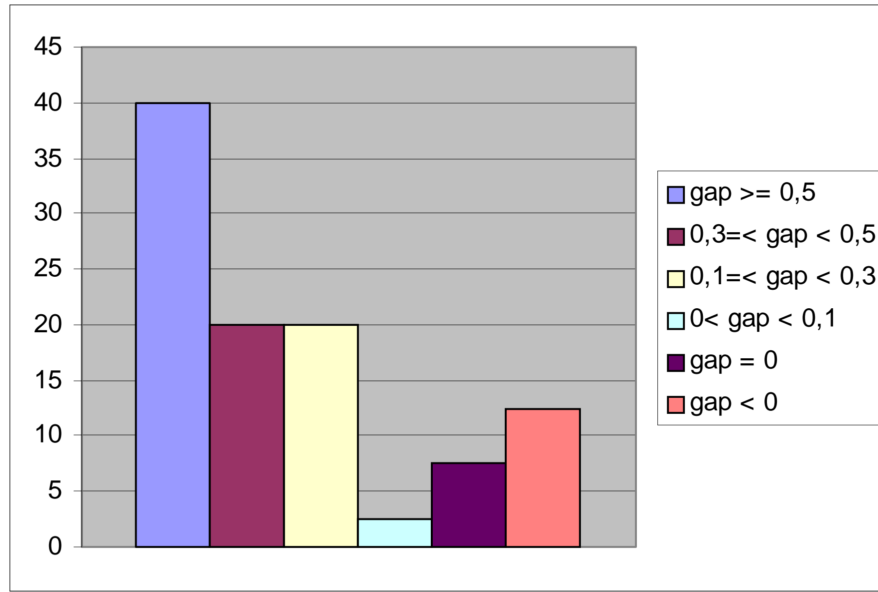


Fig. 7.1. Comparazione statistica tra il vecchio ed il nuovo procedimento di ricerca di similarità

Dal punto di vista dell'efficacia, il metodo sviluppato è sicuramente migliore del precedente in quanto reperisce, tra le frasi presenti nella base di dati delle traduzioni passate, quelle il cui significato si avvicina di più al significato della frase da tradurre. L'analisi statistica che dà un'idea dell'incremento di efficacia illustrata in Fig. 7.1 è stata eseguita, come detto, analizzando i documenti pre-tradotti per il cliente Corel. Più precisamente si è preso in considerazione un campione di 1000 documenti sui 5000 tradotti durante l'anno scorso. Il grafico riporta una suddivisione del totale delle frasi eseguita in base alla differenza (*gap*) tra i valori di similarità riscontrate nelle prove effettuate con il vecchio ed il nuovo procedimento. I valori sull'asse delle ordinate sono espressi in percentuale. Come si può vedere, l'80% delle frasi

reperite dal nuovo metodo hanno un valore di similarità superiore di almeno 0.1 rispetto a quelle reperite dal vecchio sistema e questo è un risultato lusinghiero. Il 12% per cui il gap è a favore del vecchio algoritmo comprende quelle frasi non reperite con il nuovo algoritmo. Precisiamo, comunque, che i corrispondenti risultati ottenuti con il vecchio algoritmo hanno un valore di similarità di frase molto basso e non restituiscono frasi utili per un traduttore professionista. Un esempio di questo caso è riportato nel **test 5**.

Riguardo alle prove effettuate occorre fare un'importante precisazione.

Il vecchio metodo risulta essere molto lento, per questo motivo, nonostante l'aggiornamento continuo della tabella TERMS contenente la "memoria storica" di tutte le frasi tradotte nel corso degli anni in Logos, il sistema di ricerca di similarità utilizzato finora è soggetto ad una limitazione che pone un tempo massimo di esecuzione (10 secondi). Oltre questo limite la ricerca viene fermata e vengono visualizzati i risultati reperiti fino a quel momento. Per questo motivo, spesso, non si riesce a pervenire ad un risultato buono.

7.2 Valutazione d'efficienza e punti critici

In questo paragrafo si illustrano i punti critici per quanto riguarda le prestazioni del nuovo procedimento di ricerca.

Mostriamo, a titolo di esempio, per il **Test 1**, l'interrogazione generata dal metodo **generaQuery** descritto nel paragrafo 6.3.

```
SELECT codice, COUNT(*)
    FROM (SELECT codice
              FROM terms
              WHERE cod_cliente = 6
              AND cod_settore = 5
              AND CONTAINS(stemmed_term_en,'can find update')> 0
         UNION ALL
         SELECT codice
             FROM terms
             WHERE cod_cliente = 6
             AND cod_settore = 5
             AND CONTAINS(stemmed_term_en,'find update technic')> 0
         UNION ALL
         SELECT codice
             FROM terms
```

```
            WHERE cod_cliente = 6
            AND cod_settore = 5
            AND CONTAINS(stemmed_term_en,'update technic information')> 0
        UNION ALL
        SELECT codice
            FROM terms
            WHERE cod_cliente = 6
            AND cod_settore = 5
            AND CONTAINS(stemmed_term_en,'technic information be')> 0
        UNION ALL
        SELECT codice
            FROM terms
            WHERE cod_cliente = 6
            AND cod_settore = 5
            AND CONTAINS(stemmed_term_en,'information be avail')> 0
        UNION ALL
        SELECT codice
            FROM terms
            WHERE cod_cliente = 6
            AND cod_settore = 5
            AND CONTAINS(stemmed_term_en,'be avail press')> 0
        UNION ALL
        SELECT codice
            FROM terms
            WHERE cod_cliente = 6
            AND cod_settore = 5
            AND CONTAINS(stemmed_term_en,'avail press time')> 0
        UNION ALL
        SELECT codice
            FROM terms
            WHERE cod_cliente = 6
            AND cod_settore = 5
            AND CONTAINS(stemmed_term_en,'press time release')> 0
        UNION ALL
        SELECT codice
            FROM terms
            WHERE cod_cliente = 6
            AND cod_settore = 5
            AND CONTAINS(stemmed_term_en,'time release note')> 0)
    GROUP BY codice
    ORDER BY 2 DESC
```

Questa interrogazione, nell'ambito di tutto il processo di ricerca, è sicuramente il collo di bottiglia. Essa, infatti, determina la misura più o meno elevata del tempo di esecuzione del processo stesso poiché le operazioni di normalizzazione e le altre ricerche sulla base di dati sono praticamente le stesse per ogni frase ricercata e le loro prestazioni non variano molto in base alla lunghezza della frase stessa. Dall'esempio si può, invece, capire quanto la velocità di esecuzione di interrogazioni di questo tipo dipenda fortemente dalla

lunghezza della frase normalizzata. Si ha infatti una subquery per ogni trigramma della frase stessa.

Dall'*explain plan* riportato di seguito, elaborato dall'ottimizzatore delle interrogazioni di Oracle, si può vedere che tipo di accessi vengono eseguiti sulle tabelle e quali indici vengono utilizzati.

```
Rows Id    Step
9     0    SELECT STATEMENT, parent id:0
9     1      SORT (ORDER BY), parent id:0
9     2      SORT (GROUP BY), parent id:1
9     3       VIEW, parent id:2
9     4        UNION ALL, parent id:2
      5         TABLE ACCESS(BY INDEX ROWID), TERMS, parent id:4
1     6           DOMAIN INDEX, IDX_STEMMED_TERM_EN, parent id:5
      7         TABLE ACCESS(BY INDEX ROWID), TERMS, parent id:4
1     8           DOMAIN INDEX, IDX_STEMMED_TERM_EN, parent id:7
      9         TABLE ACCESS(BY INDEX ROWID), TERMS, parent id:4
1    10           DOMAIN INDEX, IDX_STEMMED_TERM_EN, parent id:9
     11         TABLE ACCESS(BY INDEX ROWID), TERMS, parent id:4
1    12           DOMAIN INDEX, IDX_STEMMED_TERM_EN, parent id:11
     13         TABLE ACCESS(BY INDEX ROWID), TERMS, parent id:4
1    14           DOMAIN INDEX, IDX_STEMMED_TERM_EN, parent id:13
     15         TABLE ACCESS(BY INDEX ROWID), TERMS, parent id:4
1    16           DOMAIN INDEX, IDX_STEMMED_TERM_EN, parent id:15
     17         TABLE ACCESS(BY INDEX ROWID), TERMS, parent id:4
1    18           DOMAIN INDEX, IDX_STEMMED_TERM_EN, parent id:17
     19         TABLE ACCESS(BY INDEX ROWID), TERMS, parent id:4
1    20           DOMAIN INDEX, IDX_STEMMED_TERM_EN, parent id:19
     21         TABLE ACCESS(BY INDEX ROWID), TERMS, parent id:4
1    22           DOMAIN INDEX, IDX_STEMMED_TERM_EN, parent id:21
```

Nonostante siano stati creati indici (b-tree) anche sui campi cod_cliente e cod_settore, si nota che l'unico indice utilizzato per l'accesso alla tabella TERMS è l'inverted index idx_stemmed_term_en creato con *inter*Media (ved. § 6.2). Da prove effettuate imponendo l'utilizzo anche degli indici su cod_cliente e cod_settore si è notato un degrado in prestazioni dell'ordine di secondi pertanto la strada è stata abbandonata immediatamente.

Il caso illustrato nel **Test 1**, che può essere definito un caso medio come lunghezza di frase, comporta un tempo di esecuzione dell'interrogazione di

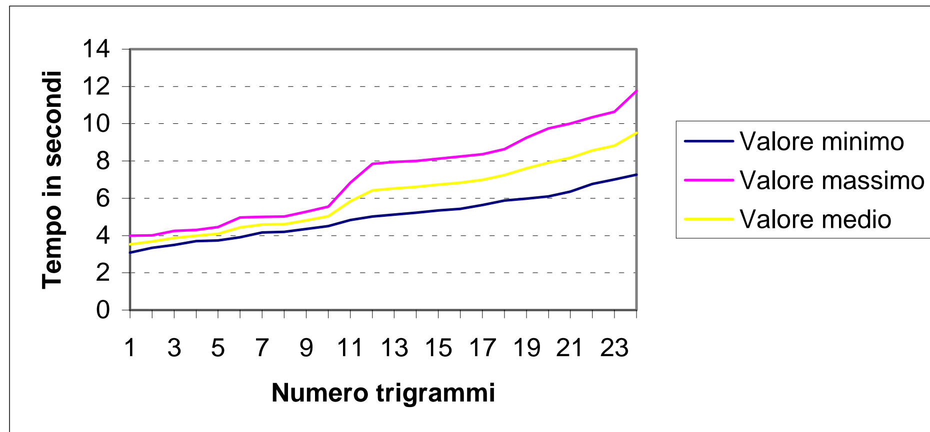


Fig. 7.2. Andamento del tempo di ricerca di una frase in funzione del numero di trigrammi che la compongono

circa 4.5 secondi. Il caso peggiore è risultato quello descritto nel **Test 2**. La frase ricercata in questo caso contiene 27 trigrammi e l'esecuzione della ricerca ha comportato un tempo di esecuzione di circa 11 secondi.

Possiamo considerare soddisfacenti queste prestazioni, in considerazione del fatto che le prove sono state effettuate su una macchina con processore Intel Pentium III 600 MHz e con disco rigido EIDE a 7200 giri. È molto probabile che queste prestazioni vengano incrementate utilizzando le macchine (più potenti e veloci) utilizzate per eseguire le operazioni di pre-traduzione.

L'andamento generale del tempo di esecuzione della ricerca in funzione della lunghezza della frase (più precisamente in funzione del numero di trigrammi) è riportato in Fig. 7.2.

Valutiamo ora, più generalmente, tutta la nuova modalità di trattamento degli approximate match, in particolare le conseguenze portate dall'utilizzo di un inverted index di *inter*Media. La tabella TERMS è soggetta a operazioni di inserimento e cancellazione (INSERT e DELETE) ogni volta che viene eseguita una pre-traduzione. Come abbiamo visto nel paragrafo 4.3, l'operazione di aggiornamento dell'indice può avvenire attraverso i metodi

sync o background. In ogni modo, qualsiasi metodo si scelga, questo aggiornamento risulta senz'altro un onere aggiuntivo che il vecchio metodo di ricerca approximate match non comportava (ved. § 2.3.1); esso infatti utilizzava una ricerca esatta sulla tabella WORDS contenente tutti i trigrammi delle frasi contenute in TERMS. La colonna di WORDS era indicizzata con un normale indice b-tree che non necessita di aggiornamento in caso di modifiche o cancellazioni di tuple.

Oltre all'aggiornamento, poi, l'inverted index creato con *inter*Media deve essere sottoposto alle operazioni di ottimizzazione viste nel paragrafo 4.4: la deframmentazione e la garbage collection. Queste operazioni, anch'esse non necessarie precedentemente, comportano un utilizzo di risorse e un dispendio di tempo che prima non erano previsti.

Potendo però programmare gli interventi nel modo seguente:

- aggiornamento dell'indice ogni volta che viene eseguita una pre-traduzione;
- una operazione di deframmentazione giornaliera (eventualmente durante la notte);
- una operazione di garbage collection settimanale (sabato e/o domenica);

si può in qualche modo minimizzare i disagi creati dalla nuova modalità di gestione della ricerca di similarità tra frasi.

7.3 Possibilità di sviluppi futuri

Dal punto di vista dell'efficacia dell'algoritmo, si è notato che qualora vengano ricercate frasi corte, composte da due o tre parole, il valore di membership può non essere così veritiero rispetto al valore calcolato per frasi di lunghezza media o elevata. Uno sviluppo futuro può consistere nel modificare l'ampiezza degli n-grammi in cui viene scomposta la frase da

ricercare. Per esempio, per le frasi composte da tre parole, si può valutare l'utilizzo di bigrammi.

Per quanto riguarda l'efficienza, come si vede dal grafico in Fig. 7.2, il tempo di risposta cresce all'aumentare del numero di trigrammi che compongono la frase ricercata. Per le frasi composte da un numero di trigrammi superiore a venti, si può considerare ancora un cambiamento nell'ampiezza degli n-grammi generati. In questo caso la modifica consisterebbe in un aumento dell'ampiezza stessa.

Appendice A

LISTATI JAVA DELL'ALGORITMO PROGETTATO

```
package StemmerOracle;


  class Regola
  {

    private String suffix;
    private int order;
    private int level;
    private String suffix1;
    private String suffix2;

    public Regola(String s, int o, int l, String s1, String s2)
     { this.suffix = s;
       this.order = o;
       this.level = l;
       this.suffix1 = s1;
       this.suffix2 = s2;
     }


    public String getSuffix() {
     return suffix;
    }

    public int getOrder() {
     return order;
    }

    public int getLevel() {
     return level;
    }

    public String getSuffix1() {
     return suffix1;
    }

    public String getSuffix2() {
     return suffix2;
    }
  }
```

```
package StemmerOracle;

import java.util.*;

abstract class Stemming  {

    public String[] applicaRegole(String parola, Regola[] regole)
    {
        int ordine = 10000;
        int livello = 0;
        String suffisso = "";
        String suffisso1 = "";
        String suffisso2 = "";

        for (int i = 0; i < regole.length; i++)
        {
            if (parola.endsWith(regole[i].getSuffix()))
            {
                if (regole[i].getOrder() < ordine)
                {
                    ordine = regole[i].getOrder();
                    suffisso = regole[i].getSuffix();
                    livello = regole[i].getLevel();
                    suffisso1 = regole[i].getSuffix1();
                    suffisso2 = regole[i].getSuffix2();
                }
             }
        }

        Vector pref = new Vector();
        for (int i = 1; i<= suffisso.length(); i++)
        {
        String s = parola.substring(0, (parola.length()-i));
        pref.add(s);
        }

        String[] prefissi = new String[pref.size()];
        pref.copyInto(prefissi);

        String[] risultati = new String[3];
        risultati[0] = "";
        risultati[1] = "";
        risultati[2] = "si";
        int pos = 0;
        switch (livello)
        {
        case 1:
        risultati[0] = prefissi[0];
        risultati[1] = prefissi[1];
        break;
```

```
        case 2:
        pos = prefissi.length - 1;
        risultati[0] = prefissi[pos] + suffisso1;
        risultati[1] = "";
        break;

        case 3:
        pos = prefissi.length - 1;
        risultati[0] = prefissi[pos] + suffisso1;
        if (prefissi[pos].endsWith("mm"))
           {prefissi[pos] = (prefissi[pos].substring(0, prefissi[pos].length()-1))+"e";}

       else if  (prefissi[pos].endsWith("d")|
                 prefissi[pos].endsWith("r")|
                 prefissi[pos].endsWith("m")|
                 prefissi[pos].endsWith("n")|
                 prefissi[pos].endsWith("s")|
                 prefissi[pos].endsWith("l")|
                 prefissi[pos].endsWith("v")|
                 prefissi[pos].endsWith("g")|
                 prefissi[pos].endsWith("mput")|
                 prefissi[pos].endsWith("at"))
                 {prefissi[pos] = prefissi[pos]+"e";}

        risultati[1] = prefissi[pos];
        break;

        case 4:
        pos = prefissi.length - 1;
        risultati[0] = prefissi[pos] + suffisso1;
        risultati[1] = prefissi[pos] + suffisso2;
        break;

        case 5:
        risultati[0] = parola;
        risultati[1] = "";
        risultati[2] = "no";
        break;

        default:
        risultati[0] = parola;
        risultati[1] = "";
        break;
        }//end_switch

        return risultati;
    }
}
```

```
package StemmerOracle;

    class Inflectional extends Stemming {
     public Inflectional () {
    //costruttore
    }


    public String cercaIrregolare(String nome) {
        String risultato = "";
        String[][] nomiIrregolari = new String[55][2];

        nomiIrregolari[0][0] = "alumni";
        nomiIrregolari[0][1] = "alumnus";
        nomiIrregolari[1][0] = "analyses";
        nomiIrregolari[1][1] = "analysis";
        nomiIrregolari[2][0] = "antennae";
        nomiIrregolari[2][1] = "antenna";
        nomiIrregolari[3][0] = "appendices";
        nomiIrregolari[3][1] = "appendix";
        nomiIrregolari[4][0] = "axes";
        nomiIrregolari[4][1] = "axis";
        nomiIrregolari[5][0] = "bacteria";
        nomiIrregolari[5][1] = "bacterium";
        nomiIrregolari[6][0] = "bases";
        nomiIrregolari[6][1] = "basis";
        nomiIrregolari[7][0] = "beaux";
        nomiIrregolari[7][1] = "beau";
        nomiIrregolari[8][0] = "bureaux";
        nomiIrregolari[8][1] = "bureau";
        nomiIrregolari[9][0] = "children";
        nomiIrregolari[9][1] = "child";
        nomiIrregolari[10][0] = "corpora";
        nomiIrregolari[10][1] = "corpus";
        nomiIrregolari[11][0] = "crises";
        nomiIrregolari[11][1] = "crisis";
        nomiIrregolari[12][0] = "criteria";
        nomiIrregolari[12][1] = "criterion";
        nomiIrregolari[13][0] = "curricula";
        nomiIrregolari[13][1] = "curriculum";
        nomiIrregolari[14][0] = "data";
        nomiIrregolari[14][1] = "datum";
        nomiIrregolari[15][0] = "deer";
        nomiIrregolari[15][1] = "deer";
        nomiIrregolari[16][0] = "diagnoses";
        nomiIrregolari[16][1] = "diagnosis";
        nomiIrregolari[17][0] = "ellipses";
        nomiIrregolari[17][1] = "ellipsis";
        nomiIrregolari[18][0] = "foci";
        nomiIrregolari[18][1] = "focus";
        nomiIrregolari[19][0] = "feet";
```

```
nomiIrregolari[19][1] = "foot";
nomiIrregolari[20][0] = "formulae";
nomiIrregolari[20][1] = "formula";
nomiIrregolari[21][0] = "fungi";
nomiIrregolari[21][1] = "fungus";
nomiIrregolari[22][0] = "genera";
nomiIrregolari[22][1] = "genus";
nomiIrregolari[23][0] = "geese";
nomiIrregolari[23][1] = "goose";
nomiIrregolari[24][0] = "hypotheses";
nomiIrregolari[24][1] = "hypothesis";
nomiIrregolari[25][0] = "indeces";
nomiIrregolari[25][1] = "index";
nomiIrregolari[26][0] = "lice";
nomiIrregolari[26][1] = "louse";
nomiIrregolari[27][0] = "men";
nomiIrregolari[27][1] = "man";
nomiIrregolari[28][0] = "matrices";
nomiIrregolari[28][1] = "matrix";
nomiIrregolari[29][0] = "means";
nomiIrregolari[29][1] = "means";
nomiIrregolari[30][0] = "media";
nomiIrregolari[30][1] = "medium";
nomiIrregolari[31][0] = "mice";
nomiIrregolari[31][1] = "mouse";
nomiIrregolari[32][0] = "nebulae";
nomiIrregolari[32][1] = "nebula";
nomiIrregolari[33][0] = "nuclei";
nomiIrregolari[33][1] = "nucleus";
nomiIrregolari[34][0] = "oases";
nomiIrregolari[34][1] = "oasis";
nomiIrregolari[35][0] = "oxen";
nomiIrregolari[35][1] = "ox";
nomiIrregolari[36][0] = "paralyses";
nomiIrregolari[36][1] = "paralysis";
nomiIrregolari[37][0] = "parentheses";
nomiIrregolari[37][1] = "parenthesis";
nomiIrregolari[38][0] = "phenomena";
nomiIrregolari[38][1] = "phenomenon";
nomiIrregolari[39][0] = "radii";
nomiIrregolari[39][1] = "radius";
nomiIrregolari[40][0] = "series";
nomiIrregolari[40][1] = "series";
nomiIrregolari[41][0] = "species";
nomiIrregolari[41][1] = "species";
nomiIrregolari[42][0] = "stimuli";
nomiIrregolari[42][1] = "stimulus";
nomiIrregolari[43][0] = "strata";
nomiIrregolari[43][1] = "stratum";
nomiIrregolari[44][0] = "syntheses";
nomiIrregolari[44][1] = "synthesis";
```

```
        nomiIrregolari[45][0] = "teeth";
        nomiIrregolari[45][1] = "tooth";
        nomiIrregolari[46][0] = "vertebrae";
        nomiIrregolari[46][1] = "vertebra";
        nomiIrregolari[47][0] = "vitae";
        nomiIrregolari[47][1] = "vita";
        nomiIrregolari[48][0] = "women";
        nomiIrregolari[48][1] = "woman";
        nomiIrregolari[49][0] = "knives";
        nomiIrregolari[49][1] = "knife";
        nomiIrregolari[50][0] = "wives";
        nomiIrregolari[50][1] = "wife";
        nomiIrregolari[51][0] = "wolves";
        nomiIrregolari[51][1] = "wolf";
        nomiIrregolari[52][0] = "shelves";
        nomiIrregolari[52][1] = "shelf";
        nomiIrregolari[53][0] = "thieves";
        nomiIrregolari[53][1] = "thief";
        nomiIrregolari[54][0] = "news";
        nomiIrregolari[54][1] = "news";

        for (int i = 0; i < 53; i++){
        if (nome.equals(nomiIrregolari[i][0])){
            risultato = nomiIrregolari[i][1];
            i = 53;}
        }//end_for
        return risultato;
    }

    public Regola[] generaRegInflect(){

        Regola [] regInflect = new Regola[9];

        regInflect[0] = new Regola("hes",1011,1,"","");
        regInflect[1] = new Regola("ses",1021,1,"","");
        regInflect[2] = new Regola("xes",1031,1,"","");
        regInflect[3] = new Regola("zes",1041,1,"","");
        regInflect[4] = new Regola("'s",1055,2,"","");
        regInflect[5] = new Regola("ies",1528,2,"y","");
        regInflect[6] = new Regola("es",1529,1,"","");
        regInflect[7] = new Regola("ss",1061,5,"","");
        regInflect[8] = new Regola("s",1530,2,"","");

        return regInflect;
    }
}
```

```
package StemmerOracle;

class Derivational extends Stemming {


    public Derivational( ) {
    //costruttore
   }

    public Regola[ ] generaRegDeriv( ){

         Regola[ ] regDeriv = new Regola[122];

         regDeriv[0] = new Regola("iless",1051,2,"y","");
         regDeriv[1] = new Regola("iest",1141,2,"y","");
         regDeriv[2] = new Regola("est",1142,3,"e","");
         regDeriv[3] = new Regola("fiable",1145,2,"fy","");
         regDeriv[4] = new Regola("cable",1201,2,"cate","");
         regDeriv[5] = new Regola("gable",1202,5,"","");
         regDeriv[6] = new Regola("able",1203,3,"","e");
         regDeriv[7] = new Regola("graphic",1251,2,"graphy","");
         regDeriv[8] = new Regola("istic",1261,2,"ist","");
         regDeriv[9] = new Regola("logic",1291,2,"logy","");
         regDeriv[10] = new Regola("mental",1311,2,"ment","");
         regDeriv[11] = new Regola("ical",1361,4,"ic","e");
         regDeriv[12] = new Regola("ional",1371,2,"ion","");
         regDeriv[13] = new Regola("iful",1411,2,"y","");
         regDeriv[14] = new Regola("ful",1412,2,"","");
         regDeriv[15] = new Regola("ive",1521,3,"ion","");
         regDeriv[16] = new Regola("onian",1530,2,"on","");
         regDeriv[17] = new Regola("logical",1360,2,"logy","");
         regDeriv[18] = new Regola("ous",1060,5,"","");
         regDeriv[19] = new Regola("ated",1131,2,"ate","");
         regDeriv[20] = new Regola("ied",1121,2,"y","");
         regDeriv[21] = new Regola("ing",1151,3,"","e");
         regDeriv[22] = new Regola("logicals",1360,2,"logy","");
         regDeriv[23] = new Regola("mentals",1311,2,"ment","");
         regDeriv[24] = new Regola("istics",1261,2,"ist","");
         regDeriv[25] = new Regola("graphics",1251,2,"graphy","");
         regDeriv[26] = new Regola("icals",1361,4,"ic","e");
         regDeriv[27] = new Regola("graphical",1360,4,"graphy","graph");
         regDeriv[28] = new Regola("graphical",1360,4,"graphy","graph");
         regDeriv[29] = new Regola("izing",1150,2,"ize","");
         regDeriv[30] = new Regola("yzing",1150,2,"yze","");
         regDeriv[31] = new Regola("ytic",1271,2,"ysis","");
         regDeriv[32] = new Regola("ytics",1271,2,"ysis","");
         regDeriv[33] = new Regola("ytical",1360,2,"ysis","");
         regDeriv[34] = new Regola("yticals",1360,2,"ysis","");
         regDeriv[35] = new Regola("onal",1371,2,"on","");
         regDeriv[36] = new Regola("llable",1201,4,"ll","l");
         regDeriv[37] = new Regola("ttable",1201,2,"t","");
```

```
regDeriv[38] = new Regola("iness",1053,2,"y","");
regDeriv[39] = new Regola("ness",1054,2,"","");
regDeriv[40] = new Regola("cter",1091,5,"","");
regDeriv[41] = new Regola("blity",1143,5,"","");
regDeriv[42] = new Regola("bility",1144,2,"ble","");
regDeriv[43] = new Regola("logist",1146,2,"logy","");
regDeriv[44] = new Regola("ist",1161,3,"","e");
regDeriv[45] = new Regola("ism",1171,3,"","e");
regDeriv[46] = new Regola("ity",1181,5,"","e");
regDeriv[47] = new Regola("ment",1301,2,"","");
regDeriv[48] = new Regola("metry",1321,4,"meter","metre");
regDeriv[49] = new Regola("nce",1331,2,"nt","");
regDeriv[50] = new Regola("ncy",1341,2,"nt","");
regDeriv[51] = new Regola("ship",1351,2,"","");
regDeriv[52] = new Regola("ihood",1421,2,"y","");
regDeriv[53] = new Regola("hood",1422,2,"","");
regDeriv[54] = new Regola("ification",1451,2,"ify","");
regDeriv[55] = new Regola("ization",1461,2,"ize","");
regDeriv[56] = new Regola("ction",1471,2,"ct","");
regDeriv[57] = new Regola("rtion",1481,2,"rt","");
regDeriv[58] = new Regola("ation",1491,3,"ate","");
regDeriv[59] = new Regola("ator",1501,2,"ate","");
regDeriv[60] = new Regola("ctor",1511,2,"ct","");
regDeriv[61] = new Regola("ician",1530,4,"ic","ical");
regDeriv[62] = new Regola("portion",1480,5,"","");
regDeriv[63] = new Regola("inesses",1053,2,"y","");
regDeriv[64] = new Regola("nesses",1054,2,"","");
regDeriv[65] = new Regola("bilities",1054,2,"ble","");
regDeriv[66] = new Regola("logists",1146,2,"logy","");
regDeriv[67] = new Regola("ists",1161,3,"","e");
regDeriv[68] = new Regola("isms",1171,3,"","e");
regDeriv[69] = new Regola("ities",1055,2,"ity","");
regDeriv[70] = new Regola("ments",1301,2,"","");
regDeriv[71] = new Regola("metries",1321,4,"meter","metre");
regDeriv[72] = new Regola("nces",1331,2,"nt","");
regDeriv[73] = new Regola("ncies",1341,2,"nt","");
regDeriv[74] = new Regola("ships",1351,2,"","");
regDeriv[75] = new Regola("ifications",1451,2,"ify","");
regDeriv[76] = new Regola("izations",1461,2,"ize","");
regDeriv[77] = new Regola("ctions",1471,4,"ct","ction");
regDeriv[78] = new Regola("rtions",1481,2,"rt","");
regDeriv[79] = new Regola("ations",1491,3,"ate","");
regDeriv[80] = new Regola("ators",1501,2,"ate","");
regDeriv[81] = new Regola("ctors",1511,2,"ct","");
regDeriv[82] = new Regola("icians",1530,4,"ic","ical");
regDeriv[83] = new Regola("izer",1461,2,"ize","");
regDeriv[84] = new Regola("izers",1461,2,"ize","");
regDeriv[85] = new Regola("ifier",1451,2,"ify","");
regDeriv[86] = new Regola("ifiers",1451,2,"ify","");
regDeriv[87] = new Regola("mentalism",1170,2,"ment","");
regDeriv[88] = new Regola("mentalisms",1170,2,"ment","");
```

```
        regDeriv[89] = new Regola("mentalist",1160,2,"ment","");
        regDeriv[90] = new Regola("mentalists",1160,2,"ment","");
        regDeriv[91] = new Regola("yst",1161,2,"ysis","");
        regDeriv[92] = new Regola("ysts",1161,2,"ysis","");
        regDeriv[93] = new Regola("ysis",1011,5,"","");
        regDeriv[94] = new Regola("yzer",1461,2,"yze","");
        regDeriv[95] = new Regola("yzers",1461,2,"yze","");
        regDeriv[96] = new Regola("thesis",1461,5,"","");
        regDeriv[97] = new Regola("an",1531,2,"a","");
        regDeriv[98] = new Regola("ier",1101,2,"y","");
        regDeriv[99] = new Regola("er",1462,1,"","");
        regDeriv[100] = new Regola("ller",1461,4,"ll","l");
        regDeriv[101] = new Regola("llers",1461,4,"ll","l");
        regDeriv[102] = new Regola("mmer",1461,4,"mm","m");
        regDeriv[103] = new Regola("mmers",1461,4,"mm","m");
        regDeriv[104] = new Regola("nner",1461,4,"nn","n");
        regDeriv[105] = new Regola("nners",1461,4,"nn","n");
        regDeriv[106] = new Regola("ers",1462,3,"e","");
        regDeriv[107] = new Regola("plication",1490,4,"plicate","ply");
        regDeriv[108] = new Regola("plications",1490,4,"plicate","ply");
        regDeriv[109] = new Regola("less",1052,2,"","");
        regDeriv[110] = new Regola("ater",1081,1,"","");
        regDeriv[111] = new Regola("like",1281,2,"","");
        regDeriv[112] = new Regola("bly",1381,2,"ble","");
        regDeriv[113] = new Regola("ily",1391,2,"y","");
        regDeriv[114] = new Regola("ly",1401,2,"","");
        regDeriv[115] = new Regola("logically",1390,2,"logy","");
        regDeriv[116] = new Regola("graphically",1390,4,"graphy","graph");
        regDeriv[117] = new Regola("ytically",1390,2,"ysis","");
        regDeriv[118] = new Regola("mentally",1390,2,"ment","");
        regDeriv[119] = new Regola("ically",1395,4,"ic","e");
        regDeriv[120] = new Regola("ally",1395,4,"","al");
        regDeriv[121] = new Regola("ifully",1390,2,"y","");

        return regDeriv;
    }
}
```

```
package StemmerOracle;

import java.sql.*;
import oracle.sqlj.runtime.Oracle;
import java.util.*;
import java.io.*;

class RicercaInDbOracle
{

 public RicercaInDbOracle(){
 try{Oracle.connect(ConnettiOracle.class, "connect.properties");}
 catch (Exception e){System.out.println("Errore: " + e);}
}  //costruttore


public String cercaVerbo(String parola) throws SQLException{

      String verbo=null;
       #sql iterator IteratoreVerbo (String verb);

       IteratoreVerbo iteraVerbo;

       #sql iteraVerbo = {SELECT verb FROM VERBI_EN WHERE idc=0
                          AND id=(SELECT DISTINCT id FROM VERBI_EN
                          WHERE verb=:parola)};

       while (iteraVerbo.next())
         verbo=iteraVerbo.verb();
        iteraVerbo.close();
        return verbo;

}

public String cercaInDizionario(String parola) throws SQLException{

        String dic=null;

       #sql iterator IteratoreDic (String word);

       IteratoreDic iteraDic;

       #sql iteraDic = {SELECT word FROM WORDENGLISH WHERE word=(:parola)};
        while(iteraDic.next())
          dic=iteraDic.word();

       iteraDic.close();
       return dic;
```

```
}


public String cercaInStoplist(String parola) throws SQLException{

       String risultato=null;
       #sql iterator IteratoreStopList (String word);

       IteratoreStopList iteraStopList;

       #sql iteraStopList = {SELECT word FROM STOPLIST WHERE word=(:parola)};
       while (iteraStopList.next())
        if (iteraStopList.word()==null)
          risultato="";
        else
          risultato=iteraStopList.word();

        iteraStopList.close();
       return risultato;

}

public String cercaInEccezioni(String parola) throws SQLException{

       String risultato=null;
      #sql iterator IteratoreEccez(String word);

       IteratoreEccez iteraEccez;

       #sql iteraEccez = {SELECT word FROM ECCEZIONI WHERE word=(:parola)};
       while(iteraEccez.next())
        risultato=iteraEccez.word();

       iteraEccez.close(); */

       return risultato;
}

public String cercaInAbbreviazioni(String parola) throws SQLException{

       String abbrev=null;

       #sql iterator IteratoreAbbrev (String verb);

       IteratoreAbbrev iteraAbbrev;

       #sql    iteraAbbrev    =    {SELECT    verb    FROM    ABBREV_EN    WHERE
                                            espressione=(:parola)};
        while (iteraAbbrev.next())
          abbrev=iteraAbbrev.verb();
```

```
      iteraAbbrev.close();
      return abbrev;

}

public static void chiudiOracle(){
try { Oracle.close(); } catch (SQLException e) { }
}//end_chiudiOracle

public  void inserisci (String frase, int codice) throws IOException, SQLException {

   #sql {UPDATE TERMS SET STEMMED_TERM_EN=:frase WHERE
          CODICE=:codice};
}//end_inserisci

}//end_Connetti



package StemmerOracle;

import java.util.*;
import java.io.*;

class Filtri {

  public Filtri() {
  }

  public static String filtraCodice(String frase) throws IOException {
    Stack pila= new Stack();
    StringBuffer risultato=new StringBuffer();
    Integer i=new Integer(0);
    for (int cont=0; cont<frase.length();cont++){
        risultato.append(frase.charAt(cont));
        if (frase.charAt(cont)=='<') {
          Integer indice=new Integer(risultato.length()-1);
          pila.push(indice);
        }
        else{
          if (frase.charAt(cont)=='>'){
            if (!pila.empty()) {
              i= (Integer) pila.pop();
              risultato=risultato.replace(i.intValue(),risultato.length()," ");
            }//end_if (!pila.empty())
          }//end_if
        }//end_else
    }//end_for
    return risultato.toString();
```

```
}//end_filtraCodice

public static boolean alfanumerica(String parola) throws IOException {
  boolean numerico=false;
  int cont=0;
  while ((!numerico) && (cont<parola.length())){
    if (Character.isDigit(parola.charAt(cont))) numerico=true;
    cont=cont+1;
  }//end_while
  return numerico;
}//alfanumerica


public static String filtraDoppiSpazi(String frase) throws IOException {
  StringBuffer risultato=new StringBuffer();
  StringTokenizer st=new StringTokenizer(frase);
  while (st.hasMoreTokens()){
    risultato.append(st.nextToken()+" ");
  }
  return risultato.toString();

}//end_filtraDoppiSpazi

public static String pulisci(String parola){
  String risultato="";
  while ((parola.charAt(0)=='"')|
        ((Character.isDigit(parola.charAt(0)))&&(parola.charAt(1)=='.'))|
        (parola.startsWith("."))|
        (parola.startsWith("("))|
        (parola.startsWith(")"))|
        (parola.startsWith(","))|
        (parola.startsWith(";"))|
        (parola.startsWith(":"))|
        (parola.startsWith("<"))|
        (parola.startsWith("-"))|
        (parola.startsWith("+"))|
        (parola.startsWith("*"))|
        (parola.startsWith("/"))|
        (parola.startsWith("_"))|
        (parola.startsWith("^"))|
        (parola.startsWith(">"))|
        (parola.startsWith("<"))|
        (parola.startsWith("="))|
        (parola.startsWith("·  "))|
        (parola.startsWith("?"))&&(!(parola.length()<2)))
        {parola=parola.substring(1,(parola.length()));}
  while ((parola.endsWith("."))|
        (parola.endsWith(","))|
        (parola.endsWith(";"))|
        (parola.endsWith(":"))|
        (parola.endsWith("("))|
```

```
        (parola.endsWith(")"))|
        (parola.endsWith("!"))|
        (parola.endsWith("?"))|
        (parola.endsWith("-"))|
        (parola.endsWith("+"))|
        (parola.endsWith("*"))|
        (parola.endsWith("/"))|
        (parola.endsWith("_"))|
        (parola.endsWith("^"))|
        (parola.charAt(parola.length()-1)=='"')|
        (parola.endsWith("="))|
        (parola.endsWith("·  "))|
        (parola.endsWith(">"))|
        (parola.endsWith("<"))&&(!(parola.length()<2)))
        {parola = parola.substring(0,(parola.length()-1));}
    if (parola.length()>= 2) return parola;
    else return " ";
  }//end_pulisci

public static StringTokenizer segmenta(String frase) {
frase = Utils.filtraCodice(frase);
StringTokenizer termini = new StringTokenizer (frase);
return termini;
}//end_segmenta
}


package StemmerOracle;

import java.util.*;
import java.io.*;

class Ricerche {

  public Ricerche() {
  }

  public static String generaTrigrammiQuery(String frase, int cod_cli, int cod_sett){
     int n=3;

     StringTokenizer st=new StringTokenizer(frase);

     if (st.countTokens()<n+1)
      {String query = "SELECT CODICE,COUNT(*) FROM TERMS WHERE
           COD_CLIENTE = "+cod_cli+" AND COD_SETTORE = "+cod_sett+"AND
           CONTAINS (STEMMED_TERM_EN,'"+frase+"')>0 GROUP BY CODICE
           ORDER BY 2 DESC";
       return query;
      }//end_if

     else
```

```
    {
    int numeroTrigrammi=st.countTokens()-2;
    int numeroParole=st.countTokens();
    String[][] ngrammi = new String[numeroTrigrammi][n];
    String[] fraseIn= new String[st.countTokens()];

    int lung=0;
    while (st.hasMoreTokens())
      {String temp =st.nextToken();
       //System.out.println("temp = "+temp);
       fraseIn[lung]=temp;
       lung=lung+1;}//end_while
    int cc=0;
    while (cc<numeroTrigrammi)
    {
     ngrammi[cc][0]=fraseIn[cc];
     ngrammi[cc][1]=fraseIn[cc+1];
     ngrammi[cc][2]=fraseIn[cc+2];
     cc=cc+1;
    }//end_while_cc

    StringBuffer ngr = new StringBuffer();
    StringBuffer query = new StringBuffer();

    for (int k=0; k<n; k++)
      {ngr.append(ngrammi[0][k]+" ");}//end_for_k

    query.append("SELECT CODICE,COUNT(*) FROM (SELECT CODICE FROM
        TERMS WHERE COD_CLIENTE = "+cod_cli+" AND COD_SETTORE =
        "+cod_sett+" AND CONTAINS(STEMMED_TERM_EN,'"+ngr.toString()+"')>0
        ");

    int cont=1;

    while (cont < numeroTrigrammi)
      {StringBuffer tt = new StringBuffer();
       for (int i=0; i<n; i++)
        {tt.append(ngrammi[cont][i]+" ");}
       query.append("UNION ALL SELECT CODICE FROM TERMS WHERE
                COD_CLIENTE = "+cod_cli+" AND COD_SETTORE = "+cod_sett+"
                AND CONTAINS(STEMMED_TERM_EN,'"+tt.toString()+"')>0 ");
       cont=cont+1;
      }//end_while
      query.append(")GROUP BY CODICE ORDER BY 2 DESC");
      return query.toString();
    }//end_else
}//end_generaTrigrammiQuery
```

```
package StemmerOracle;

public int[ ][ ] eseguiQuery(String query) throws SQLException {

   int[ ][ ] risultato = new float[10][2];

   for (int j=0;j<10;j++){for (int k=0;k<2;k++) risultato[j][k]= 0;}


   Connection
conn=sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();
   PreparedStatement pstmt = conn.prepareStatement(query);
   ResultSet rs=pstmt.executeQuery();

   int i =0;
   while ((i<10)&&(rs.next()))
     {risultato[i][0]=rs.getInt(1);
      risultato[i][1]=rs.getInt(2);
      i=i+1;}//end_while

   rs.close();
   pstmt.close();

   return risultato;
}


package StemmerOracle;

public int[ ][ ] membership(int [ ] [ ] matchCodes, String fraseFinita)  {

StringTokenizer daTradurre=new StringTokenizer(fraseFinita);
     int numero=1;
     if (!(daTradurre.countTokens()==0))
       {if (daTradurre.countTokens()<n+1) {numero=1;}
        else {numero=daTradurre.countTokens()-2;}
       }
     for (int jj=0;jj<10;jj++)
       {matchCodes[jj][1]=matchCodes[jj][1]/numero;
       //System.out.println("matchCodes = "+matchCodes[jj][1]);
       }
     int ii=0;
     while ((ii<10)&&(!((matchCodes[ii][0]==0))&&(matchCodes[ii][1])>=0.4))
     {
      normalizzazione.cercaFrase(matchCodes[ii][0],matchCodes[ii][1]);
      ii=ii+1;
     }
}
```

```
package StemmerOracle;

public static void cercaFrase(float codice, float membership) throws SQLException {

      String fraseTrovata="";
       #sql iterator IteratoreRicerca (String term_en);

       IteratoreRicerca iteraRicerca;

        #sql iteraRicerca = {SELECT term_en FROM terms WHERE codice= :codice};
          while (iteraRicerca.next())
            {fraseTrovata=iteraRicerca.term_en();
             System.out.println(fraseTrovata);
             System.out.println(membership);}//end_while
       iteraRicerca.close();
}//end_cercaFrase
```

A p p e n d i c e   B

INTRODUZIONE AL LINGUAGGIO JAVA E AL SUO UTILIZZO CONNESSO AD UNA BASE DI DATI ORACLE

Per rendere più comprensibile l'analisi del prototipo realizzato, presentiamo brevemente le principali caratteristiche del linguaggio Java e degli strumenti utilizzati per la connessione con la base di dati Oracle.

B.1 Caratteristiche del linguaggio Java

Java è un linguaggio di programmazione orientato agli oggetti sviluppato dalla Sun Microsystems, particolarmente adatto all'uso su Internet. Tale linguaggio è nato dal C++, con l'omissione di alcuni elementi quali i puntatori e la gestione della memoria, quest'ultima gestita in modo automatico. Java eredita i concetti orientati agli oggetti dal C++ e da altri linguaggi come Smalltalk: si ricorda che la programmazione orientata agli oggetti organizza un programma come una serie di componenti chiamati *oggetti*, che sono indipendenti l'uno dall'altro ma seguono regole precise per comunicare tra loro.
Uno dei motivi del successo di questo linguaggio è la sicurezza riguardo all'esecuzione di un programma Java (detto *applet*) effettuata da una pagina Web: quando viene riscontrato un applet in una pagina Web (se l'utente dispone di un browser compatibile con Java), il browser lo preleva insieme al testo e alle immagini della pagina, quindi l'applet viene eseguito sul computer dell'utente. Tutto ciò può risultare pericoloso, perché l'esecuzione di un programma può portare a virus e altri problemi. Per evitare tali insidie, Java

fornisce diversi livelli di sicurezza: innanzitutto occorre considerare che per merito della mancanza di tutti i puntatori, fatta eccezione per una forma limitata di riferimenti agli oggetti, Java risulta un linguaggio molto più sicuro; un altro livello di sicurezza è il verificatore di *bytecode* (i programmi Java sono compilati in una serie di istruzioni chiamate *bytecode*), che controlla prima dell'esecuzione ogni bytecode per garantirne l'integrità. Oltre a queste misure generiche, ve ne sono molte altre che riguardano specificatamente gli applet. Per impedire a un programma di danneggiare l'unità disco del computer, secondo l'impostazione di default gli applet non possono aprire, leggere o scrivere file sul sistema dell'utente. Inoltre, poiché gli applet possono aprire nuove finestre, queste sono identificate da un logo di Java: in questo modo si evita che qualcuno possa falsificare una finestra di dialogo per l'inserimento della password.

Un altro motivo del successo del linguaggio è l'indipendenza dalla piattaforma, ovvero la capacità di un programma di poter essere eseguito su vari sistemi diversi. I tipi di variabili di Java hanno la stessa dimensione in tutte le piattaforme di sviluppo, perciò un intero è sempre della stessa dimensione, indipendentemente dal sistema usato per la sua compilazione. Inoltre, come è testimoniato dall'uso di applet nel Web, un file con estensione *.class* di Java composto da istruzioni in bytecode può essere eseguito su qualsiasi piattaforma senza alterazioni.

Occorre infine considerare che Java mette a disposizione gli strumenti per scrivere programmi *multithreading,* cioè programmi che possono eseguire diverse serie di istruzioni in modo concorrente.

B.2 Presentazione del linguaggio Java

In questo paragrafo verranno presentati gli elementi specifici del linguaggio Java che forniscono supporto per la programmazione orientata agli oggetti: le classi, i package e le interfacce.

B.2.1 Le classi

Una classe è un modello o un prototipo che definisce un tipo di oggetto. Una classe incorpora le caratteristiche di una famiglia particolare di oggetti e fornisce una rappresentazione generale di un oggetto, mentre un'istanza ne è la rappresentazione concreta.

Nella stesura di un programma Java, si progetta e si costruisce una famiglia di classi. Durante l'esecuzione del programma, si creano e si rimuovono, secondo le necessità, istanze di tali classi.

Il compito del programmatore in Java consiste nel creare (o utilizzare) una famiglia di classi che sia adeguata agli scopi del programma.

Una classe Java si compone essenzialmente di due parti: gli attributi (variabili istanza e di classe) ed il comportamento (metodi istanza e di classe).

B.2.2 Variabili istanza

Le variabili istanza definiscono gli attributi di un oggetto. La classe definisce il tipo dell'attributo, e ogni istanza contiene il proprio valore dell'attributo.

Una variabile istanza può essere impostata al momento della creazione dell'oggetto e rimanere immutata per tutta la sua vita, oppure essere modificata a piacere durante l'esecuzione del programma.

Una variabile è considerata d'istanza se è dichiarata al di fuori delle definizioni di metodo.

B.2.3 Variabili di classe

Le variabili di classe sono visibili a tutta una classe e a tutte le sue istanze; possono essere considerate più generali delle variabili istanza.

Le variabili di classe possono servire per comunicazioni tra istanze della stessa classe o per tenere traccia di uno stato globale, relativo ad una famiglia di oggetti. Per dichiarare una variabile di classe, si premette la parola *static* alla dichiarazione.

B.2.4 Metodi istanza

Un metodo di istanza è una funzione definita all'interno di una classe, che può agire sulle istanze della classe. Non sempre un metodo agisce su un solo oggetto; gli oggetti comunicano fra loro per mezzo dei metodi. Una classe o un oggetto può richiamare i metodi di un'altra classe od oggetto per comunicare cambiamenti avvenuti nell'ambiente o per chiedere all'elemento chiamato di modificare il proprio stato.

La definizione di un metodo comprende quattro parti: il nome del metodo, il tipo di oggetto o il tipo primitivo del valore che il metodo restituisce, un elenco di parametri ed il corpo del metodo.

Se il metodo restituisce un valore, il suo corpo deve contenere l'istruzione *return,* che provvede alla restituzione del valore. Le variabili passate ad un metodo come parametri di tipo oggetto sono passate per riferimento, il che significa che le operazioni svolte su di esse nel metodo hanno effetto sugli oggetti originali (i tipi primitivi, invece, sono passati per valore).

B.2.5 Metodi di classe

I metodi di classe sono disponibili per ogni istanza della classe e possono essere messi a disposizione di altre classi; di conseguenza, alcuni metodi di

classe si possono richiamare indipendentemente dall'esistenza di un'istanza della classe relativa.

Un metodo di classe si definisce facendolo precedere dalla parola *static*, come per le variabili di classe.

In generale, i metodi che operano su un oggetto specifico dovrebbero essere definiti come metodi di istanza, mentre i metodi che forniscono funzioni di utilità generale, ma non agiscono direttamente sulle istanze, sono candidati al ruolo di metodi di classe.

B.2.6 Metodi costruttori

La definizione di una classe può contenere, oltre ai metodi normali, dei metodi costruttori. Un *metodo costruttore* non può essere richiamato direttamente, come un metodo normale, ma viene richiamato automaticamente da Java.

Quando si crea un'istanza di una classe mediante l'operatore *new*, Java compie diverse operazioni, fra le quali richiamare il metodo costruttore della classe (che può essere uno fra diversi). Se in una classe non sono definiti metodi costruttori, l'oggetto viene comunque generato, ma può essere necessario impostare le variabili istanza o richiamare altri metodi che inizializzino l'oggetto.

Definendo metodi costruttori per una classe, si possono impostare i valori iniziali delle variabili istanza, richiamare certi metodi sulla base di quelle variabili, richiamare metodi di altri oggetti o calcolare le proprietà iniziali di un oggetto. Si possono definire costruttori omonimi (*overloading* di costruttori), come per i metodi normali, così da guidare la fase di inizializzazione di un oggetto in base agli argomenti passati a *new*.

La differenza principale fra costruttori e metodi normali è che i costruttori hanno sempre lo stesso nome della classe e non restituiscono un valore.

B.2.7 Metodi conclusivi

Tale metodo è richiamato prima che un oggetto venga rimosso e la sua memoria liberata. La classe *Object* definisce un metodo conclusivo che non esegue alcuna operazione. Per definire un metodo conclusivo per una classe si ridefinisce il metodo finalize() con una particolare segnatura.

Il metodo finalize() può essere richiamato esplicitamente in qualunque momento, come ogni metodo normale, ma la sua chiamata non provoca la rimozione dell'oggetto.

I metodi conclusivi sono sfruttati soprattutto per ottimizzare la rimozione di un oggetto: ad esempio, per eliminare riferimenti ad altri oggetti, liberare risorse esterne e in generale tutto ciò che può facilitare l'eliminazione dell'oggetto.

B.2.8 Variabili locali

Le variabili locali sono dichiarate e utilizzate nelle definizioni di metodi, ad esempio per i contatori nei cicli, come variabili temporanee o per valori rilevanti solo all'interno di un metodo particolare. Le variabili locali si possono utilizzare anche all'interno dei blocchi. Al termine dell'esecuzione di un metodo (o di un blocco), le sue variabili locali e i relativi valori scompaiono.

B.2.9 Costanti

Le costanti sono utili per definire valori disponibili a tutti i metodi di un oggetto, ossia per assegnare nomi descrittivi a valori invariabili, di pertinenza di un oggetto.

In Java si possono creare costanti solo per le variabili istanza e di classe, e non per le variabili locali. Per dichiarare una costante, si utilizza la parola *final* prima della dichiarazione del tipo, e se ne indica il valore.

B.2.10 Le interfacce

Le interfacce sono classi astratte lasciate completamente senza implementazione, il che significa che nelle classi non è stato implementato nessun metodo. Inoltre, i dati membro delle interfacce sono limitati alle variabili statiche *final*, cioè costanti.

I vantaggi dell'utilizzo delle interfacce sono gli stessi offerti dall'utilizzo delle classi astratte. Le interfacce costituiscono un mezzo per definire protocolli per una classe, senza preoccuparsi dei dettagli dell'implementazione.

Un aspetto importante delle interfacce è la possibilità per una classe di implementarne diverse, simulando il concetto di ereditarietà multipla. La differenza principale tra l'ereditarietà di diverse interfacce e la vera ereditarietà multipla è che la prima permette di ereditare solo le descrizioni dei metodi, non le implementazioni. Se una classe implementa diverse interfacce, deve fornire tutte le funzionalità per i metodi definiti in queste ultime.

B.2.11 I package

I *package* vengono utilizzati per classificare e raggruppare le classi. Se un'istruzione package appare in un file sorgente in Java, deve trovarsi all'inizio del file, prima di ogni altra istruzione: in questo modo tutte le classi dichiarate all'interno dello stesso package saranno raggruppate insieme.

I package possono essere ulteriormente strutturati in una gerarchia per certi versi analoga alla gerarchia ereditaria, dove ogni livello normalmente rappresenta un gruppo più ristretto e specifico di classi (la stessa libreria di classi Java è organizzata in questo modo).

L'utilizzo dei package risolve quindi eventuali conflitti fra nomi di classi che si sono creati a causa della mancanza di questa organizzazione. Tutte le classi del prototipo appartengono a tale package.

Occorre considerare che quando ci si riferisce ad una classe per nome nel proprio codice Java, si utilizza un package. Solitamente non ci si rende conto di questo fatto, perché molte delle classi utilizzate più frequentemente nel sistema si trovano in un package che il compilatore Java importa automaticamente, denominato java.lang. Per classi non appartenenti a questo package, Java consente di far precedere qualsiasi nome di classe con il nome del package in cui sono state definite, formando un unico riferimento. Poiché tale procedimento risulta scomodo (soprattutto per nomi di package molto lunghi), Java consente di "importare" i nomi delle classi desiderate nel proprio programma, dopodiché è possibile farvi riferimento senza alcun prefisso, come per le classi java.lang. I nomi delle classi desiderate vengono importati tramite l'istruzione: *import* *<nome del package>.<nomeclasse>*. Tutte le istruzioni *import* devono trovarsi dopo *package* ma prima di qualsiasi definizione di classe. Spesso al posto di *<nomeclasse>* viene utilizzato l'asterisco (*) per consentire l'utilizzo di tutte le classi del package, ma non quelle dei sottopackage: infatti per importare tutte le classi di una complessa gerarchia di package, occorre utilizzare esplicitamente *import* per ogni livello della gerarchia.

B.2.12 Alcune parole chiave

"extends": è utilizzata nelle definizioni delle classi per indicare se la classe che si sta dichiarando è sottoclasse di un'altra.

"new": per creare un nuovo oggetto, si utilizza *new*, seguito dal nome della classe di cui si vuole creare un'istanza e dalle parentesi tonde. Le parentesi sono importanti e non possono essere omesse; il loro contenuto può essere vuoto, nel qual caso viene creato il tipo più semplice di oggetto, oppure consistere di argomenti che determinano i valori iniziali dell'oggetto. Il

numero e il tipo di tali argomenti sono definiti da ogni classe per mezzo dei metodi costruttori.

Quando si crea un'istanza di una classe mediante *new*, Java esegue tre operazioni in sequenza:

- alloca la memoria per l'oggetto;
- inizializza le variabili istanza dell'oggetto, ai valori iniziali indicati oppure ad un valore predefinito (0 per i numeri, *null* per gli oggetti, *false* per i booleani, `\0` per i caratteri);
- richiama il metodo costruttore della classe.

"this": tale parola chiave si riferisce all'oggetto corrente, e può trovarsi ovunque possa trovarsi un riferimento a un oggetto: nella notazione puntata per citare le variabili istanza dell'oggetto, come argomento di un metodo, come valore restituito dal metodo presente, e così via.

L'omissione di *this* per le variabili istanza è possibile se non esistono variabili con lo stesso nome nell'ambito locale. Poiché *this* è un riferimento all'istanza corrente di una classe, lo si può utilizzare solo nel corpo di un metodo istanza.

I metodi di classe, cioè quelli dichiarati con la parola *static*, non possono utilizzare *this.*

B.3 La classe "Vector"

In Java non esistono puntatori. Dato che gli elenchi a collegamento dinamico e le code sono implementati utilizzando i puntatori, non è possibile creare direttamente queste due strutture di dati in Java. Al loro posto è prevista la classe Vector che gestisce le occasioni in cui è necessario memorizzare dinamicamente gli oggetti.

Un aspetto positivo è che la classe Vector contribuisce a mantenere semplice il linguaggio; l'aspetto negativo principale è che la classe Vector limita i programmatori nell'utilizzo di programmi più sofisticati.

È possibile impostare la capacità dell'oggetto Vector sulle dimensioni necessarie prima di inserire un grande numero di oggetti, in questo modo si riduce la necessità di riallocare l'elenco per incrementarlo.
La capacità di memorizzazione iniziale e l'incremento della capacità possono essere entrambe specificate nel costruttore. La capacità viene incrementata automaticamente ma è possibile aumentarla di un numero minimo specifico di elementi.
Al Vector possono essere aggiunti elementi utilizzando i metodi *addElement( )* (aggiunge l'elemento passato come parametro alla fine del Vector) oppure *insertElementAt( )* (aggiunge l'elemento passato come parametro nella posizione specificata dai parametri).
Gli elementi da memorizzare nel Vector devono derivare dal tipo Object. È possibile accedere direttamente agli elementi del Vector tramite i metodi *elementAt( )* (si posiziona all'elemento di posizione specificata dal parametro passato al metodo), *firstElement( )* (si posiziona sul primo elemento del Vector), *lastElement( )* (si posiziona sull'ultimo elemento del Vector).
Un altro metodo per accedere agli elementi del Vector si ottiene utilizzando l'oggetto Enumeration.
Enumeration è un'interfaccia che specifica una serie di metodi utilizzati per enumerare, vale a dire iterare, un elenco. Un oggetto che implementa questa interfaccia può essere utilizzato per iterare un elenco una volta sola in quanto l'oggetto Enumeration viene consumato dall'utilizzo.
L'interfaccia Enumeration specifica solo due metodi: *hasMoreElements( )* (restituisce *true* se sono presenti altri elementi nell'oggetto) e *nextElement( )* (restituisce il prossimo elemento dell'oggetto Enumeration.
Vediamo ora un esempio in cui viene utilizzato l'oggetto Enumeration per accedere agli elementi di un Vector (*v*):

```
for (Enumeration e=v.elements( ); e.hasMoreElements( ); )
        {System.out.println(e.nextElement( ));}
```

Si può vedere che un oggetto Enumeration, che rappresenta tutti gli elementi nel Vector, viene creato e restituito dal metodo *elements( )* di Vector. Mediante il metodo *hasMoreElements( )* di Enumeration il ciclo controlla se vi sono ulteriori elementi nel Vector da elaborare; viene ottenuto l'elemento successivo da elaborare utilizzando il metodo *nextElement( )* di Enumeration.

B.4 Modificatori di accesso

Il modificatore di accesso predefinito specifica che solo le classi all'interno dello stesso package hanno accesso alle variabili e ai metodi di una classe. I membri di una classe con accesso predefinito hanno visibilità limitata alle altre classi all'interno dello stesso package.

Non esiste una parola chiave per dichiarare il modificatore di accesso predefinito, che viene applicato automaticamente in assenza di altri modificatori di accesso.

B.4.1 Public

Il modificatore di accesso *public* specifica che le variabili e i metodi di una classe sono accessibili a chiunque, sia all'interno che all'esterno della classe. Ciò significa che i membri *public* di una classe hanno visibilità globale e che qualsiasi altro oggetto può accedervi.

B.4.2 Protected

Il modificatore di accesso *protected* specifica che i membri di una classe sono accessibili solo ai metodi di quella classe e delle relative sottoclassi. Ciò significa che i membri *protected* di una classe hanno visibilità limitata alle sottoclassi.

B.4.3 Private

Il modificatore di accesso *private* è il più restrittivo e specifica che i membri di una classe sono accessibili solo alla classe in cui sono definiti. Ciò significa che nessun altra classe, incluse le sottoclassi, ha accesso ai membri *private* di una classe.

B.4.4 Synchronized

Tale modificatore viene utilizzato per specificare che un metodo è a *thread protetto*, vale a dire che in un metodo *synchronized* è concesso un solo percorso di esecuzione alla volta. In un ambiente multithreading come quello di Java, è possibile che nello stesso codice vengano eseguiti contemporaneamente diversi percorsi di esecuzione. Il modificatore *synchronized* modifica questa regola, permettendo a un solo thread di accedere a un metodo in un dato momento e obbligando gli altri thread ad aspettare il loro turno.

B.4.5 Native

Il modificatore *native* viene utilizzato per identificare i metodi con implementazione nativa. Esso informa il compilatore di Java che l'implementazione di un metodo si trova in un file esterno C. Per questo motivo le dichiarazioni dei metodi *native* sono diverse da quelle degli altri metodi di Java: non hanno corpo.

La dichiarazione del metodo *native* termina semplicemente con un punto e virgola, senza parentesi graffe che racchiudono il codice Java. Questo perché i metodi nativi vengono implementati nel codice C, che risiede in file sorgenti C esterni.

B.5 Garbage Collector

Come già menzionato in apertura di capitolo, Java non offre al programmatore la possibilità di deallocare gli oggetti in modo esplicito. Quando un oggetto non è più referenziato, ovvero quando non ci sono più oggetti che lo utilizzano, può essere distrutto.

Il garbage collector è una speciale routine di sistema che scandisce il Java Heap (zona di memoria che contiene gli oggetti) liberando la memoria occupata dagli oggetti non più referenziati. A causa però del continuo processo di allocazione e deallocazione, aumenta la frammentazione della memoria.

Uno svantaggio rappresentato da questa routine è la diminuzione delle prestazioni, causato dal fatto che in Java il garbage collector è un vero e proprio thread in esecuzione parallelamente al programma: anche se la sua priorità è minima (in Java le priorità vanno da 1 a 10) e quindi si avvia solo quando non ci sono altri thread attivi, è un processo da considerare in termini di tempo di CPU.

Il garbage collector è considerato un processo Demon in quanto non è gestito dal programmatore ed è sempre presente in background.

Ogni algoritmo di garbage collection deve fare tre cose:

- Determinare l'oggetto che deve essere eliminato (garbage detection);
- Liberare la corrispondente zona di memoria e renderla disponibile al programma;
- Combattere la frammentazione dello heap.

B.5.1 Garbage detection

La garbage detection deve determinare di quali oggetti possiede un riferimento: per questo motivo si definiscono le *root* (radici) che sono gli oggetti persistenti dell'applicazione ovvero quelli sempre presenti durante l'esecuzione del programma. Nel caso di una applicazione Java una *root* è

rappresentata dall'oggetto a cui appartiene il metodo *main()*. Quali altri oggetti siano considerati *root* dipende dall'implementazione del garbage collector: usualmente si considerano *root* anche quegli oggetti il cui *scope* coincide con il corpo del main.

Ad ogni altro oggetto si assegna la proprietà di *reachability* (raggiungibilità) dalla root. Un oggetto si dice raggiungibile se esiste un percorso di riferimenti dalla root, che permette di indirizzare l'oggetto stesso. È evidente a questo punto che gli oggetti raggiungibili sono quelli referenziabili e quindi vivi, mentre gli altri sono quelli da eliminare. Inoltre, oggetti raggiungibili da oggetti vivi sono pure vivi e quindi da conservare.

Gli algoritmi più comuni che permettono di determinare la proprietà di raggiungibilità di un oggetto sono due: *reference counting e tracing*.

Reference counting: ad ogni nuovo oggetto creato, viene associato un contatore inizializzato ad 1. Il contatore memorizza, in ogni istante, il numero di riferimenti all'oggetto corrispondente. Se viene creato un nuovo riferimento ad un oggetto, il relativo contatore viene incrementato, mentre se l'oggetto esce dal suo *scope* oppure al riferimento è associato un nuovo valore, il contatore viene decrementato. Per determinare se un oggetto debba essere eliminato oppure no, il garbage collector dovrà semplicemente verificare se il contatore relativo all'oggetto è nullo: se è nullo, l'oggetto non possiede riferimenti e dunque può essere eliminato.

Tracing: consiste nel prendere come riferimento gli oggetti *root* e, partendo da essi, marcare tutti gli oggetti collegati tra loro da un riferimento. Per marcare gli oggetti si può utilizzare un bit oppure una mappa distinta. Al termine di questa operazione, gli oggetti raggiungibili saranno quelli marcati, mentre gli altri non sono più referenziati e quindi dovranno essere eliminati.

B.5.2 Tecniche di deframmentazione

Altro compito del garbage collector è quello di combattere la frammentazione dello heap. Essa è la conseguenza del continuo susseguirsi delle operazioni di allocazione e deallocazione di memoria, che si verificano durante l'esecuzione di un programma. Se il garbage collector non prendesse adeguate contromisure, un oggetto potrebbe non essere allocato perché mancano una serie di locazioni contigue che lo contengano. Le tecniche di deframmentazione più comunemente utilizzate sono il *compacting* (compattazione) ed il *copying* (copia). Entrambe consistono fondamentalmente nello spostamento degli oggetti.

Compacting: tale metodo sposta tutti gli oggetti validi verso un estremo dello heap in modo tale che l'altra parte dello heap venga ad essere costituita da memoria valida per l'allocazione degli oggetti. Tutti i riferimenti all'oggetto spostato devono poi essere aggiornati alla nuova locazione. Quest'ultimo passo può costituire un problema di efficienza per cui si rende il procedimento più semplice aggiungendo un livello di indirizzamento indiretto.

Copying: tale metodo consiste nel copiare tutti gli oggetti in una nuova area di memoria. Quando gli oggetti sono spostati nella nuova area, vengono posizionati l'uno di seguito all'altro. A questo punto la vecchia area di memoria è libera. Il vantaggio di questa tecnica consiste nella sua perfetta integrazione con il metodo di *tracing* in quanto non appena un oggetto viene marcato, può essere copiato nella nuova area. Gli oggetti che non vengono marcati resteranno nell'area vecchia e quindi automaticamente eliminati.

B.6 Gestione delle eccezioni

Nella maggior parte dei casi le eccezioni vengono utilizzate per riportare condizioni di errore: esse costituiscono uno strumento per informare che vi sono degli errori e un modo per gestirli.

Per rispondere a un'eccezione, la chiamata al metodo che l'ha prodotta deve trovarsi all'interno di un blocco *try*, che è un blocco di codice che inizia con la parola chiave *try* seguita da una parentesi graffa aperta e da una chiusa. Tutti i blocchi *try* sono associati a uno o più blocchi *catch*.

Se un metodo deve intercettare le eccezioni generate dai metodi che richiama, le chiamate devono essere inserite all'interno di un blocco try. Se viene generata un'eccezione, questa viene gestita in un blocco *catch*. Diversi blocchi *catch* gestiscono diversi tipi di eccezioni.

Quando un metodo qualsiasi nel blocco *try* genera un qualsiasi tipo di eccezione, l'esecuzione del blocco *try* cessa e il controllo del programma passa immediatamente al blocco *catch* associato. Se questo è in grado di gestire quel tipo di eccezione, assume il controllo dell'esecuzione, altrimenti l'eccezione viene passata a chi ha richiamato il metodo.

In un'applicazione, questo processo continua finché un blocco *catch* intercetta l'eccezione o finché questa raggiunge il metodo *main ()* senza essere stata intercettata, facendo terminare l'applicazione.

B.6.1 Blocchi catch multipli

In alcuni casi, può succedere che un metodo debba intercettare diversi tipi di eccezioni. In Java è possibile utilizzare diversi blocchi *catch*, ognuno dei quali deve specificare un tipo diverso di eccezione.

Quando nel blocco *try* viene generata un'eccezione, questa viene intercettata dal primo blocco *catch* del tipo appropriato. Viene eseguito un solo blocco *catch* di una determinata serie. Si noti che i blocchi *catch* assomigliano alle dichiarazioni dei metodi.

L'eccezione intercettata in un blocco *catch* è un riferimento locale al vero oggetto eccezione, utilizzabile per determinare che cosa ha generato inizialmente l'eccezione.

B.6.2 La clausola finally

Java introduce un nuovo concetto nella gestione delle eccezioni: la clausola *finally*, che contraddistingue un blocco di codice che viene sempre eseguito. Ecco un esempio:

```
import java.io.*;
import java.lang.Exception;
public class MultiThrow {
public static void main (String[] args) {
try {
alpha();
}
catch (Exception e) {
System.out.println (“Intercettata eccezione”);
}
finally () {
System.out.println (“Finalmente. ”);
}
}
}
```

Nell’esecuzione normale, vale a dire quando non vengono generate eccezioni, il blocco *finally* viene eseguito immediatamente dopo il blocco *try*. Quando viene generata un’eccezione, il blocco *finally* viene eseguito prima che il controllo passi al metodo che ha effettuato la chiamata. Se *alpha()* genera un’eccezione, questa viene intercettata nel blocco *catch* e successivamente viene eseguito il blocco *finally*. Se *alpha()* non genera alcuna eccezione, il blocco *finally* viene eseguito dopo il blocco *try*. Anche se viene eseguita solo

una parte del codice di un blocco *try*, il blocco *finally* viene comunque eseguito.

B.7 Integrazione tra Java e ORACLE 8i

Descriviamo ora gli strumenti che sono stati utilizzati per accedere, attraverso il programma Java di normalizzazione, alle tabelle create sulla base di dati.

B.7.1 JDBC

JDBC (Java DataBase Connectivity) è una libreria di classi ed interfacce pensate per permettere la connessione a sorgenti di dati, l'invio di comandi di interrogazione o di aggiornamento e l'elaborazione dei dati prodotti come risultato di tali comandi. Normalmente le sorgenti di dati sono DBMS (Data Base Management Systems) relazionali e i comandi sono istruzioni SQL. Lo scopo principale della JDBC API è quello di permettere l'implementazione di interfacce alternative per l'accesso a database. Queste interfacce alternative sono di più alto livello, quindi in generale più semplici da usare per il programmatore o l'utente, ma utilizzano JDBC come base per l'accesso alla sorgente di dati. Una di queste interfacce è SQLJ (ved. § B.9.2). Il funzionamento di JDBC si basa sulla presenza di librerie di classi Java che ne implementano l'interfaccia per le sorgenti di dati. Queste librerie di classi vengono chiamate JDBC Driver. i compiti di un JDBC Driver sono:

- Stabilire una connessione con la sorgente dei dati
- Inviare comandi alla sorgente di dati. Di solito i comandi sono istruzioni SQL.
- Permettere la ricezione e l'elaborazione dei risultati dei comandi.

B.7.2 SQLJ

SQLJ è una collezione di specifiche definita da un consorzio di società nel campo dei database (per la precisione : JavaSoft, Informix, Oracle, IBM, Cloudscape, Microsoft, Sybase, Tandem e XDB) riguardanti un tipo di accesso e di interazione applicativa con i database. Presentiamone le principali funzionalità utilizzate nell'implementazione in codice Java degli algoritmi di normalizzazione e di ricerca nella base di dati Oracle. Maggiori dettagli su SQLJ possono essere reperiti in [24].

SQLJ è uno strumento che permette di includere istruzioni SQL statiche in un programma scritto in linguaggio Java. Le specifiche introdotte vengono analizzate dal componente *SQLJ Translator*. Il *SQLJ Translator* converte questo codice un puro Java e verifica la correttezza sintattica delle istruzioni SQL introdotte. Il risultato di questa operazione di pre-compilazione è un file (con estensione **.java**) che può quindi essere compilato ed eseguito interagendo con una base di dati Oracle attraverso un driver JDBC impostato.

I maggiori vantaggi portati dall'utilizzo di SQLJ riguardano:

- la compattezza delle istruzioni che possono essere inserite nel codice Java;
- la sicurezza nella programmazione grazie alla verifica di correttezza svolta nella fase di pre-compilazione;
- standardizzazione del linguaggio (e conseguente portabilità del codice) per tutti i maggiori produttori di DBMS.

B.7.2.1 Le primitive SQLJ

In un programma Java tutte le primitive SQLJ iniziano con il prefisso **#sql**. I costrutti SQLJ possono essere classificati in due categorie: *declarative* e *executable*.

Le istruzioni *declarative* introducono nel codice Java le dichiarazioni degli *iterator* (ved. § B.8.2.2) e dei *connection context* (specifiche riguardanti il driver JDBC utilizzato per la connessione alla base di dati).

I costrutti *executable* contengono istruzioni SQL all'interno di parentesi graffe, ad esempio:

#sql {update tabella set campo1=:variabile1 **where** campo2=:variabile2**};**

B.7.2.2 Gli oggetti "iterator"

In un programma Java i risultati restituiti da una interrogazione sulla base di dati possono essere rappresentati da oggetti chiamati *iterator*. Questi oggetti sono istanze della classe "iterator" e vengono dichiarati all'interno del codice con il seguente costrutto declarative:

#sql iterator <nome iteratore>;

SQLJ fornisce due tipi di iterator:

- *named iterator*: l'istanza dell'oggetto iterator conterrà in questo caso la specifica sia dei nomi delle colonne estratte dall'interrogazione che del tipo di dato da esse contenuto. Ad esempio nel nostro programma abbiamo utilizzato

  #sql iterator Iteratore (int codice**, String** term_en**);**

  Iteratore itera; //creazione dell'istanza di Iteratore

  in cui inserire il codice e la frase in inglese contenuti nella tabella TERMS.

  È da notare che un'istruzione come questa, nel file **.java** restituito dal SQLJ Translator, viene espansa nel seguente insieme di istruzioni JDBC:

  ```
  class Iteratore
  extends sqlj.runtime.ref.ResultSetIterImpl
  implements sqlj.runtime.NamedIterator
  {
  public Iteratore(sqlj.runtime.profile.RTResultSet
  resultSet) throws java.sql.SQLException
  {
  ```

```
  super(resultSet);
  codiceNdx = findColumn("codice");
  term_enNdx = findColumn("term_en");
 }
 public int codice()
   throws java.sql.SQLException
 {
   return resultSet.getIntNoNull(codiceNdx);
 }
 private int codiceNdx;
 public String term_en() throws java.sql.SQLException
 {
   return resultSet.getString(term_enNdx);
 }
 private int term_enNdx;
}
```

da questo esempio si può capire come sia d'aiuto lo strumento SQLJ nella stesura di programmi Java che instaurano connessioni con basi di dati.

Durante la fase di SQLJ Translation la dichiarazione vista genera una classe Java di nome Itera. Ogni istanza di questa classe, creata con l'istruzione

Iteratore itera;

conterrà due speciali metodi: **codice( )** e **term_en( )**. Essi restituiscono rispettivamente un intero ed una stringa e sono utilizzati per accedere ai dati estratti per mezzo di un'interrogazione del tipo

#sql myIter = {select codice,term_en **from** terms**};**

Si avrà così

int cod = itera.codice( );

String frase = itera.term_en( );

- *positional iterator*: i positional iterator specificano soltanto il tipo di dato di ogni colonna estratta. Ad esempio

  #sql iterator Itera2 (String);

Anche in questo caso viene generata una classe Java (chiamata Itera2) ma le istanze di essa non avranno i metodi speciali di accesso alla colonna.

I dati contenuti nelle colonne, qualora si utilizzino positional iterator, possono essere estratti tramite l'istruzione SQL **fetch into**.

L'unica differenza rispetto al named iterator consiste quindi nella modalità di accesso ai dati provenienti dal risultato dell'interrogazione sulla base di dati.

Come si può vedere dai listati nell'appendice A, nel nostro lavoro abbiamo utilizzato soltanto named iterator.

RIFERIMENTI BIBLIOGRAFICI

[1] C. J. van Rijsbergen. *Information Retrieval.* Butterworths & Co Publishers, 1979.

[2] Michael Brundin. Probabilistic, Vector Space and Fuzzy Logic Information Retrieval Systems. Background Report – LIS 535, 1997.

[3] Ogawa Yasushi, Mano Hiroko, Narita Masumi, Honma Sakiko. Structuring and expanding queries in the probabilistic model. *Proceedings of The Eighth Text REtrieval Conference (TREC-8), NIST Special Publication*, 1999.

[4] Ian H. Witten, Alistar Moffat, Timothy C. Bell. *Managing Gigabytes. Compressing and Indexing Documents and Images.* Second Edition, Morgan Kaufmann Publishers Inc., 1999.

[5] Christos Faloutsos, W. Oard Douglas. A Survey of Information Retrieval and Filtering Methods. Department of Computer Science, Electrical Engineering Department, University of Massachusetts.

[6] C. De Loupy, P. Bellot, M. El-Bèze, P. F. Marteau. Query Expansion and Classification of Retrieved Documents. Laboratoire d'Informatique d'Avignon (LIA).

[7] Kenney Ng. A Maximum Likelihood Ratio Information Retrieval Model. *Proceedings of The Eighth Text REtrieval Conference (TREC-8), NIST Special Publication*, 1999.

[8] A. Rungsawang, A. Tangpong, P. Laohawee, T. Khampachua. Novel Query Expansion Technique using Apriori Algorithm. *Proceedings of The Eighth Text REtrieval Conference (TREC-8), NIST Special Publication*, 1999.

[9] D. Hull. Using Statistical Testing in the Evaluation or Retrieval Experiments. *Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1993.

[10] K, Jacquelynn Kud, Paul Dixon. Oracle at Trec8: A Lexical Approach. *Proceedings of The Eighth Text REtrieval Conference (TREC-8), NIST SpeciaPublication*, 1999.

[11] *Oracle8i interMedia Text release 8.1.6.* Technical Overview. Oracle Corporation, Oracle 8i documentation..

[12] Erik Horstkotte. Fuzzy Logic Overview. Togai InfraLogic, Inc. http://www.austinlinks.com/Fuzzy/overview.html

[13] James F. Brule'. Fuzzy Systems – A Tutorial. (c) Copyright James F. Brule', 1985, http://www.austinlinks.com/Fuzzy/tutorial.html

[14] Lofti Zadeh. Fuzzy sets.*Information and Control, Vol. 8*, 1965.

[15] R. Stevens. Automatic Indexing: A State-of-the-Art Report. NDI Monograph 91, 1970.

[16] M. Porter. An Algorithm for Suffix Stripping. *Program, Vol. 14*, 1980.

[17] J. B. Lovins. Development of a Stemming Algorithm. *Mechanical Translation and computation Linguistics 11*, 1968.

[18] K. Andrews. The Development of a Fast Conflation Algorithm for English. Computer Laboratory, University of Cambridge, 1971.

[19] A. E. Petrarca, W. M. Lay. Use of an automatically generated authority list to eliminate scattering caused by some singular and plural main index terms. *Proceedings of the American Society for Information Science 6*, 1969.

[20] R. T. Dattola. FIRST: Flexible Information Retrieval System for Text. Xerox Corporation, 1975.

[21] D. S. Colombo, R. T. Niehoff. Final report on improved access to scientific and technical information through automated vocabulary switching. NSF Grant No. SIS75-12924 to the National Science Foundation.

[22] J. L. Dawson. Suffix Removal and Word Conflation. *ALLC Bulletin, Michaelmas*, 1974.

[23] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, N.J., 1992.

[24] Brian Wright. *SQLJ User's Guide and Reference.* Oracle Corporation, Oracle 8i documentation.