Università degli studi
di Modena e Reggio Emilia

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Interrogazione Efficiente di Documenti XML Temporali e una sua applicazione in Ambito Normativo

Enrico Ronchetti

Tesi di Laurea

Relatore:
Chiar.mo Prof. Paolo Tiberio

Controrelatore:
Chiar.mo Prof. Sonia Bergamaschi

Correlatori:
Dott. Federica Mandreoli
Ing. Riccardo Martoglia

Anno Accademico 2002/2003

Ringraziamenti

Ringrazio Riccardo e Federica per la loro costante assistenza e collaborazione.
Ringrazio Sara per essermi stata sempre vicina e per l'infinita pazienza.
Un grazie particolare alla mia famiglia per avermi dato fiducia e la possibilità di fare tutto questo.

Parole chiave

xml temporale
structural join
database temporale
query processor
legge

Indice

Introduzione **1**

I Introduzione **5**

1 Database temporali **7**
1.1 Introduzione ai database temporali 7
1.1.1 Modelli del tempo 8
1.1.2 Le dimensioni temporali 9
1.1.3 TSQL2: un linguaggio di interrogazione per dati temporali 10

2 Introduzione ad XML **15**
2.1 Struttura logica e presentazione 15
2.1.1 Tipologie di documenti e DTD 17
2.1.2 XML Schema 17
2.1.3 Parsing di un documento XML 18
2.1.4 Linguaggi per l'interrogazione di dati XML 21

3 Gli structural join **27**
3.1 Struttura dei dati ed algoritmi per gli structural join 27
3.2 Algoritmi in letteratura 30
3.2.1 Funzionamento di TSGeneric 32

4 Introduzione ai testi normativi ed al modello adottato **37**
4.1 Riferimenti normativi e dinamica dei testi normativi 37
4.1.1 Nozioni preliminari 38
4.1.2 Storia del testo normativo (testo normativo virtuale) . . . 38
4.1.3 Testo normativo vigente (in un momento x) 38
4.1.4 Storia di un disposizione normativa (disposizione normativa virtuale) 40
4.1.5 Ambito di applicabilità 41
4.1.6 Nessi normativi 41
4.1.7 Designazione normativa 42

4.1.8 I nessi normativi: loro distinzione in base agli effetti 44
4.2 Modello di atto normativo 51
4.2.1 Progetto NIR 51
4.2.2 Struttura 52
4.2.3 Legge 55
4.2.4 Articolato 55
4.2.5 Elementi che generano versioni 56
4.2.6 Formulazione di query sul modello adottato 59
4.2.7 Una architettura esistente 60

II Interrogazione Efficiente di Documenti XML Temporali 65

5 Architettura e strutture di un Temporal XML Query Processor 67
5.1 Una nuova architettura per la risoluzione di query su documenti XML temporali 68
5.2 Estensioni relative al modello XML temporale 70
5.3 Strutture necessarie 72
5.4 Possibile dinamica di un atto normativo 73
5.5 Parsing di una legge e generazione delle tuple 74
5.5.1 Le versioni e gli attributi strutturali 77
5.6 Memorizzazione delle strutture Tupla 83

6 Un algoritmo per la valutazione di query su XML temporale 85
6.1 Elaborazione dei vincoli su XML temporale 86
6.1.1 Vincolo strutturale 86
6.1.2 Vincolo testuale 88
6.1.3 Vincolo temporale 89
6.2 Funzionamento di TwigStack 90
6.2.1 Valutazioni sull'elaborazione 96
6.3 La generazione delle tuple di interesse 97
6.3.1 L'algoritmo di generazione GetNext2 98
6.3.2 L'algoritmo di generazione GetNextText() 103
6.3.3 La consistenza nelle dimensioni temporali 104
6.4 Metodi di accesso e generazione dei risultati 107

7 Il prototipo software 111
7.1 Il Database System 111
7.1.1 Cosa è Berkeley DB 112
7.1.2 Servizio di Data Access 114
7.1.3 Cosa non è Berkeley DB 114
7.2 I package utilizzati 116

7.2.1 Le strutture per l'elaborazione: il package Strutture 116
7.2.2 Memorizzazione dati: il Package Berkeley 118
7.2.3 La generazione delle leggi : il package GeneraLeggiXML . 123
7.2.4 L'inverted Index: il Package Lucene 123
7.2.5 Parsing di un documento XML: package Sax 124
7.2.6 Algoritmi di generazione delle tuple: package GetNext . . 125
7.2.7 Algoritmi per gli structural join: package Join 126

8 Le prove sperimentali e i risultati ottenuti 129
8.1 Le query 131
8.2 I risultati delle interrogazioni 132
8.3 Confronto prestazioni 136

Conclusioni e sviluppi futuri 139

III Appendici 143

A Il codice sviluppato 144
A.1 XML Schema 144
A.2 Package Strutture 147
A.2.1 Classe Strutture.Tupla 147
A.2.2 Classe Strutture.Query 152
A.2.3 Classe Strutture.ListaTuple 155
A.3 Package Berkeley 157
A.3.1 Classe Berkeley.AssociaDb 157
A.3.2 Classe Berkeley.DBUtility 160
A.3.3 Classe Berkeley.SKC 170
A.4 Package Lucene 172
A.4.1 Classe Lucene.CreateInvertedIndex 172
A.4.2 Classe Lucene.MyDocument 173
A.4.3 Classe Lucene.SearchInvertedIndex 174
A.5 Package GetNext 175
A.5.1 Classe GetNext.GetNext2 175
A.5.2 Classe GetNext.GetNextText 181
A.6 Package Join 186
A.6.1 Classe Join.TwigStack 186

Elenco delle figure

1.1 Operatori di TSQL2 13

2.1 Esempio di documento XML rappresentante una legge 16
2.2 Utilità di un Parser come middleware fra documento XML ed Applicazione 19
2.3 Esempio di struttura ad albero 20
2.4 Componenti di un Location Path 23
2.5 Esempi grafici di percorsi XPath 25

3.1 La numerazione degli elementi 28
3.2 Esempio di twig 32
3.3 Algoritmo TSGeneric 33
3.4 Strutture di TSGeneric 34
3.5 Data set di esempio 35
3.6 Situazione negli stack dopo l'inserimento di a1 e b1 35
3.7 Situazione negli stack dopo l'inserimento di a2 e b2 35
3.8 Situazione negli stack dopo l'inserimento di c1 36

4.1 Esempio di ambito di validità temporale 42
4.2 Esempio di modifica 47
4.3 La dinamica temporale delle modifiche testuali 49
4.4 Schema riassuntivo dei tipi di nessi normativi 51
4.5 Elementi del modello adottato 53
4.6 Dettagli dell'elemento legge 55
4.7 Dettagli dell'elemento articolato 56
4.8 Dettagli del versioning temporale 57
4.9 Esempio di query espressa con XQuery 60
4.10 Architettura adottata nell'approccio a *stratum* 62
4.11 Relazione adottata in nell'approccio a *stratum* 62
4.12 Esempio di query con XQuery 63

5.1 Architetture a confronto 70
5.2 Operatori supportati nel modello 71
5.3 Esempio di query testuale su più path 72

5.4 Esempio di documento XML 75
5.5 Suddivisione in tuple 76

6.1 Vincoli all'interno di una query 86
6.2 Esempio dettagliato di query 87
6.3 Algoritmo TwigStack descritto con pseudo-codice 92
6.4 TwigStack: tuple articolo 94
6.5 TwigStack: tupla rubrica 94
6.6 TwigStack: tupla text 95
6.7 Tuple nel database 97
6.8 Letture delle tuple in GetNext 100
6.9 Elaborazione: passo uno 100
6.10 Elaborazione: passo due 101
6.11 Elaborazione: passo tre 101
6.12 Elaborazione: passo quattro 101
6.13 Elaborazione: passo cinque 102
6.14 Risoluzione di una Query testuale 103
6.15 Esempio di query 105
6.16 Esempio di query 107
6.17 Un interno path visualizzato in una pagina web 110

7.1 Package diagram del prototipo implementato 117
7.2 Class diagram dell'oggetto Tupla 117
7.3 Class diagram dell'oggetto MyDate 118
7.4 Classe AssociaDB 119
7.5 Classe SKC 120
7.6 Classe DBUtility 121
7.7 Classe DBUtility 122
7.8 Funzionamento del metodo `cercaRecordPrecedente(Db table,int startPos,DateWrapper dw)` 123
7.9 Classi all'interno del Package Lucene 124
7.10 Classe GetNext2 126
7.11 Classe GetNextText 127
7.12 Classe TwigStack 127

8.1 Grafico dei tempi di risoluzione delle 5 query 134
8.2 Grafico dei tempi di risoluzione in linee 134
8.3 Tempi di risoluzione delle query nei due sistemi confrontati 137

Elenco delle tabelle

2.1 Tabella degli assi di XPath 24
2.2 Forme abbreviate in XPath 24

3.1 Tabella degli elementi `ELEMENT` 29
3.2 Tabella degli elementi `TEXT` 29

4.1 Significato degli attributi temporali 56

5.1 Variabili Strutturali all'interno della struttura Tupla 72
5.2 Variabili Temporali all'interno della struttura Tupla 73
5.3 Altre variabili della Struttura Tupla 73
5.4 Tuple generate dal documento di esempio 79

6.1 Strutture CoppiaNodi all'interno del twig di esempio 87
6.2 Buffer contenente gli elementi articolo 91
6.3 Buffer contenente gli elementi rubrica 91
6.4 Buffer contenente gli elementi text 91
6.5 Primo path dei risultati 95
6.6 Secondo path dei risultati 95
6.7 Tuple inutili generate da un operatore binario 96
6.8 Possibili risultati alla query 105
6.9 Tupla che non rispetta i vincoli temporali 105
6.10 Risultato *non consistente* 106

8.1 I data set utilizzati nei test 130
8.2 Risultati ottenuti con le query sul primo data set 133
8.3 Tempistiche dei risultati ottenuti 133
8.4 Tempi, numero risultati ed accessi a disco del secondo data set . . 133
8.5 Tempi, numero risultati ed accessi a disco del terzo data set . . . 134
8.6 Sintesi dei tempi di elaborazione 135
8.7 Risultati in percentuale sul numero totale dei documenti del primo data set nella presente implementazione 136
8.8 Risultati in percentuale sul numero totale dei documenti del primo data set nel progetto sviluppato in [2] 136

8.9 Confronto delle percentuali sui match 137

Introduzione

L'affidabilità e l'efficienza dei sistemi per l'interrogazione di basi di dati che seguono il modello relazionale sono oggi un dato di fatto. Più di trenta anni di ricerca hanno dato alla luce un linguaggio standard di interrogazione per dati strutturati ed una vasta gamma di *engine* capaci di risolvere le interrogazioni. Da qualche anno a questa parte è tema di ricerca l'utilizzo di XML come linguaggio per la rappresentazione di dati semi-strutturati. I produttori di DBMS relazionali hanno proposto soluzioni proprietarie per la gestione di basi di dati XML anche se oggi non è ancora presente un linguaggio standard per la loro definizione ed interrogazione.

Il linguaggio XML può essere esteso in modo da diventare *temporale*, cioè in modo che sia possibile tenere traccia della storia di un documento nel tempo attraverso il passaggio di versione in versione. Si è quindi interessati all'evoluzione del documento XML come se fosse un'*entità* capace di mutare nel tempo senza perdere la propria identità. Negli ultimi anni la ricerca sui *database temporali* basati sul modello relazionale è stata abbondante, mentre quella basata su XML temporale è piuttosto carente. Inoltre mancano *engine* capaci di risolvere interrogazioni su XML temporale.

L'obiettivo di questa Tesi di Laurea è lo sviluppo di un *Temporal XML Query Processor* che permetta l'interrogazione efficiente di grandi moli di documenti che soddisfano un modello di XML temporale. Questo *engine* supporta query su una base di dati XML temporale e verifica in modo efficiente vincoli di tipo *temporale* e *strutturale* con l'utilizzo della tecnologia relazionale a disposizione.

Recentemente si è visto che è possibile scomporre un documento XML in semplici strutture memorizzabili in tabelle secondo il modello relazionale in modo da mantenere la struttura gerarchica alla base. Allo stesso modo si è ideata una

scomposizione dei documenti XML temporali in strutture a cui è applicabile un meccanismo di *versioning* che prevede l'esplicitazione dei timestamp relativi alle dimensioni temporali secondo la teoria dei database temporali.

All'interno di questa Tesi si sviluppa un'applicazione XML di tipo *text-centric*, cioè si fa riferimento a documenti che rappresentano un'evoluzione semi-strutturata di documenti testuali. Il modello di XML temporale viene applicato concretamente all'ambito normativo, e l'efficienza dei metodi di interrogazione sviluppati è verificata su una base di dati XML temporale rappresentante atti normativi. Un atto normativo è considerato un buon esempio di documento "*dinamico*", la cui *evoluzione* nel tempo passa attraverso successive modifiche che generano nuove *versioni* ad ogni suo livello interno.

Il modello di XML temporale adottato ed i rispettivi metodi di interrogazione non sono relativi al solo ambito normativo analizzato nel presente lavoro, ma sono applicabili ad altre realtà in cui sia necessario gestire l'evoluzione di entità nel tempo.

La tesi è organizzata in otto capitoli, di cui i primi quattro servono ad introdurre tutte le tematiche necessarie allo sviluppo del lavoro di ricerca presentato nei restanti capitoli.

Nel capitolo 1 si introducono i concetti sui database temporali: vengono presentate le dimensioni temporali di interesse, gli operatori supportati da TSQL2 e viene mostrato come è possibile supportare modifiche non distruttive su dati mutevoli nel tempo.

Il capitolo 2 introduce XML come linguaggio per la rappresentazione di dati semi-strutturati. Vengono poi presentati importanti linguaggi per l'interrogazione di documenti XML come XPath e XQuery.

Nel capitolo 3 si mostra come è possibile scomporre un documento XML in tabelle senza perderne la struttura gerarchica attraverso l'utilizzo degli algoritmi conosciuti come structural join e di particolari strutture dati.

Nel capitolo 4 si introducono termini e concetti specifici all'ambito giuridico per trattare in modo rigoroso la dinamica delle norme nel tempo. Nella seconda parte del capitolo si fa riferimento ad un modello dei dati espresso con XML temporale per la rappresentazione di atti normativi mutevoli nel tempo.

Il capitolo 5 presenta l'architettura ideata con lo scopo di realizzare un Temporal XML Query Processor che permetta di effettuare interrogazioni su dati semi-strutturati aderenti ad un modello XML temporale. Nella seconda parte del capitolo viene spiegato come è stato possibile suddividere i documenti XML temporali che soddisfano il modello introdotto nel capitolo 4 nelle strutture previste dagli structural join (capitolo 3) con l'esplicitazione dei timestamp delle dimensioni temporali in pieno accordo con la teoria dei database temporali introdotta nel capitolo 1.

Nel capitolo 6 si introducono algoritmi ideati in questa Tesi per la valutazione delle interrogazioni su un'estensione del modello XML temporale proposto nel capitolo 4. Si presentano i metodi utilizzati per il soddisfacimento dei vincoli di diversa natura presenti nella query e gli accorgimenti necessari affinchè le soluzioni prodotte siano *consistenti*. Alla fine del capitolo vi è una descrizione dei metodi di accesso utilizzati nell'elaborazione.

Nel capitolo 7 si descrive l'implementazione software del Temporal XML Query Processor proposto fin qui e se ne introducono i Package e le Classi. Il prototipo sviluppato utilizza le strutture e gli algoritmi fin qui presentati per effettuare interrogazioni su una base di dati contenente una grande collezione di atti normativi espressi in XML temporale.

Nel capitolo 8 si presentano i test effettuati ed i loro risultati sperimentali per verificare l'efficienza dell'engine sviluppato. Vengono anche effettuati confronti in termini di efficienza con altri sistemi.

La presente Tesi fa parte del Programma di Ricerca MIUR 2001 "*Tecniche di Database Temporali nella Gestione della Dinamica delle Norme*" dell'Unità di Modena, interno al Progetto Nazionale MIUR 2001 "*La Dinamica delle Norme nel Tempo: aspetti Giuridici ed Informatici*".

Parte I

Introduzione

Capitolo 1

Database temporali

1.1 Introduzione ai database temporali

Il tempo è un aspetto fondamentale di tutti i fenomeni che appartengono al mondo reale. Ogni evento si verifica in un certo istante, gli oggetti e le loro relazioni hanno significato solo se legati al tempo.

La capacità di modellare il tempo è essenziale per applicazioni altamente dinamiche come sistemi informativi geografici, bancari, medici, normativi, di analisi di dati scientifici ed econometrici.

I database convenzionali (che nel seguito chiameremo DB) possono rappresentare lo stato della realtà in un singolo istante del tempo (il presente). Infatti ad ogni aggiornamento di un campo, il nuovo valore sovrascrive quello precedente, con la conseguente perdita dell'informazione passata. Inoltre la cancellazione di un record dal DB fa perdere sia l'informazione passata che la cardinalità passata della relazione, mentre l'inserimento di un record fa perdere la sola cardinalità passata.

Negli ultimi 10 anni la ricerca sui database temporali ha prodotto una discreta mole di letteratura a riguardo. Purtroppo non c'è ancora stata una standardizzazione né sul modello da adottare né sui metodi di interrogazione da seguire (attesi con SQL3).

E' possibile comunque dire che un database è temporale se:

- garantisce modifiche e cancellazioni non distruttive

- cattura le variazioni nel tempo dei fenomeni che modella

1.1.1 Modelli del tempo

E' possibile pensare al tempo come ad un insieme arbitrario di istanti a cui viene associato un ordine.

In base alla struttura può essere classificato in:

- *lineare*: il tempo avanza dal passato al futuro in modo "lineare", cioè senza diramazioni
- *branching*: il tempo è lineare fino all'istante corrente dove si dirama in più alternative
- *ciclico*: utilizzato per modellare processi ricorrenti ed aventi un periodo

In base alla densità può essere classificato in:

- *discreto*: utilizzato spesso perché di facile interpretazione e rappresentazione su computer. A questo possiamo aggiungere i concetti di:
  1. *granularità*: unità di misura temporale di riferimento
  2. *chronon*: unità di tempo non decomponibile, più piccola durata di tempo rappresentabile nel modello che ha la granularità specificata
- *continuo*
- *denso*

In base all'insieme degli istanti può essere classificato in:

- *limitato* nel passato e/o nel futuro
- *illimitato*

1.1.2 Le dimensioni temporali

Nel contesto dei database temporali, le dimensione temporali di interesse sono essenzialmente due, e vengono utilizzate per associare i fatti al loro tempo di occorrenza: esse sono il tempo di validità (*Valid Time*) ed il tempo di transazione (*Transaction Time*).

Il Valid Time è il tempo in cui un fatto è vero nella realtà, ed ha le seguenti caratteristiche:

- è l'istante in cui l'evento accade, indipendentemente dal tempo della sua registrazione nel database
- è solitamente fornito dall'utente
- può anche essere nel futuro
- è caratterizzato dal valore speciale FOREVER (F); tale valore se associato ad un fatto indica che il fatto è correntemente vero

Il Transaction Time è il tempo in cui un fatto è presente nel database come fatto memorizzato, ed ha le seguenti caratteristiche:

- identifica la transazione che lo inserisce e quella che lo cancella
- è fornito dal sistema
- è limitato nel passato dall'istante di creazione del database e nel futuro dall'istante corrente
- è caratterizzato dal valore speciale UNTIL CHANGED (UC); tale valore se associato ad un fatto indica che il fatto è correntemente presente nel database come dato memorizzato

Queste due dimensioni sono ortogonali. Esistono modelli dei dati che supportano nessuna, una o entrambe le dimensioni temporali.

Un modello dei dati che non supporta nemmeno una di queste dimensioni è detto di tipo *snapshot* siccome cattura solo una visione *"istantanea"*. Tale visione riassume sia ciò che avviene nel mondo reale sia nella realtà modellata nel database.

Un modello dei dati che supporta solo il transaction time è detto *modello dei dati transaction time* e rappresenta una sequenza di dati snapshot lungo l'asse dei tempi. Il tempo di transazione di un record è il tempo in cui il record è presente nel database. Se si verificano degli aggiornamenti i dati non vengono alterati perchè le modifiche sono realizzate su una copia dello stato corrente del database che diventa il nuovo stato corrente, possiamo quindi dire che questo tipo di relazione è *append only*. Le query che coinvolgono il tempo di transazione estraggono informazioni dallo stato della relazione in un qualche istante del passato.

Un modello dei dati che supporta solo il valid time è detto *modello dei dati valid time* e rappresenta lo stato della realtà modellata. Il tempo di validità di un record è tempo in cui il fatto da essa rappresentato è vero nella realtà. In caso di aggiornamenti di un dato l'utente specifica direttamente il nuovo tempo di validità assieme agli altri attributi da modificare, e le modifiche sono realizzate sullo stato corrente del database che è lunico esistente.

Un modello di dati che supporta entrambe le dimensioni temporali è detto *bitemporale*. In caso di aggiornamento le modifiche sono realizzate su una copia dello stato di validità corrente che diventa il nuovo stato di validità (si rispettano i modelli relativi ad entrambe le dimensioni).

1.1.3 TSQL2: un linguaggio di interrogazione per dati temporali

TSQL2 (descritto in [15]) rappresenta un'estensione temporale del linguaggio di interrogazione standard per basi di dati SQL-92. Tale linguaggio è tato progettato da un comitato di persone dedite da anni alla ricerca nel campo delle basi di dati temporali provenienti dal mondo accademico, dai produttori di DBMS commerciali e dai centri di ricerca industriali. TSQL2 è un linguaggio di interrogazione progettato per manipolare dati che variano nel tempo, memorizzati in una base di dati relazionale tramite versioni dotate di *timestamp*.

Dati primitivi in TSQL2

Nel modello di TSQL2 il tempo è considerato *discreto*, con unità minima di rappresentazione dipendente dal sistema detta *chronon*. A livello concettuale possono così essere definiti tre tipi di dati temporali:

1. *datetime*
2. *period*
3. *date*

Il tipo *datetime* corrisponde ad un evento istantaneo, privo di durata e può essere convenzionalmente rappresentato con un singolo chronon. Fra i tipi primitivi di tipo datetime vanno ricordati quelli ereditati da SQL-92: `DATE`, `TIME` e `TIMESTAMP`. Alcuni esempi sono:

- `DATE '1979-05-03'`
- `TIME '13:33:25'`
- `TIMESTAMP '1979-05-03 13:33:25'`

Il tipo *period* corrisponde ad un intervallo temporale dotato di inizio di fine e di durata. Un dato di questo tipo può essere tipicamente definito attraverso l'utilizzo di due costanti di tipo *datetime* che ne rappresentano gli estremi. A seconda della definizione, dati di tipo *period* possono rappresentare intervalli aperti o chiusi (a destra e/o a sinistra); nel caso di intervallo aperto a destra o a sinistra esso terminerà rispettivamente un chronon prima o dopo l'estremo corrispondente. Alcuni esempi di dati di tipo period sono:

- `PERIOD '[March 1996]'`
- `PERIOD '[1979-05-03 - 1990-12-31)'`
- `PERIOD '(8:37:45.0 - 12:33:45.0)'`

Il tipo *interval* è impiegato per rappresentare una pura durata, disassociata dall'asse dei tempi, e che può essere rappresentato tramite un multiplo di un chronon. Alcuni esempi di uso dei dati di tipo *interval* sono:

- `INTERVAL '10' YEAR`
- `INTERVAL '3' WEEK`
- `INTERVAL 'November' DAY`
- `INTERVAL '-20' MINUTE` (è anche possibile esprimere durate negative)

I tre tipi fondamentali possono essere combinati in espressioni miste, come negli esempi che seguono:

- `PERIOD '1995-01-01 - 1995-06-01' + INTERVAL '10' MONTH` che equivale a `PERIOD '1995-11-01 - 1996-04-01'`
- `TIMESTAMP '13:30 April 1,1990' + INTERVAL '1' YEAR +`
  `- INTERVAL '15' MINUTE` che equivale a `TIMESTAMP '1991-04-01 13:15'`

Un *temporal element*, o nel seguito più semplicemente *element*, corrisponde ad un insieme arbitrario di intervalli di tempo e può essere rappresentato tramite un insieme di chronon non necessariamente consecutivi. Come un element rappresenta un insieme di period, un insieme di datetime costituisce un *instant set*, anch'esso rappresentabile come un insieme di chronon. Questi due tipi temporali non primitivi sono molto importanti perchè permettono di definire un'algebra che garantisca la chiusura anche rispetto alle operazioni della teoria degli insiemi (unione, intersezione, differenza). Un'analoga operazione non sarebbe invece possibile con l'adozione degli intervalli come tipo primitivo, in quanto l'unione o l'intersezione di due intervalli possono non essere intervalli.

Operatori di TSQL2

Per confrontare fra loro element, period e datetime sono disponibili 5 operatori. Siano A e B due element, period o datetime, potremmo avere che:

- `A PRECEDES B`, vero se la fine di A occorre prima della fine di B (l'opposto è `B FOLLOWS A`)
- `A = B`, vero se A e B sono identici
- `A OVERLAPS B`, vero se l'intersezione di A e B è non vuota

- `A MEETS B`, vero se la fine di A precede immediatamente l'inizio di B (l'opposto è `B MET-BY A`)
- `A CONTAINS B`, vero se ogni chronon di B appartiene anche ad A (l'opposto è `B CONTAINED-IN A`)

In figura 1.1 è possibile vedere graficamente le relazioni fra due intervalli A e B. E' possibile imporre dei vincoli sulle dimensioni temporali introdotte uti-

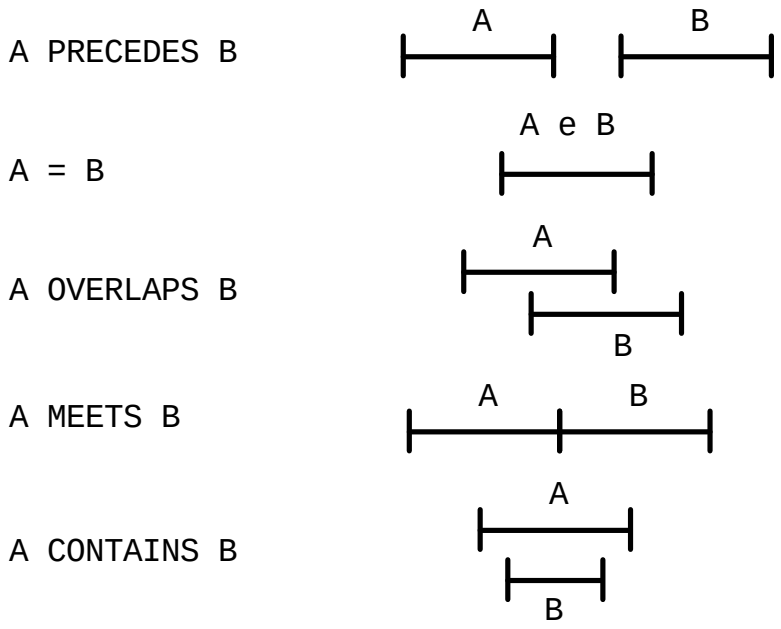

Figura 1.1: Operatori di TSQL2

lizzando la sintassi di TSQL2: `dimension [NOT] op VALUE` in cui `dimension` indica una dimensione temporale specificata, `op` indica uno degli operatori supportati e `VALUE` indica un qualsiasi *temporal element*. Ad esempio si potrebbe scrivere:

- `VALID OVERLAPS PERIOD ('1970-01-01','1999-12-31')` per indicare un vincolo che impone l'intersezione del valid time con il periodo che va dal Primo Gennaio 1970 al 31 Dicembre 1999
- `TRANSACTION CONTAINS '1980-01-01'` per indicare un vincolo che impone al transaction time di contenere il giorno Primo Gennaio 1980

Capitolo 2

Introduzione ad XML

XML, inizialmente proposto come standard per la rappresentazione dei documenti testuali, si sta ora diffondendo come standard per l'interscambio di informazioni tra le diverse applicazioni. XML stabilisce un insieme di convenzioni per definire diversi linguaggi di codifica, in base alle diverse tipologie di informazioni da rappresentare: associando ai dati di partenza informazioni descrittive (dette tag o marcatori) pone le premesse per la costruzione di inferenze automatiche sulla semantica dei dati.

2.1 Struttura logica e presentazione

XML (eXtensible Markup Language) è un linguaggio di markup mediante il quale è possibile associare a ciascuna parte di un testo un marcatore (*tag*), che la qualifica come un determinato elemento logico (per esempio legge, articolo o comma nella figura 2.1). In figura 2.1 è possibile vedere una possibile traduzione di una legge in un documento XML.

Ciascun marcatore XML è determinato da due simboli di inizio e di fine che utilizzano le parentesi angolari <> (come in `<articolo>` della figura 2.1). Un *elemento testuale* è generalmente delimitato da un tag di apertura e uno di chiusura: quest'ultimo, omonimo rispetto a quello di apertura, si distingue per il simbolo `/`, posposto alla parentesi angolare < (nell'esempio di legge di figura 2.1, la stringa "inquinamento del mare" può essere ad esempio identificata come elemento comma, con la codifica XML `<comma>inquinamento del`

```
<?xml version="1.0" ?>
<legge>
  <articolo>
    <comma>
      inquinamento del mare
    </comma >
    <comma >
      inquinamento dei fiumi
    </comma >
    <comma >
      inquinamento dell'aria
    </comma >
  </articolo>
</legge>
```

Figura 2.1: Esempio di documento XML rappresentante una legge

`mare</comma>`). Il tag di apertura di un elemento può avere degli attributi con determinati valori (nella figura 2.1 è il caso del tag `<comma>` il cui attributo `num` assume i valori "1", "2" e "3"). Il testo può stare all'interno di qualsiasi elemento (anche se questo contiene altri nodi) e fra due elementi che si trovano all'interno di un altro elemento. Malgrado questo, per avere una maggiore leggibilità, di solito si utilizzano solo elementi che contengono altri elementi oppure elementi che contengono solo testo che vengono considerati foglia (anche se non è imposto dalla sintassi).

Un documento XML ha una *struttura gerarchica*: ciascun elemento, delimitato da un tag di apertura e di chiusura è contenuto in altri elementi, ed un elemento *radice* (nell'esempio i figura 2.1, l'elemento `<legge>...</legge>`) contiene tutti gli altri. E' infatti possibile pensare ad un documento XML come ad una albero in cui la struttura si sviluppa dall'elemento radice fino alle foglie, passando per eventuali nodi intermedi.

All'interno del documento XML di esempio si possono ritrovare diversi elementi :

- legge è l'elemento *radice* o *root* : tale elemento deve sempre essere uno ed uno solo e non contiene testo

- articolo è un elemento *figlio* dell'elemento legge (figlio di root) e non contiene del testo (può essere considerato un elemento intermedio)
- comma è l'elemento figlio di articolo e contiene del testo (e viene anche chiamato elemento*foglia*)

2.1.1 Tipologie di documenti e DTD

XML è un *metalinguaggio* generico che consente di costruire diversi linguaggi di codifica, ciascuno per una diversa tipologia di documenti: XML non prescrive i nomi dei diversi marcatori, ma solo la sintassi generica per la loro definizione ed il loro utilizzo nell'identificazione degli elementi di testo.

In XML è possibile definire la peculiare struttura logica che identifica e descrive una tipologia di documenti nella cosiddetta DTD (Document Type Definition), un insieme di specifiche che situate in una sezione all'inizio del documento codificato o in un documento a parte, stabiliscono quali sono i nomi ammissibili per i marcatori, quali i nomi dei loro attributi e quali relazioni di inclusione possono sussistere tra di loro.

Un documento che rispetta la sintassi generica di XML, senza riferirsi a una specifica DTD, è detto *ben formato* (`well-formed`). Un documento XML congruente rispetto ad una determinata DTD è detto *valido* (`valid`).

Il controllo della validità di un documento rispetto ad una DTD precedentemente definita è una garanzia di *congruenza formale* necessaria per applicare con successo al documento stesso procedure automatiche di elaborazione (fogli di stile per la formattazione, istruzioni di *information retrieval*, ecc.).

2.1.2 XML Schema

Anche se a prima vista può sembrare che le DTD forniscano tutte le funzionalità necessarie per definire la struttura dei documenti XML, ad uno sguardo più attento esse presentano anche numerosi svantaggi che hanno suggerito l'adozione di soluzioni alternative.

In primi luogo esiste il problema della mancanza di uniformità fra i dati (il contenuto dei documenti XML) e i corrispondenti *metadati*, cioè le DTD usate per descriverne la struttura.

In altre parole, poichè la sintassi usata per scrivere le DTD non è essa stessa una marcatura XML valida, non è possibile elaborare i metadati con gli stessi strumenti software usati per manipolare i dati.

Oltre ad essere sintatticamente non uniformi ai dati che descrivono, le DTD sono carenti anche dal punto di vista dei *tipi di dati elementari*, cioè dei valori che possono assumere gli elementi e gli attributi.

In generale, ci si è resi conto che quando si definisce la struttura di dati in formato XML sarebbe utile disporre dello stesso repertorio di tipi di dati elementari comunemente a disposizione dei progettisti di database relazionali; ma le DTD richiedono un'obbligata definizione manuale anche di tipi di uso molto comune come le date.

La sintassi di XML Schema (o semplicemente Schema) distingue fra *tipi di dati* ed *elementi XML* appartenenti a quei tipi. Oltre ad includere una quarantina di tipi elementari incorporati, XML Schema fornisce al progettista più o meno lo stesso potere espressivo degli attuali linguaggi di programmazione ad oggetti. In altri termini XML Schema è un completo *Data Definition Language* (DDL) basato su XML che porta ad eseguire una successiva *dichiarazione* di elementi XML partendo dalla *definizione* di tipi di dati.

2.1.3 Parsing di un documento XML

XML non fa elaborazioni, ma i documenti memorizzati utilizzando tale linguaggio di marcatura godono di una rigida strutturazione interna che permette di associare un significato ad ogni loro frammento la struttura è nota.

Sono interessanti i Parser XML perchè permettono ad una applicazione di accedere direttamente ai dati XML. Ogni parser deve svolgere tre importanti funzioni, esse sono:

1. recuperare e leggere il documento XML

2. assicurarsi che il documento soddisfi determinati canoni di struttura (ben formato, valido)

3. rendere disponibile all'applicazione il contenuto del documento

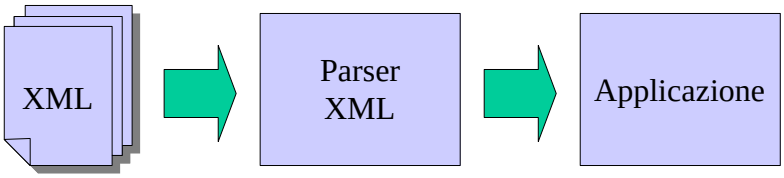

Figura 2.2: Utilità di un Parser come middleware fra documento XML ed Applicazione

Esistono due modelli di parser XML. Il modello *tree-based* si chiama DOM (Document Object Model): il documento XML in questione viene letto, compilato e ne viene generata in memoria una versione "*ad albero*" del tutto equivalente nel significato al documento di partenza. Ora l'applicazione può leggere questa struttura per accedere ai nodi dell'albero che sono rappresentativi degli elementi del documento XML originale. Il modello *event-based* si chiama SAX (Simple Api for XML): il documento XML viene letto dal primo all'ultimo carattere e ogni volta che si incontra un tag appartenente alla sintassi XML viene generato un evento che viene riportato all'applicazione. Gli eventi più importanti sono:

1. apertura / chiusura di un documento
2. apertura / chiusura di un tag
3. presenza di un nodo testo
4. presenza di un nodo attributo
5. presenza di un nodo commento
6. presenza di una processing instruction

Con l'utilizzo del modello event-based l'applicazione dovrà gestire i diversi eventi richiamando opportune routine. L'insieme di tali routine forma un Handler per un documento di una certa struttura, cioè un'applicazione che gestisce il verificarsi di tali eventi.

A differenza del modello tree-based, quando si utilizza il modello event-based non viene generata nessuna struttura interna (come prima poteva essere l'albero

equivalente al documento originale). Da ciò consegue che il parsing di un documento è notevolmente più veloce ed impiega uno spazio in memoria centrale notevolmente minore.

In figura 2.3 possiamo vedere come potrebbe apparire l'albero generato a partire dal documento XML sopra descritto.

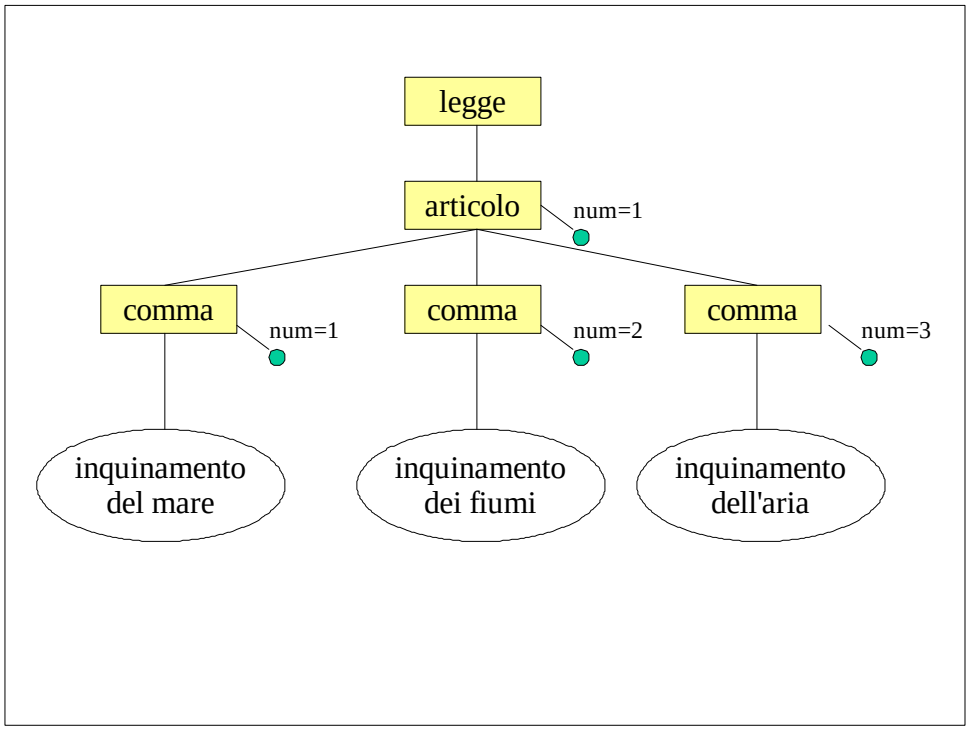

Figura 2.3: Esempio di struttura ad albero

SAX è oramai divenuto uno standard ufficiale anche se non è nato dal W3C. E' disponibile per molti linguaggi, fra cui Java. Mentre si esegue il parsing di un file XML, il parser notifica gli eventi sopra citati all'applicazione. Tali eventi vengono interpretati (o anche ignorati, se non necessari) da un Handler che dice all'applicazione come deve comportarsi quando si verifica un particolare evento. E' possibile dire che SAX è seriale, in quanto genera gli eventi man mano che li incontra nel documento XML che sta leggendo, senza mantenere nulla in memoria centrale. Un tipico Handler dovrà contenere i metodi o routine :

1. `startDocument()`: gestisce l'inizio di un documento

2. `endDocument()`: gestisce la fine di un documento

3. `startElement()`: gestisce l'inizio di un elemento
4. `endElement()`: gestisce la fine di un elemento
5. `characters()`: gestisce un nodo di testo

2.1.4 Linguaggi per l'interrogazione di dati XML

XML mette a disposizione strumenti efficaci per descrivere dati semi-strutturati mediante l'utilizzo di marcatori. Da solo, però, XML non dà i mezzi per identificare specifici sottoinsiemi di dati memorizzati in un documento; all'interno del W3C si stanno affermando due linguaggi di interrogazione appartenenti alla famiglia di XML: questi sono XPath ed XQuery.

XPath

XPath è un linguaggio basato su percorsi (paths) che consentono di raggiungere dei nodi specifici all'interno di un documento XML. Il W3C ne raccomanda la versione 1.0. I linguaggi che lo usano come proprio engine interna sono :

- XSL Trasformation (XSLT)
- XML Query (XQuery)
- XML Link (XLink)
- XML Pointer (XPointer)

XPath usa una sintassi limitata e compatta non basata su elementi XML, presenta una discreta libreria di funzioni che è adatta all'individuazione ed al reperimento di porzioni di un albero XML.

Il principale costrutto di XPath è l'*espressione*, dalla cui valutazione possono essere restituiti oggetti di tipo:

- booleano: viene restituito true o false a seconda che venga verificata o no una particolare condizione
- numerico: viene restituito un comune valore numerico (float) che può essere il risultato di uno o più operatori aritmetici

- stringa: viene restituita una semplice stringa di caratteri Unicode. Il tipo stringa può essere usato assieme agli operatori di uguaglianza (=) per verificare l'identicità di due stringhe, e di minore (<) per determinare un ordinamento

- riferimento a zero o più nodi: viene restituito un set di nodi (anche vuoto), definiti nel modello di dati utilizzato da XPath. Il modello dei dati comprende:

  1. nodo radice
  2. nodo PI (Processing Instruction): rappresenta l'istruzione di elaborazione all'interno del documento
  3. nodo elemento: rappresenta tutti i nodi del documento, compreso il nodo radice
  4. nodo commento
  5. nodo testo: rappresenta il contenuto testuale di un elemento
  6. nodo attributo: rappresenta uno degli attributi contenuti negli elementi del documento
  7. nodo namespace: rappresenta gli spazi dei nodi dichiarati nei fogli di stile XSLT (se presenti)

Location path I location path (percorsi di posizione) rappresentano il più importante strumento che XPath mette a disposizione per la navigazione di un documento XML. La valutazione di un percorso viene fatto a partire da un particolare *contesto*, composto da:

- un nodo contesto (*context node*)

- una coppia di interi positivi (*context position* e *context size*)

- un insieme di collegamenti di variabili (per il mappig dai nomi ai valori delle variabili)

- una libreria di funzioni (per il mapping dai nomi delle funzioni alle funzioni)

I percorsi possono essere assoluti o relativi:

location path relativo: consiste in una sequenza di uno o più location step separati da "/" che vengono valutati a partire dal context node

location path assoluto: consiste in un "/" seguito (opzionalmente) da un location path relativo

Ogni location path seleziona un insieme di nodi relativi al context node, ed è composto da uno o più location step, ed in ognuno di loro abbiamo (figura 2.4):

- asse : specifica le relazioni tra i nodi selezionati dal location step e il context node
- node test : specifica il tipo ed il nome del nodo
- predicato (opzionale)

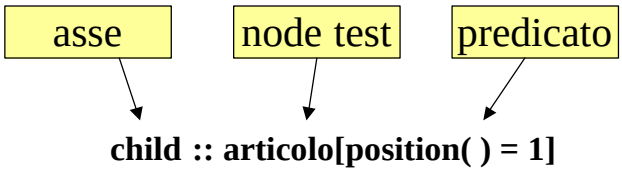

Figura 2.4: Componenti di un Location Path

In tabella 2.1 sono stati descritti tutti gli assi di XPath.

Per il node test abbiamo dei valori speciali :

1. `*`: qualunque
2. `text()`: qualunque nodo testuale
3. `commet()`: qualunque nodo commento
4. `processing-instrution()`: qualunque processing instruction.

I predicati disponibili in XPath sono molti e possono essere suddivisi in :

- funzioni booleane

asse	descrizione
self	il nodo stesso, il contest node
attribute	nodi attributo del nodo contesto
child	asse predefinito, indica i figli del nodo del contesto
descendant	nodi discendenti (non necessariamente i figli)
descendant-or-self	nodi discendenti o il contest node
parent	il padre del contest node
ancestor	i progenitori del contest node
ancestor-or-self	nodi ancestor o il contest node
following	nodi posti dopo il contest node(non discendenti,non attributi e non namespace)
following-sibling	i nodi fratelli del contest node secondo l'asse following
preceding	nodi posti prima del contest node(non discendenti,non attributi e non namespace)
preceding-sibling	i nodi fratelli del contest node secondo l'asse preceding
namespace	nodi namespace (dello spazio dei nomi) del nodo contesto

Tabella 2.1: Tabella degli assi di XPath

abbreviazione	forma standard
completa omissione	child::
@	attribute::
..	parent::node()
.	self::node()
//	/descendant-or-self::node()
[numero]	[position()=numero]

Tabella 2.2: Forme abbreviate in XPath

- funzioni numeriche
- funzioni stringa

E' anche possibile utilizzare una sintassi abbreviata nello scrivere espressioni XPath. In tabella 2.2 ci sono tutte le forme abbreviate previste da XPath.

Ad esempio, per raggiungere il nodo che parla di inquinamento dei fiumi nel documento in figura 2.1 potremmo scrivere i seguenti path assoluti:

```
/child::legge/child::articolo/
  child::comma[position()=2]/child::text()
```

che nella forma abbreviata diventa:

```
/legge/articolo/comma[2]/text()
```

Un'alternativa è considerare anche gli attributi, ad esempio:

```
/child::legge/child::articolo/
  child::comma[attribute::num = 2]/child::text()
```

che nella forma abbreviata diventa:

```
/legge/articolo/comma[@num = 2]/text()
```

Un path relativo nella forma abbreviata potrebbe essere:

```
comma[@num = 2]/text()
```

E' anche possibile esplicitare graficamente i path. Nel disegno a) della figura 2.5 c'è il corrispettivo grafico del path legge/articolo (relazione *parent-child*), mentre nel disegno b) c'è il corrispettivo grafico del path legge//comma (relazione *ancestor-descendant*).

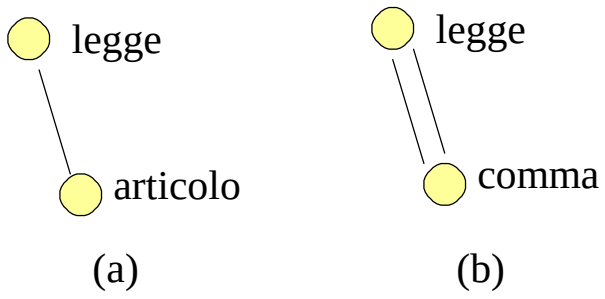

Figura 2.5: Esempi grafici di percorsi XPath

Capitolo 3

Gli structural join

Così come è stato necessario adottare *engine* efficienti per interrogare dati che seguono il modello relazionale, allo stesso modo sono necessari strumenti analoghi per l'interrogazione di documenti in XML. Come realizzare tali engine è ancora tema di ricerca, ma uno dei modi possibili di farlo passa attraverso la suddivisione dei documenti in relazioni e l'elaborazione basata sui meccanismi introdotti dagli *structural join*, o *join strutturali*.

Precedentemente all'utilizzo dei join strutturali si memorizzava sempre e solo un *documento intero*, mai una sua porzione. Di conseguenza in fase di interrogazione era necessario reperire l'intero documento per rispondere a query per cui bastava un ristretto set di dati.

Recentemente si è visto che è possibile memorizzare *solo frammenti* di documenti XML senza perderne la struttura che ne sta alla base. Tutti i frammenti vengono poi reperiti solo nel momento in cui se ne ha effettivamente bisogno in una interrogazione.

3.1 Struttura dei dati ed algoritmi per gli structural join

La struttura dei dati che sta alla base degli structural join è stata introdotta in [16] dove si è visto che suddividendo documenti XML in nodi rappresentanti elementi veri e propri (ELEMENT) e nodi rappresentanti testo semplice (TEXT) è possibile frammentare il documento senza perderne la struttura.

I nodi ELEMENT hanno la struttura: `ELEMENT (docId,startPos,endPos, level)`.

I nodi TEXT sono un po' più semplici: `TEXT (docId, startPos, level)`.

Il campo docId indica l'identificativo del documento XML, qualcosa che lo faccia distinguere dagli altri. I campi startPos ed endPos sono il risultato della numerazione *progressiva* dei tag di apertura e di chiusura di ogni elemento e l'inizio di ogni elemento di testo. Il campo level indica la profondità dell'innestamento: l'elemento root avrà sempre il valore di level minore in quanto (ciò è imposto dalla sintassi XML) contiene tutti gli altri elementi e non è contenuto in nessun altro. Nella figura 3.1 si può vedere come potremmo numerare i tag del nostro documento di esempio. Abbiamo aggiunto il campo "type" a entrambe le strutture per memorizzare il nome del tag.

```
<legge>                          1
  <articolo num ="1">            2
    <comma num ="1">             3
      inquinamento mare          4
    </comma >                    5
    <comma num ="2">             6
      inquinamento fiume         7
    </comma >                    8
    <comma num ="3">             9
      inquinamento aria          10
    </comma >                    11
  </articolo>                    12
</legge>                         13
```

Figura 3.1: La numerazione degli elementi

La tabella 3.1 rappresenta gli elementi ELEMENT, mentre la tabella 3.2 rappresenta gli elementi TEXT provenienti dal documento di esempio.

Grazie a questa strutturazione è possibile memorizzare frammenti di documenti XML in semplici tabelle. Tali tabelle potrebbero essere ad esempio quelle di un comune database relazionale .

type	docId	startPos	endPos	level
legge	1	1	13	1
articolo	1	2	12	2
comma	1	3	5	3
comma	1	6	8	3
comma	1	9	11	3

Tabella 3.1: Tabella degli elementi `ELEMENT`

type	docId	startPos	level
text	1	4	4
text	1	7	4
text	1	10	4

Tabella 3.2: Tabella degli elementi `TEXT`

E' stato poi possibile riscrivere le più importanti relazioni presenti in XPath ed XQuery alla luce di questo nuovo approccio. Le relazioni sono:

- *ancestor-descendant*
- *parent-child*
- *sibling*

Siano A e B due nodi, possiamo dire che :

A è *ancestor* di B (o B è *descendant* di A)
se e solo se
`(A.startPos < B.startPos) AND (B.endPos < A.endPos)`

A è *parent* di B (o B è *child* di A)
se e solo se
`(A.startPos < B.startPos) AND (B.endPos < A.endPos)`
`AND ( (A.level +1) = B.level)`

A è *sibling* di B , e *precede* B (o B è *sibling* di A , e *segue* B)
se e solo se
`(A.endPos < B.startPoS) AND ( A.level = B.level )`
`AND (hanno lo stesso parent)`.

3.2 Algoritmi in letteratura

L'articolo [1] introduce le strutture dati precedentemente enunciate (introdotte in [16]), conia il termine "structural join" e ne presenta interessanti algoritmi per l'elaborazione. L'algoritmo più interessante proposto nell'articolo [1] è sicuramente *stack-tree-desc*, basato sull'utilizzo di uno stack. Tale algoritmo elabora set di dati di tipo `ELEMENT` e `TEXT` ordinati secondo (docId, startPos) partendo da query complesse e strutturate.

Da ora in poi parleremo sempre di relazioni di tipo *ancestor-descendant*, quelle di tipo parent-child sono considerate identiche tranne che per un semplice controllo sul campo level.

Ipotizziamo di avere la query `a//b//c` .

Questa query complessa viene suddivisa in un insieme di semplici relazioni strutturali binarie di tipo *ancestor-descendant*. Facendo questo la query di esempio viene suddivisa nelle relazioni binarie `a//b` e `b//c`. Gli step per risolvere la query complessiva sono:

1. trovare le corrispondenze di ognuna delle relazioni strutturali binarie nel proprio database

2. unire tutte le relazioni strutturali binarie verificatesi al punto precedente

L'approccio di questo algoritmo sul set di dati è detto *set-at-a-time* perché per ogni nodo ancestor identificato vengono individuati tutti i nodi descendant possibili in un unico ciclo. Approcci precedenti sono considerati perdenti rispetto a questo in quanto erano di tipo *node-at-a-time*, cioè per ogni nodo si cercavano tutti i descendant, col problema che era necessario attraversare tutti i nodi del database molte volte.

L'algoritmo *stack-tree-desc* dà buoni risultati, ma presenta il problema della dimensione dei risultati intermedi. I risultati intermedi sono quelli generati da ognuna delle relazioni binarie della query (cioè da `a//b` e da `b//c`). La loro mole può essere un problema, in quanto:

1. rappresentano solo una soluzione parziale alla query (non si sa se parteciperanno a quella globale), ma non si possono tralasciare se si vuole una soluzione globale

2. possono essere numerose, occupando così un cospicuo spazio in memoria centrale

Il numero dei risultati intermedi è in ogni caso indipendente dal numero delle soluzioni globali delle query.

L'articolo [3] introduce per i join strutturali un punto di vista olistico.

Olismo: teoria secondo cui un'entità è un tutto superiore alla semplice somma delle parti che la costituiscono.

In tale articolo viene introdotto un algoritmo chiamato PathStack che continua a basarsi sulle stesse strutture di base, ma introduce una serie di stack collegati atti a contenere risultati parziali.

Ipotizziamo di avere la query `a//b//c` (che potremmo chiamare anche *path*, percorso) in cui a, b e c sono elementi appartenenti alla tabella ELEMENT (ricordiamo che solo gli elementi foglia possono essere di tipo TEXT, e che le stesse funzioni sono applicabili sia alle strutture ELEMENT sia alle strutture TEXT).

La terminologia legata alla query è del tipo:

- a è detto root della query

- b è figlio del nodo a della query

- c è figlio del nodo b della query ed è anche un nodo foglia per la query

L'algoritmo più importante introdotto da [3] si chiama TwigStack, che ha un funzionamento basato sugli stack come PathStack, ma permette di fare query su

più di un path. Una unione di più di un path in una query viene detto un *twig* (letteralmente: *rametto*); l'importante è che l'elemento root sia comune a tutti i path del twig.

Un esempio di twig può essere l'unione di path aventi lo stesso elemento root (figura 3.2), ad esempio:

- a // b // c
- a // d // e

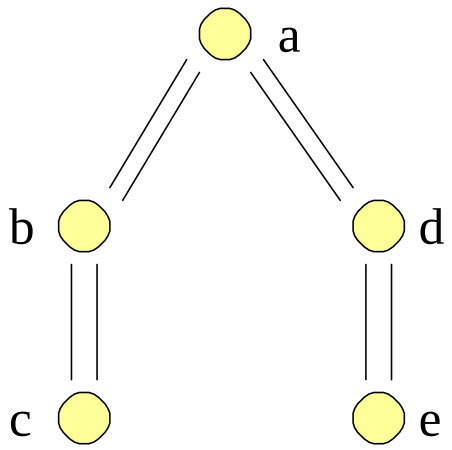

Figura 3.2: Esempio di twig

3.2.1 Funzionamento di TSGeneric

TSGeneric sta per TwigStack Generic, che è l'algoritmo *stack-based* presentato in [10].

I documenti XML di interesse vengono sottoposti al parsing, e memorizzati in tabelle consone alle strutture `ELEMENT` e `TEXT` introdotte nel paragrafo 3.1. Ora è possibile utilizzare l'algoritmo TSGeneric per risolvere una query sui dati memorizzati; tale algoritmo è descritto in figura 3.3 con l'utilizzo di pseudocodice.

Algorithm 1 *TSGeneric(root)*

```
1: while not end(root) do
2:    q = getNext(root);
3:    if not isRoot(q) then
4:       cleanStack(S_parent(q), C_q);
5:    if isRoot(q) or (not empty(S_parent(q))) then
6:       cleanStack(S_q, C_q);
7:       if not isLeaf(q) then
8:          push(S_q, C_q, top(S_parent(q)));
9:       else
10:         outputPathSolutionsWithBlocking(C_q);
11:   C_q→advance();
12: end while
13: mergeAllPathSolutions();
```

Figura 3.3: Algoritmo TSGeneric

Sia detta `Q` la query da risolvere, ad esempio `a//b//c`, e siano detti `q` i nodi della query. Associato ad ogni nodo della query avremo uno stack ed un cursore nel DB che chiameremo rispettivamente S_q e C_q.

All'interno di ogni iterazione data dal `while` viene generato un nodo della query (linea 2) che chiameremo *nodo corrente*.

L'elemento relativo al nodo corrente è quello puntato dal cursore C_q, e lo chiameremo *elemento corrente*. Ad ogni iterazione tale elemento viene analizzato e posto negli stack (se verifica certe condizioni).

Le procedure utilizzate all'interno dell'algoritmo sono:

`cleanStack(`S_q`, `C_p`)` verifica che all'interno dello stack S_q ci siano solo elementi *ancestor* dell'elemento individuato da C_p; tutti gli elementi che non soddisfano questa condizione vengono eliminati

`push(`S_q`, `C_p`, ptr)` viene inserita sullo stack S_q una struttura che comprende C_p ed un puntatore *ptr* che indica l'elemento *top* dello stack padre

`outputPathSolutionWithBloking(`C_q`)` genera una soluzione completa di un path della query (dal nodo radice al nodo foglia)

All'interno del primo `if` si controlla se il nodo corrente è un nodo diverso

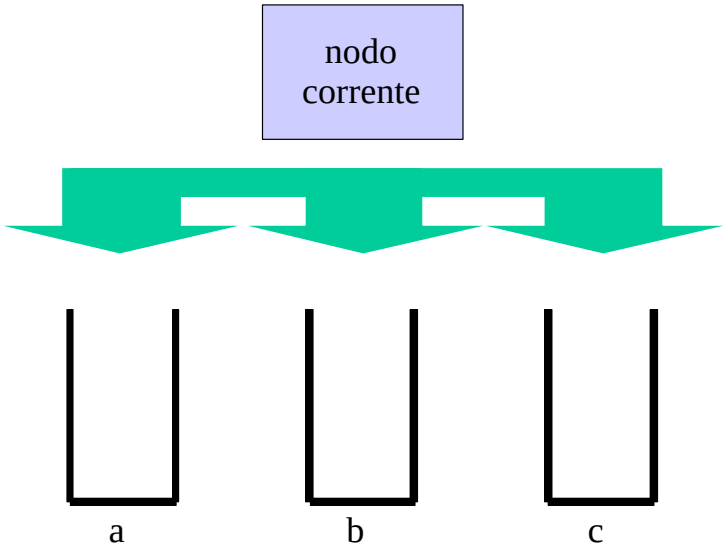

Figura 3.4: Strutture di TSGeneric

da *root*, in al caso si richiama `cleanStack(`S_q`, `C_p`)` sullo stack padre, cioè su $S_{parent(q)}$.

All'interno del secondo `if` si controlla che il nodo corrente sia root e che lo stack padre del nodo corrente sia non vuoto; se almeno una di queste due condizioni è vera allora si richiama `cleanStack(`S_q`, `C_p`)` sullo stack del nodo corrente.

Nel terzo `if` si controlla che il nodo corrente non sia un elemento foglia, in tal caso si richiama la funzione `push(`S_q`, `C_p`, ptr)` che inserisce l'elemento all'interno dello stack. Quando si entra nel ramo `else` del terzo `if` significa che si sono riempiti gli stack di tutti i nodi della query, e che il nodo corrente è un elemento foglia. La funzione `outputPathSolutionWithBloking(`C_q`)` è una funzione ricorsiva che si occupa di risalire lungo gli stack della query e reperire le soluzioni ad un path della query. Questo avviene utilizzando un riferimento al top dello stack del nodo precedente nella query, che è il puntatore *ptr* di ogni struttura memorizzata in ogni stack al momento della chiamata a `push(`S_q`, `C_p`, ptr)`.

Si supponga di avere il set di elementi in figura 3.5.

Gli elementi vengono generati da `getNext(root)` nell'ordine : a1, b1, a2, b2, c1, c2. Dopo le prime due iterazioni a1 e b1 sono negli stack (figura 3.6)

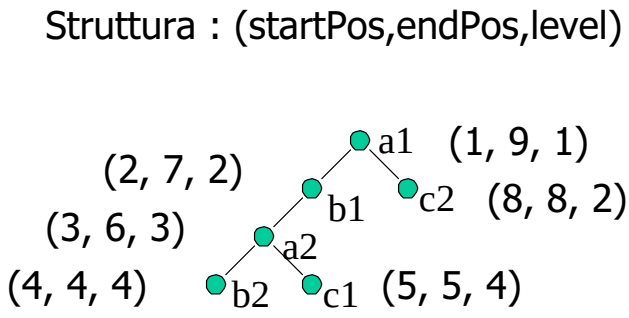

Figura 3.5: Data set di esempio

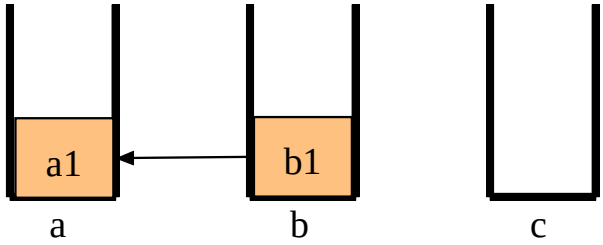

Figura 3.6: Situazione negli stack dopo l'inserimento di a1 e b1

In seguito anche a2 e b2 sono inseriti negli stack (figura 3.7). Gli elementi a1 e b1 rimangono dove sono perchè sono *ancestor* rispettivamente di a2 e b2.

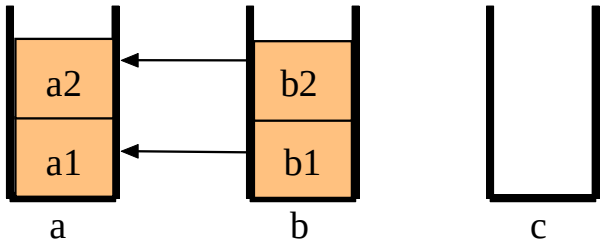

Figura 3.7: Situazione negli stack dopo l'inserimento di a2 e b2

Ora è il turno di c1 che entra e cancella il nodo b2 (che non è un suo *ancestor*), e forma le soluzioni complete del path `a//b//c` (figura 3.8). In questo caso esiste solo una soluzione completa : a1 b1 c1.

Quando entra c2, b1 viene cancellata perchè non ne è *ancestor*. A causa di questo non è possibile mettere c2 sul suo stack in quanto lo stack di b è vuoto.

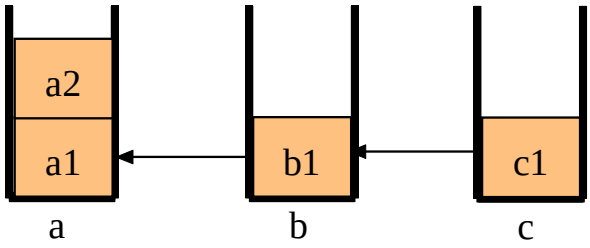

Figura 3.8: Situazione negli stack dopo l'inserimento di c1

Capitolo 4

Introduzione ai testi normativi ed al modello adottato

4.1 Riferimenti normativi e dinamica dei testi normativi

All'interno della prima parte del presente capitolo verrà introdotta la terminologia necessaria per trattare in maniera rigorosa i testi normativi. Nei testi normativi possiamo individuare diverse unità testuali autonome che chiameremo disposizioni normative. In particolare si presenterà l'*evoluzione* su testi e disposizioni causata da successive modifiche nel tempo. Questo, anche se di carattere essenzialmente nozionistico, è fondamentale per una corretta interpretazione di alcune scelte adottate nei capitoli successivi.

Nella seconda parte del capitolo verrà introdotto un modello XML temporale per la rappresentazione dell'evoluzione del tempo degli atti normativi. Tale modello, tramite l'utilizzo di un meccanismo di "*versioning temporale*", permette di esplicitare successive modifiche testuali e temporali ad ogni livello della gerarchia adottata. Inoltre, si introdurranno sia una possibile formulazione di interrogazioni sul modello adottato sia un'architettura esistente che ne permetta la risoluzione.

4.1.1 Nozioni preliminari

Si introdurranno dapprima alcuni termini specifici, che si riprenderanno nel seguito, al fine di distinguere e qualificare i diversi tipi di riferimenti normativi.

Testo normativo. Chiamiamo testo normativo ogni documento che raccolga enunciati risultanti da atti normativi.

Testo normativo originario. Il testo normativo originario, o documento normativo, corrisponde alla integrale sequenza testuale enunciata mediante un determinato atto normativo. Ad esempio, è un testo normativo il testo di legge approvato dal parlamento, promulgato il Primo dicembre 1970, e pubblicato nella gazzetta ufficiale n. 306 del tre dicembre 1970 (la legge sul divorzio).

4.1.2 Storia del testo normativo (testo normativo virtuale)

Ogni testo normativo può mutare di contenuto nel tempo, in seguito a modifiche testuali, pur mantenendo la propria individualità ed identità. E' quindi opportuno distinguere:

- le singole versioni del testo normativo, intese come le diverse formulazioni che il testo è venuto ad assumere nel corso del tempo
- la storia del testo normativo, intesa come la sequenza che comprende tutte le versioni dette prima
- il testo normativo virtuale, cioè il contenitore astratto di tali versioni che si ritiene mantenga una propria identità attraverso la storia del testo

La storia del testo normativo si svolge tra i due momenti della sua entrata in vigore e della sua uscita di vigore (abrogazione).

4.1.3 Testo normativo vigente (in un momento x)

Il testo normativo vigente in un determinato momento x corrisponde al risultato di tutte le successive operazioni di modifica testuale operate da atti normativi

successivi sul testo normativo originario. Mentre il testo normativo originario è unico, e il suo contenuto rimane per sempre fissato al momento della sua emanazione, il testo normativo vigente evolve nel tempo. Quindi, l'espressione che designa un testo di legge - ad esempio, "la legge sul divorzio", "il regolamento edilizio" - risulta ambigua in quanto:

1. può riferirsi al testo normativo originario
2. può riferirsi al testo normativo vigente
3. in questa seconda ipotesi può riferirsi all'intera storia del testo normativo, o ad un determinato momento di tale storia
4. in questa ultima ipotesi può riferirsi al risultato ultimo di tale storia (al momento nel quale il fruitore sta leggendo l'espressione designativa), o al momento in cui l'espressione designativa veniva formulata, o a un qualsiasi altro momento

Un'unica versione del testo normativo è identificata solo da formulazione come le seguenti: 'la legge n. 898 del 1970 come vigente il 15 marzo 1989', 'la legge n. 898 del 1970 come modificata dalla legge 6 marzo 1987, n. 74' nelle quali il nome del testo è accompagnato da opportune specificazioni temporali (esplicite o implicite). Tali formulazioni fanno quindi riferimento rispettivamente ai risultati di tutte le modificazioni testuali e temporali verificatesi fino ad un determinato momento (ad esempio, al 15 marzo 1989) o a tutte le modifiche testuali verificatesi fino ad un determinato atto di modifica (ad esempio, la legge modificatrice è 6 marzo 1987, n. 74).

Disposizione normativa. Chiamiamo disposizione normativa ogni unità testuale autonoma contenuta in un testo normativo (cioè ogni unità che risulta dalla suddivisione del testo in elementi ciascuno dei quali sia sintatticamente ben formato e dotato di significato compiuto). Al livello più basso, le disposizioni normative corrispondono ai periodi (enunciati) inclusi nei testi normativi. Ma potranno considerarsi disposizioni anche le partizioni superiori del testo, come i commi, gli articoli, i capi, i titoli, ecc. Anche l'intero testo di un provvedimento determinato, considerato unitariamente, costituisce una disposizione.

4.1.4 Storia di un disposizione normativa (disposizione normativa virtuale)

Anche le singole disposizioni contenute nel testo normativo, come l'intero testo, sono entità suscettibili di mantenere la propria identità nel tempo, nonostante le proprie trasformazioni. Ogni disposizione, infatti, non è identificata dal proprio contenuto, ma dal proprio nome (la propria designazione) che contrassegna una determinata posizione o locazione all'interno del testo normativo (il terzo capo della legge x, l'art. y del regolamento z, ecc.).

Qualora nel corso del tempo vengano assegnati diversi contenuti testuali al medesimo nome di disposizione, si ritiene che la disposizione abbia mantenuto la propria identità, mutando contenuto (o versione). Il nome di una disposizione normativa non accompagnato da un riferimento temporale implicito o esplicito non designa quindi una specifica versione di una disposizione normativa, ma una locazione all'interno di un testo normativo, il cui contenuto testuale può mutare nel tempo, in seguito a modifiche testuali (che possono aggiungere o eliminare, dal testo della disposizione, parole, lettere, commi, ecc.). Come per l'intero testo normativo, anche rispetto alla singola disposizione possiamo quindi distinguere le nozioni di versione di una disposizione normativa, di storia di una disposizione normativa, e di disposizione normativa virtuale.

Versione di una disposizione normativa. Parliamo di versione di una disposizione normativa per indicare lo specifico contenuto testuale che la disposizione è venuta ad assumere in un determinato un momento.

Versione originaria di una disposizione normativa (disposizione originaria). La versione originaria di una disposizione normativa è quella inclusa nel testo normativo originario. In questi casi parliamo semplicemente di disposizione originaria.

Versione vigente (nel momento x). La versione vigente (nel momento x) di un disposizione normativa è il contenuto testuale della disposizione nel momento x in cui il testo viene interpretato o applicato. In questi casi parliamo semplicemente di disposizione vigente.

4.1.5 Ambito di applicabilità

Per *ambito di applicabilità* di una disposizione normativa (o di una sua versione) **d** intendiamo l'insieme delle situazioni possibili nelle quali d trova applicazione, cioè l'insieme delle situazioni possibili in cui la conseguenza giuridica di d è destinata a valere, nel contesto normativo presente, in forza di d stessa.

La disposizione d vale nel contesto normativo presente poiché future modifiche di tale contesto normativo (ad esempio, l'emanazione di nuove eccezioni a d, o l'abrogazione di d stessa) non incidono sull'ambito di applicabilità attuale di d.

L'ambito di applicabilità è determinato mediante la caratterizzazione sostanziale delle situazioni interessate, descrizione espressa dalla fattispecie della norma interessata, e mediante la loro collocazione nello spazio e nel tempo. Distinguiamo corrispondentemente tra applicabilità sostanziale, applicabilità spaziale e applicabilità temporale (che sarà quella di maggiore interesse per questa tesi, vedi figura 4.1).

4.1.6 Nessi normativi

Per *nesso normativo* intendiamo ogni possibile rapporto che si stabilisca tra due (o più) disposizioni normative. Nei nessi normativi normalmente si può distinguere una disposizione che svolge una funzione attiva e una disposizione che svolge una funzione passiva.

La *disposizione attiva* è quella che la cui emanazione istituisce o realizza il nesso normativo (ad esempio, è la disposizione abrogante che istituisce la relazione di abrogazione rispetto alla disposizione abrogata, è la disposizione derogante che instaura la relazione di deroga rispetto alla norma derogata, ecc...).

La *disposizione passiva* è quella che subisce l'instaurazione del nesso.

La disposizione attiva è di regola successiva nel tempo rispetto alla disposizione passiva (le due disposizioni, tuttavia, possono anche essere contenute nello stesso atto), e può contenerne una designazione.

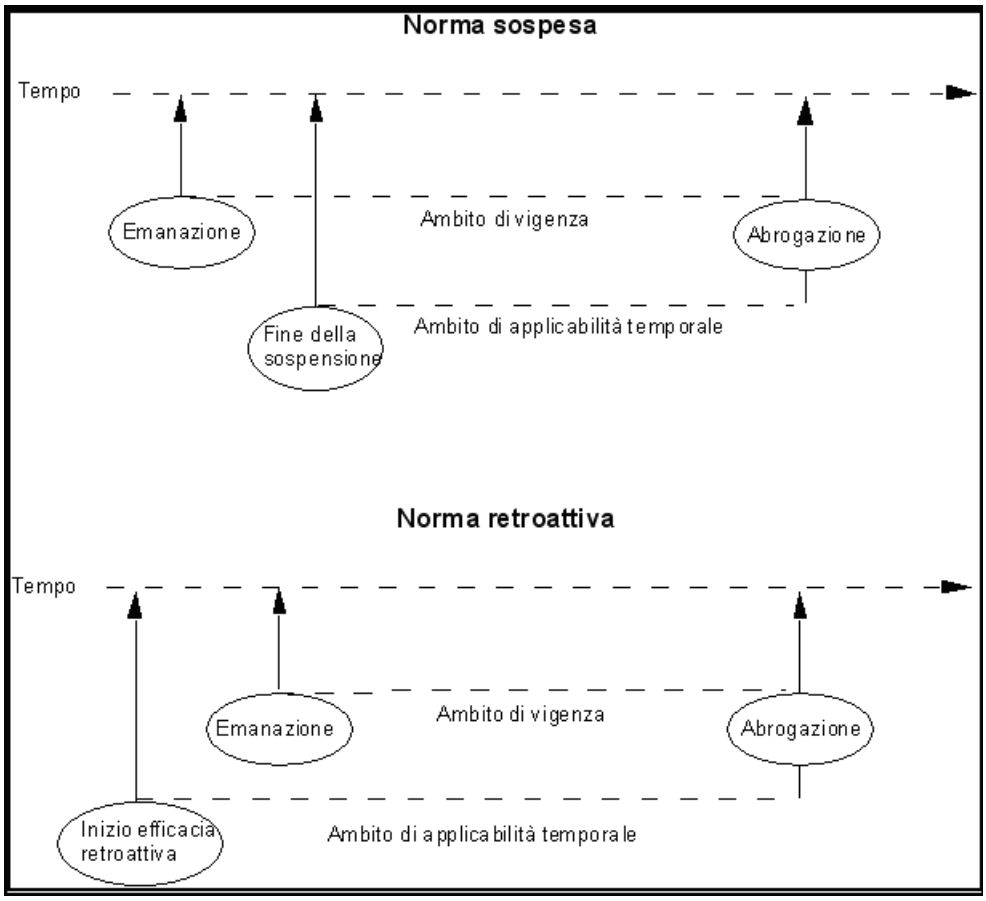

Figura 4.1: Esempio di ambito di validità temporale

4.1.7 Designazione normativa

Per *designazione normativa* intendiamo ogni espressione intesa ad identificare un atto normativo o una partizione di un atto normativo, cioè ogni espressione che funga da "nome" o "descrizione definita" di tale atto o partizione : "l'articolo quattro della legge Primo dicembre 1970, n. 898", "l'articolo 564 del codice penale", "il secondo comma dell'articolo 252 del codice civile", "gli articoli uno e 2 della presente legge", "il comma precedente", "il regio decreto 9 luglio 1939, n. 1238", "l'articolo 45 del regolamento edilizio", "l'art. 7 del regolamento d'igiene", "la delibera 16569/75".

Estendiamo la nozione di designazione normativa anche alle espressioni intese a denotare un insieme di atti normativi o di loro partizioni: "le disposizioni di cui agli articoli 155, 156, 255, 258, 260, 261, 262 del codice civile", "le disposi-

zioni in materia di arredo urbano", "le disposizioni incompatibili con il presente regolamento". Possiamo distinguere due tecniche di designazione delle disposizioni passive, che chiamiamo rispettivamente designazione sintattica (citazione in senso stretto) e designazione semantica.

Designazione sintattica (o citazione in senso stretto). La designazione sintattica (che chiameremo anche citazione) individua la disposizione passiva mediante caratteristiche "esteriori" di questa, come l'indicazione del testo normativo in cui la disposizione passiva è compresa e la denominazione della partizione del testo normativo corrispondente a tale disposizione all'interno del testo normativo ("l'articolo 6 della legge 1 dicembre 1970 n. 898", "il comma precedente") o l'indicazione dei vocaboli che vi compaiono (le disposizione che contengano l'espressione "patria potestà").

Designazione semantica. La designazione semantica individua la disposizione passiva sulla base del significato di questa: l'interpretazione della disposizione passiva è necessaria per capire se essa rientri o meno nel riferimento della disposizione attiva. Ciò si verifica, ad esempio, in espressioni come "le norme incompatibili con la presente legge", "le disposizioni in materia di contratti", "i regolamenti speciali", "le norme statali nella materia x", ecc.

Nel seguito di questo elaborato si farà riferimento principalmente alle designazioni sintattiche.

Sotto un diverso profilo, possiamo invece distinguere tra designazione statica e designazione dinamica:

- La designazione statica richiama la disposizione citata nella versione che questa aveva nel momento in cui è stata effettuata la citazione, o in un determinato momento precedente. Nell'interpretazione del testo richiamante si deve considerare solo tale versione, rispetto alla quale le modificazioni successive della disposizione richiamata non hanno alcun rilievo.

- La designazione dinamica richiama la disposizione citata nella versione di volta in volta vigente nei diversi momenti rispetto ai quali si deve ricostruire il significato della disposizione richiamante. Di regola, il momento cui far

riferimento per individuare il testo richiamato dalla citazione dinamica è quello nel quale si è verificata la fattispecie cui deve essere applicata la disposizione richiamante.

La distinzione tra designazione statica e designazione dinamica ha rilievo solo nelle designazioni sintattiche (poiché in assenza di una specifica indicazione della disposizione richiamata, non sarebbe possibile specificare la natura del riferimento).

4.1.8 I nessi normativi: loro distinzione in base agli effetti

Nelle pagine seguenti prenderemo in considerazione le principali tipologie di nessi normativi, individuate a seconda del loro impatto sulle norme interessate.

Modificazioni e rinvii

La prima grande distinzione da operare all'interno dei nessi normativi è quella tra le modificazioni e i rinvii. Le modificazioni (o modifiche) sono nessi normativi caratterizzati dal fatto che la disposizione attiva incide sulla disposizione passiva, eliminandola, cambiandone il testo o cambiandone la portata normativa (pur lasciandone immutato il testo). Questa incidenza manca invece nel rinvio, in cui la disposizione attiva si avvale della disposizione passiva per completare il proprio significato, senza influire su quest'ultima. Nelle pagine seguenti prenderemo in esame le modificazioni, considerandone approfonditamente le tipologie.

Modificazioni totali e parziali Un primo criterio di classificazione delle modificazioni attiene all'ampiezza della modifica apportata. La modificazione è totale quando comporta l'eliminazione dell'intera disposizione passiva; è parziale quando tale disposizione rimane in vigore, seppure con un contenuto testuale parzialmente diverso. Ovviamente, una modificazione potrà considerarsi totale o parziale a seconda della disposizione presa in considerazione: ad esempio, l'abrogazione di un articolo è una modificazione totale dell'articolo, mentre è una modificazione parziale dell'atto normativo che contiene quell'articolo. Più in generale, valgono le seguenti relazioni:

1. La modificazione testuale (totale o parziale) di una partizione del testo comporta una modificazione parziale, del medesimo tipo, delle partizioni di livello superiore. Ad esempio, l'abrogazione di un articolo comporta la parziale abrogazione della sezione in cui l'articolo è inserito

2. La modificazione totale di una partizione di livello superiore comporta la modificazione totale di tutte le partizioni di livello inferiore. Ad esempio, l'abrogazione di un articolo comporta l'abrogazione di tutte le partizioni che ad esso appartengono. Merita una speciale considerazione la sostituzione, che può comportare l'abrogazione delle partizioni di livello inferiore, anziché la loro sostituzione. Se un nuovo capo sostituisce l'intero capo di un provvedimento x, l'art. 13 della vecchia sezione può non trovare un corrispondente art. 13 nel nuovo regolamento. In ogni caso, il nuovo art. 13 può non avere nulla a che fare con il vecchio sotto il profilo contenutistico

3. La modificazione parziale di una partizione di livello superiore comporta la modificazione (parziale o totale) di almeno una partizione di livello inferiore

Salvo diverse specificazioni, assumeremo come punto di riferimento l'intero atto normativo, e quindi parleremo di modificazione totale nei casi in cui viene abrogato o sostituito un intero atto, e di modificazione parziale in tutti gli altri casi.

Esempio di modificazione totale: *E' abrogato il Regolamento di Polizia Urbana, approvato con deliberazione del Consiglio 6 novembre 1925*

Esempio di modificazione parziale: *Sono abrogati gli articoli da 6 a 10 del Regolamento sui diritti di informazione dei cittadini nei confronti dell'amministrazione comunale, approvato con deliberazione del Consiglio comunale 20 maggio 1987*

Modificazioni testuali, temporali, e materiali In relazione alla natura dell'impatto della modifica sulla disposizione passiva, distinguiamo:

- modificazioni testuali, che eliminano la disposizione passiva, o ne cambiano il testo

- modificazioni temporali, che incidono sull'ambito temporale di applicabilità della disposizione passiva
- modificazioni materiali, che modificano il contenuto normativo della disposizione passiva, senza incidere sul suo testo

Esempi di modificazioni testuali:

- l'articolo 26 del regolamento d'igiene del 1993, è sostituito dal seguente: "Articolo 26 ..."
- l'articolo 26 del regolamento d'igiene del 1993, è abrogato

Esempio di modificazione temporale:

- la disciplina temporanea prevista dall'articolo 22 del regolamento dei dirigenti rimane in vigore fino al quattro gennaio 1994

Esempi di modificazioni materiali:

- in deroga alla previsione dell'articolo 26 del regolamento d'igiene del 1993, è permessa la vendita di cibi all'aperto ...
- l'articolo 26 del regolamento d'igiene si applica anche alla vendita di cibi conservati

Modificazioni testuali Le modificazioni testuali hanno il seguente effetto: la disposizione passiva x nel momento t2 immediatamente successivo all'entrata in vigore della disposizione attiva ha un testo diverso da quello che x aveva nel momento t1. In altri termini, la versione della disposizione passiva vigente in t2 risulta diversa da quella vigente in t1. La differenza tra le due versioni può consistere solo nella eliminazione di una sequenza testuale (abrogazione), nella sola aggiunta di una nuova sequenza testuale (inserimento), o in entrambe le operazioni, cioè nell'eliminazione di una sequenza testuale e nell'aggiunta di una nuova sequenza al posto di quella eliminata (sostituzione).

Come risulta dalla seguente figura 4.2, le modifiche si configurano come nessi binari quando si considerino le due disposizioni virtuali implicate (o le storie delle due disposizioni), essi si configurano invece come ternari quando si considerino le

diverse versioni della disposizione passiva: l'entrata in vigore della disposizione attiva produce il passaggio da una versione della disposizione passiva alla versione successiva. Più esattamente essi includono un legame soppressivo (della versione precedente), che indichiamo con il simbolo "-", e un legame introduttivo (della nuova versione) che indichiamo con il simbolo "+". Si ha un nesso binario solo nei casi limite dell'abrogazione totale, in cui manca la versione successiva, e l'inserimento totale, in cui manca, la versione precedente.

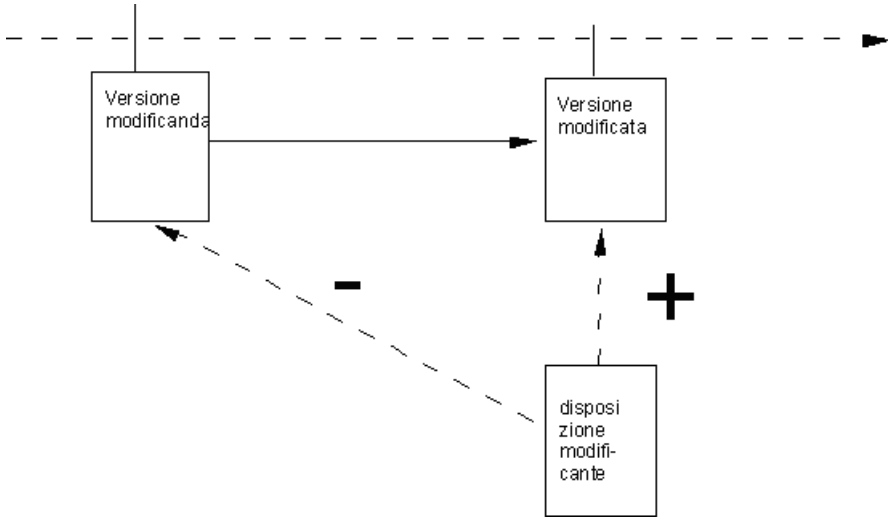

Figura 4.2: Esempio di modifica

Abrogazione L'abrogazione comporta l'eliminazione di una sequenza testuale da un testo normativo, senza che tale sequenza sia sostituita da un nuovo testo

Inserimento (o aggiunta testuale) L'inserimento comporta l'introduzione di una nuova sequenza testuale all'interno di un testo normativo

Sostituzione (o scambio testuale) La sostituzione comporta l'eliminazione di una sequenza testuale da un testo normativo e l'inserimento di una nuova sequenza al posto di quella. La sostituzione unisce quindi i momenti dell'abrogazione e dell'inserimento

Il tempo nelle modifiche testuali Di solito si ritiene che l'evento della modifica testuale (inclusa l'abrogazione) si verifichi e si concluda nel momento

nel quale la disposizione attiva modificante entra in vigore. In questa prospettiva, si reputa che la successiva abrogazione di quest'ultima disposizione non possa incidere sulla disposizione passiva già modificata: l'abrogazione della disposizione abrogante non comporta la riviviscenza della disposizione abrogata. Diverso è il caso in cui la disposizione modificante fosse illegittima e quindi possa ritenersi inefficace fin dall'inizio (come nel caso in cui detta abrogazione venga annullata da un organo giudiziario): in questa ipotesi si può ritenere che venga meno retroattivamente l'evento stesso della modifica già realizzatasi, e che quindi si ripristini la situazione precedente alla modifica. Possono esistere, inoltre, disposizioni che stabiliscono un'abrogazione (inserimento o sostituzione) per un momento successivo alla loro entrata in vigore. Tali disposizioni, che chiamiamo abrogazioni (inserimenti o sostituzioni) ritardate, potranno essere ovviamente modificate o abrogate prima che abbiano avuto efficacia.

Modifiche testuali e dinamica del testo normativo vigente Le modifiche testuali si caratterizzano per la propria attitudine ad incidere sul testo normativo vigente, o meglio, sulle successive versioni del testo normativo vigente susseguitesi nel tempo. Come abbiamo già osservato, ogni modifica testuale produce infatti una nuova versione della disposizione modificata, sulla quale agisce la modifica successiva. Una determinata versione del testo modificato rimane in vigore nell'intervallo di tempo tra la modifica che ha creato tale versione e la successiva modifica che mette termine a quella stessa versione, creando la versione successiva.

In questa prospettiva, la storia di una disposizione normativa si configura come una sequenza di versioni successive: il passaggio dall'una all'altra versione avviene tramite l'intervento di una disposizione modificante. La nostra analisi della sequenza delle versioni modificate si ricollega alla distinzione dei nessi normativi in base al momento della loro efficacia. Il momento dell'efficacia del nesso modificante segna il momento della uscita di vigore della disposizione modificanda e dell'entrata in vigore della versione modificata. Di regola tale momento è quello dell'entrata in vigore della disposizione modificante, ma può essere, come abbiamo visto, un momento successivo o precedente (qualora il nesso modificante abbia un'efficacia anticipata o posticipata).

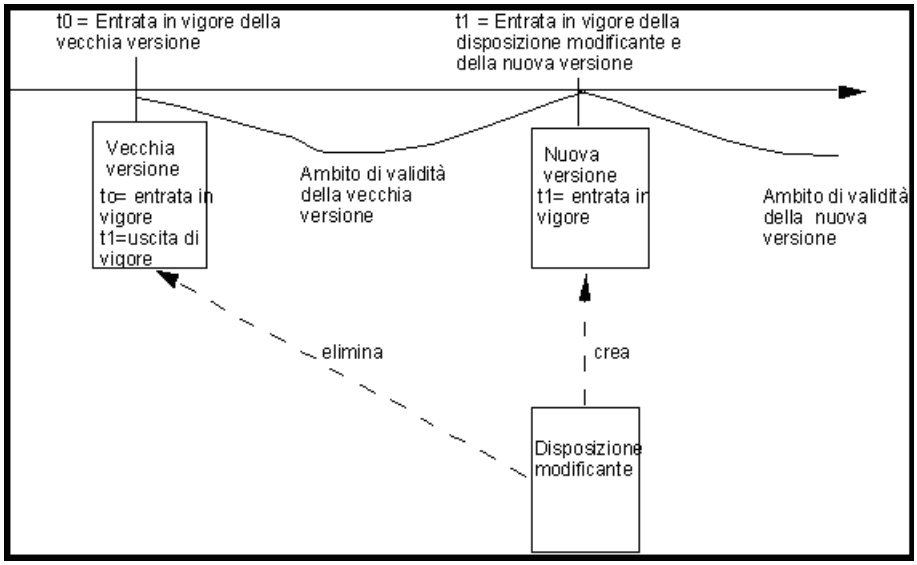

Figura 4.3: La dinamica temporale delle modifiche testuali

Quando si distinguono le due versioni della disposizione passiva (quella precedente e quella successiva alla modifica), la relazione binaria tra disposizione attiva e disposizione passiva diventa una relazione ternaria (o, se si preferisce una duplice relazione binaria), che collega la disposizione passiva con ciascuna di tali versioni. In questa prospettiva, le ipotesi di abrogazione e di introduzione rappresentano casi limite, nei quali manca, rispettivamente, la versione modificata o la versione da modificare (o, se si preferisce, in cui tali versioni sono vuote).

Il testo normativo non è quindi un unico documento testuale, ma una sequenza di versioni testuali collegate da operazioni modificatrici, o, se si preferisce, un'entità virtuale che evolve nel tempo, mutando il proprio contenuto. Il testo virtuale non è solo una costruzione operata dal giurista, ma costituisce un'entità ben presente nel discorso legislativo. Le stesse disposizioni modificatrici non operano sul testo originale, ma sul risultato delle modifiche precedenti: tali disposizioni fanno di regola riferimento, come si è visto, al testo vigente nel momento della loro entrata in vigore - l'art. x della legge y, come risulta da tutte le successive operazioni modificatrici operate fino a quel momento - e non al documento originale, che invece esiste immutabile, fissato al momento in cui è stato emanato.

Pur rappresentando il continuo riferimento ideale del legislatore, del giurista, e del cittadino, non sempre il testo "virtuale" vigente può essere identificato facilmente: la costruzione del testo vigente richiede l'esatta individuazione del documento di partenza e di tutte le successive operazioni modificatrici, la concatenazione e l'esecuzione di tali operazioni. Spesso la ricostruzione del testo non è corretta, o ancor peggio, il testo vigente viene richiamato (sia dal giurista che dal legislatore) senza averne accertato l'effettivo significato normativo. La ricostruzione automatica del testo vigente rappresenta un obiettivo fondamentale nei più avanzati sistemi di documentazione giuridica automatica.

Modificazioni temporali Tali nessi comportano una modificazione della sfera della validità o della applicabilità temporale della disposizione passiva, senza modificare il testo di tale disposizione. La disposizione modificante **d2** si limita cioè ad affermare che la disposizione **d1** è prorogata fino alla data x, sospesa fino alla data y, ecc. La modifica dell'ambito temporale può avvenire (raramente) anche modificando il testo della disposizione passiva (ad esempio, aggiungendo o modificando i riferimenti temporali contenute nelle norme interessate, cioè intervenendo su espressioni del tipo " fino alla data x i soggetti y possono presentare la dichiarazione ...", "a partire dalla data x i soggetti y sono tenuti a ...". In questo caso tuttavia, la modifica si considera testuale, e quindi rientra nelle ipotesi considerate in precedenza. Tra i nessi normativi modificanti l'ambito temporale, qui consideriamo, le proroghe e le sospensioni.

Proroga La relazione di proroga incide sulla validità della norma giuridica (la sua appartenenza all'ordinamento). La disposizione attiva (prorogante) stabilisce che la disposizione prorogata, che doveva uscire di vigore nel giorno z, rimane in vigore indefinitamente o fino ad un momento determinato successivo a z.

Sospensione La relazione di sospensione non incide sulla validità della disposizione sospesa, ma solo sul suo ambito temporale di applicabilità. La disposizione passiva sospesa dovrà cioè non applicarsi alle fattispecie verificatesi nel periodo della sospensione

Modificazioni materiali Riuniamo tra le modificazioni materiali (o meramente sostanziali) tutte le ipotesi in cui viene modificata la portata normativa di una disposizione senza modificarne il testo: la deroga, la estensione, e la modificazione interpretativa. Pur essendo molto importanti, tale tipo di modifica non è rilevante al fine di questa tesi.

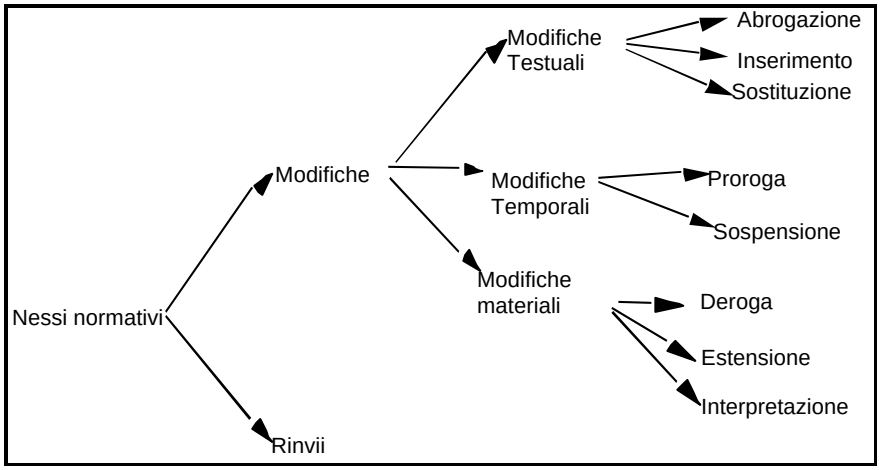

Figura 4.4: Schema riassuntivo dei tipi di nessi normativi

4.2 Modello di atto normativo

Lo sviluppo di questo elaborato parte dal lavoro svolto nella Tesi di Laurea [2] e dalle considerazioni in [8].

In particolare verrà tenuto presente il *modello di atto normativo* espresso in XML, adatto alla descrizione di testi normativi. Tutti i documenti XML rappresentanti leggi trattati nel presente elaborato dovranno essere conformi a tale modello, che è stato tradotto per comodità in un XML Schema.

4.2.1 Progetto NIR

Il modello prende forma grazie alla DTD pubblicata dal progetto Norma in Rete. Il progetto (si veda [13]), promosso dal Centro Nazionale per l'Informatica nella Pubblica Amministrazione (CNIPA, ex-AIPA) su proposta del *Ministero di Grazia e Giustizia*, intende semplificare l'accesso alle norme da parte dei cittadini e offrire supporto alle amministrazioni per la produzione, la divulgazione

e la pubblicazione attraverso Internet dei documenti di interesse giuridico. In realtà la DTD è stata arricchita della temporalità in modo da poter descrivere versioni generate da modifiche. Possiamo dire che i requisiti base di un modello rappresentante testi normativi sono:

1. rispetto della gerarchia articolato-capo-articolo-comma
2. capacità di ogni livello della gerarchia di rispettare versioni multiple, ognuna caratterizzata dai propri attributi temporali

4.2.2 Struttura

In figura 4.5 si può vedere la rappresentazione dei soli elementi e nodi testo dell'XML Schema adottato. Gli attributi nel dettaglio saranno specificati più avanti elemento per elemento, mentre il codice vero e proprio è nell'apposita appendice.

Possiamo notare che, sebbene l'XML permetta l'esistenza di testo semplice all'interno di ogni elemento, si è preferito accettare la presenza di tale testo solo all'interno di certi elementi. Infatti gli unici elementi che possono contenerlo sono :

- titolo: testo che indica il titolo della legge
- rubrica: testo che funge da riassunto o sommario per gli argomenti trattati negli articoli di ogni capo o nei commi di ogni articolo (lo stesso elemento rubrica è opzionale, e quindi lo sarà anche il corrispettivo elemento testo)
- ver (ma solo quando è figlio di un elemento comma): rappresenta il testo che ritroviamo all'interno dei commi di una legge

La DTD pubblica dal progetto NIR è stata arricchita degli attributi temporali in modo che fosse possibile descrivere atti normativi sottoposti a modifiche nel tempo. Il modello risultante è adatta allo studio dei metodi utilizzati per l'elaborazione digitale dei documenti rappresentanti leggi ad essa conformi.

Le dimensioni temporali presenti nel modello sono :

1. publication time

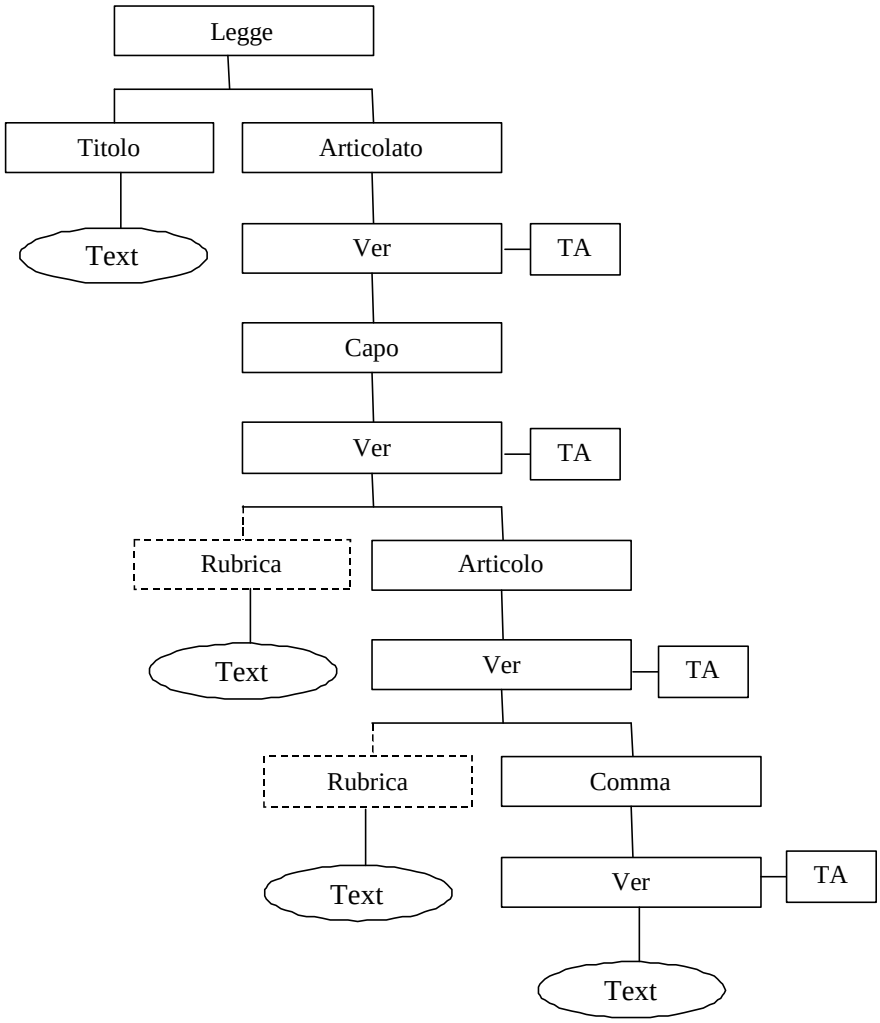

Figura 4.5: Elementi del modello adottato

2. validity time

3. efficacy time

4. transaction time

Publication Time: è la data in cui il testo normativo viene pubblicato sulla Gazzetta Ufficiale, ha la stessa semantica dell'event time. Differentemente dalle altre dimensioni temporali è considerato un momento piuttosto che un intervallo, ed è quindi rappresentabile tramite una sola data. Es : pt = "15/12/1999"

significa che la Gazzetta Ufficiale che ospita la prima pubblicazione di tale legge porta quella data.

Valid Time: è il periodo in cui la norma è in vigore nella legislazione italiana. Tale periodo ha inizio solitamente 15 giorni dopo la data di pubblicazione sulla Gazzetta Ufficiale ed è di durata indeterminata, a meno che non sia diversamente disposto da un successivo atto. La semantica è quella del valid time dei database temporali, e rappresenta il tempo in cui la norma appartiene al sistema giuridico reale.

Efficacy time: identifica il tempo in cui la norma è efficace e quindi applicabile al caso concreto. Usualmente coincide con il vigore, ma può capitare che una norma venga (ad esempio) sospesa, perdendo quindi efficacia per un certo periodo, senza uscire di vigore. La semantica è sempre quella del valid time.

Transaction time: è il tempo in cui il testo normativo è memorizzato nel database. La semantica del tradizionale transaction time delle basi di dati temporali è mantenuta interamente.

Queste ultime tre dimensioni temporali sono degli intervalli e non dei momenti come il publication time. Sono quindi necessarie 2 date per individuare un intervallo. Se in un testo normativo aderente al nostro modello è indicata solo la data iniziale dell'intervallo significa che è da considerarsi a partire dalla data specificata fino alla fine dell'asse dei tempi ($+\infty$).

Es.: ttStart = "1/1/2000" ttEnd = "31/12/2000" significa che tale norma corrisponde alla versione memorizzata nel DB per tutto l'anno 2000, mentre se manca il riferimento a ttEnd significa che tale norma corrisponde alla versione memorizzata nel DB in ogni momento successivo al giorno indicato da ttStart.

Ora entriamo nel merito di ogni elemento per spigarne dettagliatamente la struttura e le funzioni funzioni svolte all'interno del modello. Affianco ad ogni attributo è indicato se è obbligatorio (r) o opzionale (o).

4.2.3 Legge

L'elemento legge è l'elemento root del documento. Ha il compito di racchiudere attributi che interessano tutti gli altri elementi, ma che è possibile esplicitare solo una volta. In figura 4.6 possiamo vedere la struttura di un elemento legge : vediamo che contiene l'attributo num che indica il numero della legge e l'attributo natura. Natura indica il tipo di legge: ad esempio una legge può essere un Decreto Legge, Decreto del Presidente della Repubblica, ecc...

Figura 4.6: Dettagli dell'elemento legge

4.2.4 Articolato

L'elemento articolato è il primo elemento a contenere attributi temporali. L'attributo pubblicazione indica la data di pubblicazione sulla Gazzetta Ufficiale e corrisponde al publication time. In figura 4.7 possiamo vedere la struttura di un elemento articolato. Gli attributi num e natura di legge assieme all'attributo pubblicazione di articolato, formano l'identificativo della legge. Ad esempio:

- legge.natura = "DL" (che sta per decreto legge)
- legge.num = "1"
- articolato.pubblicazione = "01/01/1980"

forma l'identificativo:

```
legge.natura + / + legge.num / +
anno(articolato.pubblicazione) = DL/1/1980
```

Grazie a questo identificativo una legge è sempre distinguibile da tutte le altre, ed è pertanto considerato una designazione normativa.

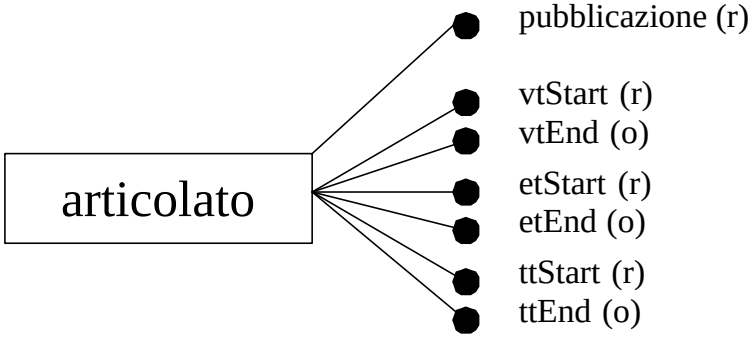

Figura 4.7: Dettagli dell'elemento articolato

Dimensione temporale	Istante Iniziale	Istante Finale
Validity Time	vtStart	vtEnd
Efficacy Time	etStart	etEnd
Transaction Time	ttStart	ttEnd

Tabella 4.1: Significato degli attributi temporali

Inoltre questo elemento racchiude gli intervalli di tutte le dimensioni temporali di cui abbiamo parlato nel paraggrafo 4.2.2. Ricordiamo che la data della fine dell'intervallo è opzionale in quanto se manca viene considerata automaticamente dal sistema come $+\infty$. In tabella 4.1 è possibile vedere il significato degli attributi che determinano gli intervalli.

Fissiamo ora uno dei cardini di questo modello: i tempi presenti all'interno dell'elemento articolato sono validi per tutti gli elementi del documento a meno che non vengano ridefiniti utilizzando gli elementi TA. Questo significa che, se non vengono trovati altri elementi TA capaci di dare la temporalità (opzionali all'interno degli elementi ver) , gli intervalli temporali di articolato vengono ereditati da tutti gli altri elementi del modello.

4.2.5 Elementi che generano versioni

Nei prossimi elementi trattati è racchiuso un meccanismo che potremo chiamare di *versioning temporale*. Tale meccanismo ci permette di fare modifiche ai documenti in modo conservativo e consistente. Possiamo dire che le modifiche sono conservative perché ad ogni modifica viene mantenuta traccia dello stato in cui

erano i dati prima della modifica (in pieno accordo con la filosofia introdotta dai database temporali), possiamo quindi dire che i nostri documenti sono *append-only*. Inoltre possiamo dire che le modifiche sono consistenti in quanto quando vengono apportate portano il documento in uno stato che può essere interrogato esattamente allo stesso modo di prima della modifica, e dare risultati esatti.

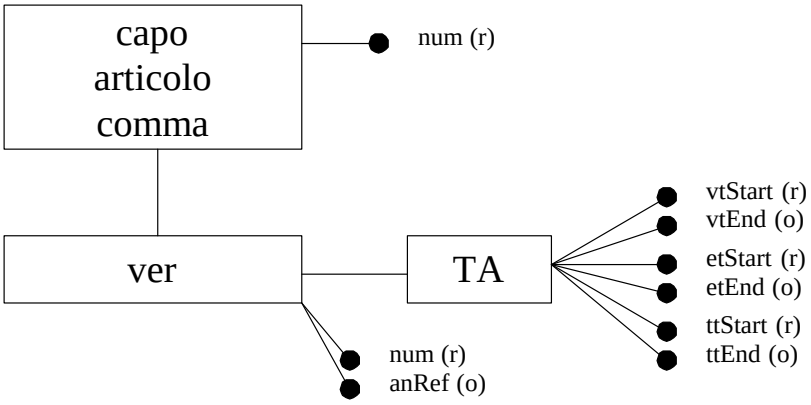

Figura 4.8: Dettagli del versioning temporale

In figura 4.8 sono racchiusi i nomi di tre elementi che si comportano allo stesso modo ed hanno la stessa struttura. La gerarchia capo - articolo - comma rappresenta la struttura che sta alla base di ogni legge italiana, e quindi anche del nostro modello. Questi tre elementi hanno l'attributo num che ne offre una numerazione progressiva, partendo da 1.

Le regole da rispettare sono molto semplici e racchiuse nel codice dell' XML Schema: ogni capo racchiude uno o più articoli; ogni articolo racchiude a sua volta uno o più commi.

La struttura dell'elemento ver comprende l'attributo num che serve per poter dare un numero alla versione in questione e l'attributo anRef (che sta per active norm Reference) che serve ad identificare la norma attiva che ha generato tale versione. Se la versione è quella relativa al documento originale allora viene omesso, altrimenti se è il risultato di una modifica testuale viene esplicitato.

TA (che sta per *temporal attributes*) sono gli elementi atti al contenimento della temporalità di ogni versione. Infatti hanno tutti gli attributi necessari

alla memorizzazione dell'inizio e della fine degli intervalli delle tre dimensioni temporali di interesse. La sintassi è la stessa che è stata adottata in articolato, ma è diverso il significato e la cardinalità. Mentre in articolato era possibile esplicitare un solo set di attributi temporali (e ciò è logico perché rispecchia la temporalità nel momento dell'approvazione della legge, che è una ed una sola), all'interno di un elemento ver è possibile esplicitare un qualsiasi numero di set di attributi temporali perché è possibile esplicitare un qualsiasi numero di TA; infatti un elemento TA coincide con un set completo di attributi temporali.

Grazie all'elemento ver è possibile esprimere ad ogni livello della gerarchia precedentemente indicata una o più versioni degli elementi sottostanti. Non solo, ma ognuna di queste versioni può essere associata ad uno o più elementi TA che ne rappresentano la temporalità. Infatti non è detto che una modifica di un articolo implichi una modifica *testuale* degli elementi comma, ma è possibile che la modifica sia di tipo temporale, e che modifichi cioè solo il TA ritenuto *corrente* (cioè quello che ha l'istante di modifica contenuto nel transaction time) e ne introduca uno nuovo con gli attributi temporali modificati, senza agire su tutti gli elementi della gerarchia sottostante. Ciò mette subito in evidenza come sia possibile evitare una grande ridondanza dei dati aggiungendo solo un elemento (ricordiamo che la ridondanza è un grande problema dei database temporali, in quanto sono *append only*).

Anche l'elemento articolato è collegato ad un elemento ver, che può a sua volta contenere dei TA. Infatti il comportamento di questo blocco è del tutto analogo a quello appena visto, la differenza sta solo nella struttura di articolato (cioè negli attributi che contiene).

Un elemento ver di articolato può contenere uno o più elementi capo. Qui il meccanismo del versioning perde un po' di significato in quanto è possibile fare una versione nuova di articolato e così cambiare tutta la parte testuale della legge, anche se una cosa del genere è difficile che accada: è preferibile farne una *ex novo*!

La temporalità presente all'interno dell'elemento articolato può essere intesa anch'essa come una versione: la versione originale della legge.

4.2.6 Formulazione di query sul modello adottato

Le collezioni di testi normativi memorizzati in formato digitale all'interno di computer sono solitamente gestite da tradizionali sistemi di *information retrieval*. Gli utenti possono accedere al contenuto di questi documenti specificando interrogazioni basate su parole chiave (*keyword*) che rappresentano l'argomento di interesse.

All'interno di [2], è stata estesa questa struttura di base offrendo la possibilità di esprimere specifiche temporali, utilizzate nella selezione e nella successiva ricostruzione temporale (consistente a tali specifiche temporali) dei testi normativi di interesse.

E' stato permesso agli utenti di integrare le richieste *testuali* di atti normativi con aspetti *temporali*. Questa temporalità specificata dall'utente diventa indispensabile qualora si voglia conoscere la versione testuale di una legge applicabile in un dato periodo, o la sua esatta formulazione in un particolare istante del passato.

La ricostruzione di versioni di un documento in modo che sia consistente a particolari vincoli temporali corrisponde ad una operazione di *temporal slicing* (si veda [6]).

In accordo alle specifiche necessità dei sistemi informativi legali che gestiscono atti normativi mutevoli nel tempo, il modello introdotto in [2] supporta le interrogazioni che utilizzano la suddetta operazione di temporal slicing ed hanno il formato XQuery:

```
FOR $a IN path
WHERE constrains on $a
RETURN const-tree(document($a), temporal specs)
```

Con il costrutto `FOR $a IN path`, la variabile `$a` assume di iterazione in iterazione il valore specificato all'interno dell'espressione XPath `path`. Con il costrutto `WHERE constrains on $a` si applicano a tutti i possibili valori che `$a` assume determinati vincoli. Questi vincoli sono di tipo testuale e si esplicitano specificando una o più *keyword*: le parole chiave dovranno essere contenute nei risultati riportati.

L'operatore `const-tree()` è utilizzato per la ricostruzione del documento corrente in modo che sia *temporalmente consistente* con le specifiche temporali esplicitate all'interno di `temporal specs`. Più precisamente le specifiche temporali definiscono le condizioni temporali che i risultati devono soddisfare. Tali specifiche sono l'unione dei predicati di selezione sui valori delle quattro dimensioni temporali supportate dal modello adottato, in accordo con la comune sintassi di TSQL2 (introdotta nel paragrafo 1.1.3).

Un esempio di query che rispetta la sintassi XQuery precedentemente espressa potrebbe essere quella in figura 4.9.

```
FOR $a IN articolo//comma
WHERE contains ($a,'inquinamento')
RETURN const-tree(document($a),
VALID CONTAINED-IN PERIOD ('1970-01-01','2000-12-31') AND
TRANSACTION CONTAINS '1980-01-01')
```

Figura 4.9: Esempio di query espressa con XQuery

L'operatore `const-tree()` prende in ingresso il documento `document($a)` ovvero l'intero documento XML. Tale documento:

- rispetta i requisiti temporali specificati in `temporal specs`
- contiene almeno un nodo che fa match con i requisiti testuali all'interno del `path` specificato

In seguito ne effettua la *ricostruzione*, cioè ad ogni livello della gerarchia seleziona solo quei nodi del documento che hanno pertinenza temporale tale da soddisfare i vincoli imposti nella query. E' importante notare che l'intero documento è stato selezionato sulla base di attributi temporali che rappresentano l'*unione* di tutte le temporalità dei nodi, è quindi possibile che porzioni del documento XML non soddisfino i requisiti imposti dalla query.

4.2.7 Una architettura esistente

Il modello descritto è stato implementato all'interno di [2] in un prototipo di sistema per la gestione di una collezione di atti normativi soggetti a variazioni

nel corso del tempo. In tale sistema è prevista l'interrogazione dei documenti memorizzati in formato XML tramite l'esplicitazione di requisiti testuali e temporali.

Quel prototipo di sistema è stato implementato utilizzando un approccio a strato (che chiameremo *stratum*, si veda anche [8]) e supporta:

1. la memorizzazione di testi normativi che variano nel tempo rappresentati come documenti XML conformi ad un particolare XML Schema

2. la modifica di testi normativi precedentemente memorizzati

3. l'interrogazione dei testi normativi memorizzati con la possibilità di ricerca tramite keyword e vincoli temporali

Fra le operazioni supportate si tratterà approfonditamente l'interrogazione. Ogni query viene trasformata dallo stratum in una espressione equivalente in modo che possa essere passata ad un *XML Engine*. Quando l'Engine ha svolto le operazioni che gli sono state richieste nell'interrogazione, questo riporta i risultati ottenuti allo stratum, che compie ulteriori elaborazioni prima di riportarli all'utente. Possiamo dire che i risultati che l'XML Engine riporta allo stratum dopo l'esecuzione di una query sono solo risultati *parziali*; infatti lo stratum esegue la ricostruzione di tali documenti attraverso l'utilizzo dell'operatore `const-tree()` prima di riportarli all'utente come risultati *finali*.

Questo approccio si basa completamente sui metodi messi a disposizione dall'Engine, compresa l'ottimizzazione della query. Nell'implementazione trattata in [2] è stato utilizzato Oracle 9i sia come Engine che come Repository.

Gli Engine esistenti non conoscono la semantica temporale espressa nella query di esempio (ricordiamo che TSQL2 non è uno standard), quindi allo stratum è lasciata la gestione degli aspetti temporali del modello adottato, in modo da provvedere ad un completo supporto per la gestione di atti normativi che mutano nel tempo. L'architettura utilizzata in [2] è rappresentata in figura 4.10; lo stratum consiste nei componenti Preprocessing, Query Processor e Update Processor. I documenti XML sono memorizzati all'interno dell'XML Repository in una tabella che ha lo schema riportato in figura 4.11. Ogni tupla appartenente

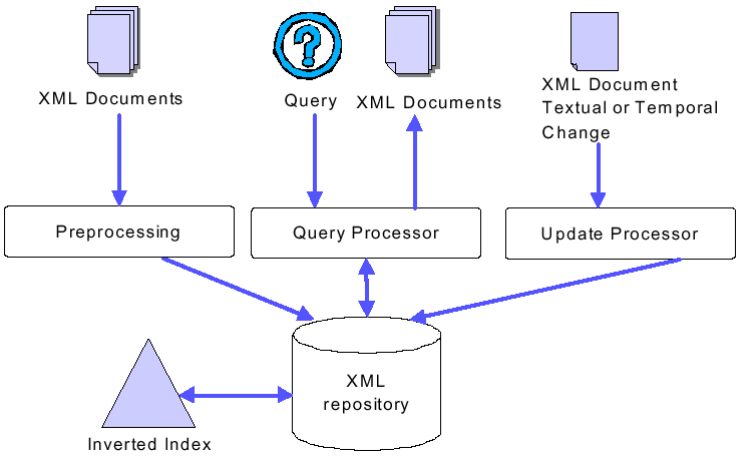

Figura 4.10: Architettura adottata nell'approccio a *stratum*

```
tnorms ( ID, XML-DOC, TYPE, PUBLICATION, VT-START,
VT-END, ET-START, ET-END, TT-START, TT-END)
```

Figura 4.11: Relazione adottata in nell'approccio a *stratum*

a tale relazione rappresenta un documento XML temporale, il cui testo è interamente memorizzato all'interno del campo `XML-DOC` in forma non strutturata (cioè testo semplice). La tabella contiene anche attributi addizionali che rappresentano la temporalità, essi sono: `PUBLICATION, VT-START, VT-END, ET-START, ET-END, TT-START e TT-END` e sono rappresentativi dell'*intera* temporalità dell'atto normativo. Infatti nel modello trattato in [2] i tempi relativi ad ogni livello della gerarchia sono interamente contenuti all'interno dei tempi degli elementi *ancestor*. E' quindi possibile dire che gli attributi nella relazione *contengono* quelli di ogni altro elemento, perchè soddisfano l'operatore `CONTAINS` di TSQL2. Inoltre, per rendere più efficiente le query testuali, è stato creato un inverted index sul testo contenuto all'interno degli elementi `titolo` e `comma`.

Quando una query di tipo XQuery come quella in figura 4.9 viene sottoposta al sistema, il Query Processor all'interno dello stratum la traduce nella corrispettiva query SQL (figura 4.12) da passare all'Engine adottata. All'interno di quella query i vincoli temporali dell'interrogazione originale sono stati tradotti nei corrispettivi vincoli SQL sulle colonne della relazione adottata (figura 4.11), mentre il vincolo strutturale viene passato all'XML Engine attraverso

```
SELECT L.XML-DOC.extract('articolo//comma').getStringVal()
FROM tnorms L
WHERE (VT-START <= '2000-12-31')
AND (VT-END >= '1970-01-01')
AND (TT-START <= '1980-01-01')
AND (TT-END >= '1980-01-01')
AND CONTAINS (L.XML-DOC, inquinamento WITHIN comma) > 0
```

Figura 4.12: Esempio di query con XQuery

l'utilizzo della funzione `extract()` richiamato sull'apposito campo `XML-DOC`. A questo punto l'interrogazione viene eseguita e, quando lo stratum riceve i risultati, si effettua l'operazione detta temporal slicing (attraverso la ricostruzione con `const-tree()`) in modo da poter restituire all'utente documenti XML consistenti alle specifiche temporali.

L'approccio adottato in [2] presenta tuttavia dei punti deboli. Dai test condotti utilizzando il prototipo sviluppato si è visto che al crescere del numero dei documenti interessati dalla query aumenta il tempo necessario per la risoluzione dell'interrogazione in modo sub-lineare per la selezione delle tuple e lineare per la ricostruzione dei documenti. Tale tempistica risulta comunque inaccettabile per query su grandi collezioni di documenti XML. Inoltre, il metodo di risoluzione adottato in [2] è interamente basato sull'utilizzo della main memory: una volta che la query (come quella descritta precedentemente) è stata tradotta nel corrispondente SQL dallo stratum, tutte le tuple che fanno match (ognuna delle quali contiene un intero documento XML) vengono riportate allo stratum stesso che applica su ognuna l'operatore di ricostruzione `const-tree()`. Allo spazio in memoria utilizzato per contenere temporaneamente le tuple restituite dal DB si somma un ulteriore spazio per una successiva elaborazione; infatti l'operatore `const-tree()` effettua la ricostruzione di una tupla per volta (quindi di un documento XML per volta) e per ognuna necessita del corrispettivo albero DOM, la cui creazione può richiedere fino a quattro volte dello spazio di memoria occupato dal documento testuale.

All'utilizzo della memoria centrale appena descritto, consegue che per rispondere a molte richieste in contemporanea, quel sistema ricorrerà inevitabilmente

all'utilizzo della memoria virtuale (su disco fisso) con un drammatico calo delle prestazioni.

Parte II

Interrogazione Efficiente di Documenti XML Temporali

Capitolo 5

Architettura e strutture di un Temporal XML Query Processor

Lo scopo della presente tesi è analizzare i meccanismi esistenti di interrogazione di documenti XML temporali e di proporne di più efficienti.

Per fare questo si adotterà una nuova architettura che non comprende lo stratum proposto in [2] in quanto non saranno più necessarie né preprocessing né postprocessing dei risultati ottenuti dalle interrogazioni.

Ciò che si intende creare è un *Temporal XML Query Processor* che supporti il meccanismo di versioning temporale introdotto nel paragrafo 4.2.5. Tale Processor è in grado di effettuare autonomamente (cioè senza l'iterazione con altri strati) interrogazioni con vincoli strutturali e temporali su una base di tali XML temporale. La nuova architettura permette anche la scalabilità e la multiutenza del sistema. Un fattore fondamentale per la multiutenza, e quindi per sottoporre più interrogazioni contemporaneamente senza che il sistema vada in *trashing* nel tentativo di fornire risposte, è un attento utilizzo della memoria principale.

Come caso particolare si farà riferimento ad un sistema per l'interrogazione di atti normativi rappresentati con documenti XML che soddisfano un determinato modello. Si ritiene infatti che un atto normativo sia un buon esempio di documento "*dinamico*", la cui *evoluzione* nel tempo passa attraverso successive modifiche che generano nuove *versioni* ad ogni suo livello interno.

Si vedrà anche come sia possibile suddividere dei documenti XML rappresentanti atti normativi in un certo numero di strutture più piccole aventi il

significato di disposizioni normative.

I concetti alla base di questa tesi non sono relativi al solo campo normativo, ma ci si propone una loro applicabilità *generale*, cioè ad ogni sistema che dia la possibilità di interrogare una base di dati contenente una grande collezione di documenti XML temporali di qualsiasi tipo.

Si tratterà solamente *l'interrogazione* dei testi normativi, in quanto è stata l'operazione che ha creato più problemi all'interno del lavoro svolto da [2].

5.1 Una nuova architettura per la risoluzione di query su documenti XML temporali

Il modello introdotto nel paragrafo 4.2 permette l'esplicitazione dei timestamp relativi al versioning temporale ad ogni livello della gerarchia (cioè su ogni disposizione normativa), ma in fase di interrogazione in [2] è consentito verificare *solo* quelli relativi all'intero documento XML in quanto vengono controllati sullo schema della relazione in figura 4.11. Per fare questo, nella fase di preprocessing, la parte temporale della query originale viene tradotta dallo stratum in SQL, mentre la parte strutturale viene affidata all'XML Engine adottato. Successivamente, per ogni documento appartenente ai risultati, è necessaria una fase di postrocessing in cui i vincoli temporali della query vengono verificati all'interno di ogni documento. Riassumeno è possibile dire che nell'architettura adottata in [2] le specifiche temporali nella query vengono verificate in due momenti distinti:

1. in fase di selezione: si confrontano con i timestamp della relazione 4.11 (che sono l'*unione* di quelli di ogni livello interno del documento)

2. in fase di ricostruzione: viene costruito in memoria principale l'albero DOM di ogni documento selezionato nella fase precedente e si confrontano i timestamp ad ogni livello della gerarchia

Infatti l'intero atto normativo *può* contenere porzioni che non verificano i vincoli temporali imposti e per questo è necessaria una fase di ricostruzione per avere solo quelle parti di documento consistenti ai vincoli imposti.

Da una attenta analisi del modello ci si è resi conto della possibilità di esplicitare i timestamp relativi alle disposizione normative e non all'intero atto normativo. Facendo questo non sarebbe più necessaria:

1. una fase di preprocessing per tradurre la query, in quanto sarebbe sufficiente un semplice confronto degli attributi temporali con i vincoli temporali imposti nell'interrogazione

2. una fase di postprocessing per analizzare gli attributi temporali delle disposizioni normative, visto che ciò può essere fatto già durante l'interrogazione stessa

Se a quanto si è appena detto applichiamo la teoria dei join strutturali del capitolo 3, è possibile far corrispondere ad ogni documento XML un certo numero di strutture, ognuna delle quali comprende:

- un set completo di attributi temporali per una caratterizzazione temporale completa

- un set completo di attributi strutturali per la sua esatta collocazione all'interno del documento XML di partenza

Con la crazione di un *Temporal XML Query Processor* che supporti il meccanismo di versioning temporale è possibile integrare in un unico componente la possibilità di porre query temporali e strutturali, sostituendo i blocchi chiamati *Stratum* e *XML Engine* nella figura 5.1(a) con l'unico componente chiamato *Structural & Temporal Engine* nella figura 5.1(b). Generare strutture dotate di un proprio set di attributi temporali permette maggiore selettività durante l'interrogazione. Utilizzando una *granularità* più fine, ad esempio passando dall'*atto normativo* alla *disposizione normativa*, è possibile confrontare direttamente gli attributi temporali con i vincoli temporali presenti e selezionare solo gli elementi di interesse. Ciò si traduce nel reperimento dei dati strettamente necessari.

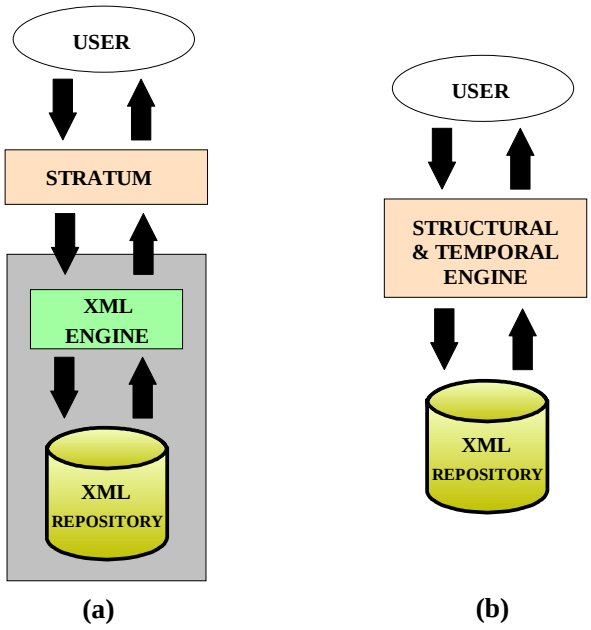

Figura 5.1: Architetture a confronto

5.2 Estensioni relative al modello XML temporale

Il modello XML temporale introdotto nel capitolo 4 è stato ampliato rispetto a come era stato presentato in [2] per essere applicato ad un contesto più generale.

L'applicazione del modello introdotto precedentemente al contesto normativo offre un interessante "banco di prova", ma ciò che si desidera fare è applicare il meccanismo di *versioning temporale* anche ad altre realtà. Per ottenere questo sono stati *rilassati dei vincoli*, al fine di avere una applicabilità più generale.

Il più importante dei vincoli rilassati è relativo al contenimento delle versioni. Secondo tale teoria, nello scendere nella gerarchia del documento XML, è consentito solo un *raffinamento* degli attributi temporali, in modo che rimanga verificato che la temporalità dei nodi *ancestor* contenga sempre quella dei nodi *descendant*. Ad esempio un articolo può avere due versioni di uno stesso comma, ma se il valid time dell'articolo è ('1970-01-01','2000-01-01') allora l'unione dei valid time delle due versioni dei commi deve essere compresa all'interno di quell'intervallo. Non sarebbe stato permesso avere come valid time di una versione di un comma l'intervallo ('1970-01-01','2002-01-01'). Questo ha senso in

un contesto normativo, ma ci possono essere applicazioni in cui è sufficiente che vi sia una intersezione non nulla; in questo caso gli intervalli dell'articolo e quello del comma sopra specificati sarebbero stati compatibili. In pratica si è passati dalla soddisfazione dell'operatore `CONTAINS` alla soddisfazione dell'operatore `OVERLAPS` di TSQL2 (figura 5.2) all'interno del documento stesso.

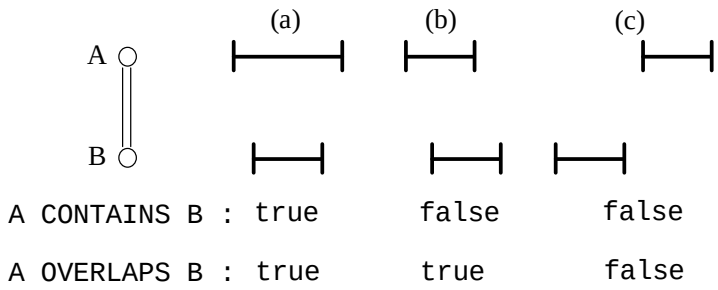

Figura 5.2: Operatori supportati nel modello

Il secondo vincolo rimosso è relativo alla dimensione dei risultati finali: in [2] i risultati restituiti all'utente sono gli interi documenti XML delle leggi contenenti il testo specificato, opportunamente privati dei nodi non consistenti con i vincoli temporali specificati. Si ritiene sufficiente riportare all'utente i soli elementi specificati nella query che soddisfano i vincoli imposti. Elementi non richiesti possono risultare inutili, così come l'intero documento contiene anche elementi inutili.

Se ad esempio l'interrogazione si propone di selezionare tutti i nodi che verificano la relazione XPath `legge//comma` contenenti la parola 'inquinamento', non servono:

- i nodi che verificano la relazione XPath `legge//comma` che non contengono tale parola
- i nodi che non verificano la relazione XPath `legge//comma` (indipendentemente dalla presenza della parola specificata)

Questo permette agli utenti di avere solo i risultati per cui è stata scritta la query, non altri.

Se l'utente desidera più porzioni di documento come un unico risultato può

specificare più path in fase di query, ognuno dei quali può contenere una interrogazione testuale distinta (figura 5.3).

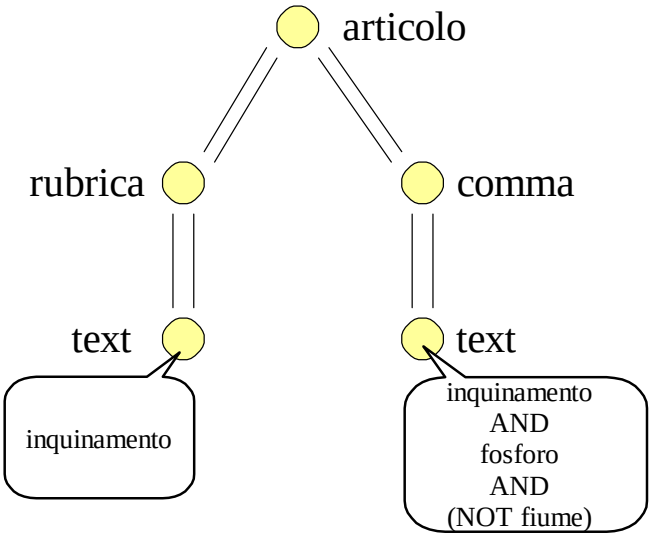

Figura 5.3: Esempio di query testuale su più path

5.3 Strutture necessarie

Sia detta *Tupla* l'unità di memorizzazione dell'implementazione adottata. All'interno di tale struttura è facile individuare sia un set completo di variabili per la memorizzazione di attributi strutturali sia un set completo di variabili per la memorizzazione di attributi temporali. Le variabili strutturali sono quelle in tabella 5.1, le variabili temporali sono in tabella 5.2 mentre le altre variabili sono in tabella 5.3.

Variabile strutturale	**Descrizione**
startPos	numero che rappresenta il tag di apertura
endPos	numero che rappresenta il tag di chiusura
level	numero che rappresenta la profondità di innestamento

Tabella 5.1: Variabili Strutturali all'interno della struttura Tupla

Come si può vedere, il significato di entrambi i set di variabili delle prime

Variabile temporale	Descrizione
pt	data di pubblicazione
vtStart	data di inizio della validità
vtEnd	data di fine della validità
etStart	data di inizio dell'applicabilità
etEnd	data di fine dell'applicabilità
ttStart	data di memorizzazione
ttEnd	data in cui dato memorizzato non è più corrente

Tabella 5.2: Variabili Temporali all'interno della struttura Tupla

due tabelle è lo stesso che è stato presentato nei capitoli precedenti in relazione agli attributi strutturali e temporali[1] .

Variabile	Tipo	Descrizione
docId	stringa	identificativo di una legge
nature	stringa	tipo di atto normativo
num	intero	numero (se presente)
type	stringa	tipo di tupla
anRef	stringa	riferimento all'atto normativo attivo (quello che produce una modifica, se presente)
testo	stringa	stringa di testo (se presente)

Tabella 5.3: Altre variabili della Struttura Tupla

5.4 Possibile dinamica di un atto normativo

Anche se le query formulate comprendono una ristretta tipologia di elementi di un singolo documento XML rappresentante una legge, fino ad oggi si è sempre preferito reperire l'intero documento piuttosto che una sua porzione.

Questo può essere dovuto al fatto che la struttura concorre a definire il significato della nostra query, e reperire l'intero documento è il metodo più facile per mantenere intatta tale struttura. Per essere più chiari ora possiamo fare riferimento al modello presentato nel capitolo 4 utilizzando gli strumenti di interrogazione presentati nel capitolo 2.

[1] ad ogni struttura Tupla corrispondono tutte le veriabili qui introdotte, anche se alcune di esse non vengono mostrate in seguito

La nostra query può essere: reperire tutte le porzioni di testo che sono presenti all'interno dell'elemento rubrica. Facendo attenzione al nostro modello possiamo accorgerci che abbiamo due elementi che corrispondono al nome di rubrica. La loro differenza sta nella loro posizione all'interno dell'albero XML. Infatti abbiamo un elemento rubrica che è figlio di un capo, e rappresenta la parte riassuntiva degli articoli presenti in quel capo, poi abbiamo un elemento rubrica che è figlio di un articolo, e rappresenta la parte riassuntiva dei commi presenti in quell'articolo. Questi elementi rubrica hanno i seguenti path: `capo/rubrica` e `articolo/rubrica`. Se avessimo voluto trattare entrambi gli elementi rubrica il path nella query sarebbe stato `capo//rubrica`; poi per avere il nodo testo dell'elemento selezionato mi basta aggiungere alla query `/text()`

Con l'aiuto degli Structural Join introdotti nel capitolo 3 è possibile suddividere un documento XML in porzioni più piccole, e mantenerne interamente il significato legato alla sua posizione all'interno dell'intero documento.

Per fare ciò è sufficiente conoscere in partenza la struttura dei documenti con cui si ha a che fare.

In questo lavoro di tesi si vedrà come mantenere il significato legato alla struttura di un documento XML, ed estenderlo al versioning temporale grazie alla teoria alla base dei Database Temporali.

Possiamo ora introdurre un esempio di atto normativo che sia conforme al modello già presentato. Tale documento XML è una semplificazione di una possibile legge che si pronunci in merito di scarichi in acqua di sostanze; l'intero codice XML è visibile nella figura 5.4.

5.5 Parsing di una legge e generazione delle tuple

Ora sappiamo come evolve nel tempo un atto normativo e quali sono le strutture che si intendono usare per la sua memorizzazione, rimane da chiarire come è possibile tradurre un documento XML (magari come quello nel documento di esempio) in una collezione di tali strutture; per fare ciò useremo una serie di meccanismi ideati in questa tesi.

```
<?xml version="1.0" encoding="UTF-8"?>
<legge num="1" natura="DL"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="C:\codice\libreria_leggi\schema.xsd">
  <titolo>Scarico in acqua di sostanze liquide</titolo>
  <articolato pubblicazione="1979-12-15"
    vt_start="1980-01-01" et_start="1980-01-01" tt_start="1979-12-20">
    <ver num="1">
      <capo num="1">
        <ver num="1">
          <rubrica> introduzione agli articoli</rubrica>
          <articolo num="1">
            <ver num="1">
              <rubrica> introduzione ai commi</rubrica>
              <comma num="1">
                <ver num="1">
                  versione1 comma1 : scarico in mare fosforo
                    soglia 20 mg/litro
                  <ta vt_start="1980-01-01" et_start="1980-01-01"
                    tt_start="1979-12-20" tt_end="1980-12-31"/>
                  <ta vt_start="1980-01-01" vt_end="1981-12-31"
                    et_start="1980-01-01" et_end="1981-12-31"
                    tt_start="1981-01-01" tt_end="1991-12-31"/>
                </ver>
                <ver num="2" an_ref="DPR/5/1992">
                  versione2 comma1 : scarico in mare fosforo
                  soglia 15 mg/litro
                  <ta vt_start="1992-01-01" et_start="1992-01-01"
                    tt_start="1992-01-01" tt_end="1992-12-31"/>
                  <ta vt_start="1993-01-01" et_start="1995-01-01"
                    tt_start="1993-01-01"/>
                </ver>
              </comma>
              <comma num="2">
                <ver num="1">
                  versione1 comma2 : scarico in fiume fosforo
                    soglia 20 mg/litro
                </ver>
              </comma>
              <ta vt_start="1980-01-01" et_start="1980-01-01"
                tt_start="1979-12-20" tt_end="1984-12-31"/>
              <ta vt_start="1980-01-01" vt_end="1986-12-31"
                et_start="1980-01-01" et_end="1986-12-31"
                tt_start="1985-01-01"/>
            </ver>
          </articolo>
        </ver>
      </capo>
    </ver>
  </articolato>
</legge>
```

Figura 5.4: Esempio di documento XML

Per comodità tutte le strutture che vengono create sono di tipo Tupla, anche se ciò può generare strutture con attributi inutili a seconda del caso (ad esempio il numero in un elemento rubrica).

Il primo passo è fare il parsing del documento utilizzando un parser.

Importante : non c'è corrispondenza *uno ad uno* fra gli elementi e gli oggetti Tupla, ma si è preferito fare una traduzione che fosse fedele alle disposizioni normative (si veda paragrafo 4.1.3) piuttosto che agli elementi veri e propri.

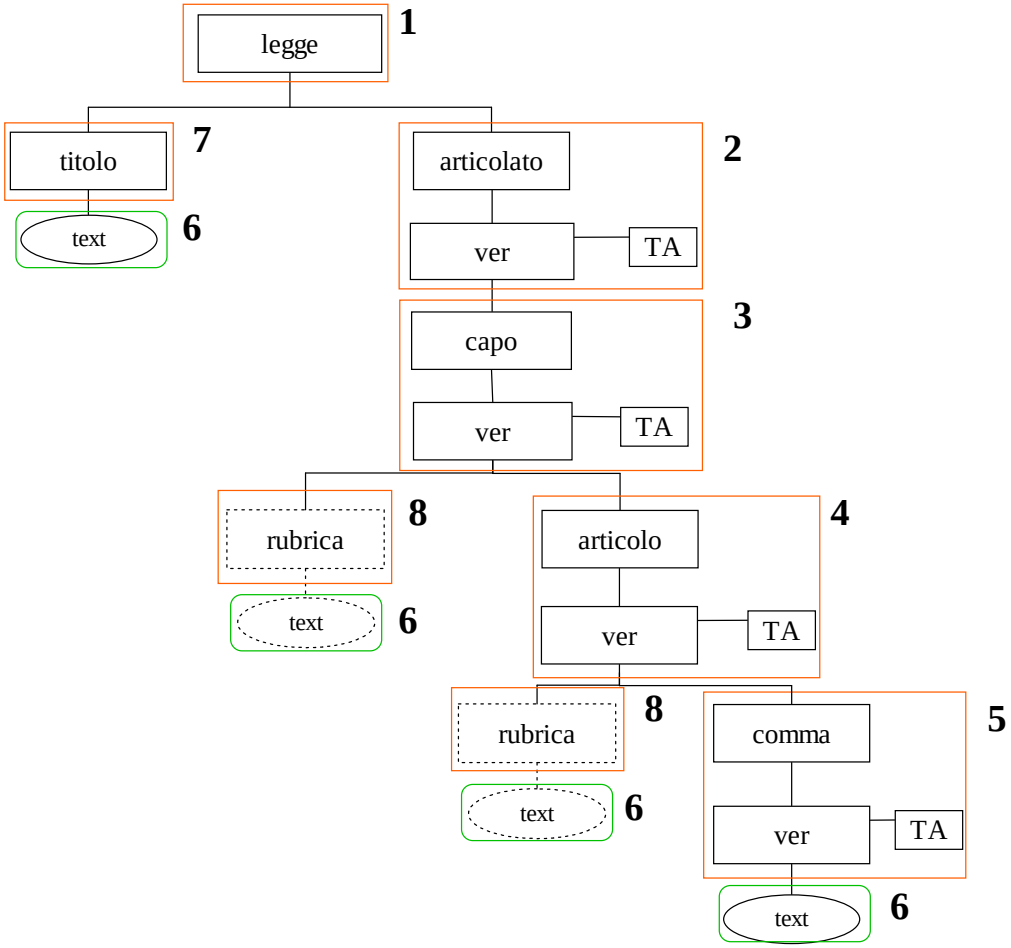

Figura 5.5: Suddivisione in tuple

Nella figura 5.5 possiamo vedere come è possibile suddividere l'intero documento XML in porzioni più piccole conservandone la corrispondenza ad una disposizione normativa. I rettangoli indicano che il blocco all'interno viene tradotto come un `ELEMENT`, rispettando la sintassi `ELEMENT (docId, startPos, endPos, level)`. Rettangoli ad angoli smussati indicano elementi foglia di tipo `TEXT`, che pertanto rispettano la sintassi `TEXT (docId, startPos, level)`.

Di fianco ai rettangoli di ogni tipo abbiamo messo un numero che ci permette di individuare come verrà tradotto il blocco di elementi racchiusi, in particolare:

1. tupla **legge**

2. tupla **articolato**: raccoglie in sè sia il significato della versione sia delle temporalità

3. tupla **capo**: raccoglie in sè sia il significato della versione sia delle temporalità

4. tupla **articolo**: raccoglie in sè sia il significato della versione sia delle temporalità

5. tupla **comma**: raccoglie in sè sia il significato della versione sia delle temporalità

6. tupla **text**: testo semplice all'interno di un nodo foglia

7. tupla **titolo**

8. tupla **rubrica**

Il tipo di tupla viene racchiuso all'interno della variabile `type` della struttura Tupla.

5.5.1 Le versioni e gli attributi strutturali

La traduzione dei blocchi di tipo 1, 6, 7 ed 8 non dà particolari problemi: basta seguire fedelmente la sintassi XML. Quando invece si traducono i blocchi 2, 3, 4 e 5 (cioè quelli che raccolgono versioni ed elementi temporali) si utilizza una routine capace di trattare versioni testuali e temporali ed effettuare la conversione in strutture di tipo Tupla. Possiamo dire che ad ogni versione testuale corrisponde un set completo di attributi strutturali con gli opportuni attributi temporali ed a ogni nuova versione temporale (ad esempio quelle che si hanno in seguito di una modifica temporale) corrisponde lo stesso set di attributi strutturali della versione in questione mentre si impostano gli attributi temporali come è specificato nell'apposito elemento TA.

Nella tabella 5.4 possiamo vedere come potrebbero essere le tuple generate a partire dal documento di esempio[2]; l'unione delle tuple in tale tabella rappresenta *interamente* ogni significato racchiuso nel documento XML originale.

[2]per brevità non verranno mostrati alcuni campi: pt, docId, natura valgono rispettivamente '1979-12-15', 'DL/1/1979' e 'DL' per ogni tupla generata ed il testo è presente new campo testo delle tuple di tipo text

id	type	startPos	endPos	level	vtStart	vtEnd	etStart	etEnd	ttStart	ttEnd
1	legge	1	26	1	01/01/1980	F	01/01/1980	F	20/12/1979	UC
2	titolo	2	4	2 n	01/01/1980	F	01/01/1980	F	20/12/1979	UC
3	text	3	-	3	01/01/1980	F	01/01/1980	F	20/12/1979	UC
4	articolato	5	25	2	01/01/1980	F	01/01/1980	F	20/12/1979	UC
5	capo	6	24	3	01/01/1980	F	01/01/1980	F	20/12/1979	UC
6	rubrica	7	9	4	01/01/1980	F	01/01/1980	F	20/12/1979	UC
7	text	8	-	5	01/01/1980	F	01/01/1980	F	20/12/1979	UC
8	articolo	10	23	4	01/01/1980	F	01/01/1980	F	20/12/1979	31/12/1984
9	articolo	10	23	4	01/01/1980	31/12/1986	01/01/1980	31/12/1086	01/01/1985	UC
10	rubrica	11	13	5	01/01/1980	F	01/01/1980	F	20/12/1979	UC
11	text	12	-	6	01/01/1980	F	01/01/1980	F	20/12/1979	UC
12	comma	14	16	5	01/01/1980	F	01/01/1980	F	20/12/1979	31/12/1980
13	comma	14	16	5	01/01/1980	31/12/1981	01/01/1980	31/12/1981	01/01/1981	31/12/1991
14	text	15	-	6	01/01/1980	F	01/01/1980	F	20/12/1979	31/12/1991
15	comma	17	19	5	01/01/1992	F	01/01/1992	F	01/01/1992	31/12/1992
16	comma	17	19	5	01/01/1993	F	01/01/1995	F	01/01/1993	UC
17	text	18	-	6	01/01/1992	F	01/01/1992	F	01/01/1992	UC
18	comma	20	22	5	01/01/1980	F	01/01/1980	F	20/12/1979	31/12/1984
19	comma	20	22	5	01/01/1980	31/12/1986	01/01/1980	31/12/1986	01/01/1985	UC
20	text	21	-	6	01/01/1980	F	01/01/1980	F	20/12/1979	UC

Tabella 5.4: Tuple generate dal documento di esempio

Nella tabella abbiamo aggiunto la colonna **id** che ci serve per identificare la tupla univocamente, in quanto alcune hanno gli stessi attributi strutturali (perchè sono differenti versioni temporali della stessa versione testuale).

Attributi strutturali dell'oggetto Tupla

Abbiamo già detto che l'attribuzione degli attributi strutturali è determinato dalla numerazione progressiva di:

- tag di apertura e di chiusura di elementi
- elementi testo all'interno di nodi foglia

Questo meccanismo è applicabile direttamente solo per le tuple di tipo legge, titolo, rubrica e text (rispettivamente blocchi 1, 7, 6 e 8 in figura 5.5), cioè in quelle porzioni di documento in cui non compare il meccanismo del versioning.

Per le tuple di tipo articolato, capo, articolo e comma (rispettivamente blocchi 2, 3, 4 e 5) la numerazione tiene conto dei tag di apertura e di chiusura degli elementi ver. Per dare ad ogni tupla il significato di una disposizione normativa esistente (vedi paragrafo 4.1.3) abbiamo deciso di associare gli elementi ver (e quindi gli attributi strutturali conseguenti alla loro numerazione) al tipo di elemento padre.

In breve possiamo dire di aver numerato non il singolo elemento ver, ma l'insieme di `*/ver` (che nel nostro caso significa `articolato/ver`, `capo/ver`, `articolo/ver` e `comma/ver`).

Ne consegue che da due versioni di una disposizione normativa si origineranno due tuple del tipo corrispondente aventi due set di attributi strutturali distinti. Nel nostro esempio ciò avviene quando si incontrano le due versioni del comma uno dell'articolo uno; nella tabella tali versioni sono rappresentate dalle tuple di tipo comma che hanno startPos rispettivamente 14 e 17.

In pieno accordo con la teoria che sta alla base degli structural join vediamo che il testo (tipo text) corrispondente agli enunciati di tali versioni è caratterizzato da uno startPos compreso fra startPos ed endPos delle tuple comma prima citate (che sono le tuple padre).

Riassumendo possiamo dire che a seconda del tipo di elemento numeriamo :

1. il tag di apertura e chiusura dell'elemento

2. l'insieme `*/ver` sia in apertura che in chiusura

3. il testo semplice

Attributi temporali dell'oggetto Tupla

Gli attributi temporali vengono attribuiti in due fasi.

Fase 1 Nella prima fase è direttamente il parser che gestisce gli attributi temporali attribuendoli solo a quegli elementi che hanno una temporalità propria. Possiamo dire che ogni versione di disposizione normativa ha una temporalità propria se l'elemento ver corrispondente ha fra i suoi figli degli elementi TA (che sono sempre opzionali).

Guardando il documento di esempio vediamo che entrambe le versioni del primo comma ed il primo articolo hanno degli elementi TA a loro riferiti.

Questo fa sì che quella versione di disposizione normativa venga duplicata con gli stessi elementi strutturali per ogni TA disponibile, e che ad ognuno di questi duplicati venga attribuito un set completo di attributi temporali preso direttamente dall'elemento TA che ha originato il duplicato stesso.

All'interno dell'unica versione di articolo troviamo due elementi TA, che giustificano le tuple con id 8 e 9, cioè quelle che hanno gli stessi attributi strutturali (provengono dalla stessa versione) ma due set di attributi temporali differenti. Non solo, ma tale temporalità coincide con quella dei TA presenti.

Lo stesso vale anche per i commi con id 12 e 13, e 15 e 16: hanno gli stessi attributi strutturali in quanto discendono della stessa versione, ma set temporali differenti che provengono direttamente dagli elementi TA corrispondenti.

L'eccezione a questa regola è data dalla tupla articolato che è dotata di una temporalità interna (senza il bisogno di elementi TA) che viene estesa già dal parser alle tuple di tipo capo, articolo e comma che non hanno già una temporalità propria.

Fase 2 Alla fine della fase uno le uniche tuple che hanno una temporalità sono quelle che hanno il meccanismo del versioning. Ora possiamo distinguere tre momenti.

Momento 1 La prima cosa che si fa è trovare una temporalità appropriata per la tupla legge. Guardando gli attributi strutturali tale tupla è *ancestor* di ogni altra, allo stesso modo la tupla legge deve *contenere la temporalità* di ogni altra. Questa temporalità è data dalla unione di tutte le temporalità trovate finora.

Momento 2 Ora all'interno della porzione di documento che utilizza il meccanismo del versioning (articolato, capo, articolo e comma) avviene l'estensione delle modifiche temporali alle porzioni inferiori della gerarchia.

L'estensione di tali modifiche viene fatta seguendo il naturale corso delle modifiche, cioè ordinandole secondo ttStart, e viene applicato su quelle tuple che verificano tutte le seguenti ipotesi:

1. tuple discendenti della tupla che determina tale modifica
2. tuple che corrispondono ad un dato corrente, cioè quelle che hanno ttEnd = UNTIL CHANGE
3. tuple che hanno il proprio ttStart precedente al ttStart della tupla che determina tale modifica

Questa estensione origina le tuple con id 18 e 19. In questo caso la tupla che innesca la modifica è quella con id 9, cioè la seconda versione temporale di articolo. L'output generato dal parser nella fase uno comprendeva solo uno comma:

id	**type**	**startPos**	**endPos**	**level**	**vtStart**	**vtEnd**	**ttStart**	**ttEnd**
18	comma	20	22	5	01/01/1980	F	20/12/1979	UC

Tale comma ha la parte temporale totalmente ereditata da articolato in quanto la versione del comma era sprovvista di elementi TA. La tupla con id 9 origina la modifica, la tupla comma con id 18 viene modificata di conseguenza nel suo

transaction time e viene introdotta una nuova versione temporale di tale comma (id 19).

Momento 3 A questo punto gli unici elementi che sono ancora sprovvisti di temporalità sono text, rubrica e titolo. Siccome non sono sottoposte al meccanismo del versioning è opportuno che prendano la stessa temporalità delle tuple di cui sono *child*. E' possibile che alcune tuple abbiano più di una tupla *parent*, è quindi necessario che vi sia una fusione delle loro temporalità prima di estenderle al nodo *child*. Un esempio è la tupla con id 14, che ha come *parent* sia 12 che 13, cioè due versioni temporali della stessa versione testuale.

All'interno del presente paragrafo si è visto come sia possibile tradurre documenti che soddisfano il modello XML temporale adottato in semplici strutture Tupla. Tali strutture non sono altro che una collezione di attributi. Più in generale potremmo dire che grazie ai meccanismi ideati per la traduzione dei documenti si è passati da dati XML a dati relazionali; non solo, ma questi dati relazionali hanno anche una temporalità e sono coerenti con la teoria dei database temporali.

Al parsing di ogni documento segue una fase di memorizzazione delle strutture tupla all'interno di un database system. L'unico accorgimento è che la numerazione per le variabili strutturali è *progressiva anche da documento a documento*: questo significa che se il primo documento di cui viene fatto il parsing ha endPos di legge uguale a 26, startPos della tupla legge del documento successivo che si sottopone al parsing sarà 27. In questo modo è possibile mantenere l'identità di un intero documento utilizzando il meccanismo degli structural join: infatti una volta individuato l'elemento legge (che è root) è possibile dire che ogni altro elemento che ne è *descendant* appartiene allo stesso documento. In questo modo è possibile evitare controlli sulla variabile docId della struttura Tupla, e considerare l'insieme di tutte leggi nel repository come un unico grande documento avente una *root virtuale*.

5.6 Memorizzazione delle strutture Tupla

Una volta generate le strutture Tupla dal parsing di un documento XML è necessario memorizzarle all'interno di un database system. Ogni tupla viene memo-

rizzata in una tabella corrispondente al suo tipo; in questo modo si avrà bisogno di 8 tabelle con la stessa struttura, ciascuna delle quali porta come nome il tipo di tupla a cui è associato. Non è possibile utilizzare il valore di startPos di ogni tupla come chiave primaria in quanto più tuple possono avere gli stessi attributi strutturali (in quanto differenti versioni temporali della stessa versione testuale), allora ad ogni inserimento di una tupla è stata associata una chiave unica intera.

In realtà sarebbe possibile usare startPos come chiave unica, ma solo per le tuple che non sono soggette al meccanismo del versioning; tuttavia si è preferito mantenere l'attribuzione della chiave unica intera per tutte le tabelle per far sì che tutto possa essere applicabile anche a contesti differenti da quello normativo.

Su ognuna delle 8 tabelle viene costruito un B-TREE sull'attributo startPos, su cui faremo ogni accesso alla base di dati.

Capitolo 6

Un algoritmo per la valutazione di query su XML temporale

Nel presente capitolo verrà introdotto un algoritmo per risolvere le query su documenti XML temporali come quelli che rispettano il modello introdotto nel capitolo 4.

La fase di elaborazione di questa tesi è affidata all'algoritmo che chiameremo TwigStack. Tale algoritmo è stato ideato all'interno del presente lavoro anche se ci si è ispirati a quanto è stato presentato in [3] e ripreso in [10]. Entrambi i lavori hanno affrontato con interessanti prospettive il processo di query su più di un percorso (twig). TwigStack elabora le strutture Tupla introdotte nel precedente capitolo per risolvere query *complesse* e generare risultati *consistenti*. Per leggere le strutture Tupla dal DB l'algoritmo TwigStack si avvale di altri due algoritmi che vengono richiamati automaticamente a seconda del tipo di query; questi algoritmi, che chiameremo GetNext2 e GetNextText, sono stati ideati completamente all'interno di questa tesi e provvedono alla lettura delle tuple dal database system memorizzate in precedenza.

Indipendentemente dalla presenza e dal tipo di vincoli specificati in una query è *indispensabile* che i risultati generati siano consistenti, cioè coesistano in almeno un chronon tutte le dimensioni temporali del modello. Questo significa che per ogni dimensione temporale tutte le strutture Tupla coinvolte in una soluzione devono avere intersezione non nulla.

Alla fine del capitolo saranno anche spiegati i metodi di accesso utilizzati negli algoritmi ed il significato dei risultati finali.

6.1 Elaborazione dei vincoli su XML temporale

All'interno di una query è possibile specificare tre tipi di vincoli (vi veda figura 6.1):

1. *vincolo strutturale* (necessario)
2. *vincolo testuale* (opzionale)
3. *vincolo temporale* (opzionale)

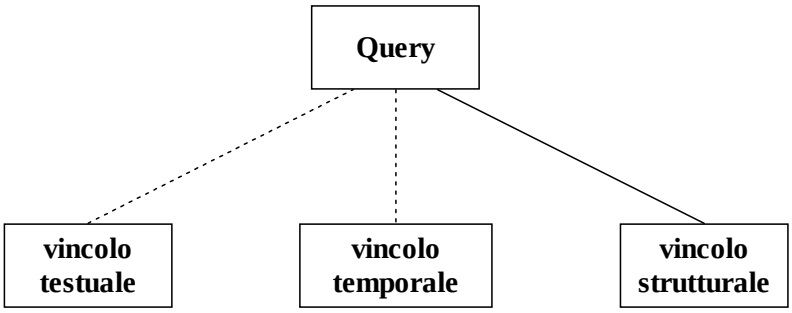

Figura 6.1: Vincoli all'interno di una query

6.1.1 Vincolo strutturale

Tutte le strutture qui introdotte concorrono alla formulazione del vincolo strutturale di una query in cui troviamo una struttura Twig, che comprende una o più strutture Path ognuna delle quali a sua volta contiene due o più strutture CoppiaNodi. Tale vincolo ci dà la possibilità di accedere a particolari frazioni del documento tramite l'esplicitazione di uno o più percorsi.

Path: è una struttura che serve a memorizzare un percorso (per l'appunto, un path). I path in figura 6.2 sono :
`articolo//rubrica//text` (path 0)
e
`articolo//comma//text` (path 1).

Twig: è una struttura che serve a memorizzare una struttura ad albero, in cui possono essere individuati più percorsi (più path). La figura 6.2 è un

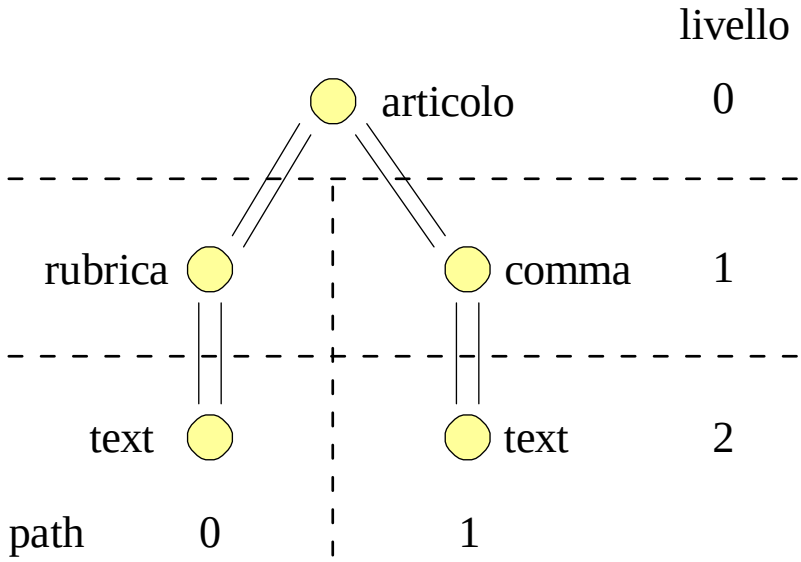

Figura 6.2: Esempio dettagliato di query

nodo	nodo parent	CoppiaNodi
articolo	`null`	(articolo, `null`)
rubrica	articolo	(rubrica, articolo)
text	rubrica	(text, rubrica)
comma	articolo	(comma, articolo)
text	comma	(text, comma)

Tabella 6.1: Strutture CoppiaNodi all'interno del twig di esempio

twig, ed è importante notare che tutti i path del twig hanno lo stesso elemento root (in questo caso è articolo). Per ogni nodo possiamo individuare il *path* di appartenenza ed il *livello* all'interno del path. Un twig rappresenta i *vincoli strutturali* della query.

CoppiaNodi: è una struttura che serve a memorizzare un nodo ed il suo nodo parent. All'interno di un twig è possibile specificare una qualsiasi coppia di tipo (*nodo*, *nodo parent*) e ritrovarne la sua posizione, sia riguardo il livello di innestamento, sia riguardo al path di appartenenza. Nella tabella 6.1 possiamo vedere come da ogni nodo del twig in figura 6.2 si origina una struttura *CoppiaNodi*. Una struttura dati di questo tipo è molto utile in quanto permette di specificare più volte un elemento all'interno dello stesso twig: è sufficiente che siano distinte le coppie (*nodo*, *nodo parent*). Grazie a questa struttura è possibile esplicitare twig come quello in figura, dove compare due volte l'elemento

text, ma con due elementi *ancestor* differenti. Seguendo la definizione di query presente in [3] e [10] ciò non sarebbe stato possibile. La struttura *CoppiaNodi* è quindi necessaria ad individuare l'esatto posizionamento di un nodo all'interno del twig di interrogazione, e per ottenere questo si è ricorsi all'individuazione di coppie distinte di tipo (*nodo*, *nodo parent*).

I vincoli strutturali sono risolti dall'algoritmo TwigStack.

6.1.2 Vincolo testuale

Nella presente tesi si tratta un'applicazione XML di tipo *text-centric* in quanto i documenti considerati sono un'evoluzione semi-strutturata di documenti testuali; da questo consegue che l'interrogazione basata su *keyword* è di particolare interesse.

L'*Information Retrieval* (IR) è un settore della Computer Science che tratta i problemi relativi alla memorizzazione, rappresentazione e reperimento di documenti. L'IR non è ristretta al trattamento di documenti testuali, ma la maggior parte di sistemi e tecniche sono, per motivi storici, di fatto relativi a questi.

Nel capitolo 4 si è visto che è concessa la presenza di testo solo all'interno di certi elementi del documento XML, che sono titolo, rubrica e ver (solo quando è all'interno degli elementi comma). Nel capitolo 5 è emerso che nella fase di creazione delle tuple, vengono generate strutture di tipo `text` che hanno un set completo di attributi strutturali e temporali proprio come le tuple di tutti gli altri tipi. Inoltre, le tuple di quel tipo contengono testo semplice all'interno del campo testo.

Le tuple text hanno significati differenti, legati ai path che soddisfano:

`/legge/titolo/text` : testo relativo al titolo dell'atto normativo

`/legge/articolato/capo/rubrica/text` : testo relativo al sommario degli articoli presente all'interno di un elemento capo

`/legge/articolato/capo/articolo/rubrica/text` : testo relativo al sommario dei commi presente all'interno di un elemento articolo

`/legge/articolato/capo/articolo/comma/text` : testo all'interno dei commi

Visto che il testo può essere presente solo all'interno delle tuple text, è stato costruito un *inverted index* sul campo testo di tutte le tuple della relativa tabella di DB.

Questo ha fatto in modo che fosse creata la corrispondenza (K_i, W_i), dove W_i indica una parola all'interno di una tupla text (non appartente alla *stoplist*) e K_i indica il valore di startPos.

I vincoli testuali che vengono gestiti nell'interrogazione sono di tipo booleano: da ogni espressione specificata si è in grado di trovare tutte le tuple con identificativo K_i che soddisfano *esattamente* tale espressione. E' possibile creare query molto espressive combinando gli operatori booleani (`AND`, `OR` e `NOT`) ai termini della query.

Un utente interessato ad una legge che si pronunci in merito all'inquinamento dei mari causato da fosforo o piombo potrebbe formulare una espressione del tipo: `inquinamento AND ( fosforo OR piombo ) AND mare`.

Ad una espressione di questo tipo corrisponderanno un certo numero di elementi, ognuno dei quali verifica l'espressione. Gli identificativi delle tuple che verificano l'espressione sono il punto di partenza di tutte le query che contengono vincoli testuali, in quanto possono essere considerate come il primo passo per ogni interrogazione che comprenda anche altri vincoli.

In realtà il vincolo testuale può essere visto come un caso particolare di vincolo strutturale. Infatti, una volta che si ha a disposizione un path in cui è presente un nodo di tipo text, è possibile effettuare ricerche testuali senza l'ausilio di altro. Purtroppo tale operazione risulta molto inefficiente rispetto all'utilizzo di un inverted index in quanto può dar luogo ad un cospicuo numero di letture inutili da disco.

I vincoli testuali sono risolti in fase di generazione delle tuple di interesse dagli appositi aloritmi.

6.1.3 Vincolo temporale

I vincoli temporali servono per poter limitare la mole di dati con cui si ha a che fare durante una interrogazione. Infatti un grande problema dei database temporali è la ridondanza dei dati creata per garantire modifiche e cancellazioni non distruttive (si ricorda che i database temporali sono *append only*); questo si traduce in un grande aumento dei dati memorizzati.

Per ogni dimensione temporale su cui è espresso un vincolo temporale nella query si controllano i corrispettivi attributi direttamente nelle strutture tuple trattate, e solo quelle che verificano tutti i vincoli espressi possono continuare

nell'elaborazione. La soddisfazione di tale vincolo non ha niente a che fare con la consistenza delle soluzioni di cui si parlerà più avanti.

I vincoli temporali sono risolti in fase di generazione delle tuple di interesse dagli appositi algoritmi.

6.2 Funzionamento di TwigStack

Ora verrà spiegato il funzionamento di TwigStack su una query di un solo path, sarà poi immediato estenderne il significato ad un twig (che comprende più di un path)

Si assuma che la query sia quella relativa al solo path 0 mostrato in figura 6.2, e che i dati a disposizione siano le tuple in tabella 5.4 (cioè quelle provenienti dal documento di esempio) opportunamente memorizzate nelle tabelle messe a disposizione da un database system.

A questo punto si hanno 8 tabelle, ognuna delle quali memorizza le tuple di un tipo. Su ogni tabella è stato costruito un indice sul valore startPos della tupla, e sarà proprio tramite questo indice che accederemo ai dati del database. Purtroppo startPos delle tuple non è un valore di chiave *unique*, infatti come abbiamo visto ci sono tuple che hanno lo stesso valore degli attributi strutturali (e quindi anche startPos), pertanto abbiamo prima memorizzato le tuple attribuendo loro una chiave generica data da un intero (chiave *unique*) e poi abbiamo costruito un indice *secondario* sul campo startPos (si veda il paragrafo 5.6).

Si inizia col selezionare le tuple di un tipo di nostro interesse (quelli che compaiono nella query) in ordine di startPos. Selezionare *solo le tuple di interesse* è un tema che sta alla base di questo lavoro, ma verrà trattato in seguito; per ora pensiamo di leggerle dal DB, metterle in tanti buffer quanti sono i nodi della query ed all'interno di ogni buffer in una lista che segua l'ordine di startPos.

Le tuple di interesse sono state suddivise a seconda del tipo, e sono mostrate nelle tabelle 6.2, 6.3 e 6.4 (in cui sono visibili solo i campi principali delle strutture Tupla).

In partenza l'algoritmo TwigStack si presenta con tanti stack quanti sono i livelli della query, quindi avremo tre stack: il primo per il nodo root che è articolo, il secondo per rubrica ed il terzo per il nodo text. Gli stack servono per

type	startPos	endPos	level	ttStart	ttEnd
articolo	10	23	4	20/12/1979	31/12/1984
articolo	10	23	4	01/01/1985	UC

Tabella 6.2: Buffer contenente gli elementi articolo

type	startPos	endPos	level	ttStart	ttEnd
rubrica	7	9	4	20/12/1979	UC
rubrica	11	13	5	20/12/1979	UC

Tabella 6.3: Buffer contenente gli elementi rubrica

type	startPos	endPos	level	ttStart	ttEnd
text	3	-	3	20/12/1979	UC
text	8	-	5	20/12/1979	UC
text	12	-	6	20/12/1979	UC
text	15	-	6	20/12/1979	31/12/1991
text	18	-	6	01/01/1992	UC
text	21	-	6	20/12/1979	UC

Tabella 6.4: Buffer contenente gli elementi text

contenere le strutture tupla che l'algoritmo elabora. Se la nostra query contiene più di un path allora si generano tanti stack quanti sono i livelli del path più lungo (quello che ha *livello* maggiore); le tuple vanno nello stack relativo al loro livello all'interno del path di query, indipendentemente dal path di appartenenza.

In figura 6.3 possiamo vedere TwigStack descritto utilizzando uno pseudocodice.

I metodi utilizzati nell'algoritmo (alcuni sono molto simili a quelli mostrati nel capitolo 3) sono:

`isRoot(q)` metodo che verifica se la struttura `CoppiaNodi q` passata come parametro è root del twig

`isLeaf(q)` metodo che verifica se la struttura `CoppiaNodi q` passata come parametro è leaf del twig

`cleanStackParent(`$S_{parent(q)}$`, t)` verifica che all'interno dello stack S_q ci siano solo tuple *ancestor* di t; tutti gli elementi che non soddisfano questa condizione vengono eliminati

`cleanStackSelf(`S_q`, t)` verifica che all'interno dello stack S_q ci siano solo tuple

```
1  Tupla t   ;
2  CoppiaNodi q    ;

4  while not end(t) do
5      t = getTupla();
6      q = getNodo();

8      if not isRoot(q) then
9          cleanStackParent(S_parent(q), t);

11     if ( isRoot(q) or (not empty(S_q)) ) then
12         cleanStackSelf(S_q, t);
13         push(S_q, t, ptr);

15         if isLeaf(q) then
16             outputPathSolution();
17             pop(S_q);

19 endwhile
20 mergeAllPathSolution();
```

Figura 6.3: Algoritmo TwigStack descritto con pseudo-codice

ancestor o con lo stesso startPos di t; tutti gli elementi che non soddisfano questa condizione vengono eliminati

`push(`S_q`, t, ptr)` viene inserita sullo stack S_q una struttura che comprende t ed un puntatore *ptr* che indica l'elemento *top* dello stack padre

`outputPathSolutionWithBloking()` genera una soluzione completa di un path della query (dal nodo radice al nodo foglia)

L'elaborazione inizia con la creazione delle strutture di cui si ha bisogno; infatti si creano una `CoppiaNodi q` e una `Tupla t` vuoti (che saranno rispettivamente il nodo e la tupla *correnti*).

Il ciclo `while` procede fintanto che ci sono delle tuple con i requisiti adatti per la query espressa (nel nostro caso si procede finchè non vengono elaborate tutte le tuple nelle tabelle viste sopra).

Ora i metodi `getTupla()` e `getNodo()` generano la `Tupla t` ed la `CoppiaNodi q` corrispondente, seguendo l'ordine di startPos. Di ogni lista di tuple viene esaminata solo la prima (*top della lista*), visto che quelle successive avranno uno startPos sicuramente maggiore o uguale (si ricorda che sono state ordinate proprio su questo campo). L'ordine con cui le tuple vengono generate rispecchia quello degli startPos delle tuple top delle liste.

Guardando le tuple di esempio si può vedere che verrà generata per prima la tupla di tipo text che ha startPos uguale a tre (infatti nessun'altra tupla ha startPos minore fra le tuple che sono al top delle liste). Il nodo relativo a tale tupla è *(text, rubrica)*, che sta ad indicare che lo stack chiamato in causa è il terzo e che il nodo è una foglia.

E' possibile entrare nel codice del primo `if` solo se il nodo non è di tipo root. Se ciò è vero si richiama la procedura `cleanStackParent(`$S_{parent(q)}$`, t)`, la quale verifica che tutti gli elementi dello stack padre (*parent stack*, lo stack di rubrica) contengano tuple *ancestor* della tupla `t`; tutti gli elementi per i quali questa condizione non è verificata vengono eliminati dallo stack. Ora non abbiamo elementi all'interno dello stack di rubrica, quindi non succede nulla.

Entriamo nel codice del secondo `if` solo se il nodo `q` è il nodo root, o se lo stack parent non è vuoto. Nel nostro caso nessuna delle due precedenti condizioni è verificata perchè il nodo `q` non è root ed abbiamo appena detto che lo stack precedente a quello di text è vuoto.

Il ciclo arriva alla sua ultima istruzione, la tupla corrente è scartata, e si ricomincia dall'inizio con una nuova tupla ed un nuovo nodo.

Appare evidente che la prima tupla che potrà entrare in uno degli stack è di tipo articolo, perchè corrisponde alla root di `twig`. Così le tuple con startPos 7 e 8 vengono analizzate ma sono scartate coma la precedente. Quando arriva la tupla con startPos 10, che è di tipo articolo, possiamo eseguire il codice all'interno del secondo `if`. La prima cosa che si fa è richiamare la funzione `cleanStackSelf(`S_q`, t)` che controlla all'interno dello stack relativo al nodo corrente (non del parent stack, come era successo prima) e cancella tutti gli elementi che contengono tuple che non sono ancestor (e fin qui il funzionamento è uguale a `cleanStackParent(`$S_{parent(q)}$`, t)`), ma lasciando gli elementi che contengono tuple che hanno startPos uguale a quello della tupla corrente. Questo permette di inserire nello stesso stack tuple che hanno lo stesso startPos, cioè differenti versioni temporali della stessa versione testuale di disposizione normativa.

Conseguentemente alla pulizia dello stack relativo al nodo corrente viene richiamata la funzione `push(`S_q`, t, ptr)`, in cui viene creato un elemento contenente la tupla `t` corrente e un *riferimento ptr* all'elemento più in alto (*top element*) del parent stack e tale struttura viene inserita all'interno dello stack relativo al nodo corrente. Siccome le due tuple appena inserite sono relative al

nodo root non ci sono parent stack, quindi i riferimenti sono privi di significato.

Le due tuple di tipo articolo che hanno startPos 10 vanno entrambe nel primo stack (vedi figura 6.4).

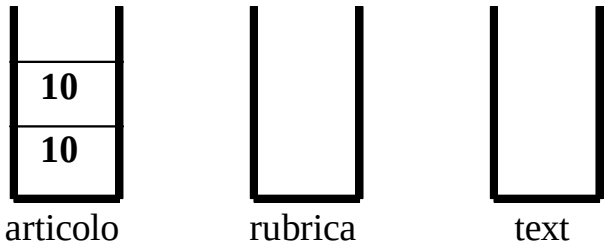

Figura 6.4: TwigStack: tuple articolo

Il codice interno al terzo `if` non viene eseguito in quanto le due tuple articolo non sono appartenenti a nodi foglia, ma a nodi root. Successivamente viene elaborata la tupla di tipo rubrica che ha startPos 11, quella con startPos 7 era stata scartata prima dell'elaborazione delle tuple articolo. Tale tupla appartiene al secondo nodo della query. Quando si esegue il codice del primo `if` non viene cancellato nessun elemento dallo stack di articolo in quanto contengono tutti tuple *ancestor* della tupla corrente. Nel secondo `if` viene creato un elemento contenente la tupla corrente (rubrica) e viene inserito nel secondo stack (figura 6.5). L'elemento appena creato comprende un riferimento all'elemento top dello stack di articolo. All'iterazione successiva del ciclo `while` viene messa nel terzo

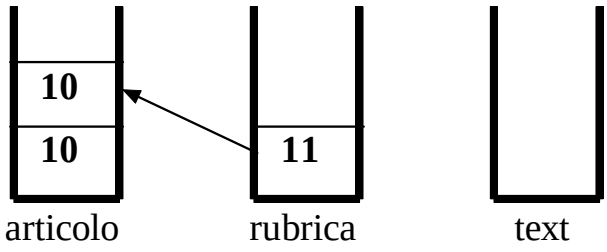

Figura 6.5: TwigStack: tupla rubrica

stack un elemento contenente la tupla di tipo text con startPos 12, e viene creato un riferimento all'elemento top dello stack di rubrica. La coppia di nodi relativa alla tupla corrente è di tipo *(text, rubrica)*, ciò significa che si sta trattando un nodo foglia; da questo consegue che è possibile eseguire quella parte di codice all'interno del terzo `if`. La funzione `outputPathSolutionWithBloking()` è una

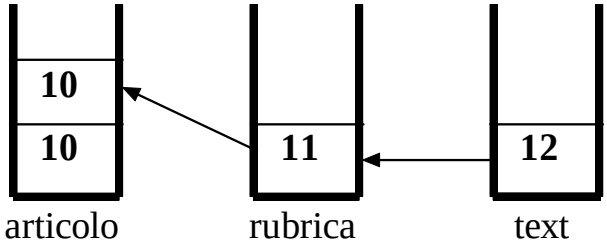

Figura 6.6: TwigStack: tupla text

type	startPos	endPos	level	ttStart	ttEnd
articolo	10	23	4	20/12/1979	31/12/1984
rubrica	11	13	5	20/12/1979	UC
text	12	-	6	20/12/1979	UC

Tabella 6.5: Primo path dei risultati

funzione ricorsiva che si occupa di risalire lungo gli stack della query e reperire le soluzioni ad un path. Questo avviene utilizzando il riferimento al top dello stack del nodo precedente nella query.

Grazie a tale riferimento è possibile sapere che l'elemento puntato dal riferimento e tutti quelli sottostanti contengono tuple *ancestor* di quella appena trattata. Ricorsivamente si risale di stack in stack fino a quello del nodo root della query. Ogni volta che si arriva fino ad un elemento nello stack di root si può dare in output un intero path. Nelle tabelle 6.5 e 6.6 ci sono i path che soddisfano la query iniziale. Le soluzioni presentate in tali tabelle sono distinte perchè, sebbene le tuple rubrica e text siano gli stessi, le tuple articolo sono due tuple distinte.

Il metodo chiamato `mergeAllPathSolution()` ora non ha significato in quanto serve alla fusione di soluzioni provenienti da più path in un unico oggetto composto da soluzioni che hanno elementi radice identici, ma ora stiamo lavorando su un solo path.

type	startPos	endPos	level	ttStart	ttEnd
articolo	10	23	4	01/01/1985	UC
rubrica	11	13	5	20/12/1979	UC
text	12	-	6	20/12/1979	UC

Tabella 6.6: Secondo path dei risultati

La normale elaborazione dell'algoritmo TwigStack può essere applicata a twig contenenti più di un path, basta che non ci siano due strutture `CoppiaNodi` identiche all'interno del twig di query. Ciò significa che non ci possono essere due nodi di tipo x che hanno il nodo parent di tipo y. Questo controllo viene fatto durante la fase di creazione del twig, e se ciò avviene si interrompe l'elaborazione. I buffer contenenti le liste di tuple sono ordinati secondo startPos, gli stack vengono creati automaticamente in modo corretto e l'elaborazione avviene esattamente come è stato descritto sopra anche quando si tratta un twig.

Riassumendo potremmo dire che da un punto di vista puramente strutturale TwigStack funziona come TSGeneric introdotto nel paragrafo 3.2.1, ma permette la coesistenza nello stesso stack di tuple caratterizzate dallo stesso startPos; facendo riferimento al modello adottato potremmo dire che tali tuple rappresentano distinte versioni temporali della stessa versione testuale.

6.2.1 Valutazioni sull'elaborazione

La differenza fra l'utilizzo dell'algoritmo *stack-tree-desc* (introdotto nel paragrafo 3.2) e TwigStack sono molte. La prima è l'eliminazione dei risultati intermedi. Per elaborare la query appena risolta utilizzando *stack-tree-desc* avremmo dovuto trovare tutti i risultati degli operatori binari `articolo//rubrica` e `rubrica//text` e poi fonderli insieme. Con il nostro set di dati avremmo avuto fra i risultati degli operatori binari anche la coppia di tuple in tabella 6.7. Tale

type	**startPos**	**endPos**	**level**	**ttStart**	**ttEnd**
rubrica	7	9	4	20/12/1979	UC
text	8	-	5	20/12/1979	UC

Tabella 6.7: Tuple inutili generate da un operatore binario

coppia verifica il path `rubrica//text`, ma non il path `articolo//rubrica` in quanto è la rubrica interna ad un elemento capo e quindi non concorre alla soluzione finale della query. Da questo si capisce che utilizzando *stack-tree-desc* è facile spendere tempo di CPU ed accessi a disco che potrebbero essere evitati.

TwigStack, al prezzo di una modesta complicazione dell'algoritmo, offre solo risultati "sicuri" dal nodo root al nodo foglia di ogni path della query; non solo ma è aumentato anche il *significato* delle query risolvibili in quanto non si parla più di un singolo path ma di un twig contenente più di un path di query.

Considerando di avere una query a twig con `n` nodi distinti, in [3] è stato dimostrato che TwigStack ha il caso peggiore di I/O e tempo di CPU lineari alla somma delle `n` liste di input e della lista di output.

E' importante ricordare che il caso peggiore del tempo di elaborazione richiesta dall'algoritmo è indipendente dalla dimensione delle soluzioni di una qualsiasi query.

Nell'articolo [3] è stato dimostrato che TwigStack comporta l'ottimalità in termini di tempo di CPU e di I/O con le query di tipo *ancestor-descendant* e risulta comunque efficiente con query di tipo *parent-child*.

6.3 La generazione delle tuple di interesse

Nel paragrafo precedente si è spiegato il funzionamento di TwigStack al presentarsi di ogni nuova tupla: ora è noto il suo funzionamento ogni volta che si presenta un tupla `t` relativa ad un certo nodo `n` della query.

Per la realizzazione di un Temporal XML Query Processor efficiente è di grande importanza **come** viene generata tale tupla `t`, perchè dietro a questo si cela l'aspetto che più di tutti ha peso quando si parla di efficienza di un sistema informativo: l'accesso a disco.

In figura 6.7 vi è una rappresentazione della memorizzazione delle tuple interessanti per la nostra query di esempio.

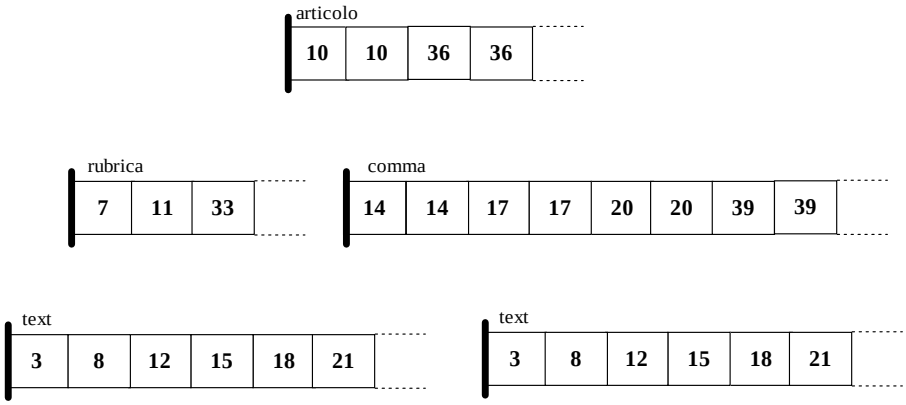

Figura 6.7: Tuple nel database

Oggi, come nelle ultime due decadi, il collo di bottiglia o *bottleneck* è l'hard disk. Un buon algoritmo di lettura deve comprendere metodi per ridurre al

minimo gli accessi a disco (senza sacrificare la sua efficacia), se si cercano delle buone performance. Purtroppo l'algoritmo `getNext(root)` di [3] non provvede ad una generazione della tupla più indicata all'elaborazione, in quanto vengono passati a TwigStack anche tuple che non andranno mai a finire negli stack e quindi vengono effettuate una serie di operazioni inutili che devono essere evitate se si vogliono avere buone performance (vedi anche esempio uno all'interno di [10]).

Nel seguito di questo paragrafo verranno proposti due algoritmi ideati in questa tesi per la generazione *efficiente* delle tuple.

6.3.1 L'algoritmo di generazione GetNext2

L'algoritmo base di generazione delle tuple e nodi ideato in questa tesi si chiama `GetNext2`: prevede un numero minore di accessi a disco del metodo `getNext()` in [3] e riporta comunque tutte le tuple necessarie; ne consegue una maggiore efficienza senza tralasciare l'efficacia.

E' importante tenere presente che il primo accesso fatto a disco riguarda una tupla root (nel nostro caso articolo): è inutile reperire tuple appartenenti ad altri nodi che non siano root, in quanto anche se tali tuple vengono restituite a TwigStack sono automaticamente scartate. Abbiamo detto in precedenza che la prima tupla ad entrare negli stack è una tupla del nodo root (altrimenti non è concesso eseguire quella parte di codice all'interno del secondo `if`, cioè dove viene eseguita l'operazione `push(`S_q`, t, ptr)` sulla tupla). Quando la tupla (o le tuple, come nel caso del nostro esempio) vengono reperite aggiorniamo la variabile `g_startPos` che indica il valore di startPos a cui è giunta l'elaborazione. Ricordiamo che al momento della lettura delle tuple da una tabella del DB utilizzando l'indice su startPos è possibile che si leggano più tuple in una volta sola, tutte con gli stessi attributi strutturali (e quindi stesso startPos).

Ogni volta che si prendono delle tuple dal DB, queste vengono memorizzate all'interno di una struttura a lista che chiameremo `ListaTuple` (ce ne è una per ogni nodo), cancellando quelle già presenti.

Essere sempre a conoscenza del valore corrente di g_startPos ci permette di reperire dal DB solo quelle tuple che hanno la possibilità di essere *descendant* delle tuple root; infatti per esserlo sono obbligate ad avere `startPos > g_startPos`.

Per ogni altro nodo diverso dal primo nodo di root, si esaminano solo le tuple relative ad un nodo *"raggiungibile dal nodo corrente"*, che chiameremo `current`.

Ora il nostro nodo `current` è root, e da questo nodo possiamo raggiungere solo il nodo stesso ed i nodi rubrica e comma. Per ora è inutile analizzare le tuple relative ad ogni altro nodo (ad esempio text). Sono considerati raggiungibili:

- in nodo root
- i nodi figli di root
- il nodo `current` ed ogni suo *ancestor*
- il nodo figlio di `current`

In ogni istante dell'elaborazione tutti i nodi raggiungibili dal nodo `current` devono contenere tuple con startPos maggiore o uguale a `g_startPos`. I nodi che soddisfano questa condizione sono considerati *aggiornati*, mentre quelli che non la soddisfano *devono essere aggiornati* con uno o più accessi a disco; se non sono più disponibili tuple che hanno startPos maggiore o uguale a `g_startPos` la rispettiva `ListaTuple` è posta uguale a `null` ed il corrispettivo stream di tuple non viene più considerato.

Regola: quando le tuple relative ai nodi raggiungibili dal nodo `current` sono state reperite si seleziona il blocco con startPos minore.

A questo punto si aggiorna la situazione corrente :

- si dà a `g_startPos` il valore startPos del blocco di tuple appena selezionato incrementato di uno, in questo modo si ha un'idea della progressione nella ricerca
- si aggiorna il nodo corrente con quello rappresentato dal blocco di tuple appena selezionato

All'interno della figura 6.8 possiamo vedere i blocchi di tuple che vengono letti dal DB, ed il loro ordine di lettura con `GetNext2`.

E' ora possibile fare un'analisi dettagliata dell'elaborazione. Le figure che seguono rappresentano le strutture ListaTuple ed il loro contenuto; quelle in evidenza sono relative ad i nodi *raggiungibili*, cioè quelle fra cui si sceglie il valore minore. Come è ben visibile in figura 6.9 si inizia con la selezione delle tuple relative a root. Questo comporta un aggiornamento di :

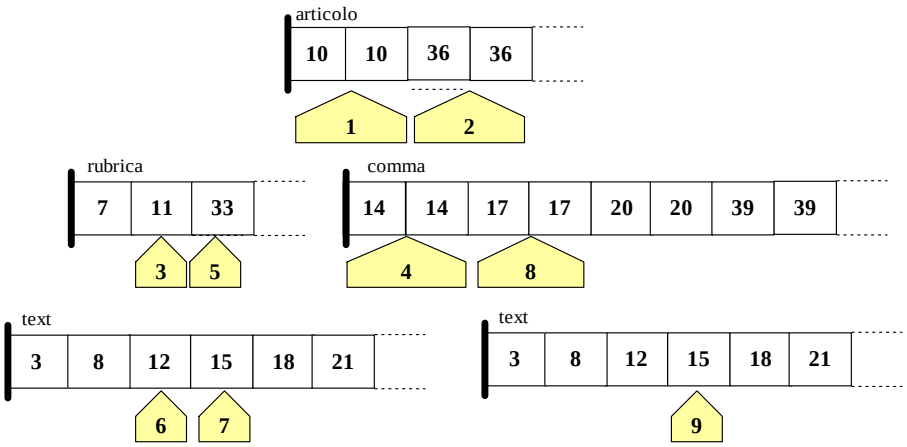

Figura 6.8: Letture delle tuple in GetNext

- `g_startPos = 11` (cioè lo startPos delle tuple selezionate incrementato di uno)
- `current =` articolo

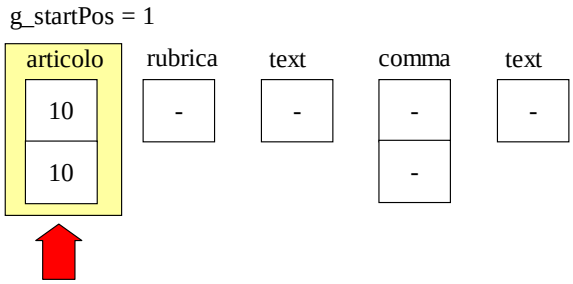

Figura 6.9: Elaborazione: passo uno

Una volta passate a TwigStack entrambe le tuple articolo, si effettua una nuova ricerca (figura 6.10). Stavolta i nodi raggiungibili sono articolo, rubrica e comma e fra questi lo startPos minore lo troviamo in rubrica; `g_startPos` diventa 12 e `current` diventa rubrica.

Nell'elaborazione successiva (figura 6.11) si vede che non è necessario aggiornare le tuple contenute in articolo e comma in quanto potrebbero ancora essere scelte perchè hanno startPos maggiore o uguale `g_startPos`. I nodi raggiungibili sono articolo, rubrica, text di rubrica e comma ed il nodo scelto è il nodo text di rubrica. Il nodo `current` passa da rubrica a text di rubrica. Si ricorda che

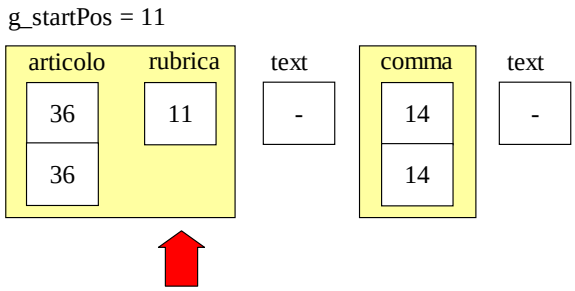

Figura 6.10: Elaborazione: passo due

text di rubrica e text di comma sono due strutture `CoppiaNodi` ben distinte e pertanto facilmente distinguibili.

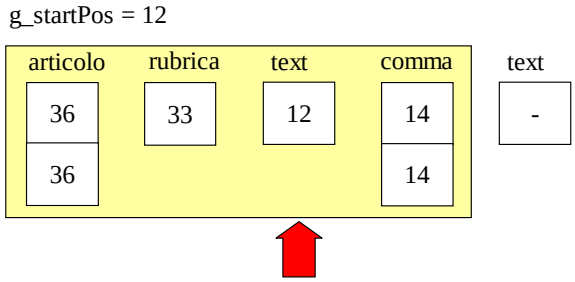

Figura 6.11: Elaborazione: passo tre

I nodi raggiungibili nell'iterazione successiva (figura 6.12) sono gli stessi di quella precedente, ma ora è il nodo comma ad essere selezionato, di conseguenza si ha che `current` = comma.

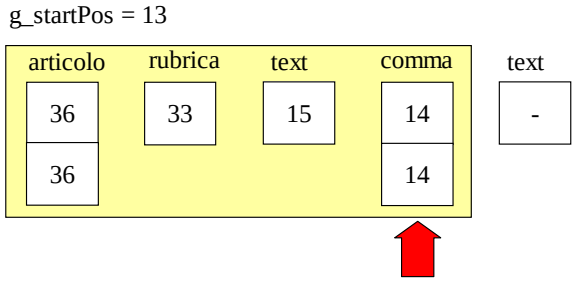

Figura 6.12: Elaborazione: passo quattro

Ora che il nodo corrente è comma diventa raggiungibile anche il nodo text di comma, che contiene la tupla che ha startPos minore di ogni altra tupla, e pertanto verrà selezionata (figura 6.13) e riportata a TwigStack.

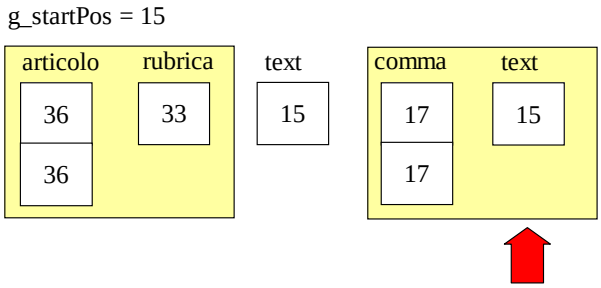

Figura 6.13: Elaborazione: passo cinque

Arrivati a questo punto si sono passati a TwigStack un numero sufficiente di tuple per formare la prima soluzione completa al twig di query, che è quello in figura 6.2.

Vantaggi di `GetNext2`

L'utilizzo di `GetNext2` al posto del metodo `getNext(root)` ha portato molti vantaggi. Il primo è che vengono effettuate molte meno letture su disco e questo velocizza l'elaborazione. Potremmo dire che ogni tupla viene reperita *on the fly*, perchè:

- viene reperita solo quando se ne ha bisogno, cioè solo quando è possibile che tali tuple siano inserite negli stack e non quando si sa già in partenza che verranno scartate (questo lo si capisce grazie al nodo `current`)
- la ricerca utilizza la variabile globale (sempre aggiornata) `g_startPos` per effettuare la ricerca nel DB attraverso l'indice secondario su startPos, in modo da ricercare solo tuple che siano compatibili con tale variabile. Questo significa che le tuple avranno startPos maggiore o uguale a `g_startPos` perchè le relazioni strutturali da ricercare sono di tipo *ancestor-descendant* e *parent-child*

Agni accesso al DB vengono lette *tutte le versioni temporali di una particolare disposizione normativa*: infatti sappiamo che tali versioni sono consecutive in quanto l'accesso al DB viene fatto attraverso l'indice su startPos, e differenti versioni temporali della stessa versione testuale hanno gli stessi attributi strutturali (fra cui c'è anche startPos) da cui consegue che sono record consecutivi utilizzando tale indice.

Un altro punto di forza di questo algoritmo è che spesso ci si trova all'interno delle strutture `ListaTuple` già il blocco di tuple di interesse, senza bisogno di andare a cercarlo direttamente sul DB.

Al momento di ogni lettura delle tuple dal DB è possibile verificare se le tuple lette soddisfano i vincoli temporali della query (se presenti); se questi non sono verificati le tuple vengono ignorate. Appare evidente che GetNext2 si incarica della valutazione dei vincoli temporali nella fase della generazione delle tuple.

6.3.2 L'algoritmo di generazione GetNextText()

Qualora la query contenga al suo interno dei vincoli *testuali*, è possibile utilizzare una routine che permette di fare molti meno accessi a disco di quelli che farebbe `GetNext2`, e quindi migliorare significativamente le prestazioni. Infatti ogni volta che abbiamo a disposizione dei vincoli testuali, possiamo iniziare la ricerca delle sole tuple di tipo text che soddisfano tale richiesta. Partendo da questa è possibile risalire dal nodo foglia (text) fino all'elemento root, attraverso ogni nodo intermedio (figura 6.14).

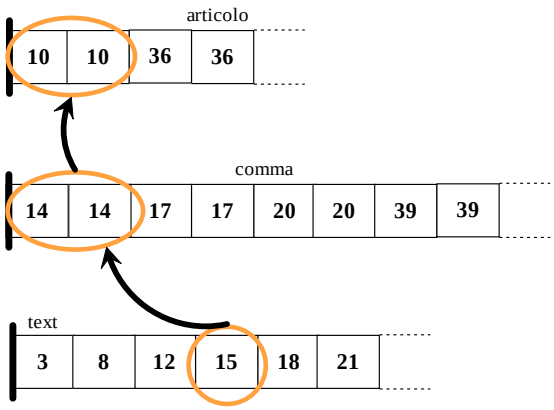

Figura 6.14: Risoluzione di una Query testuale

Quando sappiamo che la tupla di tipo text che verifica la query testuale `'mare AND fosforo'` nel path `articolo//comma//text` (si veda il documento di esempio usato finora) è quella che ha startPos 15, è facile trovare nella tabella delle tuple comma tutte quelle che ne sono *ancestor*: è sufficiente guardare gli attributi strutturali. Quando si sono reperite queste si risale di un livello, cercando le tuple di tipo articolo. Si parte dal nodo foglia ed ad ogni passo si risale verso la root.

Quando si è arrivati alla root si saranno reperite le tuple che formano un path completo in accordo con quello richiesto nella query, ed in più verifica l'inclusione del testo specificato.

Se si utilizza una semplice struttura come una lista per la memorizzazione delle tuple lette con `GetNextText`, e si effettuano solo inserimenti in testa alla lista, ora è possibile passare tali elementi nel loro ordine di lista a TwigStack, ed avere i risultati completi.

Al momento di ogni lettura dal DB con GetNextText è possibile verificare sia i vincoli testuali che (eventuali) vincoli temporali.

6.3.3 La consistenza nelle dimensioni temporali

All'interno del presente lavoro di tesi è stata data una notevole rilevanza all'aspetto temporale. Nella fase di suddivisione di documenti XML in strutture tupla si è adottata una trattazione rigorosa di tutte le dimensioni temporali coinvolte (paragrafo 5.5.1). Nelle tuple è possibile ritrovare completamente la caratterizzazione delle quattro dimensioni temporali adottate nel modello: publication time, valid time, transaction time ed efficacy time.

Nel paragrafo 6.1 si è visto che all'interno di una interrogazione è possibile specificare dei vincoli temporali. Tali vincoli sono imponibili su ognuna delle quattro dimensioni temporali trattate con l'utilizzo dell'operatore `OVERLAPS` (anche se non si è obbligati a specificarle tutte); sulle dimensioni temporali che non sono presenti in una query non è imposto nessun vincolo. Per esprimere i vincoli temporali si fa ricorso alla sintassi di TSQL2 (presentata nel capitolo 1); un esempio potrebbe essere:

```
TRANSACTION OVERLAPS '1980-01-01'
AND VALID OVERLAPS PERIOD('1970-01-01','1999-12-31')
```

Una query contenente le specifiche appena mostrate significa che si è interessati a tutte le versioni testuali degli atti normativi così come erano nel DB il Primo Gennaio 1980 (vincolo sul transaction time) e che hanno un periodo di validità che si interseca con il periodo che va dal Primo Gennaio 1970 al 31 Dicembre 1999.

Grazie alla suddivisione adottata è possibile controllare i vincoli temporali specificati nell'interrogazione su *ogni* struttura tupla direttamente nella fase della loro lettura da DB da parte degli algoritmi analizzati nel presente capitolo.

Le tuple così generate vengono passate a TwigStack che si occupa della loro

collocazione negli stack e della costruzione dei risultati in forma di percorsi. In tale algoritmo, quando una tupla viene inserita nello stack relativo al nodo foglia, si avvia la routine `outputPathSolution()` che risale di stack in stack fino a quello relativo al nodo root per poi generare le soluzioni complete ad un singolo path della query.

Ad esempio se la query in figura 6.15 potrebbe avere come soluzione il percorso in tabella 6.8.

Query:

- *vincolo strutturale*:
  articolo//rubrica//text
- *vincolo testuale*:
  nessuno
- *vincolo temporale*:
  TRANSACTION OVERLAPS '1980-01-01'
  AND VALID OVERLAPS PERIOD('1970-01-01','1999-12-31')

Figura 6.15: Esempio di query

type	startPos	endPos	level	ttStart	ttEnd	vtStart	vtEnd
articolo	10	23	4	20/12/1979	31/12/1984	01/01/1980	F
rubrica	11	13	5	20/12/1979	UC	01/01/1980	F
text	12	-	6	20/12/1979	UC	01/01/1980	F

Tabella 6.8: Possibili risultati alla query

Invece, una tupla che non verrà mai letta da `GetNext2` (e quindi mai riportata a TwigStack) è quella in tabella 6.9, perchè non rispetta il vincolo imposto sul transaction time ma solo sul valid time. Vale un ragionamento analogo per ogni dimensione temporale coinvolta nella query.

type	startPos	endPos	level	ttStart	ttEnd	vtStart	vtEnd
articolo	10	23	4	01/01/1985	UC	01/01/1980	31/12/1996

Tabella 6.9: Tupla che non rispetta i vincoli temporali

Oltre al controllo in fase di selezione appena visto, viene effettuato un *controllo di consistenza*, sulle tuple che appartengono ad una soluzione. Questo significa che tutte le strutture tuple di un possibile risultato devono verificare

l'operatore `OVERLAPS` fra loro per ogni dimensione temporale specificata nel modello: infatti è possibile che le tuple di una soluzione verifichino i vincoli espressi dalla query, ma non siano consistenti fra loro. Le tuple della soluzione in tabella 6.8 sono consistenti, mentre quelle della tabella 6.10 non sono consistenti anche se verificano i vincoli nella query sia sul valid che sul transaction time.

type	**startPos**	**endPos**	**level**	**ttStart**	**ttEnd**	**vtStart**	**vtEnd**
articolo	10	23	4	01/06/1980	31/12/1984	01/06/1980	F
rubrica	11	13	5	20/12/1979	01/03/1980	01/01/1980	F
text	12	-	6	20/12/1979	F	01/01/1980	F

Tabella 6.10: Risultato *non consistente*

In tali tuple il transaction time di articolo non interseca in nemmeno un chronon il transaction time di rubrica, da ciò consegue che il risultato non è consistente.

All'interno di [2], come è già stato detto in precedenza, nei risultati era verificato l'operatore `CONTAINS`; l'utilizzo di `OVERLAPS` al posto di `CONTAINS` permette di applicare il modello studiato ad un maggior numero di casi concreti, rendendolo di validità più "generale".

E' importante notare che la soddisfazione dei vincoli temporali non ha nulla a che vedere con il *controllo di consistenza*, anzi possiamo dire che sono due *concetti ortogonali*. Infatti i vincoli temporali sono verificati durante la fase di generazione delle tuple e servono a limitare i dati coinvolti nella query, mentre la *consistenza* viene valutata *contemporaneamente* o *successivamente* all'elaborazione di TwigStack. Quanto è stato detto finora spiega il controllo di consistenza *successivo* all'elaborazione (avviene infatti quando sono già disponibili le soluzioni), ma può avvenire anche *contemporaneamente* all'elaborazione in quanto ogni volta che una tupla viene inserita negli in uno stack è possibile tenere traccia della sua temporalità. Questo permette di prevedere quali tuple non sono consistenti prima del loro effettivo inserimento negli stack (e quindi anche prima della generazione dei risultati). Questo accorgimento è particolarmente prezioso perchè permette inserire negli stack solo le tuple *strettamente necessarie*, evitando il più possibile problemi di *overflow degli stack*. Tale problema è molto sentito quando si ha a che fare con database temporali, nei quali è possibile avere un elevato numero versioni successive della stessa entità che potrebbero dare luogo ad un overflow. Possiamo dire che sono possibili due metodi di cancellazione delle tuple *non consistenti* ideati nella presente tesi:

1. *cancellazione a posteriori*: avviene quando sono già state inserite le tuple negli stack e si sono generate le soluzioni; tale metodo è molto semplice e per la sua attuazione non sono necessarie ulteriori strutture ed elaborazioni, per questo può risultare molto efficiente anche se si corre il rischio di overflow sugli stack

2. *cancellazione a priori*: avviene prima di inserire le tuple negli stack; tale metodo necessita di ulteriori strutture ed elaborazioni, per questo può essere meno efficiente del precedente ma eliminare il rischio di overflow sugli stack

6.4 Metodi di accesso e generazione dei risultati

Quando una query come quella in figura 6.16 è stata formulata i metodi di accesso a disposizione sono:

1. l'indice secondario (un semplice B-TREE) costruito sul campo startPos di ogni tupla. Ogni tabella del DB (sono 8, ciascuna relativa ad un particolare tipo di tupla) ha un proprio B-TREE, che costituisce l'unico metodo di accesso vero e proprio alle strutture tupla

2. l'inverted index costruito sul testo, utilizzabile solamente in concomitanza di query che contengano dei vincoli testuali

Query:

- *vincolo strutturale*:
  legge//titolo//text
- *vincolo testuale*:
  contains 'informatica'
- *vincolo temporale*:
  VALID OVERLAPS PERIOD ('1970-01-01','2000-12-31')
  AND TRANSACTION OVERLAPS '1980-01-01'

Figura 6.16: Esempio di query

Il secondo metodo di accesso si basa sull'inverted index per trovare i valori degli startPos che fanno match con il vincolo testuale della query, ma per reperire effettivamente le tuple dal DB viene utilizzato ancora il B-TREE. Anche se vengono analizzate più strutture di accesso, le interrogazioni che contengono testo sono molto più veloci nella loro esecuzione rispetto a quelle che non lo contengono, in quanto viene applicato un *filtro* dotato di una *selettività* proporzionale alla complessità dell'espressione da valutare. Ad esempio, se nella tabella text in cui ci sono 1000 tuple di tipo text vi potrebbero essere solo 10 di queste tuple che contengono la parola 'inquinamento'; utilizzando l'inverted index è possibile conoscere i loro startPos (che corrisponde al K_i specificato nel paragrafo 6.1.2) ed utilizzare l'altra struttura di accesso per il reperimento. Se l'espressione fosse più complicata allora le tuple che farebbero match potrebbero essere meno di 10, e ciò restringerebbe il numero di tuple da cercare nel DB.

Una volta che si è reperita la tupla text che verifica l'espressione contenuta nella query è possibile accedere alla tupla corrispondente al nodo della query che vi sta direttamente sopra, utilizzando ancora una volta il B-TREE sul campo startPos, e così arrivare fino al nodo root del vincolo strutturale specificato (si veda 6.3.2).

Quando nella query non sono presenti vincoli testuali l'unico metodo di accesso utilizzabile è il B-TREE. Ci si affida all'algoritmo `GetNext2` per la generazione delle tuple in modo da soddisfare i vincoli strutturali della query, ed in secondo luogo quelli temporali. Per la verifica delle condizioni strutturali è di grande utilità accedere alle tuple secondo il loro valore di startPos (cosa che il B-TREE permette di fare) in quanto applicando la teoria dei join strutturali è possibile ridurre al minimo le letture inutili da disco. E' da notare che non ci sono strutture di accesso costruite sui campi che rappresentano attributi temporali, da ciò consegue che per la loro verifica è prima necessario reperire la tupla su cui tale verifica deve essere fatta.

Durante l'esecuzione della query, le strutture tupla vengono lette dal DB grazie a `GetNext2` (o `GetNextText` se si è in presenza di interrogazioni testuali) e poi vengono passate a TwigStack che si preoccupa di metterle sugli stack e generare le soluzioni finali. Ogni soluzione finale corrisponde ad un set di tuple *consistenti* (si veda il paragrafo 6.3.3) che verificano tutti i vincoli dell'interrogazione, e viene di volta in volta memorizza su un apposito file.

A differenza di quanto succede in [2], ad ogni iterazione conclusasi con successo all'interno della fase di elaborazione si ha a disposizione *immediatamente* il corrispettivo risultato finale. Non solo, ma il risultato viene scritto utilizzando marcatori XML, in modo che sia possibile:

- una visualizzazione "user friendly" con un semplice browser
- una successiva (ed ipotetica) fase di elaborazione automatica

Ad esempio è possibile applicare al file di output una semplice trasformazione XSLT per ottenere una pagina web contenente i risultati: in figura 6.17 vi è un possibile risultato della query in figura 6.16.

All'interno del lavoro [2], i risultati della selezione non erano *finali* ma *parziali* in quanto necessitavano delle ricostruzione effettuata con `const-tree()` per rispondere alle specifiche della query.

Nella figura 6.17 è possibile vedere come le tuple che formano il risultato abbiano:

- attributi strutturali *coerenti* con la teoria degli structural join
- attributi temporali che soddisfano la query e che sono consistenti fra tupla e tupla
- la presenza del testo esplicitato nella query all'interno della tupla text

Si può notare inoltre che tutte le tuple appartengono allo stesso documento (infatti hanno lo stesso docId) anche non sono stati fatti controlli su tale campo: questo avviene perchè tutte le tuple specificate sono *descendant* della stessa tupla legge che identifica l'intero documento.

Mozilla

type:legge	docId:DL/21/1966	num:21
startPos: 510	ensPos: 532	level: 1

vtStart: 05/04/1967	vtEnd: ---
etStart: 20/04/1967	etEnd: ---
ttStart: 09/01/1967	ttEnd: ---

type:titolo	docId:DL/21/1966	num:0
startPos: 511	ensPos: 513	level: 2

vtStart: 05/04/1967	vtEnd: ---
etStart: 20/04/1967	etEnd: ---
ttStart: 09/01/1967	ttEnd: ---

type:text	docId:DL/21/1966	num:0
startPos: 512	ensPos: 512	level: 3

vtStart: 05/04/1967	vtEnd: ---
etStart: 20/04/1967	etEnd: ---
ttStart: 09/01/1967	ttEnd: ---

text: titolo informatica

Done

Figura 6.17: Un interno path visualizzato in una pagina web

Capitolo 7

Il prototipo software

Nei capitoli precedenti si sono presentati gli algoritmi e le strutture necessarie ad una interrogazione efficiente di documenti XML temporali, spiegandone i concetti ed i meccanismi alla base. Tutto questo è stato fatto mantenendo un approccio il più possibile indipendente dall'effettiva implementazione. In questo capitolo verrà approfondito proprio l'aspetto del progetto e dell'implementazione: verrà motivata la scelta di un particolare database system e verranno mostrati i diagrammi dei package e delle classi che compongono il programma, al fine di proporre un prototipo funzionante del lavoro svolto.

Si è deciso di lavorare utilizzando Java perchè permette una discreta rapidità nella stesura di programmi di una certa complessità ed un'ottima integrazione con strumenti per il parsing di documenti XML.

7.1 Il Database System

Quando si progetta un'applicazione, è di fondamentale importanza scegliere gli strumenti giusti in base alle proprie esigenze. Oggi si ha a disposizione una scelta molta vasta di *database system*, che potremmo etichettare utilizzando i termini "network", "relational", "object-oriented" ed "embedded", o combinazioni di queste.

Fondamentalmente, ogni database system provvede a due tipologie di servizi.

Il primo servizio è il *Data Access*. Data Access significa poter inserire nuovi record (inserting), trovare i dati di interesse (searching), modificare dati precedentemente inseriti (updating) o rimuoverli completamente (deleting). In pra-

tica ogni database system provvede a questo servizio, ma con meccanismi e prestazioni differenti.

Il secondo servizio è il *Data Management*. Data Management è più complicato rispetto al Data Access, così provvedere un buon servizio di gestione dati è la parte più difficile che si ha nel costruire un database system.

Data Management significa :

- permettere a molti utenti di lavorare sullo stesso database contemporaneamente (concorrenza)
- permettere di agire su molti dati in uno stesso istante (supporto alle transazioni, che rispettano le proprietà ACID)
- permettere il ripristino in caso di crash dell'applicazione o del sistema (recovery)

7.1.1 Cosa è Berkeley DB

Berkeley DB is an open source embedded database library that provides scalable, high-performance, transaction-protected data management services to applications.

Questa è la miglior definizione che si possa dare di Berkeley DB, in italiano suonava diversamente.

Con open source intendiamo che Berkeley DB è distribuito una licenza conforme alla *Open Source License*. Questo garantisce che Berkeley DB è liberamente disponibile per l'uso e la sua ridistribuzione in altri prodotti Open Source. L'intero codice sorgente di Berkeley DB è disponibile gratuitamente per l'uso, e la *Sleepycat Software* (si veda [14]), produttrice del software, mette a disposizione licenze commerciali per ridistribuzioni proprietarie.

Berkeley DB è *embedded* (letteralmente *integrato, interno*), perchè è direttamente collegabile all'applicazione. Infatti "gira" nello stesso spazio degli indirizzi dell'applicazione che lo utilizza.

Come risultato si ha che non c'è necessità di comunicazione fra processi (sia fra processi della stessa macchina, che fra processi su macchine sulla stessa rete) per l'utilizzo delle sue funzionalità di database.

Berkeley DB mette a disposizione una semplice API (Application Program

Interface) disponibile per molti linguaggi fra cui Java, C, C++, Perl, TCL, Python e PHP.

Ogni operazione riguardante il database può essere utilizzata richiamando l'apposita API all'interno della libreria.

Diversi processi, o differenti thread all'interno di un singolo processo possono usare il database nello stesso tempo se ognuno utilizza la libreria di Berkeley DB.

Le operazioni a basso livello come il meccanismo del locking, del log delle transazione, della gestione dei buffer condivisi e della gestione della memoria centrale sono trattate dalla libreria in modo trasparente all'utente (ed all'occorrenza anche configurabile).

La libreria è estremamente portabile. Funziona su quasi tutte le versioni dei sistemi UNIX e Linux, su Windows e su gran parte dei sistemi operativi embedded e real-time. Funziona su macchine a 32 ed a 64 bit. E' stata utilizzata nell'implementazione si server per internet, macchine per ufficio, palmari e switch di rete. Una volta che Berkeley DB è collegato all'applicazione l'utente finale generalmente non sa nemmeno della presenza di un database.

Berkeley DB è scalabile in un vasto numero si aspetti. La sua libreria in sè è molto compatta , ma può gestire database fino a 256 TeraByte di dimensione e supporta molteplici utenti che operano sullo stesso database nello stesso momento (concorrenza). Inoltre aumentando le risorse hardware è immediato avere un miglioramento delle prestazioni, magari dopo una opportuna fase di system tuning.

Berkeley DB generalmente supera nelle prestazioni i tradizionali DBMS e object-DBMS nelle applicazioni che hanno bisogno di un embedded database. Ciò avviene grazie a :

1. la libreria, che gode dello stesso spazio degli indirizzi dell'applicazione, quindi non ci sono costi di comunicazione da pagare fra processi che vogliono comunicare fra loro

2. le chiamate a funzione, che sono fatte all'applicazione sulle API della libreria e interessano direttamente il database, evitando elaborazioni che altri sistemi (come ad esempio i classici R-DBMS) sono obbligati a fare. Infatti utilizzando le API della libreria evitiamo l'esecuzione del parsing del

linguaggio utilizzato dalla query (ad esempio SQL), e la produzione di un piano di esecuzione della query stessa (da parte del Query Optimizer)

7.1.2 Servizio di Data Access

I metodi di accesso ai dati memorizzati in Berkeley DB possono essere scelti dal programmatore a seconda dell'applicazione. Berkeley DB mette a disposizione le seguenti strutture di memorizzazione (*storage structures*):

- B-Tree
- Hash
- Recno, (record number) per dare una semplice numerazione ai record
- Queue, per sistemi a coda

La struttura di accesso maggiormente utilizzata all'interno dell'implementazione adotttata è il B-Tree, in quanto consente il facile reperimento di record caratterizzati da un ordinamento (es: gli oggetti tupla in ordine startPos)

7.1.3 Cosa non è Berkeley DB

Il punto debole di Berkeley DB rispetto a molti altri database system tradizionali è l'avere dei metodi di accesso abbastanza poveri. Infatti i record sono memorizzati come coppie *(key,value)*, su cui Berkeley DB può fare ben poche operazioni:

insert aggiungo un record ad una tabella

find trovo un record in una tabella specificando il suo valore di chiave

update modifico un record che ho già trovato (con **find**)

delete cancello un record, specificando la chiave

E' importante notare che Berkeley DB non opera mai sulla parte *value* del record, che viene considerata *payload*, cioè qualcosa che segue semplicemente la chiave.

Sia *key* che *value* sono considerate semplici stringhe di byte, sia di lunghezza fissa che variabile. Il risultato di questo è che il programmatore può mettere

strutture native della propria applicazione direttamente in tali campi, memorizzandole nel database senza nessun tipo di conversione ad altri formati di dati. Il salvataggio ed il reperimento dei dati sono molto semplici, ma l'applicazione **deve sapere** fin da subito che strutture verranno messe in tali campi. Berkeley DB non conosce le strutture che vengono memorizzate, è l'applicazione che lo deve sapere.

Siccome entrambi i campi possono essere di una lunghezza massima di quattro GygaByte, in un singolo record possono essere memorizzati anche dati che necessitano di molto spazio come i dati multimediali : immagini, filmati e stream audio. I dati di grandi dimensioni non vengono trattati in maniera differenti da quelli di piccole dimensioni: vengono semplicemente spezzettati in pacchetti grandi quanto una pagina logica (impostabile tramite parametri) e memorizzati così come sono. Quando arriverà una richiesta per quel dato, tali pagine varranno riassemblate e restituite all'applicazione che ha richiesto il dato.

Berkeley DB non è un database relazionale. Per prima cosa non supporta query scritte in SQL, ma tutte le operazioni di accesso ai dati avvengono utilizzando le API della libreria. Il supporto di SQL avrebbe significato sicuramente una maggiore espressività nelle query, ma anche overhead per il parsing della query, ottimizzazione ed esecuzione.

Se il programmatore conosce in partenza che tipo di strutture saranno memorizzate può scrivere delle routine a basso livello con la libreria di Berkeley DB per leggere e scrivere i dati. In questo modo è possibile avere delle performance migliori; è esattamente lo scenario che si presenta all'interno di questo elaborato, infatti:

- si ha una conoscenza *a priori* del modello
- non sono necessarie particolari operazioni sui dati
- non è necessario l'utilizzo del modello relazionale
- si cercano buone prestazioni

Se viene confrontato con dei sistemi relazionali, Berkeley DB può essere considerato come una *high-performance, transactional library.*

7.2 I package utilizzati

I package più importanti sono:

1. *Strutture*: comprende tutte le strutture di base utilizzate nell'implementazione
2. *Berkeley*: contiene i metodi per l'iterazione col database system
3. *GeneraLeggiXML*: consente la creazione di ampi set di leggi in formato XML che soddisfano uno specifico XML Schema
4. *Sax*: permette la suddivisione di documenti XML rappresentanti leggi in strutture tupla
5. *Lucene*: contiene metodi e strutture per la generazione e l'interrogazione dell'inverted index
6. *GetNext*: contiene gli algoritmi in grado di reperire le tuple dai DB a seconda della query specificata
7. *Join*: contiene le routine utilizzate per effettuare i join strutturali sulle tuple riportate dagli algoritmi del package GetNext

Nella figura 7.1 è possibile vedere le dipendenze fra i vari package. I package Strutture e Berkeley sono stati racchiusi nel package *Base* in quanto possono essere considerati strettamente collegati all'interno della maggior parte delle elaborazioni effettuate.

7.2.1 Le strutture per l'elaborazione: il package Strutture

All'interno del Package Strutture sono presenti tutte le classi rappresentanti strutture; ora verranno descritte le più importanti.

La classe Tupla è composta da un vasto set di variabili interne e da opportuni metodi per ricavarle. Nella figura 7.2 è possibile vedere tutte le variabili di un oggetto Tupla e le interfacce che implementa.

A questo punto è lecito chiedersi come mai è stato scelto un tipo di dati intero per rappresentare una data. Si è deciso di fare questo perchè la granularità necessaria a questa tesi era quella dei giorni (vedi capitolo 2), e perchè

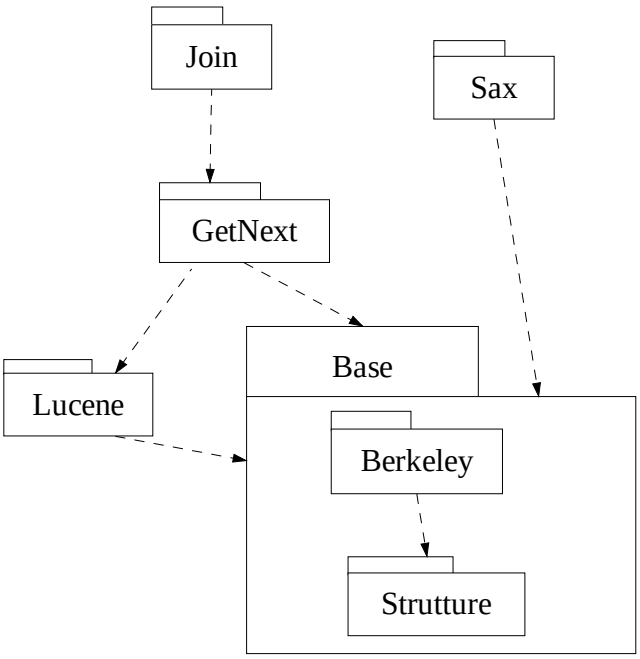

Figura 7.1: Package diagram del prototipo implementato

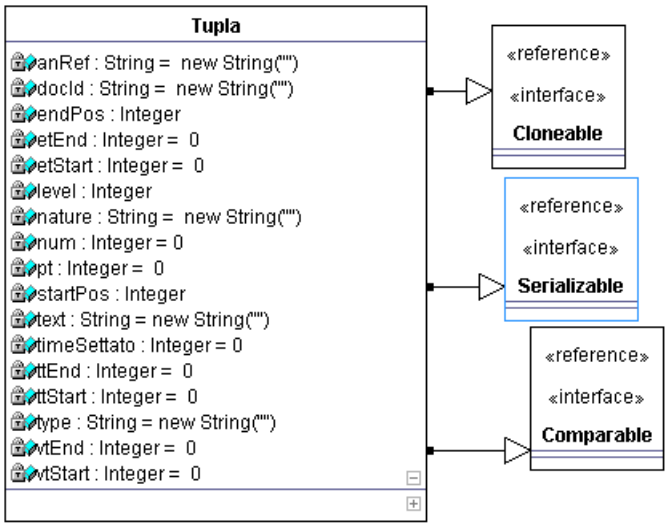

Figura 7.2: Class diagram dell'oggetto Tupla

non era nel nostro interesse descrivere atti normativi molto distanti nel tempo: così abbiamo pensato di limitare l'asse dei tempi. Infatti si è pensato di mettere un'origine sull'asse dei tempi corrispondente al Primo Gennaio dell'anno 1800, e di mettere la fine dell'asse dei tempi corrispondente con il 31 Dicembre 9999; facendo questo è lecito supporre di non avere atti normativi in formato digitale

al di fuori di questo intervallo. Per esprimere una data basta considerare l'offset (spiazzamento) a partire dall'origine dei tempi fino alla data desiderata, e per fare ciò è più che sufficiente un dato di tipo int (4 byte). Java è dotato di classi atte alla descrizione di una data (come `java.util.Data`), ma la loro memorizzazione occupa più di 80 byte per data, il che sarebbe stato un inutile spreco. Ricordiamo che l'oggetto Tupla sta alla base dell'elaborazione, è quindi fondamentale renderlo il più *snello* possibile. Inoltre in tale oggetto compaiono ben 7 date: con la memorizzazione con interi si occupano 28 byte, mentre con l'utilizzo dell'oggetto Data se ne sarebbero utilizzati circa 560 byte.

In figura 7.3 è possibile vedere la classe MyDate, creato dall'estensione `java.util.Data` proprio per la conversione di un int in una data in formato inglese (`yyyy-MM-dd` utilizzata nei file XML) o in formato italiano (`dd/MM/yyyy`, utilizzato per una comprensione più immediata dell'output).

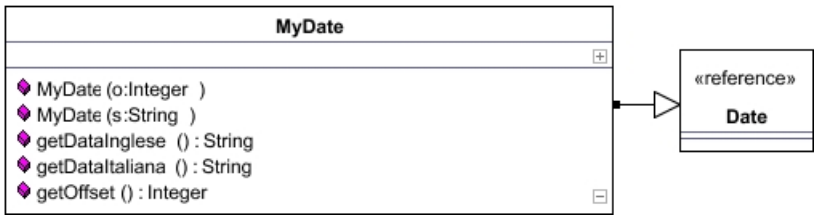

Figura 7.3: Class diagram dell'oggetto MyDate

7.2.2 Memorizzazione dati: il Package Berkeley

All'interno del package `Berkeley` sono presenti tutte le classi necessarie alla creazione, all'interrogazione ed al mantenimento della base di dati utilizzando Berkeley DB. Prima che sia possibile svolgere qualsiasi tipo di operazione sui database è necessario crearli e configurarli correttamente. La classe `AssociaDb` serve proprio a questo (figura 7.4).

Le tabelle di Berkeley DB sono oggetti di tipo `Db`, e che chiameremo DB ; per istanziare tale oggetto bisogna:

1. associarlo ad un file fisico presente nel DB (se non esiste verrà creato)
2. associarlo ad un nome logico
3. impostare appropriatamente una serie di *flag*

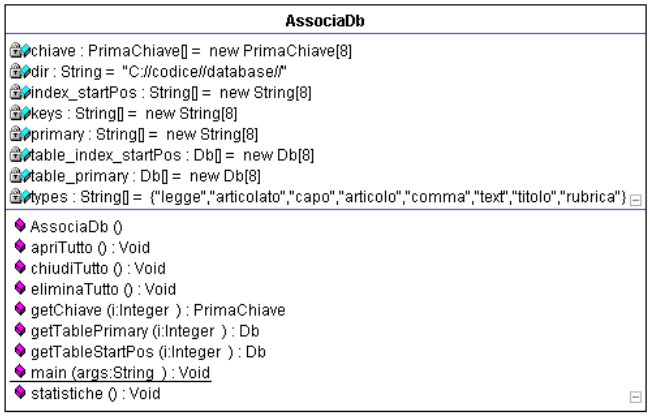

Figura 7.4: Classe AssociaDB

I flag che abbiamo impostato per i DB primari sono :

DB_CREATE indica che il DB (se non è presente) deve essere creato

DB_BTREE indica il metodo di accesso utilizzato

L'insieme di questi tag indica che viene creato un DB sul cui campo *key* verrà costruito un metodo di accesso (un vero e proprio indice) di tipo B-Tree. Tale operazione viene effettuata per ogni tipo di tupla (sono 8, come appare in figura 5.5 ma ora inizio a contare da 0 e non da 1), utilizzando un semplice ciclo `for`.

Nel campo *key* del record abbiamo messo un valore intero che viene generato di volta in volta attraverso l'utilizzo di una opportuna routine appositamente scritta. Generare una chiave unica intera è una operazione che molti DBMS tradizionali offrono, ma che Berkeley DB non può mettere a disposizione in quanto non ha coscienza del tipo di dato memorizzato nel campo *key*. La struttura di accesso di tipo RECNO offre questa possibilità, ma genera problemi con le strutture di accesso secondarie.

Utilizzando la routine descritta sopra per la generazione dell'intero da memorizzare nel campo *key* siamo sicuri che nel DB non ci sono più record con lo stesso valore di chiave (l'unicità della chiave è indispensabile per i DB principali su cui si deve costruire un metodo di accesso secondario).

Nel campo *value* del record viene memorizzato un oggetto tupla opportunamente serializzato.

Nella creazione della tabella primaria si è usato come campo chiave un semplice intero. Per la generazione dell'intero appropriato anche dopo modifiche ed

inserimenti nel DB, si fa riferimento all' apposita routine che prevede l'accesso ad un file in cui ci sono tutte le informazioni necessarie per effettuare un inserimento con una chiave che non sia già presente nel DB.

Se si fosse utilizzato il valore startPos di ogni tupla come campo *key* non sarebbe stato un valore unico, in quanto abbiamo visto che è possibile avere tuple con lo stesso startPos (corrispondono a più versioni temporali della stessa versione testuale).

I metodi di accesso secondari si basano su quello primario, e per il loro utilizzo è necessario che il relativo DB primario non contenga chiavi duplicate. Successivamente si procede alla creazione di un vero e proprio oggetto Db (come quello primario) e lo si associa utilizzando l'istruzione *associate* ad un DB primario. Al momento dell'associazione del DB primario è necessario specificare una routine che implementi l'interfaccia `SecondaryKeyCreator` che ci permetta di creare la chiave secondaria a partire dal campo *value* di ogni record. Tale routine si chiama SKC ed il suo class diagram è visibile in figura 7.5.

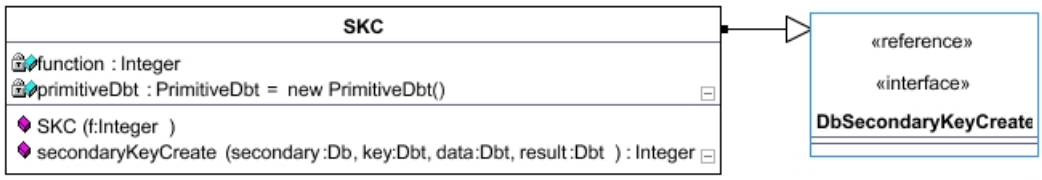

Figura 7.5: Classe SKC

Per ogni record presente nel DB primario viene creato un record nel DB secondario che avrà nel campo *key* ciò che viene generato dalla routine SKC, mentre nel campo *value* ci sarà la chiave primaria della tupla in questione. In particolare SKC ricava il valore di startPos da ogni tupla. Nella figura 7.6 è rappresentata la creazione del db secondario. Ogni accesso al DB secondario può restituire sia il campo *value* dello stesso DB sia il campo value del DB primario a cui è associato (che è l'oggetto che più ci interessa: l'oggetto Tupla), a seconda dell'API utilizzata.

Nella classe `AssociaDb` ci sono le procedure necessarie alla creazione dei DB ed alla loro associazione ai relativi DB secondari. Inoltre il momento dell'apertura di un DB esistente da parte dell'applicazione è considerata una sorta di "connessione" ai database. Dopo tale operazione i dati nel db possono essere acceduti attraverso il ricorso alle apposite API. Tale connessione è considerata

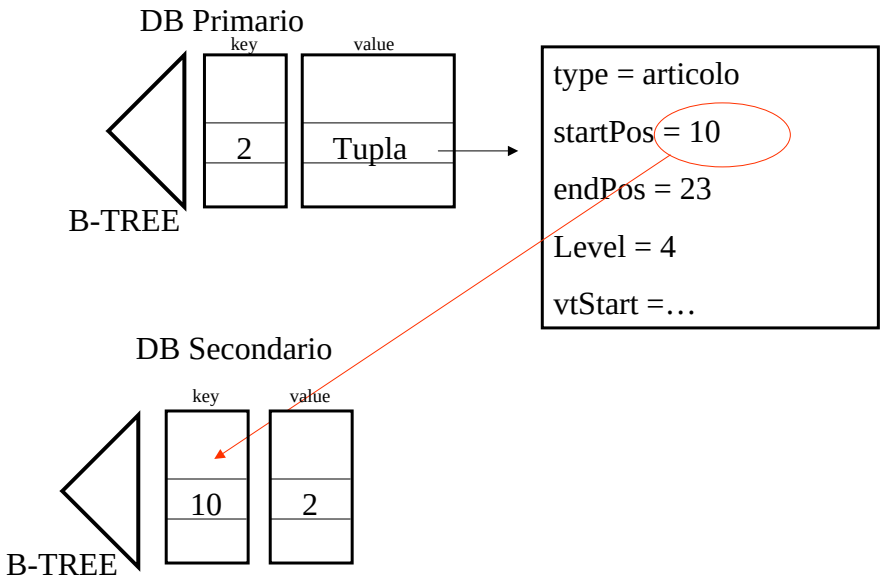

Figura 7.6: Classe DBUtility

una operazione piuttosto lenta, e quindi si preferisce farla all'inizio dell'applicazione e non precedentemente alla richiesta di ogni tupla. Prima della conclusione dell'applicazione è necessario chiudere tale connessione.

Nella stessa classe troviamo anche metodi per verificare le chiavi utilizzate per gli inserimenti nei DB principali e per cancellare l'intero database.

Un'altra classe molto importante è `DBUtility` che incorpora una serie (figura 7.7) di metodi più ad alto livello rispetto a quelli messi a disposizione dalle API. Si è detto che se si ha la conoscenza delle strutture che si intendono memorizzare con Berkeley DB è possibile scrivere opportune routine a basso livello per avere performance migliori, ed è proprio quello che si è fatto. Sono metodi scritti appositamente per sopperire ad esigenze particolari di questa tesi. I metodi più importanti sono :

- `memorizzaFile()`: serve per memorizzare le tuple provenienti dal parsing di un documento XML rappresentante un atto normativo

- `memorizzaDirectory()`: serve per memorizzare le tuple provenienti dal

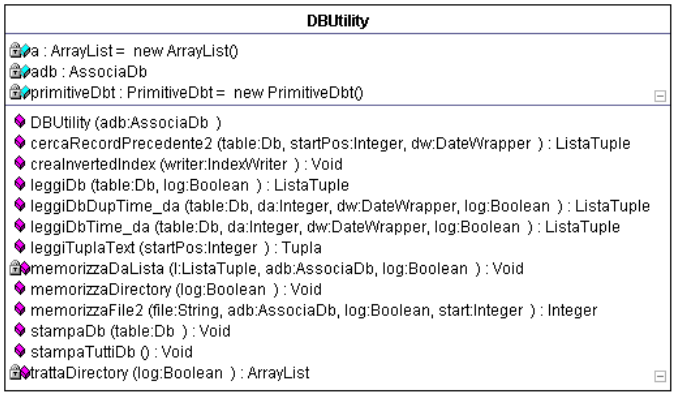

Figura 7.7: Classe DBUtility

parsing dei documenti XML rappresentante un atto normativo che si trovano all'interno di una directory

- `leggiDb()`: serve per leggere tutte le tuple memorizzate all'interno di un DB

- `leggiDbTime_da()`: serve per leggere tutte le tuple memorizzate all'interno di un DB, e ritornarle solo se verificano certi vincoli sugli attributi temporali e se hanno startPos maggiore o uguale ad un certo valore

- `cercaRecordPrecedente()`: serve per trovare le tuple ancestor di una particolare tupla

Il metodo più importante è `cercaRecordPrecedente(Db table, int startPos,DateWrapper dw)`, in quanto permette di reperire la tupla *precedente* a quella indicata dal valore in ingresso `startPos` (che si solito coincide con lo startPos di una tupla già reperita di livello inferiore nella query). Il funzionamento è descritto all'interno della figura 7.8: le tuple all'interno dei box rettangolari sono quelle realmente accedute, anche se si risale a quelle appena precedenti (negli ovali) che rappresentano gli *ancestor*. Se tale tupla (o tuple) non soddisfano i requisiti temporali specificati all'interno dell'istanza di `dw` allora vengono scartate e se, dopo tale controllo, non ci sono più tuple allora ci si può fermare.

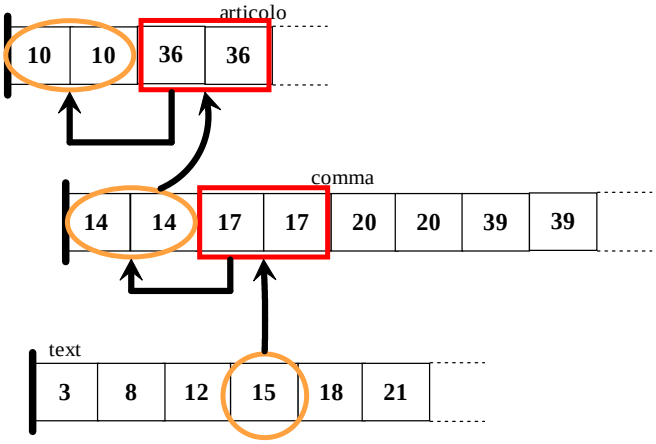

Figura 7.8: Funzionamento del metodo `cercaRecordPrecedente(Db table,int startPos,DateWrapper dw)`

7.2.3 La generazione delle leggi : il package GeneraLeggiXML

La classe GeneraXML all'interno del package GeneraLeggiXML è stata utilizzata per la creazione di documenti XML *validi* secondo un particolare XML Schema. Si sono creati 6 differenti *modelli* di atto normativo, ognuno con un numero diverso di articoli, commi, versioni di commi, ma senza testo né attributi temporali.

Ad ogni iterazione i campi testuali sono stati riempiti con stringhe di diverse lunghezze generate casualmente a partire la una lista rappresentante un *vocabolario* ed un'altra lista rappresentante una *stoplist*. I campi contenenti le date relativi agli attributi temporali sono anch'essi generati casualmente all'interno di intervallo centrato sul publication time (primo attributo temporale ad essere estratto). Si è fatta particolare attenzione a generare solo date *verosimili*, cioè che non ci fossero sovrapposizioni fra differenti versioni temporali della stessa entità, o che ci fossero date troppo distanti fra loro. Dai 6 modelli sono nate leggi con una dimensione che varia dai 4 ai 140 KB.

7.2.4 L'inverted Index: il Package Lucene

L'inverted index utilizzato in questa tesi è Jakarta Lucene, un software open source per la ricerca di testo scritto interamente in Java. Le classi utilizzate

nel package `Lucene` allo scopo della creazione e del mantenimento dell'inverted index necessario per le ricerche testuali sono (figura 7.9):

MyDocument classe che permette la creazione di strutture memorizzabili da Lucene a partire da tuple di testo

CreateInvertedIndex classe usata per la creazione dell'inverted index

SearchInvetedIndex classe utilizzata per la ricerca di *espressioni* all'interno dell'inverted index

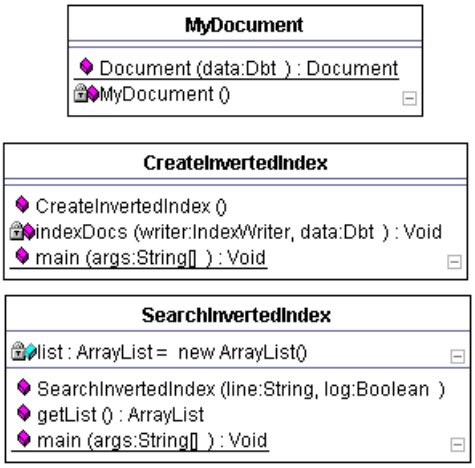

Figura 7.9: Classi all'interno del Package Lucene

Sia la creazione che la ricerca all'interno dell'inverted index sono effettuate su termini che non appartengono ad una *stoplist* memorizzata su un file, in modo da ignorare tutti i quei termini che non sono utili nelle ricerche testuali (articoli, avverbi, pronomi, ecc ...).

Si è visto che Lucene è un ottimo strumento per la ricerca di testi, in quanto è semplice da utilizzare e veloce sia nella creazione dell'inverted index sia nella sua interrogazione; inoltre mette a disposizione tool che permettono una veloce *ottimizzazione* dell'inverted index, operazione che risulta molto utile dopo un cospicuo numero di modifiche.

7.2.5 Parsing di un documento XML: package Sax

All'interno del package Sax si trovano tutte quelle classi che concorrono alla creazione delle tuple memorizzate nel Db a partire da documenti XML che rap-

presentano gli atti normativi. Per fare ciò LexParser è l'handler utilizzato dal parser Sax per la generazione delle corrispettive tuple. Per ben definire gli attributi temporali in relazione all'ereditarietà imposta dal modello adottato, la classe EstendiTempi effettua le ultime operazione sulle tuple.

7.2.6 Algoritmi di generazione delle tuple: package GetNext

All'interno del package `GetNext` ci sono le classi che si preoccupano di generare le tuple da passare poi a TwigStack per formare i risultati. Le classi più importanti sono

- `GetNext2`: permette la generazione delle tuple necessarie alla risoluzione di query senza vincoli testuali, rispettando quelli strutturali e temporali
- `GetNextText`: permette la generazione delle tuple necessarie alla risoluzione di query con vincoli testuali, strutturali e temporali.

All'interno di TwigStack basta istanziarle e richiamare le funzioni `getNodo()` e `getTupla()` (presenti in entrambe) per avere il nodo corrente e la tupla corrente (quella che sarà, forse, inserita negli stack).

La classe GetNext2

La classe `GetNext2` (figura 7.10) si occupa di generare le tuple necessarie alla query quando non sono presenti vincoli testuali. Tale generazione segue le specifiche già esposte nel paragrafo 6.3

La classe GetNextText

Nella classe GetNextNext (figura 7.11) è utilizzato l'inverted index sulle tuple di tipo text per ritrovare gli startPos delle tuple text interessate dalla query. Ogni volta che si ha a disposizione uno di questi startPos si può reperire la corrispettiva tupla dal DB utilizzando il B-TREE e controllare che soddisfi i vincoli temporali espressi nella query. Se ciò è verificato allora si procede con la ricerca delle tuple relative al nodo *parent* nella query accedendo alla relativa tabella ancora attraverso l'indice su startPos. Grazie al metodo `cercaRecordPrecedente()` della classe `DBUtility` è possibile risalire alle tuple *ancestor* con un numero

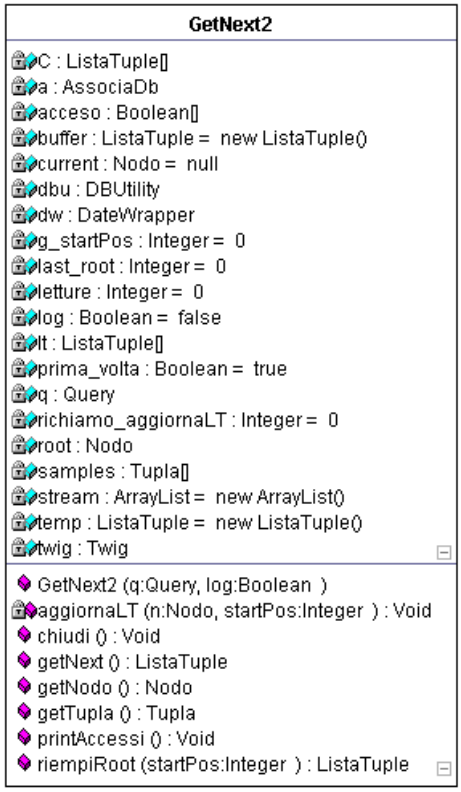

Figura 7.10: Classe GetNext2

minimo di accessi a disco (si veda 7.2.2). Questo procedimento si ripete finchè non si trova il nodo root della query o tutte le tuple così trovate non soddisfano le specifiche temporali della query ad uno qualsiasi dei livelli interessati.

7.2.7 Algoritmi per gli structural join: package Join

La classe più importante all'interno di questo package è TwigStack (figura 6.3). Tale algoritmo si avvale della generazione delle tuple da parte di GetNext2 o di GetNextText a seconda della presenza dei vincoli testuali nella query. Ogni tupla così generata viene esaminata esattamente come è già stato esposto nel paraggrafo 6.2, infatti è stato mantenuto anche il nome dei metodi utilizzati. E' possibile ridirigere l'intero output su un file per effettuare una lettura dei risultati in un momento successivo al termine dell'esecuzione.

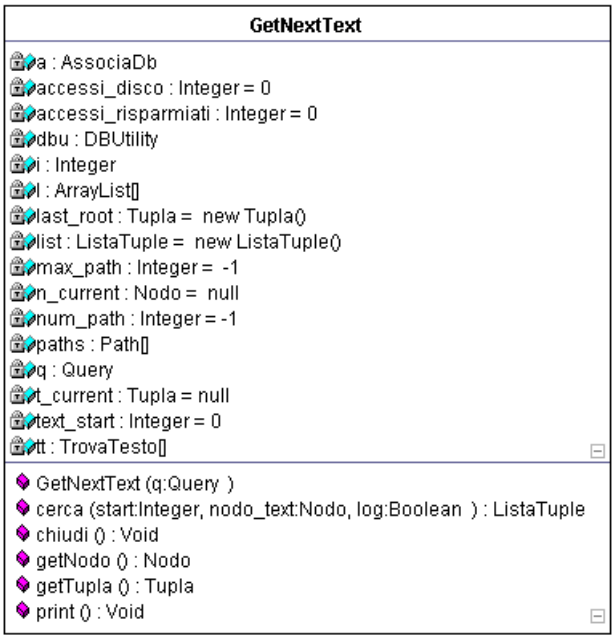

Figura 7.11: Classe GetNextText

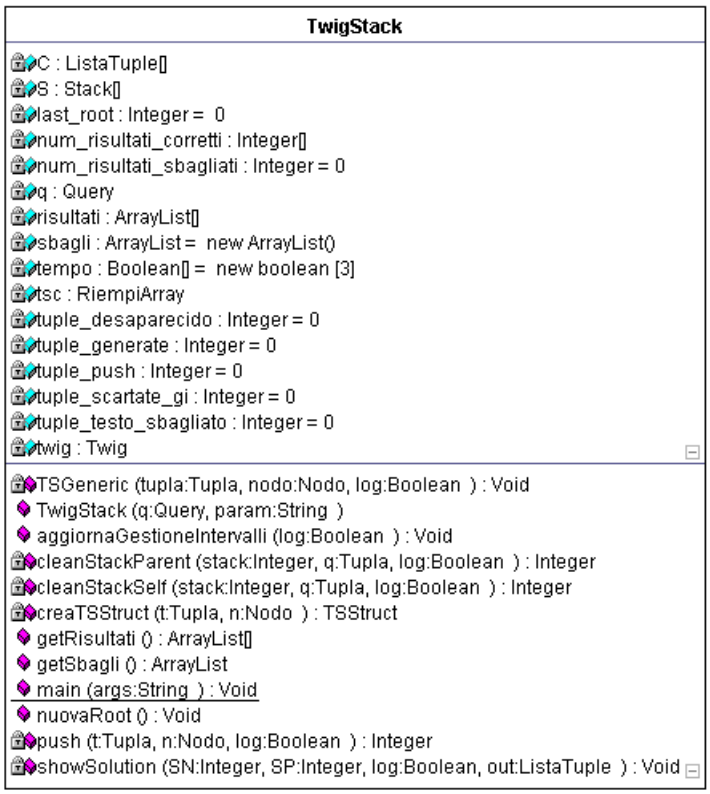

Figura 7.12: Classe TwigStack

Capitolo 8

Le prove sperimentali e i risultati ottenuti

All'interno di quest'ultimo capitolo verranno presentati i risultati sperimentali del prototipo software sviluppato. In particolare verrà tenuta in grande considerazione l'efficienza dimostrata in termini di prestazioni e di occupazione di memoria. Collezioni di documenti XML temporali rappresentanti atti normativi sono suddivisi in tuple e memorizzati in un database system. Si utilizzeranno l'architettura introdotta nel capitolo 5 e gli algoritmi presentati nel capitolo 6 per effettuare interrogazioni.

L'elaborato [2], in cui si introduce il sistema TeX, è considerato un punto di partenza per la presente tesi. Ne consegue che un confronto fra i risultati sperimentali ottenuti sia fondamentale per verificare la qualità del lavoro svolto. Sebbene gli approcci adottati siano molto differenti, nella fase di test del presente prototipo si è cercato di sottoporre al sistema interrogazioni il più possibile simili a quelle di TeX. Un aiuto alla rigorosità di questo confronto ci è dato dall'utilizzo di documenti XML rappresentanti atti normativi:

1. conformi allo stesso XML Schema adottato in [2]

2. aventi all'incirca le stesse dimensioni (da 4 a 140 KB, con una dimensione media di 24 KB per documento)

Tutti i test sono stati eseguiti su un personal computer dotato di processore AMD Athlon XP 2000+, un comune disco rigido IDE da 20 GB, una memoria centrale di 256 MB con installato Windows XP come sistema operativo. Come

database system è stato utilizzato Berkeley DB (versione 4.2.52) già introdotto nel paragrafo 7.1.

Al prototipo software sviluppato sono state sottoposte solo interrogazioni, le operazioni di inserimento e di modifica sono da considerarsi esterne alla presente tesi (si tratta solamente la parte di Norma-Query specificata in [12]).

Sono stati generati tre differenti data set di documenti XML; il minore dei data set usati nel presente prototipo corrisponde al più grande utilizzato nel progetto TeX, cioè in 5000 documenti XML con una occupazione di memoria di 120 MB. Sono poi stati generati altri due set, rispettivamente di dimensioni doppie e quadruple di quello di partenza; questo significa che il secondo set contiene 10000 documenti su 240 MB ed il terzo set contiene 20000 su 480 MB di memoria su disco.

In tabella 8.1 è possibile vedere il numero di tuple generate a partire da ogni data set.

data set	**documenti**	**MB su disco**	**tuple**
1	5000	120	290000
2	10000	240	580000
3	20000	480	1160000

Tabella 8.1: I data set utilizzati nei test

Le query sottoposte al sistema contengono *sempre* vincoli di tipo strutturale, ed almeno uno fra il vincolo testuale e quello temporale (anche entrambi). Sono state tenute presenti le considerazioni a cui si è giunti in [2], e sono state create 5 analoghe interrogazioni (da Q1 a Q5):

1. Q1 e Q2 contenenti solo vincoli testuali; all'interno di queste query non sono espressi vincoli di tipo temporale (è richiesta solo la consistenza delle soluzioni, si veda il paragrafo 6.3.3)

2. Q3 contenente solo vincoli temporali; all'interno di tale query non sono espressi vincoli di tipo testuale

3. Q4 e Q5 contenenti sia vincoli temporali che testuali

8.1 Le query

Le prime quattro query rispecchiano fedelmente le prime quattro query affrontate in [2], infatti:

- in presenza di vincoli testuali, l'espressione utilizzata è la stessa: stesso numero di termini (3 per Q1 e 2 per Q2) e stesso operatore (AND logico)
- in presenza di vincoli temporali, le dimensioni interessate e i periodi adottati sono del tutto equivalenti

Q1:

- *vincolo strutturale*:
  legge//comma//text
- *vincolo testuale*:
  contains 'parola1 AND parola2 AND parola3'
- *vincolo temporale*:
  nessuno

Q2:

- *vincolo strutturale*:
  legge//comma//text
- *vincolo testuale*:
  contains 'parola1 AND parola2'
- *vincolo temporale*:
  nessuno

Q3:

- *vincolo strutturale*:
  legge//comma//text
- *vincolo testuale*:
  nessuno

- *vincolo temporale*:
  VALID OVERLAPS PERIOD ('1970-01-01','2000-12-31')
  AND EFFICACY OVERLAPS PERIOD ('1970-01-01','2000-12-31')
  AND TRANSACTION OVERLAPS '1980-01-01'

Q4:

- *vincolo strutturale*:
  legge//comma//text
- *vincolo testuale*:
  contains 'parola1 AND parola2'
- *vincolo temporale*:
  VALID OVERLAPS PERIOD ('1970-01-01','2000-12-31')
  AND EFFICACY OVERLAPS PERIOD ('1970-01-01','2000-12-31')
  AND TRANSACTION OVERLAPS '1980-01-01'

Q5:

- *vincolo strutturale*:
  path 1:legge//titolo//text
  path 2:legge//comma//text
- *vincolo testuale*:
  path 1:contains 'parola1 AND parola2'
  path 2:contains 'parola3 AND parola4'
- *vincolo temporale*:
  VALID OVERLAPS PERIOD ('1970-01-01','2000-12-31')
  AND EFFICACY OVERLAPS PERIOD ('1970-01-01','2000-12-31')
  AND TRANSACTION OVERLAPS '1980-01-01'

In particolare nella query Q5 si hanno due differenti path, e su entrambi si possono esprimere vincoli testuali distinti. I risultati saranno i twig che hanno la medesima tupla root e che soddisfano i vincoli della query.

8.2 I risultati delle interrogazioni

Nel sottoporre le query al sistema oggetto di questa tesi sono stati considerati:

1. il numero dei risultati ottenuti in forma di path (o di twig)
2. il numero di accessi a disco
3. il tempo necessario all'interrogazione dell'inverted index
4. il tempo necessario per reperire le tuple nel DB

Nella tabella 8.2 vi è il numero dei risultati originati dalle 5 query ed il numero degli accessi a disco per ognuna (nel data set minore). In tabella 8.3 vi sono i

	Q1	**Q2**	**Q3**	**Q4**	**Q5**
numero dei risultati ottenuti	302	2018	1459	24	73
numero di accessi a disco	906	6057	7430	2067	4015

Tabella 8.2: Risultati ottenuti con le query sul primo data set

tempi di elaborazione per l'interrogazione dell'inverted index, quelli per reperire le tuple e la loro somma.

	Q1	**Q2**	**Q3**	**Q4**	**Q5**
interrogazione dell'inverted index (in ms)	271	541	-	532	1064
reperimento delle tuple (in ms)	1406	2859	18485	1468	2736
totale (in ms)	**1677**	**3400**	**18485**	**2000**	**3800**

Tabella 8.3: Tempistiche dei risultati ottenuti

Ora si proverà a sottoporre le stesse query al sistema quando abbiamo memorizzato i data set da 10000 e 20000 leggi, e se ne trarranno conclusioni sulla scalabilità; i risultati sono rispettivamente in tabella 8.4 e tabella 8.5. In figura 8.1 vi è una sintesi dei tempi di risoluzione dei tre data set utilizzati nei test del prototipo sviluppato.

	Q1	**Q2**	**Q3**	**Q4**	**Q5**
numero dei risultati ottenuti	584	4079	2859	57	153
numero di accessi a disco	1755	12249	14665	4201	8027
interrogazione dell'inverted index (in ms)	396	813	-	849	1821
reperimento delle tuple (in ms)	1790	4594	36068	2190	5132
tempo totale di elaborazione (in ms)	2186	5407	36068	3039	6953

Tabella 8.4: Tempi, numero risultati ed accessi a disco del secondo data set

	Q1	Q2	Q3	Q4	Q5
numero dei risultati ottenuti	1195	8336	5811	125	343
numero di accessi a disco	3588	25032	29557	8596	16147
interrogazione dell'inverted index (in ms)	531	1260	-	1262	2626
reperimento delle tuple (in ms)	2255	8167	67302	2828	9969
tempo totale di elaborazione (in ms)	2786	9427	67302	4090	12595

Tabella 8.5: Tempi, numero risultati ed accessi a disco del terzo data set

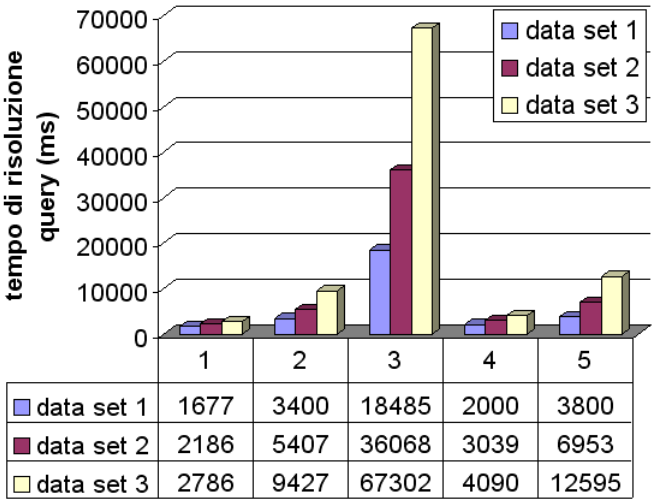

Figura 8.1: Grafico dei tempi di risoluzione delle 5 query

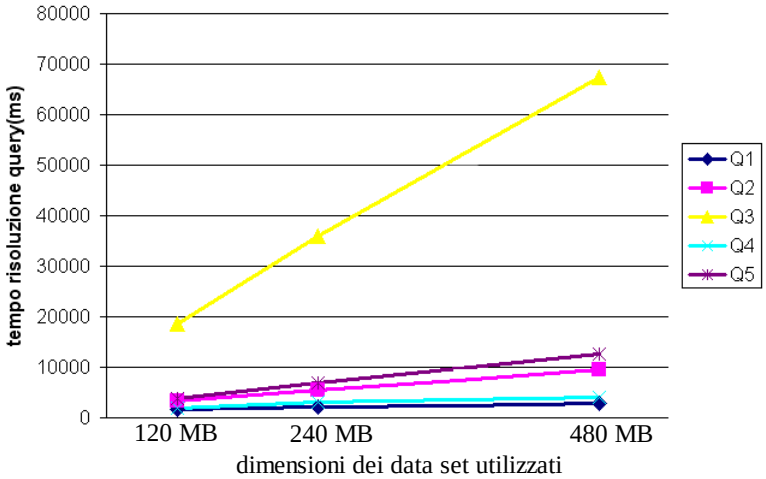

Figura 8.2: Grafico dei tempi di risoluzione in linee

Nella figura 8.2 gli stessi risultati sono presentati sotto forma di linee in modo che si possa vedere l'andamento dei tempi di risposta al crescere delle dimensioni

del data set utilizzato. In particolare è ben visibile che i tempi di risposta alle query giacciono sulla stessa retta anche al crescere delle dimensioni dei data set utilizzati.

Dai grafici e dalle tabelle mostrate è possibile vedere che il prototipo software sviluppato ha ottime prestazioni anche su interrogazioni su grandi set di dati (come il terzo data set). Nella tabella 8.6 è possibile vedere che l'andamento dei tempi di elaborazione cresce in modo sub-lineare al crescere della dimensione dei data set utilizzati. Solo la query Q3 ha una crescita del tempo di elaborazione quasi lineare alla crescita della grandezza del set di dati analizzato.

	Q1	**Q2**	**Q3**	**Q4**	**Q5**
data set 1 (in ms)	1677	3400	18485	2000	3800
data set 2 (in ms)	2186	5407	36068	3039	6953
data set 3 (in ms)	2786	9427	67302	4090	12595

Tabella 8.6: Sintesi dei tempi di elaborazione

Le query che coinvolgono vincoli testuali (Q1, Q2, Q4 e Q5) sono state risolte molto rapidamente grazie alla selettività data dall'espressione valutata; in questo modo si è ridotto drasticamente il numero delle tuple da valutare e quindi da leggere da DB al prezzo del minimo tempo di elaborazione richesto dall'interrogazione dell'inverted index.

La risoluzione della query Q3 significa effettuare una selezione temporale di tutte le tuple che soddisfano i vincoli strutturali; ciò è sicuramente un'operazione molto onerosa ed infatti è la query che ha il tempo di risoluzione maggiore fra quelle proposte.

Il presente prototipo richiede una minima quantità di memoria centrale per l'elaborazione in quanto:

- permette di acquisire di volta in volta le strutture necessarie alla produzione di un risultato finale (un path o un twig)
- non è necessario costruire strutture *accessorie* che concorrono all'elaborazione, come l'albero DOM di ogni documento da ricostruire in [2]

Si è verificata l'occupazione di memoria centrale durante l'elaborazione del prototipo sviluppato in questa tesi e si è visto che rimane praticamente costante per l'intera durata dell'interrogazione e per tutte le interrogazioni. Ciò dimostra che è stato effettuato un uso *attento* della memoria centrale.

8.3 Confronto prestazioni

Nella fase di specifica delle interrogazioni è stata considerata anche la *selettività* delle query in modo da fare un confronto il più possibile paritario. Anche se la granularità minima adottata nei due elaborati è decisamente differente (in TeX è il documento intero, mentre in questo progetto è la tupla) è possibile far corrispondere ad ogni risultato in [2] 10 dei path risultanti nella presente implementazione (ognuno dei quali è composto da almeno tre tuple). Ciò significa che 160 path risultanti si possono far corrispondere a 16 documenti fra i risultati di TeX.

Grazie a questo è possibile calcolare la percentuale dei risultati selezionati all'interno di uno specifico data set. Nelle tabella 8.7 e 8.8 è possibile vedere la percentuale dei risultati sul numero totale dei documenti (del primo data set) relativa al presente prototipo ed al progetto TeX, e questo ci autorizza a fare i dovuti confronti sulle prestazioni. Nella tabella 8.9 vi è una sintesi delle due tabelle precedenti in cui sono visibili le percentuali di mach per ogni query. Si nota che nel presente prototipo la percentuale di match è sempre maggiore a quella ottenuta all'interno del progetto sviluppato in [2].

	Q1	**Q2**	**Q3**	**Q4**	**Q5**
numero dei path risultanti	302	2018	1459	24	73
percentuale dei path risultanti	0,6 %	4,02 %	2,9 %	0,68 %	1,46 %

Tabella 8.7: Risultati in percentuale sul numero totale dei documenti del primo data set nella presente implementazione

	Q1	**Q2**	**Q3**	**Q4**	**Q5**
numero dei documenti risultanti nel progetto [2]	2-10	160	133	2-10	2-10
percentuali dei documenti risultanti nel progetto [2]	0,04-0,2 %	3,2 %	2,66 %	0,04-0,2 %	0,04-0,2 %

Tabella 8.8: Risultati in percentuale sul numero totale dei documenti del primo data set nel progetto sviluppato in [2]

Purtroppo non è stato possibile fare altrettanto con i data set più grandi in quanto non sono stati trattati nel progetto TeX.

All'interno di [2] i risultati finali si avevano solo dopo una fase di selezione (molto veloce, da 1,4 secondi a 4 secondi) ed una di ricostruzione con l'operatore

	match nel presente implementazione , match nel progetto TeX
Q1	**0,6** % $\geq$ 0,2 %
Q2	**4,02** % $\geq$ 3,2 %
Q3	**2,9** % $\geq$ 2,66 %
Q4	**0.68** % $\geq$ 0,2 %
Q5	**1,46** % $\geq$ 0,2 %

Tabella 8.9: Confronto delle percentuali sui match

`const-tree()` (molto lenta, all'incirca un secondo per documento); da questo consegue che i tempi da utilizzare per un confronto sono (in una ipotesi molto migliorativa per il progetto TeX e peggiorativa per questo progetto):

Q1: $2 + 1 \cdot 2 = 4$ secondi

Q2: $2 + 1 \cdot 160 = 162$ secondi

Q3: $2 + 1 \cdot 133 = 135$ secondi

Q4: $2 + 1 \cdot 2 = 4$ secondi

Q5: $2 + 1 \cdot 2 = 4$ secondi

Ora è possibile confrontare i risultati ottenuti (figura 8.3, in scala logaritmica sull'asse verticale). E' possibile vedere che le prestazioni che si hanno nel proto-

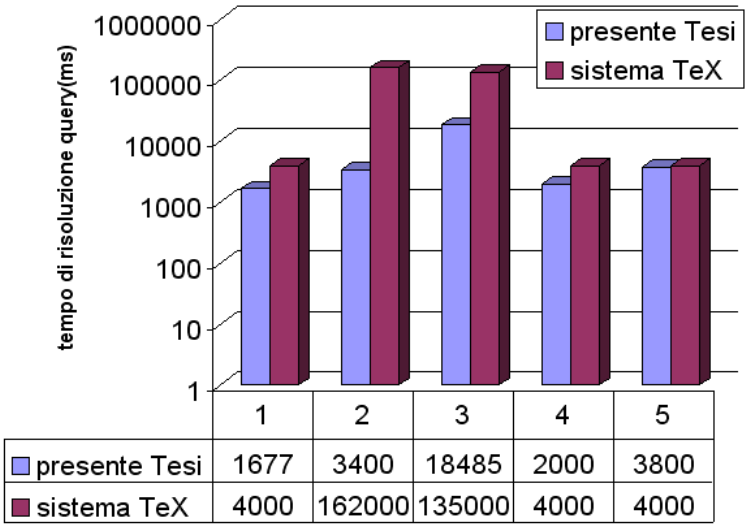

Figura 8.3: Tempi di risoluzione delle query nei due sistemi confrontati

tipo sviluppato in questa tesi sono decisamente migliori di quelle in [2]. Se per alcune query i risultati sono confrontabili, per altre la differenza può essere di più di un ordine di grandezza. Inoltre l'occupazione di memoria richiesta dal presente prototipo è minima rispetto al sistema TeX. Con il prototipo realizzato nella presente tesi è possibile pensare all'implementazione di server che rispondano alle richieste di un gran numero di utenti in quanto l'elaborazione non necessita della costruzione di grandi strutture in memoria principale come succede in [2].

Conclusioni e sviluppi futuri

All'interno di questo lavoro di tesi è stata ideata una nuova architettura chiamata *Temporal XML Query Processor* per l'interrogazione efficiente di documenti XML temporali. Per fare ciò si è partiti dal modello XML temporale adottato e lo si è esteso per fornirne una applicabilità più generale. Su questo modello sono state svolte due operazioni fondamentali:

1. abbiamo esplicitato la temporalità ad ogni livello della gerarchia
2. abbiamo suddiviso ogni documento in strutture adeguate all'utilizzo degli algoritmi noti in letteratura come *structural join*

Da ogni documento XML generiamo un certo numero di strutture, ognuna delle quali comprende:

- un set completo di attributi temporali per una caratterizzazione temporale completa
- un set completo di attributi strutturali per la sua esatta collocazione all'interno del documento XML originale

L'*engine* realizzato offre la possibilità di porre query temporali e strutturali, evitando i cosiddetti approcci a *stratum*. A questo punto interrogare documenti XML temporali significa in realtà interrogare dati che rispettano il modello relazionale.

Facendo riferimento all'ambito normativo analizzato nella presente tesi potremmo dire di usare una *granularità* più fine: infatti abbiamo ideato una scomposizione dell'*atto normativo* in strutture minori che hanno il significato di *disposizioni normative*. Abbiamo chiamato Tuple tali strutture ed abbiamo utilizzato la tecnologia relazionale esistente per verificare i vincoli imposti dalle interrogazioni.

Abbiamo valutato l'efficienza del *Temporal XML Query Processor* ideato su basi di dati contenenti grandi collezioni di testi normativi e possiamo dire di aver ottenuto dei risultati molto interessanti. Sono state sottoposte al prototipo sviluppato vari tipi di interogazioni contenenti vincoli di tipo strutturale, testuale e temporale ed i tempi di elaborazione si sono dimostrati in ogni caso molto contenuti. Dal confronto di prestazioni effettuato con il prototipo in [2], in cui si utilizza un'architettura *"a stratum"*, è emerso che la presente implementazione rappresenta un netto miglioramento per quanto riguarda la risoluzione di query su documenti che soddisfano un modello XML temporale. Sono stati analizzati i dati risultanti da query su entrambi i prototipi in cui:

- il modello XML temporale è lo stesso
- i vincoli imposti sono molto simili
- le percentuali di documenti selezionati sono simili
- i documenti hanno la stessa grandezza media
- il numero dei documenti è lo stesso
- la dimensione su disco dei documenti è la stessa

Dai risultati sperimentali è emerso che l'approccio implementato in questa tesi è sempre più efficiente sia in termini di prestazioni sia in termini di utilizzo della memoria centrale. Se per alcune interrogazioni i tempi di risposta sono comparabili, per altri la differenza può essere anche di più di un ordine di grandezza. Inoltre l'oculato utilizzo della memoria centrale effettuato con questa architettura ci permette di pensare all'utilizzo dell'*engine* sviluppato anche per l'implementazione di *server* che rispondano alle interrogazioni su grandi collezioni di documenti nel modello XML temporale. Un'applicazione specifica potrebbe essere un *web server* per l'interrogazione di una banca dati contenente testi normativi attraverso l'utilizzo di un semplice client (ad esempio un browser), come previsto nel progetto Norma-System (si veda [12]). In particolare il presente prototipo consente una elevata multiutenza del sistema considerando la minima quantità di memoria necessaria alla risoluzione di una singola richiesta. Una simile multiutenza era impensabile con l'approccio adottato in [2].

In fase di test si è visto che l'engine sviluppato permette una elevata scalabiblità: il tempo necessario alla risoluzione di una interrogazione aumenta rispetto

alla crescita delle dimensioni dei data set utilizzati in modo sub-lineare per le interrogazioni contenenti vincoli testuali e lineare per le altre. Inoltre al crescere dei data set utilizzati l'occupazione di memoria centrale per l'elaborazione rimane costante.

Eventuali sviluppi futuri del lavoro fin qui svolto potrebbero essere:

- migliorare le strutture di accesso: il B-TREE sul valore startPos di ogni tupla (che è l'unico indice utilizzato) non permette la verifica di nessuna condizione temporale senza la lettura della tupla dal DB. Costruire indici sugli attributi temporali potrebbe essere utile sia in fase di verifica dei vincoli della query sia in fase di verifica della consistenza. In particolare si pensa ad indici di tipo R-TREE per rappresentare gli *intervalli* delle dimensioni temporali;

- permettere l'esplicitazione di vincoli strutturali più complessi di quelli consentiti all'interno della presente tesi;

- provare l'integrazioni del presente prototipo con altri software del progetto Norma-System, ad esempio per l'inserimento e la modifica dei testi normativi presenti;

- potenziare i metodi di *information retrieval* utilizzati finora. Si potrebbe pensare di permettere l'utilizzo di:

  1. *wildcard* come `?` o `*` per aumentare l'espressività della query;
  2. tecniche *stemming*, per una indicizzazione automatica delle *radici* dei termini;
  3. tecniche di *ranking* delle soluzioni.

Parte III

Appendici

Appendice A

Il codice sviluppato

Considerata la grande mole del codice sviluppato (più di 6000 linee), si riporteranno solo le classi java principali ed il modello adottato in forma di XML Schema.

A.1 XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSPY v5 rel. 4 U (http://www.xmlspy.com) by rth77 (rth77) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
 <xs:simpleType name="valid_time">
  <xs:restriction base="xs:date"/>
 </xs:simpleType>
 <xs:simpleType name="transaction_time">
  <xs:restriction base="xs:date"/>
 </xs:simpleType>
 <xs:simpleType name="event_time">
  <xs:restriction base="xs:date"/>
 </xs:simpleType>
 <xs:simpleType name="ud_time">
  <xs:restriction base="xs:date"/>
 </xs:simpleType>
 <xs:element name="legge">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="titolo"/>
    <xs:element name="articolato">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="ver" maxOccurs="unbounded">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="capo" maxOccurs="unbounded">
           <xs:complexType>
```

```
<xs:sequence>
 <xs:element name="ver" maxOccurs="unbounded">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="rubrica" minOccurs="0"/>
    <xs:element name="articolo" maxOccurs="unbounded">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="ver" maxOccurs="unbounded">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="rubrica" minOccurs="0"/>
          <xs:element name="comma" maxOccurs="unbounded">
           <xs:complexType>
            <xs:sequence>
             <xs:element name="ver" maxOccurs="unbounded">
              <xs:complexType mixed="true">
               <xs:sequence>
                <xs:element name="ta" minOccurs="0"
                maxOccurs="unbounded">
                 <xs:complexType>
                  <xs:attribute name="vt_start" type="valid_time"
                  use="required"/>
                  <xs:attribute name="vt_end" type="valid_time"
                  use="optional"/>
                  <xs:attribute name="tt_start"
                  type="transaction_time" use="required"/>
                  <xs:attribute name="tt_end" type="transaction_time"
                  use="optional"/>
                  <xs:attribute name="et_start" type="ud_time"
                  use="required"/>
                  <xs:attribute name="et_end" type="ud_time"
                  use="optional"/>
                 </xs:complexType>
                </xs:element>
               </xs:sequence>
               <xs:attribute name="num" type="xs:integer"
               use="required"/>
               <xs:attribute name="an_ref" use="optional"/>
              </xs:complexType>
             </xs:element>
            </xs:sequence>
            <xs:attribute name="num" type="xs:integer"
            use="required"/>
           </xs:complexType>
          </xs:element>
          <xs:element name="ta" minOccurs="0" maxOccurs="unbounded">
           <xs:complexType>
            <xs:attribute name="vt_start" type="valid_time"
            use="required"/>
            <xs:attribute name="vt_end" type="valid_time"
            use="optional"/>
            <xs:attribute name="tt_start" type="transaction_time"
```

```
                use="required"/>
                <xs:attribute name="tt_end" type="transaction_time"
                use="optional"/>
                <xs:attribute name="et_start" type="ud_time"
                use="required"/>
                <xs:attribute name="et_end" type="ud_time"
                use="optional"/>
               </xs:complexType>
              </xs:element>
             </xs:sequence>
             <xs:attribute name="num" type="xs:integer" use="required"/>
             <xs:attribute name="an_ref" use="optional"/>
            </xs:complexType>
           </xs:element>
          </xs:sequence>
          <xs:attribute name="num" type="xs:integer" use="required"/>
         </xs:complexType>
        </xs:element>
        <xs:element name="ta" minOccurs="0" maxOccurs="unbounded">
         <xs:complexType>
          <xs:attribute name="vt_start" type="valid_time"
          use="required"/>
          <xs:attribute name="vt_end" type="valid_time"
          use="optional"/>
          <xs:attribute name="tt_start" type="transaction_time"
          use="required"/>
          <xs:attribute name="tt_end" type="transaction_time"
          use="optional"/>
          <xs:attribute name="et_start" type="ud_time" use="required"/>
          <xs:attribute name="et_end" type="ud_time" use="optional"/>
         </xs:complexType>
        </xs:element>
       </xs:sequence>
       <xs:attribute name="num" type="xs:integer" use="required"/>
       <xs:attribute name="an_ref" use="optional"/>
      </xs:complexType>
     </xs:element>
    </xs:sequence>
    <xs:attribute name="num" type="xs:integer" use="required"/>
   </xs:complexType>
  </xs:element>
  <xs:element name="ta" minOccurs="0" maxOccurs="unbounded">
   <xs:complexType>
    <xs:attribute name="vt_start" type="valid_time" use="required"/>
    <xs:attribute name="vt_end" type="valid_time" use="optional"/>
    <xs:attribute name="tt_start" type="transaction_time"
    use="required"/>
    <xs:attribute name="tt_end" type="transaction_time" use="optional"/>
    <xs:attribute name="et_start" type="ud_time" use="required"/>
    <xs:attribute name="et_end" type="ud_time" use="optional"/>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
```

```
          <xs:attribute name="num" type="xs:integer" use="required"/>
          <xs:attribute name="an_ref" use="optional"/>
         </xs:complexType>
        </xs:element>
       </xs:sequence>
       <xs:attribute name="pubblicazione" type="event_time" use="required"/>
       <xs:attribute name="vt_start" type="valid_time" use="required"/>
       <xs:attribute name="vt_end" type="valid_time" use="optional"/>
       <xs:attribute name="tt_start" type="transaction_time" use="required"/>
       <xs:attribute name="tt_end" type="transaction_time" use="optional"/>
       <xs:attribute name="et_start" type="ud_time" use="required"/>
       <xs:attribute name="et_end" type="ud_time" use="optional"/>
      </xs:complexType>
     </xs:element>
    </xs:sequence>
    <xs:attribute name="num" use="required"/>
    <xs:attribute name="natura" use="required"/>
   </xs:complexType>
  </xs:element>
 </xs:schema>
```

A.2 Package Strutture

A.2.1 Classe Strutture.Tupla

```
package Strutture;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

/** Classe che memorizza una tupla, che e' l'output del processo di parsing del
 * documento in XML.
 */
public class Tupla  implements Cloneable,Serializable,Comparable{
    String docId= new String("");
    String type=new String("");
    String text=new String("");
    String nature= new String("");

    //attributi strutturali
    int startPos;
    int endPos;
    int level;


    //attributi temporali
    int pt = 0;
    int vtStart= 0;
```

```
int vtEnd= 0;
int etStart= 0;
int etEnd= 0;
int ttStart= 0;
int ttEnd= 0;

/* indica se ho gi specificato gli elementi temporali
 * 0=no
 * 1=si
 * 2=non servono
 */
int timeSettato=0;

String anRef = new String("");
int num=0;


/** Il costruttore in realta' non setta nessun parametro, che dovranno essere
 * settati con gli opportuni metodi.
 */
public Tupla() {
}
public Tupla(String docId,int startPos,int endPos,int level,int num,
             String type,String nature,String anRef,
             int pt,int vtStart,int vtEnd,int etStart,int etEnd,
             int ttStart,int ttEnd,
             String text){
    this.docId = docId;
    this.startPos = startPos;
    this.endPos = endPos;
    this.level = level;
    this.num = num;

    this.type = type;
    this.nature = nature;
    this.anRef = anRef;

    this.pt =  new MyDate((int)pt).getOffset();
    this.vtStart =  new MyDate((int)vtStart).getOffset();
    this.vtEnd = new MyDate((int)vtEnd).getOffset();
    this.etStart =  new MyDate((int)etStart).getOffset();
    this.etEnd =  new MyDate((int)etEnd).getOffset();
    this.ttStart =  new MyDate((int)ttStart).getOffset();
    this.ttEnd =  new MyDate((int)ttEnd).getOffset();
    this.text = text;
}

public void setDocId(String s) {
    docId=s;
}
public String getDocId(){
    return docId ;
}
```

```
public void setStartPos(int s) {
    startPos=s;
}
public int getStartPos(){
    return startPos;
}

public void setEndPos(int s) {
    endPos=s;
}
public int getEndPos(){
    return endPos;
}

public void setLevel(int s) {
    level=s;
}
public int getLevel(){
    return level;
}
/* questo metodo non delinea il tipo di atto normativo ma il tipo di nodo!!
 */
public void setType (String s) {
    type=s;
}
public String getType(){
    return type;
}

public void setText (String s) {
    text=s;
}
public String getText(){
    return text;
}

public void setNature (String s) {
    nature=s;
}
public String getNature(){
    return nature;
}

public void setPt(String s) {
    pt= new MyDate(s).getOffset();
}
public int getPt() {
    return pt;
}

/** Settaggio dell'inizio del Valid Time */
public void setVtStart(String s){
```

```
        vtStart = new MyDate(s).getOffset();
    }
    public void setVtEnd(String s){
        vtEnd = new MyDate(s).getOffset();
    }

    public int getVtStart(){
        return vtStart;
    }
    public int getVtEnd(){
        return vtEnd;
    }

    public void setEtStart(String s){
        etStart = new MyDate(s).getOffset();
    }
    public void setEtEnd(String s){
        etEnd = new MyDate(s).getOffset();
    }
    public int getEtStart(){
        return etStart;
    }
    public int getEtEnd(){
        return etEnd;
    }

    public void setTtStart(String s){
        ttStart = new MyDate(s).getOffset();
    }
    public void setTtEnd(String s){
        ttEnd = new MyDate(s).getOffset();
    }
    public int getTtStart(){
        return ttStart;
    }
    public int getTtEnd(){
        return ttEnd;
    }

    public void setAnRef(String s){
        anRef=s;
    }
    public String getAnRef(){
        if (anRef.equals(""))
            return new String("---");
        else
            return anRef;
    }
    public void setNum(int n){
        num=n;
    }
    public int getNum(){
        return num;
```

```
}

public void setTime(int i){
    timeSettato=i;
}
public int getTime(){
    return timeSettato;
}

/** Metodo che consente di settare in una volta sola tutti i campi Data nella
 * maniera opportuna: quelli che ci sono verranno settati con la corrispondente
 * data, mentre quelli mancanti verranno settati secondo quanto previsto dalla
 * classe Data da me definita.
 */
public void setDateWrapper(DateWrapper a) {
    vtStart=a.getVtStart();
    vtEnd=a.getVtEnd();
    etStart=a.getEtStart();
    etEnd=a.getEtEnd();
    ttStart=a.getTtStart();
    ttEnd=a.getTtEnd();
}

public DateWrapper getDateWrapper (){
    DateWrapper dw = new DateWrapper(
    vtStart,
    vtEnd,
    etStart,
    etEnd,
    ttStart,
    ttEnd
    );

    return dw;
}


public Object clone() {
    Tupla d = null;
    try {
        d = (Tupla)super.clone();

    } catch (CloneNotSupportedException e) {} // Won't happen
    return d;
}

/** Metodo che mi permette di ordinare oggetti di tipo tupla:
 *  1 - startpos
 *  2 - ttStart
 *  3 - vtStart
 *  4 - ttEnd
 *  5 - vtEnd
 */
```

```
    public int compareTo(Object o) {
        Tupla t =(Tupla)o;

        if (this.equals(t))
            return 0;

        //prima startpos
        if (this.getStartPos() < t.getStartPos())
            return -1;
        else
            if (this.getStartPos() > t.getStartPos())
                return 1;

        //poi ttstart
        if (this.getTtStart() < t. getTtStart())
            return -1;
        else
            if (this.getTtStart() > t. getTtStart())
                return 1;

        //poi ttend
        if (this.getTtEnd() < t. getTtEnd())
            return -1;
        else
            if (this.getTtEnd() > t. getTtEnd())
                return 1;

        //poi vtstart
        if (this.getVtStart() < t. getVtStart())
            return -1;
        else
            if (this.getVtStart() > t. getVtStart())
                return 1;

        //poi vtend
        if (this.getVtEnd() < t. getVtEnd())
            return -1;
        else
            if (this.getVtEnd() > t. getVtEnd())
                return 1;

        return 0;
    }
}
```

A.2.2 Classe Strutture.Query

```
package Strutture;

import GetNext.Nodo;
import GetNext.Twig;
```

```
import Join.TrovaTesto;
import Utility.TieniIlTempo;

public class Query {
    Twig twig;
    DateWrapper dw ;
    TrovaTesto [] tt;
    boolean [] t= new boolean [3];

    /** Creates a new instance of Query */
    public Query(Twig twig,DateWrapper dw,TrovaTesto tt []) {
        this.twig = twig;
        this.dw = dw;
        this.tt = tt;
    }

    public Twig getTwig () {
        return twig;
    }

    public void setDateWrapper(DateWrapper dw ) {
        this.dw = dw;
    }

    public DateWrapper getDateWrapper (){
        return dw;
    }


    public void print (){
        System.out.println("ecco il twig");
        twig.print();
        System.out.println("ecco la temporalit");
        dw.print();
    }



    /** Metodo che controlla quali sono gli intervalli che devo controllare :
     * infatti se
     * nella query abbiamo per un qualsiasi tipo di intervallo (vt,et,tt)
     * ?tStart = 1800-01-01 e ?tEnd = 9999-12-31
     * significa che ne accettiamo ogni temporalit (infatti tali vincoli non sono
     * considerati stringenti per la nostra applicazione e sono considerati l'origine e
     * la fine dei tempi);
     * per ogni altro intervallo specificato nella query viene effettuato il controllo
     * suu quel tipo di intervallo.
     * Es di temporalit in una query:
     * VtStart = 1800-01-01 e VtEnd = 9999-12-31
     * EtStart = 1900-01-01 e EtEnd = 9999-12-31
     * TtStart = 2000-01-01 e TtEnd = 2003-12-31,
     * in questo caso il controllo verr fatto su et e tt
     */
```

```
public  boolean [] intervalliNecessari(boolean log) {

    if ( (dw.getVtStart() == new MyDate("1800-01-01").getOffset()) &&
         (dw.getVtEnd() == new MyDate("9999-12-31").getOffset()) ) {
             t[0] = false;
    }
    else
        t[0] = true;

    if ( (dw.getEtStart() == new MyDate("1800-01-01").getOffset()) &&
         (dw.getEtEnd() == new MyDate("9999-12-31").getOffset()) ) {
             t[1] = false;
    }
    else
        t[1] = true;

    if ( (dw.getTtStart() == new MyDate("1800-01-01").getOffset()) &&
         (dw.getTtEnd() == new MyDate("9999-12-31").getOffset()) ) {
             t[2] = false;
    }
    else
        t[2] = true;

    if (log) {
        System.out.println("------ECCO GLI INTERVALLI CHE CONTROLLERO'");
        System.out.println("vt :"+t[0]);
        System.out.println("et :"+t[1]);
        System.out.println("tt :"+t[2]);
    }

    return t;
}

public void eseguiQueryTesto(){
    TieniIlTempo tit = new TieniIlTempo();
    tit.setTempo();
    int i;
    for (i=0;i<tt.length;i++) {
        tt[i].eseguiRicerca();
    }
    tit.getTempo("fine ricerca ed ordinamento di tutte le query testuali");
}

public boolean verificaTesto(Tupla t,Nodo text){
    int i;
    boolean ret = false;
    for (i=0;i<tt.length;i++) {
        if (tt[i].getNodo().equals(text))
            ret = tt[i].testoOK(t, text);

        if (ret == true)
            return true;
    }
```

```
//        System.out.println("nessun nodo");
        return false;
    }

    public TrovaTesto [] getTrovaTesto(){
        return tt;
    }

    public int verificaNodo(Tupla t,Nodo text){
        int i;
        boolean ret = false;
        for (i=0;i<tt.length;i++) {
            if (tt[i].getNodo().equals(text))
                return i;
        }

//        System.out.println("nessun nodo");
        return -1;
    }

}
```

A.2.3 Classe Strutture.ListaTuple

```
package Strutture;

import GetNext.Nodo;
import Strutture.Query;
import Strutture.Tupla;
import java.io.PrintStream;
import java.util.ArrayList;

public class ListaTuple extends ArrayList{

    /** Creates a new instance of MyArrayList */
    public ListaTuple() {
        super();
    }

    public Tupla getTupla(int i) {
        Tupla t = (Tupla)this.get(i);
        return t;
    }


    public void printAll(){
        int i;
        for (i=0;i<this.size();i++) {
            Tupla t =this.getTupla(i);
            new TuplaUtility(t).printAll();
```

```
        }
    }

    public void printHTML(PrintStream ps){
        System.setOut(ps);
        int i;
        System.out.println("<p>**********************************************</p>");
        for (i=0;i<this.size();i++) {
            Tupla t =this.getTupla(i);
            System.out.println(new TuplaUtility(t).printHTML());

        }
    }

    public boolean containsType(Tupla t_type ){
        int i;
        for (i=0;i<this.size();i++) {
            Tupla t = this.getTupla(i);
            if (t.getType().equals(t_type.getType())) {
                return true;
            }
        }
        return false;
    }

    /** Metodo che torna un intero che rappresenta l'indice dell'elemento in cui
     * contenuta una tupla che ha lo stesso type di quella passata come argomanto.
     * se non  presente torno -1;
     */
    public int indexOf(Tupla t_type) {
        int i;
        for (i=0;i<this.size();i++) {
            Tupla t = this.getTupla(i);
            if (t.getType().equals(t_type.getType())) {
                return i;
            }
        }
        return -1;
    }

    public int removeNotAncestor(Tupla t){
        int count =0;
        int i;
        for (i=0;i<this.size();i++) {
            boolean anc = new TuplaUtility(this.getTupla(i)).isAncestor(t);

            if (anc == true) ;
            else {
                remove(i);
                i--;
                count++;
            }
```

```
        }
        return count;
    }

    public boolean cercaAncestor(Tupla t,Nodo parent){

        int i;
        for (i=0;i<this.size();i++) {
            Tupla temp = this.getTupla(i);
            if (temp.getType().equals(parent.getNodo())) {
                boolean anc = new TuplaUtility(temp).isAncestor(t);
                return true;
            }
        }
        return false;
    }
}
```

A.3 Package Berkeley

A.3.1 Classe Berkeley.AssociaDb

```
package Berkeley;

import Strutture.DateWrapper;
import Strutture.ListaTuple;
import Strutture.MyDate;
import Utility.PrimaChiave;
import com.sleepycat.db.Db;
import com.sleepycat.db.DbBtreeStat;
import java.io.File;
import java.util.ArrayList;

/** Classe che mi iauta nel maneggiare gli handle ai database, associando ogni volta
 * i db secondari al giusto db primario. Il costruttore crea gli handle, ma poi
 * <CODE>apriTutto</CODE> che fa le associazioni. Quando i db non sono pi
 * necessari, prima dell'uscita del programma bisogna chiudere gli handle dei db
 * con <CODE>chiudiTutto</CODE> altrimenti si hanno problemi (es: non riesco ad
 * aprire il db per un'altra lettura e non riesco nemmeno ad eliminarlo con new
 * File(file).delete() ).
 * Se voglio cancellare i db devo chiamare <CODE>eliminaTutto</CODE> prima di
 * richiamare <CODE>apriTutto</CODE>, ma dopo il costruttore.
 * Una volta creato l'oggetto  facile sisalire ai db che servono attraverso i
 * metodi <CODE>getTableXXX</CODE> a cui si passa un intero che rappresenta il tag
 * a cui fare riferimento. Per ottenere tale indice ci sono dei metodi apposta in
 * DBUtility
 */
public class AssociaDb {
    String  types[] = {"legge","articolato","capo","articolo",
    "comma","text","titolo","rubrica"};
```

```
//directory in cui memorizzare i database
String dir = "C://codice//database//";

String keys [] = new String[8];

String primary [] = new String[8];
String index_startPos [] = new String[8];


Db table_primary [] = new Db[8];
Db table_index_startPos [] = new Db[8];


PrimaChiave chiave [] = new PrimaChiave[8];


/** Creates a new instance of AssociaDb */
public AssociaDb() throws Exception{
    int i;
    for (i=0;i<types.length;i++) {
        keys[i] = dir + types[i] + "_keys.dat";

        primary[i] = dir + types[i] + ".db";
        index_startPos[i] = dir + types[i] + "_startPos.db";


        table_primary[i] = new Db(null, 0);
        table_index_startPos[i] = new Db(null, 0);


        chiave[i] = new PrimaChiave(keys[i]);
    }


}

public void apriTutto() throws Exception{

    int i;
    for (i=0;i<types.length;i++) {
        table_primary[i].setErrorStream(System.err);
        table_primary[i].open(null,primary[i],"namedb_"+primary[i],
        Db.DB_BTREE, Db.DB_CREATE, 0644);

        table_index_startPos[i].setErrorStream(System.err);
        table_index_startPos[i].setFlags(Db.DB_DUP | Db.DB_DUPSORT);
        table_index_startPos[i].open(null,index_startPos[i],"namedb_"
        +index_startPos[i], Db.DB_BTREE, Db.DB_CREATE, 0644);

        SKC secStartPos = new SKC(1);
        table_primary[i].associate(null,table_index_startPos[i],secStartPos, 0);
```

```
        }
    }

    public void eliminaTutto() throws Exception{
        int i;
        for (i=0;i<types.length;i++) {
            new File(primary[i]).delete();
            new File(index_startPos[i]).delete();

            chiave[i].firstSave();
        }


    }

    public void chiudiTutto() throws Exception{
        int i;
        for (i=0;i<8;i++) {
            table_primary[i].close(0);
            table_index_startPos[i] .close(0);

        }

    }

    public Db getTablePrimary(int i){
        return table_primary[i];
    }

    public Db getTableStartPos(int i){
        return table_index_startPos[i];
    }


    public PrimaChiave getChiave(int i){
        return chiave[i];
    }

    public void statistiche() throws Exception{
        System.out.println("\n*****************STATISTICHE**************");
        int i;
        for (i=0;i<8;i++) {
            System.out.println("database : "+table_primary[i].getDatabaseName());
            System.out.println("record :"+(chiave[i].count())+ "\n");

        }
        System.out.println("*******************************************");
    }

    public static void main (String args [] ) throws Exception{

        AssociaDb a = new AssociaDb();
        a.eliminaTutto();
```

```
        a.apriTutto();

        DBUtility dbu = new DBUtility(a);
        dbu.memorizzaDirectory(false);


        a.statistiche();

        a.chiudiTutto();
    }
}
```

A.3.2 Classe Berkeley.DBUtility

```
package Berkeley;

import Lucene.*;
import Sax.FromXMLToRows;
import Strutture.DateWrapper;
import Strutture.ListaTuple;
import Strutture.Tupla;
import Strutture.TuplaUtility;
import Utility.PrimitiveDbt;
import Utility.TieniIlTempo;
import com.sleepycat.db.*;
import java.io.File;
import java.util.ArrayList;
import org.apache.lucene.index.IndexWriter;

public class DBUtility {
    PrimitiveDbt primitiveDbt = new PrimitiveDbt();
    AssociaDb adb;

    ArrayList a = new ArrayList();


/** Costruttore dell'oggetto che permette di richiamare una serie di utility sui
* database. E' necessario passare un oggetto AssociaDb per poter risalire a tutte
* le tabelle e alle varie associazioni.
*/
    public DBUtility(AssociaDb adb) {
        this.adb = adb;

        a.add("legge");
        a.add("articolato");
        a.add("capo");
        a.add("articolo");
        a.add("comma");
        a.add("text");
        a.add("titolo");
        a.add("rubrica");
```

```
}

/** Metodo che ritorno tutti i file *.xml presenti nella directory passata come
 * parametro.
 */
private  ArrayList trattaDirectory(boolean log) {
    String dir = "C://codice//modelli_leggi//leggi_generate";
    File direttorio = new File(dir);
    ArrayList al = new ArrayList();

    if (direttorio.canRead()==true)
        System.out.println("posso leggere da "+dir);
    else {
        System.out.println("non posso leggere nel direttorio:"+dir);
        System.exit(0);
    }

    File [] list=direttorio.listFiles();

    for (int i=0;i<list.length;i++) {
        String url = new String(list[i].toString());

        if (url.endsWith(".xml")==true) {
            // alla lista aggiungo un oggetto file.
            if (log)
                System.out.println("tratto :"+list[i].toString());
            al.add(list[i].toString());
        }
        else
            System.out.println("il file "+url+" non mi interessa");
    }


    return al;
}


public int memorizzaFile2(String file,AssociaDb adb,boolean log,int start)
throws Exception{
    //          TieniIlTempo tit = new TieniIlTempo();
    //          tit.setTempo();


    FromXMLToRows txtr = new FromXMLToRows(file,start +1,false);
    ListaTuple l = txtr.getList();
    if (log)
        System.out.println("FromXMLToRows produce un array lungo "+l.size());


    this.memorizzaDaLista(l, adb,log);


    //          tit.getTempo("memorizzaFile2");
```

```
        start = txtr.getList().getTupla(0).getEndPos();

        return start;
    }

/** Metodo che memorizza nel db <CODE>table</CODE> gli oggetti della lista <CODE>al</CODE>
* , usando come chiave ci che restitutisce <CODE>p</CODE> (oggetto che consulta
* il file delle chiavi).
*/
    private void memorizzaDaLista(ListaTuple l,AssociaDb adb,boolean log)
    throws Exception{

        int ret=-1;


        if (l.size()==0) {
            System.out.println("la lista  vuota");
            return;
        }


        Dbt data = new Dbt();
        Dbt key = new Dbt();

        for (int i=0;i<l.size();i++) {

            Tupla t = l.getTupla(i);

            int index = Tag.getIndex(t.getType());
            /* la chiave viene settata a seconda di quello che c' nel file
             * di PrimaChiave
             */
            int chiave =adb.getChiave(index).getIndex();

            //l' oggetto primitiveDbt  globale
            primitiveDbt.setIntDbt(chiave);
            key=primitiveDbt.getDbt();

            data.setObject(t);


            try {
                if (log)
                    System.out.println("type : "+t.getType()+" ,index: "+index+
                    " ,chiave:" +chiave);

                Db table = adb.getTablePrimary(index);
                ret = table.put(null,key, data, Db.DB_NOOVERWRITE);

                if (ret==Db.DB_KEYEXIST) {
                    System.out.println("la chiave che sto inserendo esiste gi");
                    System.exit(1);
```

```
                }

            }
            catch (DbException dbe){
                System.out.println("***:"+dbe);
            }
        }

    }

    public void memorizzaDirectory(boolean log) throws Exception{
        int start = 0;
        TieniIlTempo tit = new TieniIlTempo();
        tit.setTempo();

        ArrayList list = trattaDirectory(log);
        int i;
        for (i=0;i<list.size();i++) {
            start = memorizzaFile2((String)list.get(i),adb,log,start);
        }

        tit.getTempo("memorizzaDirectory");
    }

    /** Metodo che scrive sullo standard output il contenuto di un db, che ha come
     * chiave un int o una String . (sono ammessi i
     * duplicati)
     */
    public void stampaDb(Db table) throws Exception{
        Dbc iterator = table.cursor(null, 0);
        Dbt key = new Dbt();
        Dbt data = new Dbt();
        int count=0;

        System.out.println("\n****************************************");
        System.out.println("stampo il db :"+table.getDatabaseName());
        System.out.println("preso dal file : "+table.getFileName());
        System.out.println("****************************************");

        while (iterator.get(key, data, Db.DB_NEXT) == 0) {

            if (key.getSize()==4) {
                primitiveDbt.setDbt(key);
                int num=primitiveDbt.getInt();
                System.out.println((count++)+" - key:"+num+".");
            }
            else {
                String s = (String)key.getObject();
                System.out.println((count++)+" - key:"+s+".");
            }

            Tupla t = (Tupla)data.getObject();
            new TuplaUtility(t).printAll();
```

```
        }
        iterator.close();
    }

    public void stampaTuttiDb() throws Exception{
        int i;
        for (i=0;i<8;i++) {
            System.out.println("\n*********************************");
            System.out.println("Database di ***"+Tag.getIndex(i)+"***");
            System.out.println("*********************************");

            stampaDb(adb.getTablePrimary(i));

            stampaDb(adb.getTableStartPos(i));

            //              stampaDb(adb.getTableDocId(i));
        }
    }

/** Metodo che legge un db e memorizza tutte le tuple cos lette in una ListaTuple
* che vienme ritornata.
* @param table tabella da cui sto leggendo le tuple
* @param log flag che indica se voglio o meno l'output su schermo
*/
    public ListaTuple leggiDb(Db table,boolean log) throws Exception{
        Dbc iterator = table.cursor(null, 0);
        ListaTuple l = new ListaTuple();
        Dbt key = new Dbt();
        Dbt data = new Dbt();
        int count=0;

        if (log) {
            System.out.println("\n****************************************");
            System.out.println("leggo le tuple del db :"+table.getDatabaseName());
            System.out.println("preso dal file : "+table.getFileName());
            System.out.println("****************************************");
        }

        while (iterator.get(key, data, Db.DB_NEXT) == 0) {

            Tupla t = new Tupla();
            t = (Tupla)data.getObject();
            l.add(t);
            count ++;
        }
        iterator.close();

        if (log) {
            System.out.println("ho letto "+count+" record dal db");
        }
        return l ;
    }
```

```
/** Metodo che legge un db e memorizza tutte le tuple cos lette in una ListaTuple
* che vienme ritornata. Permette la lettura da una certa chiave intera in poi, per
* migliorare le scansioni !!!
* @param table tabella da cui sto leggendo le tuple
* @param log flag che indica se voglio o meno l'output su schermo
* @param da intero da cui si desidera far partire la scansiosne
* @param dw DataWrapper con cui testare la compatibilit di ogni Tupla
* estratta. Se <CODE>null</CODE> allora non viene testata la temporalit ma solo la
* strutturalit
*/
    public ListaTuple leggiDbTime_da(Db table,int da,DateWrapper dw,boolean log)
    throws Exception{
        Dbc iterator = table.cursor(null, 0);
        ListaTuple l = new ListaTuple();
        Dbt key = new Dbt();
        Dbt data = new Dbt();
        int ret =-1;
        int count=0;

        if (log) {
            System.out.println("\n*************************************");
            System.out.println("leggo le tuple del db :"+table.getDatabaseName());
            System.out.println("preso dal file : "+table.getFileName());
            System.out.println("*************************************");

            if (da != 0) {
                System.out.println("*************************************");
                System.out.println("valore minimo di startPos: "+da);
                System.out.println("*************************************");
            }

            if (dw !=null) {
                System.out.println("*************************************");
                System.out.println("verifico compatibilit temporale con ");
                dw.print();
            }

            System.out.println("*************************************");
        }

        //l' oggetto primitiveDbt  globale
        primitiveDbt.setIntDbt(da);
        key=primitiveDbt.getDbt();

        ret = iterator.get(key, data, Db.DB_SET_RANGE);
        if (ret==Db.DB_NOTFOUND) {
            System.out.println("\nATTENZIONE***************************");
            System.out.println("non ci sono chiavi maggiori o uguali a "+da);
            System.exit(1);
        }

        if (ret==0) {
            Tupla t = new Tupla();
```

```
            t = (Tupla)data.getObject();

            if ( (dw == null) ||
            (new TuplaUtility(t).compatibileCon(dw) == true) ){
                l.add(t);
                count ++;
            }
        }


        while (iterator.get(key, data, Db.DB_NEXT) == 0) {

            Tupla t = new Tupla();
            t = (Tupla)data.getObject();

            if ( (dw == null) ||
            (new TuplaUtility(t).compatibileCon(dw) == true) ){
                l.add(t);
                count ++;
            }
        }
        iterator.close();

        if (log) {
            System.out.println("ho letto "+count+" record dal db");
        }
        return l ;
    }

/** Metodo che legge su un db a partire da un certo valore di chiave intera. Poi
* prosegue leggendo tutte le tuple che hanno lo stesso valore della prima chiave
* trovata (duplicati). Ogni tupla letta (sia la prima che le successive) vengono
* controllate con l'elemento temporale della query (<CODE>DateWrapper dw</CODE>)
* e poi se c' compatibilit vengono messe nella ListaTuple che viene ritornata in
* output. Se la ListaTuple ritornata  nulla vuol dire che nel db non ho nessun
* valore >= del valore intero indicato nel parametro <CODE>da</CODE>, se invece ho
* una ListaTuple non nulla ma vuota vuol dire che ho trovato degli elementi ma che
* non soddisfacevano le condizioni temporali imposte.
* @param table tabella da cui sto leggendo le tuple
* @param log flag che indica se voglio o meno l'output su schermo
* @param da intero da cui si desidera far partire la scansiosne
* @param dw DataWrapper con cui testare la compatibilit di ogni Tupla
* estratta. Se <CODE>null</CODE> allora non viene testata la temporalit ma solo la
* strutturalit
* @return ritorno una ListaTuple contenente gli elementi trovati; se vuota vuol dire che
* ne ho trovati ma che non soddisfacevano <CODE>dw</CODE>, se == null vuol dire
* che <CODE>da</CODE> un intero troppo grande per questo DB.
*/
    public ListaTuple leggiDbDupTime_da(Db table,int da,DateWrapper dw,
    boolean log) throws Exception{

        //          TieniIlTempo tit = new TieniIlTempo();
        //          tit.setTempo();
```

```
Dbc iterator = table.cursor(null, 0);
ListaTuple l = new ListaTuple();
Dbt key = new Dbt();
Dbt data = new Dbt();
int ret =-1;
int count=0;

//l' oggetto primitiveDbt  globale
primitiveDbt.setIntDbt(da);
key=primitiveDbt.getDbt();

ret = iterator.get(key, data, Db.DB_SET_RANGE);
if (ret==Db.DB_NOTFOUND) {
    iterator.close();
    return null;
}

if (ret==0) {
    Tupla t = new Tupla();
    t = (Tupla)data.getObject();

    if ( (dw == null) ||
    (new TuplaUtility(t).compatibileCon(dw) == true) ){
        l.add(t);
        count ++;
    }
}


while (iterator.get(key, data, Db.DB_NEXT_DUP) == 0) {

    Tupla t = new Tupla();
    t = (Tupla)data.getObject();

    if ( (dw == null) ||
    (new TuplaUtility(t).compatibileCon(dw) == true) ){
        l.add(t);
        count ++;
    }
}
// inizio pezzo nuovo
while (l.size() == 0 ) {
    ret = iterator.get(key, data, Db.DB_NEXT_NODUP);

    if (ret==Db.DB_NOTFOUND) {
        iterator.close();
        return null;
    }

    if (ret==0) {
        Tupla t = new Tupla();
        t = (Tupla)data.getObject();
```

```
            if ( (dw == null) ||
            (new TuplaUtility(t).compatibileCon(dw) == true) ){
                l.add(t);
                count ++;
            }
        }


        while (iterator.get(key, data, Db.DB_NEXT_DUP) == 0) {

            Tupla t = new Tupla();
            t = (Tupla)data.getObject();

            if ( (dw == null) ||
            (new TuplaUtility(t).compatibileCon(dw) == true) ){
                l.add(t);
                count ++;
            }
        }
    }
    //fine pezzo nuovo

    iterator.close();

    if (log) {
        System.out.println("ho letto "+count+" record dal db");
    }

    //          tit.getTempo("fine lettura  unitaria:leggiDbDupTime_da ");


    return l ;
}

public void creaInvertedIndex(IndexWriter writer) throws Exception{
    Db table = adb.getTablePrimary(Tag.getIndex("text"));
    Dbc iterator = table.cursor(null, 0);

    Dbt key = new Dbt();
    Dbt data = new Dbt();
    int count=0;

    while (iterator.get(key, data, Db.DB_NEXT) == 0) {

        writer.addDocument(MyDocument.Document(data));
        count ++;
    }
    iterator.close();
    System.out.println("elementi text letti:  "+count);
}

public ListaTuple cercaRecordPrecedente2(Db table,int startPos,
DateWrapper dw) throws Exception{
```

```
Dbc iterator = table.cursor(null, 0);

Dbt key = new Dbt();
Dbt data = new Dbt();
int count=0,ret;
//l' oggetto primitiveDbt  globale
primitiveDbt.setIntDbt(startPos);
key=primitiveDbt.getDbt();

ret = iterator.get(key, data, Db.DB_SET_RANGE);

ret = iterator.get(key, data, Db.DB_PREV);

Tupla t = (Tupla)data.getObject();
//valore di partenza perla ricerca
int a = t.getStartPos();
//        System.out.println("prima di "+startPos+" ho trovato "+a);



//l' oggetto primitiveDbt  globale
primitiveDbt.setIntDbt(a);
key=primitiveDbt.getDbt();


ListaTuple l = new ListaTuple();

ret = iterator.get(key, data, Db.DB_SET_RANGE);
if (ret==Db.DB_NOTFOUND) {
                iterator.close();
               return null;
}

if (ret==0) {
    t = (Tupla)data.getObject();

    if ( (dw == null) ||
    (new TuplaUtility(t).compatibileCon(dw) == true) ){
        l.add(t);
        count ++;
    }
}


while (iterator.get(key, data, Db.DB_NEXT_DUP) == 0) {

    t = new Tupla();
    t = (Tupla)data.getObject();

    if ( (dw == null) ||
    (new TuplaUtility(t).compatibileCon(dw) == true) ){
        l.add(t);
        count ++;
```

```
            }
        }



        iterator.close();
        return l;
    }

    public Tupla leggiTuplaText (int startPos) throws Exception{
        Db table = adb.getTableStartPos(Tag.getIndex("text"));
        Dbc iterator = table.cursor(null, 0);

        Dbt key = new Dbt();
        Dbt data = new Dbt();
        int count=0,ret;
        //l' oggetto primitiveDbt  globale
        primitiveDbt.setIntDbt(startPos);
        key=primitiveDbt.getDbt();

        ret = iterator.get(key, data, Db.DB_SET);
        Tupla t = (Tupla)data.getObject();

        iterator.close();
        return t;
    }
}
```

A.3.3 Classe Berkeley.SKC

```
package Berkeley;

import Strutture.*;
import Utility.PrimitiveDbt;
import com.sleepycat.db.*;
import java.io.IOException;

/** Classe che permette il reperimento dell'informazione necessaria
 *alla costruzione
 * di un indice secondario su una tabella.
 */
public class SKC implements DbSecondaryKeyCreate{
    int function;
    PrimitiveDbt primitiveDbt = new PrimitiveDbt();

    /** 1- campo startpos da una Tupla
     * 2- campo docId da una Tupla
     */
    public SKC (int f){
        function=f;
    }
```

```
    public int secondaryKeyCreate (Db secondary, Dbt key, Dbt data, Dbt result)
    throws DbException {
        Tupla t= null;

        String secondaryKey = null;
        int secondaryKeyint =0;

        Dbt tdbt = new Dbt();
        tdbt.setData(data.getData());
        tdbt.setSize(data.getSize());

        Dbt s=new Dbt();

        try {
            switch (function) {
                case 1:
                    t=(Tupla)tdbt.getObject();
                    secondaryKeyint= t.getStartPos();
                    primitiveDbt.setIntDbt(secondaryKeyint);
                    s=primitiveDbt.getDbt();
                    break;

                case 2:
                    t = (Tupla)tdbt.getObject();
                    secondaryKey= t.getDocId();
                    s.setObject(secondaryKey);
                    break;


                default :
                    System.out.println("erroneo cast per la chiave secondaria;
                    function="+function);
                    System.exit(0);
                    break;
            }
        }catch (Exception ioe) {
            System.out.println("SKC:Exception: "+ioe);
        }

            result.setData(s.getData());
            result.setSize(s.getSize());

            return 0;
    }
}
```

A.4 Package Lucene

A.4.1 Classe Lucene.CreateInvertedIndex

```
package Lucene;

import Berkeley.*;
import Sax.FromXMLToRows;
import Strutture.*;
import Utility.TieniIlTempo;
import com.sleepycat.db.Dbt;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.index.IndexWriter;

public class CreateInvertedIndex {



    /** Creates a new instance of CreateInvertedIndex */
    public CreateInvertedIndex() {

    }

    private void indexDocs(IndexWriter writer,Dbt data) {
        try{
            writer.addDocument(MyDocument.Document(data));
        } catch (Exception e) {
            System.out.println(" caught a " + e.getClass() +
            "\n with message: " + e.getMessage());
        }
    }


    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) throws Exception{

        //cancello quello che c'era prima e ne faccio uno nuovo vuoto
        IndexWriter writer = new IndexWriter("C://codice//invertedIndex",
        new StandardAnalyzer(new StopWords().getStopWords()), true);
        writer.close();

        IndexWriter writer2 = new IndexWriter("C://codice//invertedIndex",
        new StandardAnalyzer(new StopWords().getStopWords()), true);


        AssociaDb a = new AssociaDb();
        a.apriTutto();

        DBUtility dbu = new DBUtility(a);
        System.out.println("inizio a scrivere l'indice");
        TieniIlTempo tit = new TieniIlTempo();
```

```
            tit.setTempo();
            dbu.creaInvertedIndex(writer2);
            tit.getTempo("creazione dell'indice:");
            tit.setTempo();
            writer2.optimize();
            tit.getTempo("ottimizzazione dell'indice:");

            a.chiudiTutto();
            writer2.close();
        }

    }
```

A.4.2 Classe Lucene.MyDocument

```
package Lucene;

import Strutture.Tupla;
import Utility.PrimitiveDbt;
import com.sleepycat.db.Dbt;

import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.document.DateField;

public class MyDocument {

    public static Document Document(Dbt data) {

        // make a new, empty document
        Document doc = new Document();


        try {

            Tupla t = (Tupla)data.getObject();


//              doc.add(Field.Keyword("key", new Integer(chiave).toString()));

            doc.add(Field.Keyword("key", new Integer(
            t.getStartPos()).toString()));

            doc.add(Field.Text("contenuto", t.getText()));


        }
        catch (Exception e) {
            System.out.println(" caught a " + e.getClass() +
            "\n with message: " + e.getMessage());
        }
```

```
        // return the document
        return doc;
    }

    private MyDocument() {}
}
```

A.4.3 Classe Lucene.SearchInvertedIndex

```
package Lucene;

import java.io.IOException;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.ArrayList;

import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.search.Searcher;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.Hits;
import org.apache.lucene.queryParser.QueryParser;

public class SearchInvertedIndex {
    ArrayList list = new ArrayList();
    /** Creates a new instance of SearchInvertedIndex */
    public SearchInvertedIndex(String line,boolean log) {

        try {
            Searcher searcher = new IndexSearcher("C://codice//invertedIndex");
            Analyzer analyzer = new StandardAnalyzer(
            new StopWords().getStopWords());


            QueryParser qp = new QueryParser("contenuto",analyzer);
            qp.setOperator(QueryParser.DEFAULT_OPERATOR_AND);
//              qp.parse(line);

            Query query = qp.parse(line);

            System.out.println("Searching for: " + query.toString("contenuto"));

            Hits hits = searcher.search(query);
            System.out.println(hits.length() + " total matching documents");

            int i;
            for (i=0;i<hits.length();i++) {
                Document doc = hits.doc(i);
```

```
                    String id = doc.get("key");
                    String contenuto = doc.get("contenuto");

                    if (log) {
                        System.out.println("\n\nmatch:");
                        System.out.println("key(startPos): "+id);
                        System.out.println("contenuto: "+contenuto);
                    }

                    try {
                        Integer integer = new Integer(id);
                        list.add(integer);
                    }catch (NumberFormatException n) {
                        System.out.println("NumberFormatException :"+n);
                    }
                }


                searcher.close();

            } catch (Exception e) {
                System.out.println(" caught a " + e.getClass() +
                "\n with message: " + e.getMessage());
            }
        }

        public ArrayList getList(){
            return list;
        }
        /**
         * @param args the command line arguments
         */
        public static void main(String[] args) {
            SearchInvertedIndex s = new SearchInvertedIndex("mafia condono",false);
        }

    }
```

A.5 Package GetNext

A.5.1 Classe GetNext.GetNext2

```
package GetNext;

import Berkeley.*;
import GUI.ScrivoQuery;
import Join.TrovaTesto;
import Strutture.*;
import Utility.TieniIlTempo;
import java.util.ArrayList;
```

```
public class GetNext2 {
    Query q;
    Twig twig ;
    ListaTuple [] C ;
    DateWrapper dw ;

    Nodo root,current = null;
    ListaTuple lt[] ;

    /* ArrayList che contiene una collezione di nodi, che rappresentano tutti
     * i nodi della mia query
     */
    ArrayList stream = new ArrayList();

    AssociaDb a;
    DBUtility dbu;

    boolean [] acceso ;
    /*indica lo strartPos da cui inizier la ricerca delle tupel alla prossima
     *chiamata
     */

    int g_startPos = 0;
    Tupla samples [] ;
    int letture = 0;
    int richiamo_aggiornaLT = 0;
    int last_root = 0;
    boolean log = false;

    //per il metodo getTupla
    boolean prima_volta = true;
    ListaTuple buffer = new ListaTuple();
    ListaTuple temp = new ListaTuple();

    /** Creates a new instance of GeneraNodoOnTheFly */
    public GetNext2(Query q, boolean log){
        this.q = q;
        this.log  = log;
        twig =q.getTwig();
        dw = q.getDateWrapper();

        root = twig.getRoot();

        try {
            a = new AssociaDb();
            a.apriTutto();

            dbu = new DBUtility(a);
        }catch (Exception e ) {
            System.out.println("Exception : " + e.toString());
        }
```

```
        stream = twig.getStreams() ;

        lt = new ListaTuple[twig.getNumeroStream()];
        acceso = new boolean [twig.getNumeroStream()];
        samples = new Tupla[twig.getNumeroStream()];


        int i;
        for (i=0;i<twig.getNumeroStream();i++) {
            /* inizializzo le streutture di cui ho bisogno
             */
            lt[i] = new ListaTuple();
            acceso[i] = false;
            samples [i] = null;

            /* riempio tutti gli stream prima di iniziare
             */
            Nodo temp = new Nodo();
            temp = (Nodo)stream.get(i);


            try {
                lt[i] = dbu.leggiDbDupTime_da(a.getTableStartPos(
                Tag.getIndex(temp.getNodo())),1,dw,false);
            }catch (Exception e ) {
                System.out.println("Exception : " + e.toString());
            }
        }
    }

    public ListaTuple riempiRoot(int startPos) throws Exception{
        int index = twig.getStream(root);

        aggiornaLT(root,startPos);

        samples[index] = lt[index].getTupla(0);

        g_startPos = lt[index].getTupla(0).getStartPos()+1;


        current = root;

        if (log) {
            System.out.println("current:");
            current.print();
        }

        return lt[index];
    }

    public ListaTuple getNext() throws Exception{
        if (log) {
            System.out.println("\n\ngetNext");
```

```
        System.out.println("current:");
        current.print();
    }

    int deep_temp = -1;
    int path_temp = -1;
    int path = -1;
    if (twig.isRoot(current) == false) {
        path = twig.getPath(current);
    }

    int i;
    Nodo temp = new Nodo();
    for (i=0;i<twig.getNumeroStream();i++) {
        temp = (Nodo)stream.get(i);

        if (lt[i]==null) continue;


        if (twig.isRoot(temp) == false) {
            path_temp = twig.getPath(temp);
        }

        deep_temp = twig.getDeep(temp);


        if ( ( deep_temp <= 1 ) || (path == path_temp) ) {
            if (log)
                System.out.println("\n\naggiorno lo stream di temp");

            aggiornaLT(temp,g_startPos);
            acceso[i]= true;
        }
        else
            acceso[i] = false;

    }

    int min = Integer.MAX_VALUE;
    int index = -1;
    for (i=0;i<twig.getNumeroStream();i++) {


        if ( (lt[i]==null) || (acceso[i] == false) )
            continue;

        if (lt[i].getTupla(0).getStartPos()<min) {
            min = lt[i].getTupla(0).getStartPos();
            index = i;
        }
    }

    if (log)
```

```
            System.out.println("index uguale a "+index);

        if (index == -1 ) {
            System.out.println("fine lettura DB:ritorno null e chiudo");
            this.chiudi();
            return null;
        }

        /* aggiorno il tipo di nodo e lo startPos globale
         */
        current = (Nodo)stream.get(index);
        g_startPos = lt[index].getTupla(0).getStartPos()+1;

        /* prima di fare il return controllo che le tuple reperite servano
         * realmente,confrontandole con le ultime ritornate corrispondenti al
         * nodo parent. ci lo posso fare solo per le tuple che non sono del nodo
         * root. Se c' compatibilit faccio l'aggiornamento del sample e ritorno
         * lt[index], altrimenti richiamo getNext() [current e g_startPos li ho
         * aggiornati sopra ].Importante : tutte le volte che ritorno dei nodi
         * root devo aggiornare il sample,perch su root non faccio controlli.
         */
        if ( twig.isRoot(current) == false) {
            Nodo nodo_parent = twig.getParent(current);
            Tupla tupla_parent = samples[twig.getStream(nodo_parent)];

            if (new TuplaUtility(tupla_parent).isAncestor(lt[index].getTupla(0))
            ==true ){
                if (log)
                    System.out.println("compatibilit col sample del parent");
                samples [index] = lt[index].getTupla(0);
                return lt[index];
            }
            else {
                if (log)
                    System.out.println("NON C'E' compatibilit col sample del
                    parent");
                return getNext();
            }
        }
        else{
            //se  root ritorno subito dopo l'aggiornamento del sample
            samples [index] = lt[index].getTupla(0);
            return lt[index];
        }
    }
    /** metodo per reperire le tuple dal db: dentro lt[i] ci sono sempre delle
     * tuple che hanno t.getStartPos() >= startPos passato come parametro,
     * altrimenti faccio un aggiornamento. inoltre gli lt[i] contengono sempre
     * almeno una tupla, e faccio l'aggiornamento finch ci non avviene.
     * se lt[i]==null vuol dire che non ci sono pi tuple con t.getStartPos() >
     * startPOs nel db.
     */
    private void aggiornaLT(Nodo n,int startPos) {
```

```
        if (log) {
            System.out.println("aggiornaLT");
            n.print();
        }

        int index = twig.getStream(n);

        //in teoria non serve,faccio il controllo gi nel ciclo sopra
        if (lt[index] == null)
            return ;

        //lo stream non pu essere vuoto.
        if (lt[index].size() == 0){
            System.out.println("lt[index] vuoto !!!");
            System.exit(0);
        }

        richiamo_aggiornaLT ++;


        try {
            //faccio l'aggiornamento
            if (lt[index] != null)  {
                if ( lt[index].getTupla(0).getStartPos() < startPos ) {
                    lt[index] = dbu.leggiDbDupTime_da(a.getTableStartPos(
                    Tag.getIndex(n.getNodo())),startPos,dw,false);
                    if (log)
                        System.out.println("aggiornamento effettuato");
                    letture ++;
                }
                else
                    if (log) System.out.println("stream gi a posto");
            }
            else
                if (log) System.out.println("stream gi terminato");
        }catch (Exception e ) {
            System.out.println("Exception : " + e.toString());
        }
    }

    public Nodo getNodo(){
        return current;
    }

    public void chiudi() throws Exception{
        a.chiudiTutto();
    }
    /** moetodo da usare alternativamente a riempiRoot() r getNext(), ci pensa
     * lui a richiamare le cose giuste, mi basta sapere che ad ogni chiamata
     * ho la tupla giusta da inserire negli stack finch non arriva null.
     */
    public Tupla getTupla() {
```

```
        try{

            if (temp.size() == 0) {
                if (prima_volta == true ){
                    //                  System.out.println("riempiRoot");
                    buffer = this.riempiRoot(g_startPos);
                    prima_volta = false;
                }
                else {
                    //                  System.out.println("getNext");
                    buffer = this.getNext();
                }

                if (buffer == null)
                    return null;
                else {
                    temp.addAll(buffer);
                }
            }


        }catch (Exception e ) {
            System.out.println("Exception : " + e.toString());
        }

        /* quando proprio non si trova nemmeno una tupla...
         */
        if (temp.size()==0)
            return null;

        //qui buffer  sempre > 0
        Tupla t = temp.getTupla(0);
        temp.remove(0);

        return t;

    }

    public void printAccessi(){
        System.out.println("\n*************GeneraNodoOnTheFly**************");
        System.out.println("ho richiamato aggiornaLT "+richiamo_aggiornaLT+ "
        volte");
        System.out.println("ma ho letto dal db solo "+letture+" volte");
        System.out.println("*******************************************");
    }

}
```

A.5.2 Classe GetNext.GetNextText

```
package GetNext;
```

```
import Berkeley.*;
import GUI.ScrivoQuery;
import GetNext.*;
import Join.TrovaTesto;
import Strutture.*;
import Utility.TieniIlTempo;
import com.sleepycat.db.Db;
import java.util.ArrayList;

public class GetNextText {
    Query q;
    AssociaDb a;
    DBUtility dbu;
    ListaTuple list = new ListaTuple();
    int accessi_disco=0;
    int accessi_risparmiati=0;

    int i;
    Tupla t_current=null;
    Nodo n_current = null;
    Tupla last_root= new Tupla();
    int num_path=-1;
    int max_path = -1;

    TrovaTesto tt [];
    ArrayList l[] ;
    int text_start =0;
    Path paths [];
    /** Creates a new instance of RiempiArray */
    public GetNextText(Query q) {

        this.q = q;
        tt = q.getTrovaTesto();
        paths = q.getTwig().getPaths();

        max_path = q.getTwig().getPaths().length;
        l = new ArrayList[max_path];
        int i;
        for (i=0;i<max_path;i++) {
            l[i] = new ListaTuple();
            l[i].addAll(tt[i].getList());
        }


        try {
            a = new AssociaDb();
            dbu = new DBUtility(a);
            a.apriTutto();
        }catch (Exception e ) {
            System.out.println("Exception : " + e.toString());
        }
    }
```

```
public ListaTuple cerca(int start,Nodo nodo_text,boolean log)
throws Exception{

    num_path = q.getTwig().getPath(nodo_text);
    Nodo current = new Nodo();
    current = nodo_text;
    Tupla text = dbu.leggiTuplaText(start);
    accessi_disco++;

    if (new TuplaUtility(text).compatibileCon(q.getDateWrapper())==false){
        list.clear();
        return list;
    }


    if (log) {
        System.out.println("tupla text");
        new TuplaUtility(text).printAll();
    }

    int rimossi = list.removeNotAncestor(text);
    if (log) {
        System.out.println("ho rimosso "+rimossi+" elementi " );
        System.out.println("dopo la rimozione list continene");
        list.printAll();
    }


    while (current != null) {
        current = q.getTwig().getParent(current);

        if (current == null) {
            break;
        }

        boolean nodo_ok = list.cercaAncestor(text, current);


        if (nodo_ok == false) {

            Db table_current = a.getTableStartPos(Tag.getIndex(
            current.getNodo()));
            ListaTuple temp_list = new ListaTuple();

            temp_list = dbu.cercaRecordPrecedente2(table_current,
            text.getStartPos(),q.getDateWrapper());
            accessi_disco++;

            if (temp_list == null) {
                System.out.println("temp_list  nulla");
                continue;
            }
```

```
                if (temp_list.size() == 0) {
                    list.clear();
                    return list;
                }


                if (new TuplaUtility(temp_list.getTupla(0)).isAncestor(text)
                == true) {
                    list.addAll(temp_list);
                }
                else{
                    list.clear();
                    return list;
                }

            }
            else {
                accessi_risparmiati++;
            }
        }

        java.util.Collections.sort(list);
        list.add(text);


        if (log) {
            this.print();
            System.out.println("ho trovato:");
            list.printAll();
        }

        return list;
    }

    public void print(){
        System.out.println("accessi disco: "+ accessi_disco);
        System.out.println("accessi risparmiati: "+ accessi_risparmiati);
    }

    public void chiudi() throws Exception{
        a.chiudiTutto();
    }

    public Tupla getTupla(){
        Tupla t = new Tupla();

        while (list.size() == 0) {

            int min_index=-1;
            //determino l'intero minore e di consegueza il nodo
            int a;
            int min = Integer.MAX_VALUE;
```

```
                for (a=0;a<max_path;a++) {

                    if (l[a].size()==0) continue;

                    int min_temp = ((Integer)l[a].get(0)).intValue();
//                    System.out.println("in  l["+a+"] ho "+min_temp);

                    if (min_temp<min) {
                        min = min_temp;
                        min_index = a;
                    }
                }
//                System.out.println("min_index = "+min_index);
//                System.out.println("min_ = "+min);

                if (min_index == -1) {
//                    System.out.println("finito");
//                    for (a=0;a<max_path;a++) {
//                        System.out.println("l["+a+"] ha dim : "+l[a].size());
//                    }
                    return null;
                }

                text_start = min;
                l[min_index].remove(0);
                try {
                    list = this.cerca(text_start,tt[min_index].getNodo(),false);
                }catch(Exception e){
                     System.out.println("Exception : " + e.toString());
                }


            }

            t = list.getTupla(0);
            list.remove(0);

            t_current = (Tupla)t.clone();

            if (getNodo().equals(q.getTwig().getRoot())) {

                if ( (last_root.getStartPos() == t.getStartPos()) &&
                     (last_root.getTtStart() == t.getTtStart()) ){
//                    System.out.println("ricorsione*******+++++");
                    return getTupla();
                }
                else{
                    last_root = (Tupla)t.clone();
                }
            }

            return t;
        }
```

```
    public Nodo getNodo(){
        return paths[num_path].getNodo(t_current.getType());
    }


}
```

A.6 Package Join

A.6.1 Classe Join.TwigStack

```
package Join;

import GetNext.*;
import Join.TwigResults;
import Lucene.SearchInvertedIndex;
import Strutture.*;
import Utility.TieniIlTempo;
import java.io.FileWriter;
import java.util.ArrayList;
import java.util.Stack;

/** Classe che esegue il TwigStack (generic) come descritto nell'articolo. Per
 * generare le tuple che di volta in volta possono essere inserite negli stack usa
 * la classe <CODE>GeneraNodo</CODE>, che rappresenta la funzione <CODE>getNext()</CODE>
 * dell'articolo. Al normale sviluppo dell'algoritmo viene aggiunto il controllo su
 * ogni stack con la classe <CODE>GestisciIntervalli</CODE> ceh valuta gli aspetti
 * temporali.
 * @author pippo
 */
public class TwigStack {
    Query q;
    Twig twig ;
    ListaTuple [] C ;
    Stack [] S;
    //    GestisciIntervalli gi [] ;

    ArrayList [] risultati ;
    ArrayList sbagli = new ArrayList();
    int num_risultati_corretti [] ;
    int num_risultati_sbagliati=0;

    int last_root = 0;

//    GetNext tsc ;
    GetNextText tsc;

    boolean [] tempo= new boolean [3];
```

```
//controllo le tuple che entrano negli stack
int tuple_generate=0;
int tuple_push=0;
int tuple_scartate_gi=0;
int tuple_desaparecido=0;
int tuple_testo_sbagliato =0;

/** Creates a new instance of TSNew */
public TwigStack(Query q, String param) {
    TieniIlTempo tit = new TieniIlTempo();
    tit.setTempo();
    int i;

    this.q = q;

    twig = q.getTwig();

    //controllo che intervalli temporali devo controllare a seconda della query
    tempo = q.intervalliNecessari(false);

    tsc = new RiempiArray(q);

    // genero le strutture che mi servono per questa query


    //genero gli stack
    int numStack = twig.getNumStack();
    S = new Stack[numStack];
    for (i=0;i<numStack;i++) {
        S[i] = new Stack();
    }


    //genero gli ArrayList per i risultati (uno per ogni Path della query)
    int numPaths = twig.getPaths().length;
    risultati = new ArrayList[numPaths];
    num_risultati_corretti = new int [numPaths];
    for (i=0;i<numPaths;i++) {
        risultati[i] = new ArrayList();
        num_risultati_corretti[i] = 0;
    }

    /* ad ogni ciclo genero una Tupla ed un Nodo che controllo prima dell'
     * inserimento sugli stack
     */
    Tupla tupla = new Tupla();
    Nodo nodo = new Nodo();

    while (tupla != null) {

        tupla = tsc.getTupla();
```

```
            nodo = tsc.getNodo();
            /* inizia la parte dell'algoritmo che usa gli stack
             */

            if (tupla == null)
                break;

            tuple_generate++;


            //meccanismo per sapere quando cambia la root
            if (nodo.equals(twig.getRoot())) {
                if (tupla.getStartPos()>last_root) {
                    last_root = tupla.getStartPos();
                    nuovaRoot();
                }
            }


            this.TSGeneric(tupla,nodo,false);

             /* fine della parte dell'algoritmo che usa gli stack
              */

        }

        tit.getTempo("fine TSGenericNoGI");

        tsc.print();
        System.out.println("\n***********TSGenericNoGI2***********");
        System.out.println("Riassunto sulle tuple coinvolte");
        System.out.println("tuple generate: "+tuple_generate);
        System.out.println("tuple finite negli stack: "+tuple_push);
        System.out.println("tuple scartate da GestisciIntervalli: "+
        tuple_scartate_gi);
        System.out.println("tuple desaparecido: "+tuple_desaparecido);
        System.out.println("tuple con testo sbagliato: "+tuple_testo_sbagliato);
        System.out.println("");
        for (i=0;i<this.getRisultati().length;i++) {
            System.out.println("risultati esatti (ramo "+i+") : "+
            num_risultati_corretti[i]);
        }

        System.out.println("risultati sbagliati : "+num_risultati_sbagliati);
        System.out.println("**********************************");

    }


    private void TSGeneric(Tupla tupla,Nodo nodo,boolean log) {
        int used = 1;
```

```
        if (log) {
            System.out.println("tupla generata:");
            new TuplaUtility(tupla).printAll();
        }

        if ( twig.isRoot(nodo) == false ){
            this.cleanStackParent(twig.getDeep(nodo)-1, tupla, log );

            if (log)
                System.out.println("la tupla NON un elemento root");
        }

        if ( ( twig.isRoot(nodo) == true ) ||
        (!S[twig.getDeep(nodo)-1].empty()) ){

            if (log)
                System.out.println("o la tupla un elemento root o lo
                stack parent NON vuoto");



            this.cleanStackSelf(twig.getDeep(nodo), tupla, log );

            if (log)
                System.out.println("faccio push sulla tupla");

            used = this.push(tupla,nodo, false);

            if (twig.isLeaf(nodo)==true) {

                //cerco ricorsivamente le soluzioni
                if (S[twig.getDeep(nodo)].size() ==1) {
                    if (log)
                        System.out.println("la tupla FOGLIA, cerco le
                        soluzioni");

                    this.showSolution( twig.getDeep(nodo),0,log,null);
                    //tolgo la foglia che ho messo nello stack
                    S[twig.getDeep(nodo)].remove(0);
                }

            }

        }

        if (used==1){
            tuple_desaparecido++;
        }
    }

    private void showSolution(int SN,int SP,boolean log,ListaTuple out) {
        if (out==null) {
            out = new ListaTuple();
```

```
        }

        if (SN==0){
            if (log)
                System.out.println("showSolution: ho trovato root:SN="+SN+"
                SP="+SP);

            TSStruct t_root=(TSStruct)S[SN].get(SP);

            out.add(0,t_root.getTupla());

            /* prima di aggiungere il rilusltato controllo che ci sia intersezione
             * fra tutte le tuple
             */
            boolean intersectAll = new ListaRisultati(out,q).intersectAll();
            if (intersectAll == true) {
                int fit = twig.fit(out);
                if (fit==-1) {
                    System.exit(0);
                }
                risultati[fit].add(out.clone());
            }
            else {
                sbagli.add(out.clone());
            }
        }
        else {
            if (log)
                System.out.println("showSolution: ricorsione normale:SN="+SN+"
                SP="+SP);

            TSStruct tss = (TSStruct)S[SN].get(SP);

            out.add(0,tss.getTupla());

            /* prima di ogni successiva chiamata ricorsiva a showSolution ripristino
             * lo stato del vettore out a prima della ricorsione.
             */

            ListaTuple temp = new ListaTuple();
            temp.addAll(out);
            for (int i=tss.getHigh();i>=0;i--) {
                showSolution(SN-1,i,log,out);
                out.clear();
                out.addAll(temp);
            }
        }
    }


    private int cleanStackParent(int stack,Tupla q,boolean log) {
        if (log)
            System.out.println("\n--- INIZIO : cleanStackParent(int stack,
```

```
            Tupla q,boolean log)");
        int count =0;

        if (stack<0) {
            System.out.println("stack="+stack);
            return -1;
        }

        if (S[stack].empty() == true)
            return 0;

        int i;
        for (i=0;i<S[stack].size();i++) {
            TSStruct temp = (TSStruct)S[stack].get(i);
            Tupla t_temp = temp.getTupla();

            if (new TuplaUtility(t_temp).isAncestor(q) == false) {

                S[stack].remove(i);
                count++;
                i=-1;
            }
        }

        if (log) {
            System.out.println("cleanStackParent  :ho eliminato "+count+"
            oggetti TSStruct dallo stack "+stack);
            System.out.println("--- FINE : cleanStackParent(int stack,Tupla q,
            boolean log)\n");
        }

        //          this.aggiornaGestioneIntervalli(true);

        return count ;

    }

    private int cleanStackSelf(int stack,Tupla q,boolean log) {
        if (log)
            System.out.println("\n--- INIZIO : cleanStackSelf(int stack,Tupla q,
            boolean log)");
        int count =0;

        if (stack<0) {
            System.out.println("stack="+stack);
            return -1;
        }

        if (S[stack].empty() == true) {
            if (log) {
                System.out.println("cleanStackSelf  :ho eliminato "+count+"
                oggetti TSStruct dallo stack "+stack);
                System.out.println("--- FINE : cleanStackSelf(int stack,Tupla q,
```

```
            boolean log)\n");
        }
        return 0;
    }

    int i;
    for (i=0;i<S[stack].size();i++) {
        TSStruct temp = (TSStruct)S[stack].get(i);
        Tupla t_temp = temp.getTupla();

        if (new TuplaUtility(t_temp).isAncestorOrTwin(q) == false) {
            S[stack].remove(i);
            count++;
            i=-1;
        }
    }

    if (log) {
        System.out.println("cleanStackSelf  :ho eliminato "+count+" oggetti
        TSStruct dallo stack "+stack);
        System.out.println("--- FINE : cleanStackSelf(int stack,Tupla q,
        boolean log)\n");
    }

    //          this.aggiornaGestioneIntervalli(true);

    return count ;
}

/** Mette una struttura del tipo TSStruct sullo stack appropriato */
private int push(Tupla t,Nodo n, boolean log){
    if (log)
        System.out.println("\n--- INIZIO: push(Tupla t, boolean log)");




    int index=twig.getDeep(n);
    TSStruct tss = this.creaTSStruct(t,n);

    if (log) {
        System.out.println("\naggiungo la tupla :");
        tss.print();
        System.out.println("--- FINE: push(Tupla t, boolean log)\n");

    }

    S[index].add(tss);
    tuple_push++;

    //          this.aggiornaGestioneIntervalli(true);
    return 0;
}
```

```
/** Crea una Tsstruct in base alla tupla che gli passo: non gli serve nessun altro
 * parametro perch sfrutta (come altri meotdi) alcune delel variabili globali
 * all'interno della classe.
 */
private TSStruct creaTSStruct(Tupla t,Nodo n) {
    int index=twig.getDeep(n);
    int index_parent=index-1;
    int high =-1;
    if (index_parent>-1) {
        high=S[index_parent].size()-1;
    }

    TSStruct tss =new TSStruct(t,index_parent,high);
    return tss;
}

public ArrayList [] getRisultati(){
    return risultati;
}

public ArrayList getSbagli(){
    return sbagli;
}

/*metodi per il controllo dei tempi prima dell'inserimento delle tuple degli
 stack
 */
public void aggiornaGestioneIntervalli(boolean log){
    int i;
    for (i=0;i<twig.getNumStack();i++) {

        //              if (i==0)
        //                  gi[i] = new GestisciIntervalli(q,S[i],i,null);
        //              else
        //                  gi[i] = new GestisciIntervalli(q,S[i],i,gi[i-1]);
        //
        //              gi[i].aggiornaVettori();
    }
}



public void nuovaRoot(){
    //          System.out.println(" cambiata la root!(nuovo startPos di root
    "+last_root+" )");

    TwigResults t_result = new TwigResults(risultati,sbagli,false);

            t_result.printHTML();

    int i;
    for (i=0;i<risultati.length;i++) {
```

```
            num_risultati_corretti[i] += risultati[i].size();
            risultati[i].clear();
        }
        num_risultati_sbagliati += sbagli.size();
        sbagli.clear();
    }

    public static void main(String args[]) {
        ScrivoQuery cq = new ScrivoQuery(args[0],args[1],args[2]);
//        ScrivoQuery cq = new ScrivoQuery("si","uno","due");

        TSGenericNoGI2 tsg = new TSGenericNoGI2(cq.getQuery(),args[0]);
    }
}
```

Bibliografia

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural Join: A Primitive for Efficient XML Query Matching. In *Proceedings of the IEEE International Conference on Data Enegeneering*, 2002.

[2] M. Bergonzini. TeX System: un sistema per la gestione temporale di testi normativi. Tesi di laurea, Università degli studi di Modena e Reggio Emilia, 2003.

[3] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *ACM SIGMOD*, 2002.

[4] S. Chien, Z. Vagena, D. Zhang, v. J. Tsottras, and C. Zaniolo. Efficient Structural Join on Indexed Documents. In *Proceeding of the 28th VLDB Conference*, 2002.

[5] D. Gao, S. Jensen, R. T. Snodgrass, and M. D. Soo. Join Operations in Temporal Databases. Technical report, A TimeCentre Report, 2002.

[6] D. Gao and R. T. Snodgrass. Temporal Slicing in the Evaluation of XML Queries. In *Proceedings of the 29th VLDB Conference*, 2003.

[7] F. Grandi, F. Mandreoli, M. Bergonzini, and P. Tiberio. A Temporal data model and Managmente System for Normative Texts in XML Format. In *ACM*, 2003.

[8] F. Grandi, F. Mandreoli, and P. Tiberio. Temporal Modelling and Management of Normative Documents in XML Format. In *Elsevier Science*, 2004.

[9] H. Jiang, H. Lu, and B. C. Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In *ACM SIGMOD*, 2002.

[10] H. Jiang, W. Wang, H. Lu, and J.X. Yu. Holistic Twig Joins on Indexed XML Documents. In *Proceedings of the 29th VLDB Conference*, 2003.

[11] B. Oliboni, E. Damiani, E. Quintarelli, and L. Tanca. Temporal Aspects of semistructured data. In *SEBD*, 2001.

[12] M. Palmirani and R. Brighi. Norma-System: A Legal Document System for Managing Consolidated Acts. In *Springer-Verlag Berlin Heidelberg*, 2002.

[13] Progetto Norme In Rete. www.normeinrete.it.

[14] Berkeley DB. www.sleepycat.com.

[15] R. T. Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.

[16] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Managment Systems. In *ACM SIGMOD*, 2001.

[17] D. Zhang, V. J. Tsotras, and B. Seeger. A Comparison of Indexed Temporal Joins. Technical report, A TimeCentre Report, 2000.

[18] D. Zhang, V. J. Tsottras, and B. Seeger. Efficient Temporal Join Pocessing using indices. 2001.