UNIVERSITA' DEGLI STUDI DI MODENA E REGGIO EMILIA

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea

TeX System: UN SISTEMA PER LA GESTIONE TEMPORALE DI TESTI NORMATIVI

Relatore: Chiar.mo Prof. Paolo Tiberio
Correlatori: Dott.ssa Federica Mandreoli
Prof. Fabio Grandi

Laureando: Marco Bergonzini

Anno Accademico 2002/2003

Ringraziamenti

Ringrazio la Dottoressa Federica Mandreoli e l'Ing Riccardo Martoglia per la disponibilità e la costante assistenza fornita.
Desidero inolte ringraziare il Professor Paolo Tiberio e il Professor Fabio Grandi, per l'appoggio offertomi durante lo sviluppo della tesi.

Un grazie particolare alla mia famiglia per avermi sostenuto in tutti questi anni.

Parole chiave

XML
Database Temporali
Testi normativi
Dati semistrutturati
Oracle
Versionamento

Indice

Indice delle figure

C a p i t o l o 1

CONSIDERAZIONI INTRODUTTIVE

L'archiviazione e la gestione dei testi normativi sono da tempo tematiche di grande rilevanza. La costante produzione di leggi e decreti da parte di un gran numero di soggetti spesso non ben coordinati tra loro, unitamente alla mancanza di una precisa metodologia per la stesura dei testi, ha sempre reso di difficile interpretazione le disposizioni normative. In particolare, un problema sempre più sentito, è quello della determinazione della versione applicabile di una disposizione. Col passare del tempo infatti, i testi normativi subiscono modifiche ed aggiornamenti che ne cambiano sia il contenuto che l'ambito di applicabilità. La storia di un testo normativo deve perciò tenere memoria di tutte queste modifiche e delle diverse versioni che da esse scaturiscono.

Gli obiettivi fondamentali della presente tesi sono stati lo studio, la progettazione e infine la realizzazione di un sistema in grado di semplificare l'interpretazione temporale delle disposizioni normative.

La prima fase del progetto TeX è stata la determinazione di un modello temporale, basato sullo standard XML, in grado di rappresentare efficacemente un testo normativo, catturandone la struttura e gli aspetti temporali collegati ad ogni elemento che lo compone, e di definire la semantica di alcuni operatori per la gestione della dinamica e l'accesso delle norme nel tempo. In questo stadio nel nostro lavoro un importante ruolo è stato ricoperto dalle tecniche sviluppate nella

ricerca sui database temporali. Grazie ad esse siamo riusciti ad esprimere i requisiti del nostro modello attraverso formalismi comunemente riconosciuti ed accettati.
Si sono poi prese in considerazione le varie alternative presenti sul mercato, in termini di DBMS e linguaggi di programmazione, allo scopo di realizzare un prototipo che implementasse il modello e gli operatori in modo efficiente, senza però perdere di vista l'esigenza di creare un software portabile, basato il più possibile su protocolli standard e strumenti di comune utilizzo, con un occhio di riguardo per il mondo Web. La scelta dello standard XML per la rappresentazione dei testi normativi è stata perciò una naturale conseguenza. XML, infatti, risponde appieno alle nostre necessità, sia in termini di potenza espressiva che di portabilità e presenza su Internet.
La tesi è organizzata in 7 capitoli e in 5 appendici. Nel capitolo 2 si introducono le nozioni di base sui testi normativi definendone la tipologia e la dinamica. Si procederà a un'introduzione della terminologia per arrivare a una descrizione dei nessi normativi e a una loro distinzione in base agli effetti, in modo da chiarire come le disposizioni normative interagiscono tra loro.
I capitoli da 3 a 5 descrivono in dettaglio gli strumenti che si sono decisi di utilizzare per la realizzazione di modello e prototipo, in particolare:

- Nel capitolo 3 si analizzano le caratteristiche principali dei database temporali. Si introducono le diverse dimensioni temporali e si discutono gli aspetti che è possibile rappresentare grazie ad esse.
- Nel capitolo 4 si descrive lo standard XML come metodo per rappresentare dati semistrutturati e se ne discutono efficacia e potenza espressiva in presenza di dati temporali.
- nel capitolo 5 si discutono le potenzialità di un prodotto commerciale, Oracle 9i, in particolare dei pacchetti XML DB, per l'archiviazione e la

ricerca di documenti XML, e interMedia, che abbiamo utilizzato per costruire indici e per interrogare i testi normativi.

Il capitolo 6 riguarda il progetto vero e proprio del sistema TeX, cominciando con la fase di modellazione e un'analisi delle varie alternative prese in considerazione, per poi proseguire con la progettazione del sistema e delle principali procedure di cui si è deciso di dare una rappresentazione semi-formale attraverso il modello UML (Unified Modeling Language).

Infine, nel capitolo 7 si illustrano i risultati ottenuti e si confrontano le prestazioni delle alternative considerate.

Nelle appendici si riportano le nozioni di contorno e la documentazione aggiuntiva, in particolare:

l'appendice A contiene i listati Java di una selezione di classi e metodi descritti nel capitolo 6;

l'appendice B riporta i file sorgente degli schemi XML e alcuni documenti conformi al modello proposto;

l'appendice C è una breve descrizione dello strumento Inverted Index;

l'appendice D contiene una breve introduzione al linguaggio di programmazione Java facendo particolare attenzione agli strumenti che permettono la connessione a una base di dati Oracle;

l'appendice E contiene i testi di alcune query che sono state eseguite durante la fase di test del prototipo.

C a p i t o l o 2

INTRODUZIONE AI TESTI NORMATIVI

Il presente capitolo contiene tutte le nozioni sui testi normativi[1] essenziali per la comprensione del lavoro svolto.

La prima parte presenterà, in modo molto conciso, il sistema dei testi normativi attraverso alcune stipulazioni terminologiche. Si fornirà una classificazione dei diversi tipi di riferimenti normativi al fine di introdurre un dizionario specialistico, al quale si farà riferimento nel seguito .

E' importante ricordare che uno degli argomenti cruciali nel nostro lavoro è rappresentato dalla gestione della dinamica delle modifiche dei testi normativi. Va sottolineato in particolare, che ogni volta che una disposizione normativa subisce una modifica, esiste sempre un'altra disposizione che provoca tale modifica. Esiste quindi un collegamento tra modificata e modificante, questo collegamento prende il nome di *nesso normativo*.

La seconda parte del capitolo si occuperà di classificare i nessi normativi in base agli effetti che questi hanno sulle norme interessate.

La comprensione di come le disposizioni normative vengono modificate nel tempo e di quali sono gli aspetti temporali delle modifiche rappresenta la base di partenza del nostro lavoro. Quanto segue, pur essendo di carattere essenzialmente nozionistico, è quindi fondamentale per una corretta interpretazione di alcune scelte adottate durante lo sviluppo del progetto TeX.

2.1. Nozioni preliminari

Questo paragrafo elenca una serie di termini in relazione coi testi normativi. Per ognuno di essi viene spiegato il significato, anche attraverso alcuni esempi.

2.1.1. 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 documento normativo, il testo della legge approvato dal parlamento, promulgato il 1 dicembre 1970, e pubblicato sulla gazzetta ufficiale n.306 del 3 dicembre 1970.

Storia del testo normativo. Ogni testo normativo può subire modifiche testuali che ne mutano il contenuto nel tempo, pur mantenendo la propria identità. E' opportuno fare alcune distinzioni:

- *singole versioni* del testo normativo; esse sono le diverse formulazioni che il testo ha assunto nel corso del tempo.
- *storia* del testo normativo, intesa come la sequenza di tutte le versioni.
- *testo normativo virtuale*, il contenitore astratto di tali versioni, esso mantiene la propria identità attraverso le modifiche del testo.

La storia del testo normativo ha inizio con l'entrata in vigore e termine con l'abrogazione.

Testo normativo vigente. In un determinato momento x il *testo normativo vigente* corrisponde a tutte le successive operazioni di modifica testuale operate da atti

normativi successivi, fino al momento x, sul testo normativo originario. A differenza del testo normativo originario, il cui contenuto è fissato e immutabile, il testo normativo vigente si evolve a seguito delle modifiche a cui è soggetto.
Espressioni che designano un testo di legge, come ad esempio questa: "la legge n. 898 del 1970", possono risultare ambigue in quanto possono riferirsi alle varie forme del testo normativo. Tuttavia esse sono frequentemente intese come testo normativo vigente nel "momento attuale", dove con "momento attuale" si indica il presente, o più precisamente, il momento rispetto al quale si deve individuare la versione applicabile.
Dette espressioni fanno quindi riferimento al risultato di tutte le modifiche testuali verificatesi fino al momento dell'applicazione del testo normativo.

2.1.2. Disposizione normativa

Chiamiamo *disposizione normativa* ogni unità testuale autonoma contenuta nel testo normativo, si tratta quindi di ogni unità risultante da una suddivisione del testo in elementi sintatticamente ben formati e di significato compiuto. Il livello più basso di disposizioni normative sono quindi i periodi inclusi nei testi normativi. Sono considerate disposizioni normative anche partizioni del testo a livello superiore, come commi, articoli, ecc. L'intero testo di un provvedimento può quindi essere considerato una disposizione.

Storia di una disposizione normativa. Esattamente come per l'intero testo, anche le singole disposizioni mantengono la propria identità nel tempo, nonostante le modifiche a cui sono soggette. Ogni disposizione infatti è identificata dal proprio nome, che identifica una posizione all'interno del testo normativo. Si ritiene che anche se nel corso del tempo vengano assegnate diversi contenuti testuali alla medesima disposizione, essa mantenga la propria identità, mutando versione.

Come per il testo normativo abbiamo le seguenti distinzioni:

- *versione* di una disposizione normativa. E' lo specifico contenuto testuale che la disposizione è venuta ad assumere in un determinato momento.
- *versione originaria* di una disposizione normativa. Si tratta della versione contenuta nel testo normativo originario, si può chiamare anche *disposizione originaria*.
- *Versione vigente*. In un determinato momento x la versione vigente di una disposizione normativa è il contenuto testuale della disposizione in tale momento.

2.1.3. Ambito di applicabilità

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

L'espressione "contesto normativo presente" sta ad indicare che future modifiche di tale contesto, come ad esempio l'introduzione di nuove limitazioni alla disposizione, o una sua abrogazione, non incidono sull'ambito di applicabilità attuale.

Per "in forza della disposizione stessa" intendiamo che non rientrano nell'ambito di applicabilità le situazioni nelle quali la conseguenza giuridica della disposizione valga in quanto stabilita da norme diverse.

2.1.4. Riferimenti normativi

L'espressione riferimento normativo è notevolmente ambigua e comprende almeno i sensi che andremo ad analizzare nel seguito.

Nesso normativo. Per *nesso normativo* intendiamo ogni possibile rapporto che si stabilisca tre due o più disposizioni normative. Normalmente è possibile individuare una disposizione che ha funzione attiva e un'altra con funzione passiva.

- La disposizione attiva è quella la cui emanazione istituisce o realizza il nesso normativo; ad esempio, è la disposizione abrogante che istituisce il nesso con l'abrogata.
- La disposizione passiva è quella che subisce l'instaurazione del nesso.

La disposizione attiva è normalmente successiva nel tempo rispetto alla disposizione passiva, e può contenere una denominazione esplicita o meno di quella.

Designazione normativa. Per *designazione normativa* intendiamo ogni espressione intesa ad identificare un atto normativo o una partizione di esso, cioè ogni espressione che funga da "nome" di tale atto o partizione, ad esempio: "il primo comma dell'articolo 252 del codice civile". La nozione di designazione va estesa anche alle espressioni intese a denotare un insieme di atti normativi o loro partizioni, ad esempio: "le disposizioni di cui gli articoli 155, 156, 255 del codice civile".

Possiamo distinguere due tecniche di designazione delle disposizioni passive:

- designazione sintattica, o citazione in senso stretto. La *citazione* individua la disposizione passiva mediante le caratteristiche "esteriori" di questa, come l'indicazione del testo normativo e partizione all'interno di esso che la comprendono o l'indicazione di vocaboli che vi compaiono. Sotto un diverso profilo un'ulteriore distinzione può essere fatta a riguardo della designazione sintattica: la *designazione statica* richiama la disposizione passiva nella versione che questa aveva nel momento in cui è stata effettuata

la citazione o in un determinato momento precedente; la designazione dinamica richiama la disposizione passiva nella versione di volta in volta vigente nei diversi momenti rispetto ai quali si deve ricostruire il significato della disposizione richiamante.

- 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. Esempio: "le norme incompatibili con la presente legge".

Designazione sintattica e semantica possono essere combinate tra loro.

Menzione normativa. Chiamiamo *menzioni normative* le porzioni di un testo normativo riportate all'interno di un altro testo normativo. Sono frequenti nelle norme sostitutive o nelle norme interpretative.

Richiamo normativo. Per *richiamo normativo*, o riferimento normativo in senso stretto, intendiamo la funzione linguistica consistente nell'identificare le disposizioni interessate da un nesso normativo. La funzione del richiamo normativo è solitamente svolta dalla disposizione attiva, che assume quindi il ruolo di *richiamante*, al contrario la disposizione passiva funge da *richiamata*.

2.2. I nessi normativi: distinzione in base agli effetti

Nelle pagine seguenti prenderemo in considerazione le tipologie principali di nessi normativi, distinte a seconda dell'impatto del nesso normativo sulle norme interessate.

2.2.1. Modificazioni e rinvii

I nessi normativi possono essere suddivisi in due grandi categorie:

- 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;
- nel *rinvio* invece la disposizione attiva si avvale della disposizione passiva al fine di completare il proprio significato, senza influire su quest'ultima.

Qui di seguito ci occuperemo soltanto delle modificazioni, esse hanno rappresentato un punto fondamentale nello sviluppo del progetto TeX.

2.2.2. Modificazioni totali e parziali.

Un primo criterio di classificazione delle modificazioni consiste nella valutazione dell'ampiezza della modifica apportata.

La modificazione si definisce *totale* quando comporta l'eliminazione dell'intera disposizione passiva, è *parziale* quando invece tale disposizione rimane in vigore, anche se con contenuto testuale parzialmente differente.

Chiaramente, una modificazione si può ritenere totale o parziale a seconda della disposizione prese in considerazione: ad esempio, l'abrogazione di un articolo è totale per l'articolo stesso e parziale per l'atto normativo che lo contiene.

In generale sono valide le seguenti relazioni:

- la modificazione testuale, totale o parziale, di una partizione del testo comporta una modificazione parziale delle partizioni di livello superiore;
- la modificazione totale di una partizione comporta la modificazione totale di tutte le partizioni di livello inferiore;
- la modificazione parziale di una partizione comporta la modificazione, parziale o totale, di almeno una partizione di livello inferiore,

Salvo diverse specificazioni, da questo punto in avanti, parleremo di modificazione totale nei casi in cui viene abrogato o sostituito l'intero atto, e parziale negli altri casi.

2.2.3. Modificazione 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.

2.2.4. 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 aveva al momento t1. Quindi la versione della disposizione passiva vigente al momento t1 è diversa da quella vigente in t2.
La differenza tra le due versioni può consistere in una *abrogazione*, in un inserimento, o in una combinazione di entrambi, che prende il nome di *sostituzione*.
Come intuibile in figura 2.1, le modifiche si configurano come nessi ternari tra le diverse versioni della disposizione passiva e la disposizione attiva: l'entrata in vigore della disposizione attiva produce il passaggio da una versione della disposizione passiva alla versione successiva. Più precisamente i nessi 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 estremi di abrogazione totale, in cui manca la versione successiva, e inserimento totale, in cui manca la versione precedente.

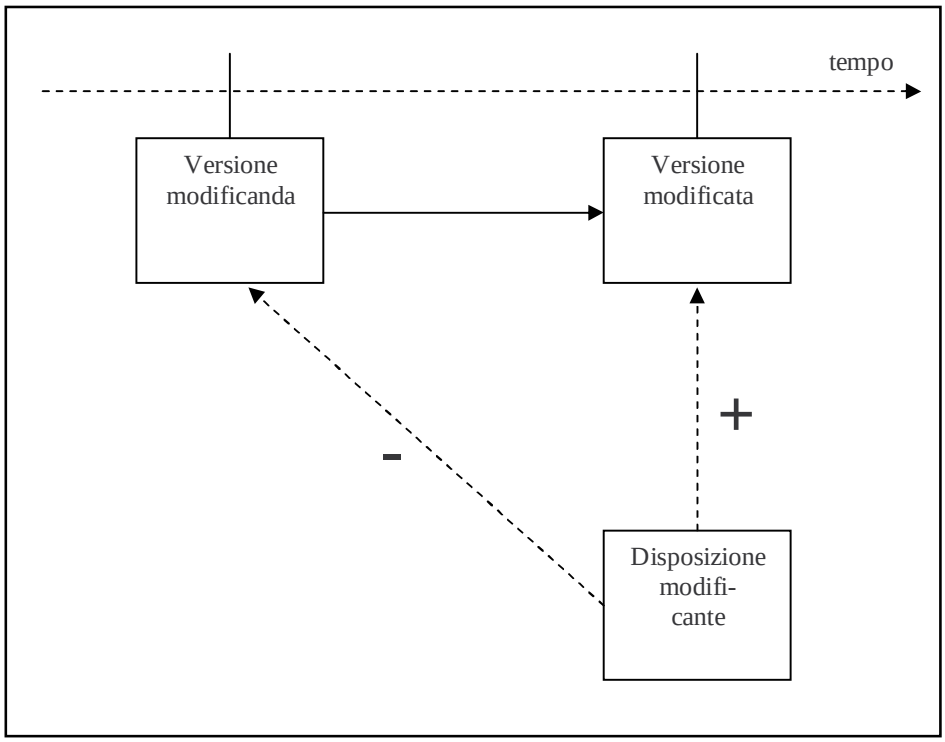


Fig. 2.1 – Le modifiche testuali

Passiamo ora brevemente in rassegna i tipi di modifica testuale.

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: l'inserimento comporta l'introduzione di una nuova sequenza testuale all'interno di un testo normativo.

Sostituzione: la sostituzione comporta l'eliminazione di una sequenza testuale da un testo normativo e l'inserimento di una nuova sequenza al posto di quella tolta. Si tratta quindi di una combinazione delle due modifiche precedenti.
Nella figura 2.2 se ne fornisce anche una rappresentazione grafica.

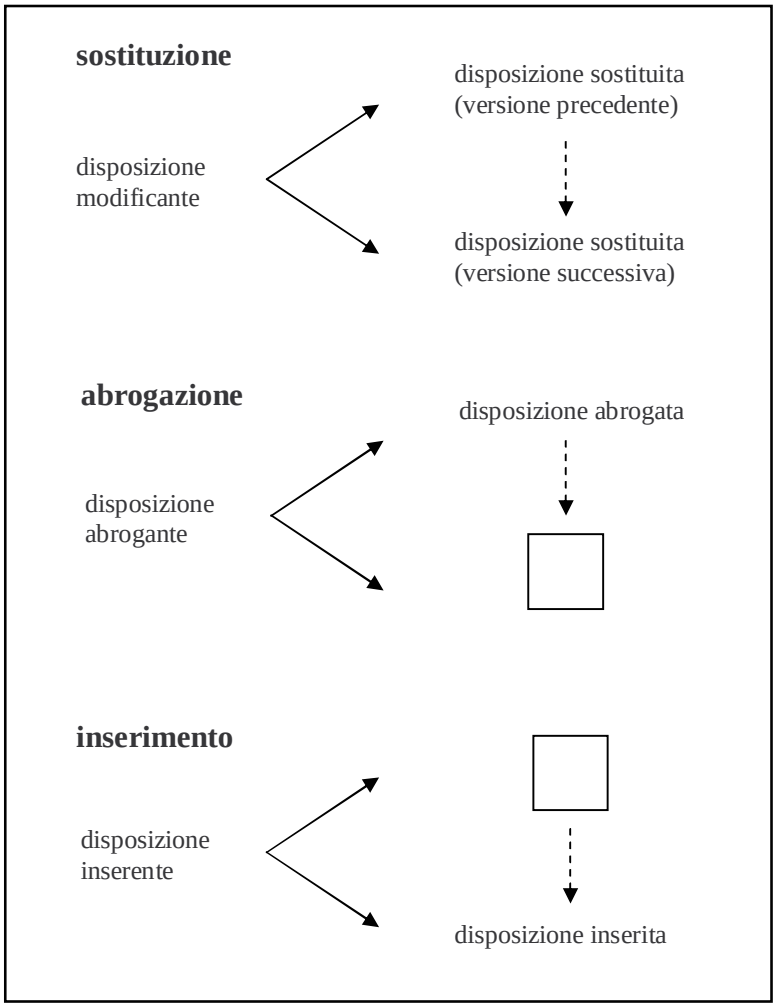


Fig 2.2 – Gli effetti delle modifiche testuali

2.2.4.1. Il tempo nelle modifiche testuali

Solitamente si ritiene che l'evento della modifica testuale si verifichi e si concluda nel momento nel quale la disposizione attiva modificante entra in vigore. Di conseguenza la successiva modifica o abrogazione di quest'ultima non incide sulla disposizione passiva già modificata.

E' possibile, inoltre, che disposizioni attive stabiliscano che la modifica avrà luogo in un tempo successivo alla loro entrata in vigore. Tali disposizioni, chiamate abrogazioni (inserimenti o sostituzioni) ritardate, potranno ovviamente essere modificate prima che abbiano avuto efficacia.

2.2.4.2. Modifiche testuali e dinamica del testo normativo vigente

Come abbiamo osservato, ogni modifica testuale produce una nuova versione della disposizione modificata, sulla quale agisce la modifica successiva. Una determinata versione del testo 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 ottica, la storia di una disposizione normativa si configura come una sequenza di versioni successive, il passaggio da una all'altra avviene con l'intervento di una disposizione modificante come mostrato in figura 2.3.

Il testo normativo non è quindi un unico documento testuale, ma una sequenza di versioni testuali collegate da operazioni modificatrici, oppure, se si preferisce, può essere considerato un'entità virtuale che evolve nel tempo.

Il testo "virtuale", pur rappresentando il continuo riferimento ideale del giurista, non è sempre di facile identificazione: la ricostruzione del testo vigente richiede l'esatta individuazione del documento di partenza e di tutte le successive modifiche.

La ricostruzione automatica del testo vigente rappresenta una delle finalità del progetto TeX.

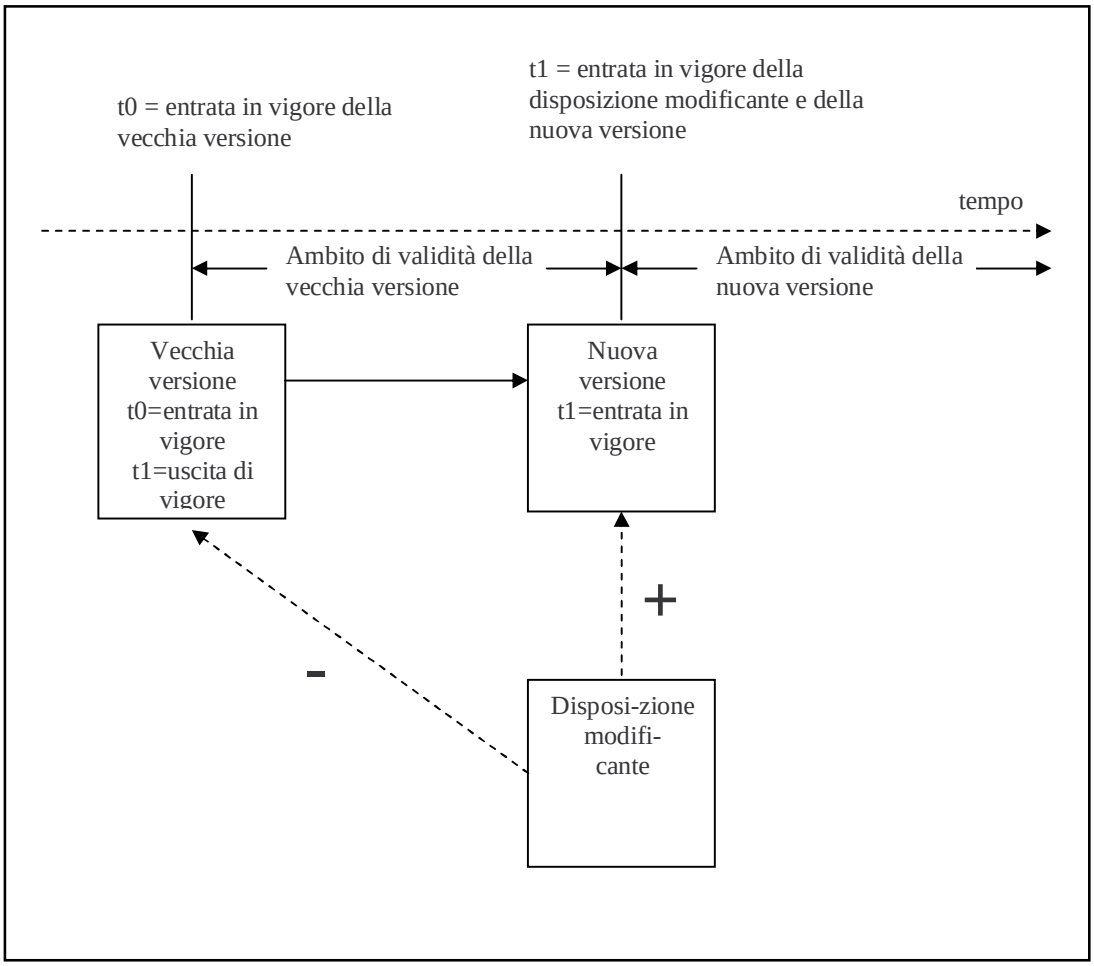


Fig. 2.3 – La dinamica temporale delle modifiche testuali

2.2.5. Modificazioni temporali

Le modificazioni temporali comportano una modifica della sfera di validità o della applicabilità temporale della disposizione passiva, senza modificarne il testo.

Raramente la modifica dell'ambito temporale può avvenire anche modificando il testo della disposizione passiva, ad esempio modificando i riferimenti temporali contenuti nella norma interessata, agendo cioè su espressioni del tipo: "fino alla data x i soggetti y possono presentare la dichiarazione…". In questi casi però, la modifica è considerata testuale.

Tra le modificazioni temporali prenderemo in considerazione le *proroghe* e le *sospensioni*.

2.2.5.1. Proroga

La relazione di proroga incide sulla validità della norma giuridica, la disposizione prorogante stabilisce che la disposizione prorogata, invece di uscire di vigore il giorno x, rimane in vigore indefinitamente, oppure fino a un momento determinato successivo a x.

Possiamo distinguere due tipi di proroga.

- La proroga *propria*, che interviene prima dell'efficacia di una disposizione abrogante ritardata, e si risolve nell'abrogazione di tale disposizione, sostituendola eventualmente con una seconda disposizione abrogante destinata ad avere effetto in un momento successivo.
- La proroga *impropria*, che interviene dopo che una disposizione abrogante ha avuto effetto. Tale tipo di proroga può essere destinata ad entrare in vigore retroattivamente, dal momento dell'effettiva abrogazione della disposizione da prorogare oppure non retroattivamente, dal momento dell'emanazione della disposizione prorogante, in questi casi si parla rispettivamente di *riviviscenza* e *richiamo in vigore*.

2.2.5.2. Sospensione

La 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 durante il periodo di sospensione.

2.2.6. Modificazioni materiali

Le modificazioni materiali influiscono sulla portata normativa di una disposizione senza modificarne il testo. Di seguito ne descriviamo brevemente le tipologie.

2.2.6.1. Deroga

Nella deroga la disposizione attiva derogante limita gli effetti giuridici della disposizione passiva derogata. Più precisamente, la conseguenza giuridica della disposizione derogata non si verifica nel caso in cui sia soddisfatta la fattispecie della disposizione derogante, che in questi casi è spesso chiamata eccezione.

2.2.6.2. Estensione

Nell'estensione, la disposizione attiva richiama l'effetto giuridico della disposizione passiva collegandolo a una nuova fattispecie.

2.2.6.3. Modificazione interpretativa

Nella modificazione interpretativa la disposizione attiva stabilisce il significato della disposizione passiva.

C a p i t o l o 3

DATABASE TEMPORALI

I database temporali[2][3] sono stati concepiti per automatizzare la gestione degli aspetti temporali dei dati rendendola il più possibile trasparente all'utente.
In questo capitolo se ne introdurranno le tematiche fondamentali, collegandole in particolare all'esigenza di rappresentare testi normativi dipendenti dal tempo. La gestione in maniera automatica di alcuni aspetti temporali è, infatti, una delle finalità principali del progetto TeX.
Analizzeremo i tipi di dati temporali e le dimensioni temporali proposte dalla letteratura scientifica, discuteremo come i modelli dei dati temporali si comportano e il tipo di informazioni che sono in grado di gestire.
Nella parte finale ci spingeremo poi più a fondo, prendendo in considerazione due dimensioni temporali che sono state introdotte in tempi più recenti e sono ancora oggetto di discussione: l'event time e l'availability time, ne descriveremo le caratteristiche e il modo in cui si relazionano alle altre.
I paragrafi che seguono pongono le basi per la fase di modellazione del progetto TeX, soprattuto per quanto riguarda la scelta delle dimensioni temporali che il nostro modello dovrà supportare.

3.1. Introduzione ai Database Temporali

Il tempo è un aspetto fondamentale della realtà. Gli eventi accadono in specifici istanti nel tempo, gli oggetti e le relazioni tra di essi, nel nostro caso disposizioni e

nessi normativi, esistono nel tempo. La capacità di modellare la dimensione temporale sta diventando sempre più importante per molte applicazioni software.
I database convenzionali rappresentano lo stato della realtà in uno specifico momento nel tempo, i continui cambiamenti sono visti come modifiche dello stato, mentre le vecchie informazioni, non più attuali, sono cancellate. Il contenuto corrente di un database può essere considerato come una fotografia della realtà modellata.
Un database temporale supporta alcuni aspetti del tempo: in generale garantisce cancellazioni e modifiche non distruttive e cattura le variazioni nel tempo dei fenomeni che modella.
I vantaggi di tale strumento sono molteplici, l'utente può modificare non soltanto la stato attuale del database, ma anche stati passati e futuri; può eseguire ricerche complesse che coinvolgano il tempo, come ad esempio la ricostruzione della versione vigente, in una determinata data, di una disposizione normativa.

3.2. Modelli del tempo e tipi di dati temporali

Il tempo è un insieme arbitrario di istanti a cui viene associato un ordine, può essere classificato secondo diversi criteri ed essere amministrato in modo differente in accordo con le necessità del caso. Ad esempio, un database universitario può gestire il tempo con una granularità giornaliera, mentre un'agenzia pubblicitaria avrà certamente bisogno di un livello più fine per tener conto degli orari in cui gli spot sono andati in onda.
Di seguito sono riportate le principali classificazioni del tempo.
In base alla *struttura* il tempo può essere:

- *lineare*: il tempo avanza dal passato al futuro in modo lineare su un unico percorso;

- *branching*: il tempo è lineare fino all'istante corrente per poi diramarsi in più alternative differenti;
- *ciclico*: il tempo avanza linearmente da un punto iniziale a un punto finale, per poi riprendere a scorrere dall'inizio;

In base alla *densità* il tempo si può classificare come:

- continuo;
- discreto: questo è il caso, solitamente utilizzato nei database, in cui preso in considerazione l'aspetto della *granularità* del tempo, e cioè dell'unità di misura temporale di riferimento.

  E' detto *chronon* l'unità di tempo non decomponibile, la più piccola durata di tempo quindi rappresentabile nel modello alla granularità specificata.

In base all'insieme degli istanti il tempo si considera:

- limitato nel passato e/o nel presente;
- illimitato.

Il punto chiave della modellazione del tempo è spesso la decisione sulla granularità appropriata, una granularità troppo fine appesantirà inutilmente il database, al contrario chronon troppo grandi non rappresenteranno in modo adeguato gli aspetti della realtà che si desiderano modellare.

Gli aspetti della realtà che interessa modellare nelle basi di dati, possono essere considerati come un susseguirsi nel tempo di fatti che, per qualche motivo, hanno inizio, durano per un certo periodo e poi si concludono. Il tempo in cui un fatto è presente nel mondo reale viene chiamato *tempo di occorrenza* del fatto.
Al fine di modellare il tempo di occorrenza di fatti ed eventi sono stati introdotti diversi tipi di dati temporali, andiamo ad elencarli:

- l'*istante* è un particolare chronon nell'asse dei tempi: questo tipo di dato temporale è in grado di modellare fatti istantanei, detti eventi, il cui tempo di occorrenza è quindi è quindi l'istante in cui accadono;
- il *periodo di tempo* è il tempo compreso tra due istanti: per modellare fatti che abbiano una durata è necessario utilizzare questo tipo di dato temporale;
- l'*intervallo di tempo* è una durata assoluta, ovvero una quantità di tempo di cui è nota solo la lunghezza, non gli istanti di inizio e di fine;
- l'*elemento temporale* è un insieme disgiunto di periodi.

3.3. Dimensioni temporali

Le diverse dimensioni temporali[4] sono utilizzate per associare i fatti al loro tempo di occorrenza. Due sono le dimensioni comunemente riconosciute come le dimensioni fondamentali per i database temporali: il v*alid time* e il t*ransaction time*. Grazie ad esse è possibile rappresentare in una base di dati la maggior parte delle informazioni temporali sui fatti registrati.

La dimensioni temporali possono essere gestite in modo del tutto indipendente una dall'altra, perché rappresentano aspetti del tempo che sono tra loro "ortogonali".

Vediamo ora le principali caratteristiche delle dimensioni temporali e a quali aspetti del tempo si riferiscono.

Valid time: il valid time (VT) di un fatto è il tempo in cui il fatto è vero nella realtà.

- E' solitamente fornito dall'utente;
- il valid time di un evento è l'istante in cui accade;
- il valore speciale *forever (F.)* associato ad un fatto indica che esso è correntemente vero.

Transaction time: il transaction time (TT) di un fatto è il tempo in cui il fatto è corrente nel database come dato memorizzato.

- E' gestito dal sistema;
- è limitato nel passato dall'istante di creazione del database e nel futuro dall'istante corrente;
- il transaction time di un fatto identifica la transazione che lo inserisce e quella che lo cancella;
- il valore speciale *until changed (U.C.)* associato ad un fatto indica che il fatto è correntemente presente nel database come dato memorizzato.

3.4. Modelli dei dati

Come già accennato, è possibile che un'applicazione implementi entrambe le dimensioni temporali o una sola di esse, ciò dipende essenzialmente da quali sono aspetti temporali di cui interessa tenere traccia.
In questo paragrafo illustriamo brevemente i modelli dei dati temporali e le tipologie di informazioni che questi sono in grado di rappresentare in relazione alle dimensioni temporali che implementano.

Modello dei dati snapshot: in questo caso nessuna dimensione temporale è implementata, si tratta del tradizionale database che conserva soltanto lo stato corrente della realtà modellata.

Modello dei dati transaction time: una relazione transaction time è una sequenza di stati snapshot indicizzati lungo il tempo di transazione.
A ogni tupla è associato un transaction time che rappresenta il tempo in cui tale tupla è presente nel database. Nel caso di modifiche non si va ad agire sullo stato corrente del database, ma su una copia di esso, che dopo le modifiche diventa il nuovo stato corrente. Ciò permette di non alterare i dati passati conservandone le copie.

Le interrogazioni al database che coinvolgono il tempo di transazione estraggono quindi informazioni sullo stato della relazione in un qualche istante del passato.

Modello dei dati valid time: una relazione valid time rappresenta lo stato della realtà modellata.
Ad ogni tupla è associato un valid time che rappresenta il tempo in cui il fatto da essa identificata è vero nella realtà, l'utente è tenuto a inserire, oltre agli altri dati, anche le indicazioni sul tempo di validità. Nel database è presente il solo stato corrente e le modifiche agiscono su di esso trasformandolo, non sono perciò conservati i dati non più attuali.
Le query che coinvolgono il valid time estraggono insieme ai dati correnti anche informazioni sulla loro validità.

Modello dei dati bitemporale: combina i vantaggi dei modelli precedenti conservando gli stati passati della relazione oltre ai dati sulla validità delle tuple. Le modifiche agiscono su una copia dello stato di validità del database corrente generandone il nuovo stato di validità corrente.

Al fine di spiegare in modo intuitivo il significato delle dimensioni temporali fondamentali e dei modelli dei dati, consideriamo un semplice esempio, ispirato all'ambiente medico.
Esempio: il 10 di agosto 1998, il Dottor Rossi prescrive una terapia antibiotica basata sul farmaco X a partire dal giorno dopo. I dati relativi alla terapia sono inseriti nel database alle 9:00 del 10 agosto. A causa di un inaspettato reazione al farmaco da parte del paziente, l'infusione dell'antibiotico X è sospesa il 13 agosto e sostituita con una terapia di antibiotico Y da eseguirsi a partire dal 14 agosto, il medico di turno al momento delle complicazioni era il Dottor Bianchi. I nuovi dati sono inseriti il 14 alle 12:00.

Vediamo come i vari modelli dei dati rappresentano i fatti di interesse.

La relazione che memorizza le informazioni sulle prescrizioni di antibiotici è la seguente:

Prescrizioni (Paziente, Farmaco, Medico)

Il modello del tempo adottato è il modello lineare discreto.

I fatti vengono rappresentati facendo uso di periodi.

Le granularità del tempo sono il giorno per il valid time e il minuto per il transaction time.

Transazione eseguita il 10 agosto 1998 alle 9:00

Relazione Snapshot

Paziente	Farmaco	Medico
Verdi	X	Rossi

Relazione Transaction Time

Paziente	Farmaco	Medico	TT
Verdi	X	Rossi	[98Ago10;9:00, U.C.)

Relazione Valid Time

Paziente	Farmaco	Medico	VT
Verdi	X	Rossi	[98Ago11, F.)

Relazione Bitemporale

Paziente	Farmaco	Medico	VT	TT
Verdi	X	Rossi	[98Ago11, F.)	[98Ago10;9:00, U.C.)

Transazione eseguita il 14 agosto 1998 alle 12:00

Relazione Snapshot

Paziente	Farmaco	Medico
Verdi	Y	Bianchi

Relazione Transaction Time

Paziente	Farmaco	Medico	TT
Verdi	X	Rossi	[98Ago10;9:00, 98Ago14;12:00.)
Verdi	Y	Bianchi	[98Ago14;12:00, U.C.)

Relazione Valid Time

Paziente	Farmaco	Medico	VT
Verdi	X	Rossi	[98Ago11, 98Ago13]
Verdi	Y	Bianchi	[98Ago14, F.)

Relazione Bitemporale

Paziente	Farmaco	Medico	VT	TT
Verdi	X	Rossi	[98Ago11, F.)	[98Ago10;9:00, 98Ago14;12:00]
Verdi	X	Rossi	[98Ago11, 98Ago13]	[98Ago14;12:00, U.C.)
Verdi	Y	Bianchi	[98Ago14, F.)	[98Ago14;12:00, U.C.)

3.5. Altre dimensioni temporali

Negli ultimi anni gli studiosi di database temporali hanno rivolto sempre maggiori sforzi nella formalizzazione di nuove dimensioni temporali che arricchiscano il livello di dettaglio con cui possono essere trattati dati temporali.

Non tutti gli aspetti temporali del mondo reale possono, infatti, essere gestiti dalle dimensioni temporali classiche. Non è possibile, ad esempio, distinguere se una modifica è retroattiva, oppure stabilire se il sistema informativo nel suo complesso fosse già al corrente di tale modifica prima che questa venisse effettivamente registrata nel database.

Altre due dimensioni sono ritenute necessarie per una più completa descrizione della realtà, esse sono l'*event time* e l'*availability time*[4].

3.5.1. Event time

L'event time (ET) di un fatto è il tempo di occorrenza dell'evento che ha generato il fatto nella realtà.

Grazie all'event time si sopperisce all'incapacità del valid e del transaction time di distinguere tra modifiche retroattive o ritardate, in particolare considerato in relazione al valid time permette di distinguere gli eventi in tre categorie:

- eventi "*on-tme*": l'intervallo di validità ha come inizio il tempo di occorrenza dell'evento che ha generato il fatto;
- eventi *retroattivi*: l'intervallo di validità comincia prima del tempo di occorrenza dell'evento;
- eventi *proattivi*: L'inizio dell'intervallo di validità è successivo al tempo di occorrenza dell'evento.

Per quanto riguarda le modifiche esse sono classificate in base agli eventi che le hanno generate, di conseguenza una modifica retro/proattiva sarà generata da un evento retro/proattivo.

In relazione al transaction time l'uso dell'event time permette una diversa classificazione delle modifiche:

- modifiche "*on-time*": il transaction time coincide con l'event time, in questo caso i dati sono inseriti nel database non appena vengono generati nella realtà;
- modifiche *ritardate*: il transaction time è maggiore dell'event time, in questo caso i dati sono inseriti in un tempo successivo alla loro generazione;
- modifiche *anticipate*: il transaction time è inferiore dell'event time, questo è il caso in cui i dati sono inseriti prima del tempo di occorrenza dell'evento che li genera. La nozione di modifica anticipata è utile per modellare ipotetiche serie di eventi.

La definizione originale dell'event time, soffre però di una limitazione intrinseca: assume implicitamente che sia sufficiente associare un singolo evento a ogni intervallo di validità. In molte situazioni questa assunzione è accettabile ma esistono casi in cui è opportuno distinguere tra il tempo di occorrenza dell'evento che identifica l'inizio di un fatto e l'evento che ne provoca la fine.
Introduciamo brevemente la definizione completa dell'event time, senza addentrarci in analisi approfondite in quanto nel progetto TeX si farà uso della formulazione originale: nel nostro caso specifico, come vedremo in seguito, l'event time sarà associato alla data di pubblicazione sulla gazzetta ufficiale di un testo normativo, un singolo evento sarà quindi sufficiente a modellare tale aspetto.

Event time (seconda formulazione): l'event time di un fatto è il tempo di occorrenza di un evento che nella realtà provochi l'inizio o la fine dell'intervallo di validità del fatto.

Questa definizione determina quindi l'introduzione di due event time, che chiameremo *initiating event* time (ETi) e *terminating event* time (ETt) e che identificheranno rispettivamente l'inizio e la fine del fatto in questione.

3.5.2. Availability time

Per completezza in questo paragrafo si introduce e si descrive brevemente l'availability time, nel progetto TeX non se ne viene fatto uso, o meglio si considera che l'event time e l'availability time coincidano entrambi con la data di pubblicazione sulla gazzetta ufficiale.
L'availability time (AT) di un fatto è il periodo di tempo durante il quale il fatto è noto e ritenuto corretto nel sistema informativo.
In pratica l'availability time può essere considerato il transaction time del sistema informativo, spieghiamo con un breve esempio.

Esempio: un medico di famiglia viene a conoscenza, in data 14 Aprile 2003, di alcuni particolari sintomi presentati da un suo paziente successivamente a un incidente sportivo, provvede a registrare tali sintomi nel database solo tre giorni più tardi.
In questo caso il transaction time sarà [2003Apr17,U.C.), mentre l'availability time sarà [2003Apr14,U.C.); anche se non registrata nel database fino al 17 Aprile l'informazione è già disponibile al sistema informativo nel suo complesso a partire dal 14 Aprile.

3.6. Relazioni tra le dimensioni temporali

La figura 3.1 riassume le relazioni tra le dimensioni temporali analizzate, la realtà modellata e il sistema informativo.

Riepilogando, il transaction time è “append-only”, non è permessa cioè la cancellazione dei dati precedentemente registrati, è possibile soltanto di registrare nuovi fatti il cui intervallo di tempo di transazione include il presente, la rimozione logica delle informazioni non più attuali è ottenuta “chiudendo” i transaction time dei fatti corrispondenti, le modifiche sono ottenute combinando inserimenti e cancellazioni.

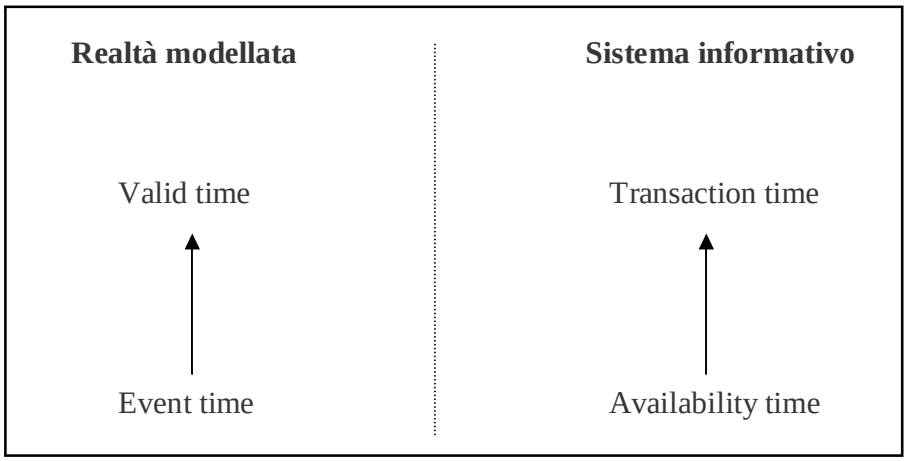


Fig. 3.1 – Dimensioni temporali, realtà modellata, sistema informativo

Al contrario il valid e l’event time, essendo collegati a tempi nel mondo reale possono essere posizionati sia nel passato che nel futuro e sono liberamente modificabili.

Per quanto riguarda l’availability time è anch’esso, per natura, “append-only”, perché fatti precedentemente conosciuti e ritenuti corretti dal sistema informativo non possono essere cambiati. Comunque, dal punto di vista del database, l’availability time può essere considerato “append-only” solo nel caso che non ci siano errori nella registrazione dei dati; tale evetualità non può chiaramente essere

esclusa, di conseguenza, stati precedenti del sistema informativo possono essere riconsiderati attraverso l'inserimento di nuovi dati.

C a p i t o l o 4

DATI SEMISTRUTTURATI - XML

Nel presente capitolo si analizza il protocollo XML[5] (eXtensible Markup Language). Inizialmente proposto come standard per la rappresentazione dei documenti testuali, XML si sta ora imponendo come infrastruttura generale del World Wide Web per l'interscambio di informazioni tra le diverse applicazioni. XML stabilisce un insieme di convenzioni per la definizione di 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.
La capacità di XML di descrivere testi dotati di una struttura gerarchica e in particolare, la possibilità di collegare agli elementi di un documento dei metadati temporali che ne qualifichino cronologicamente il contenuto, ne fanno uno strumento di grande interesse per la rappresentazione dei testi normativi. Inoltre, il sempre maggior successo di XML come standard per lo scambio di dati su Internet, ci permette di non perdere di vista uno degli obiettivi fondamentali che ci siamo prefissi: la portabilità del nostro modello dati e, per quanto possibile, del prototipo.
Nella parte finale del capitolo, descriveremo brevemente gli approcci proposti in letteratura per la rappresentazione degli aspetti temporali nell'ambito di XML che più hanno influito sulla fase di progettazione del nostro modello dati.

4.1. Introduzione a XML

XML è uno standard abbastanza recente, pubblicato ufficialmente nel febbraio 1998 dal W3C[6] (World Wide Web Council), consorzio che si occupa delle definizione di standard per il Web.

A differenza dei normali formati di videoscrittura e di stampa (Word, PostScript…), XML è uno standard documentato e indipendente dai produttori di software e inoltre risponde a un'esigenza specifica, avvertita da editori e studiosi umanisti, quella cioè di rappresentare la struttura logica di un documento, come ad esempio la suddivisione in articoli e commi, creando un'astrazione dell'aspetto grafico con cui il documento si presenta sulla pagina stampata.

XML può essere considerato come il diretto discendente di SGML (Standard Generalized Markup Language), un linguaggio di markup nato nel 1980 su specifiche dell'ANSI e successivamente approvato dall'ISO nel 1986. In comune con esso ha la maggior parte delle convenzioni sintattiche e semantiche per la costruzione dei tag e la possibilità di definire diversi linguaggi di codifica in base alla tipologia di documento, cioè di definire diverse DTD o schemi. La differenza fondamentale consiste nella maggior semplicità e flessibilità di XML.

Un'altra importante parentela è quella con HTML (HyperText Markup Language), esso non è altro che l'applicazione più popolare di SGML, o più precisamente, la sua più diffusa DTD. In pochi anni HTML è divenuto il formato standard per la pubblicazione di documenti su Internet, costituendo la base del World Wide Web, la "ragnatela" ipertestuale che ha reso Internet popolare.

La finalità originale di HTML, cioè la rappresentazione della struttura logica di documenti ipertestuali, è andata perduta a causa della tumultuosa crescita del Web e delle esigenze di rapida pubblicazione dei documenti: una mescolanza inestricabile tra marcatori logici e grafici ha reso impossibile la distinzione tra struttura logica e aspetto grafico del testo. Non essendo quindi sufficientemente in grado di

rappresentare la struttura testuale attraverso HTML, l'esigenza di specificare il complesso logico di differenti tipologie di documenti, in modo distinto rispetto alla loro rappresentazione sul Web, ha portato alla nascita di XML.
In definitiva XML può essere considerato una versione semplificata di SGML, definita in base all'esperienza intercorsa nel frattempo di HTML, tenendo conto delle esigenze di pubblicazione rapida delle pagine Web.

4.2. La struttura di XML

XML è un linguaggio dichiarativo, non definisce quindi le operazioni da fare per visualizzare la pagina, ma definisce le varie parti del testo e le loro rispettive relazioni; più precisamente XML è un linguaggio di markup mediante il quale è possibile associare a ciascuna parte di un testo un *marcatore*, o *tag*, che la qualifica come un determinato elemento logico senza preoccuparsi di come apparirà fisicamente nel documento.
Ciascun marcatore XML è delimitato da due simboli di inizio e di fine, le parentesi angolari < > e può avere o meno degli *attributi*.
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 figura 4.1 la stringa "Direttiva" può essere identificata come elemento natura, con la codifica <natura>**Direttiva**</natura>.
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, <legge></legge> in figura 4.1, contiene tutti gli altri. Sono ammessi anche elementi vuoti, in cui il marcatore di apertura e di chiusura possono essere sostituiti da un unico tag, con la sintassi </tag>.

Come già accennato, oltre a un testo un elemento può avere degli attributi, cioè campi ai quali sono assegnati determinati valori, ad esempio l'elemento <legge> del testo normativo in figura 4.1 ha l'attributo "num" inizializzato a valore "1".

```
<legge num="1">
  <titolo>Titolo Legge 1</titolo>
  <natura>Direttiva</natura>
  <articolato>
    <capo num="1">
      <rubrica>Testo Rubrica</rubrica>
      <articolo num="1">
        <comma num="1">
          Testo del comma 1
        </comma>
        <comma num="2">
          Testo del comma 2
        </comma>
      </articolo>
    </capo>
  </articolato>
</legge>
```

Fig. 4.1 – Esempio: testo normativo.

4.3. XML come protocollo per lo scambio di dati sul Web, XML infoset

Nei paragrafi precedenti XML è stato presentato unicamente come formato per la definizione, attraverso la marcatura, della struttura e della semantica di documenti di testo.

Questa descrizione, benché perfettamente coerente con la storia di XML, non rispecchia appieno lo stato attuale dello standard, che si rivolge alla rappresentazione di qualsiasi tipo di informazione che possa essere scambiata tra sistemi software.

La parte oggi considerata più importante dell'intero standard XML, il cosiddetto *Infoset*[7], descrive i documenti XML come insiemi di *oggetti astratti*, che hanno

una o più proprietà dotate di nomi convenzionali, senza fare alcuna ipotesi sul loro formato di memorizzazione a basso livello, detto anche *formato di serializzazione*.
Grazie ad Infoset i documenti XML sono quindi del tutto svincolati dai dettagli tecnici di come i dati sono effettivamente rappresentati su una specifica piattaforma, XML diventa così una sorta di "linguaggio comune" del mondo Web.
Anche la modellazione di Infoset segue la struttura gerarchica dei testi, l'intero documenti XML costituisce, infatti, un oggetto-documento paragonabile alla radice di un albero *multi-sorte*, cioè composto da oggetti di vario tipo.
Ogni sistema software che deve manipolare dati XML è libero di definirne una propria rappresentazione interna di basso livello, purché rispetti lo standard Infoset.
Nella terminologia di Infoset, la struttura interna di un elemento XML viene chiamata *modello di contenuto*; nell'esempio in figura 4.1 per l'elemento <capo>, si parla di *modello di contenuto di soli elementi*, mentre per gli elementi che contengono solo dati e non hanno elementi figlio, come <rubrica> nell'esempio, si parla di *modello di contenuto di soli dati*.
Il *modello a contenuto misto*, che contiene sia dati che elementi figlio, finora piuttosto comune, è sempre meno usato.

4.4. XML Schema e DTD

XML è un metalinguaggio generico che consente di costruire diversi linguaggi di codifica, ciascuno per una diversa tipologia di documenti: non prescrive i nomi dei diversi marcatori, ma solo la sintassi generica per la loro definizione e il loro utilizzo.
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 stabiliscono quali sono i nomi ammissibili per i marcatori e quali relazioni di inclusione possono sussistere tra di loro.

In questa tesi non ci addentreremo in analisi approfondite sulle DTD: nel progetto TeX non se ne fa uso, e inoltre, il loro utilizzo è in calo in favore dello standard XML Schema[8].

XML Schema risolve molti dei problemi che si presentavano con le DTD: in primo luogo consente una uniformità di linguaggio tra dati (il contenuto dei documenti XML) e i corrispondenti metadati, cioè lo Schema stesso, uniformità non presente nelle DTD. Diventa quindi possibile manipolare dati e metadati con gli stessi strumenti software. Inoltre XML Schema fornisce un maggior numero di *tipi di dati elementari*.

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 elementari incorporati, comunemente a disposizione dei progettisti di database relazionali; ma le DTD richiedono una definizione manuale anche di tipi comunemente usati come le date e non consentono di porre vincoli sui dati, come ad esempio un campo numerico limitato agli interi tra 1 e 10.

Lo standard XML Schema è una sintassi che distingue tra tipi di dati ed elementi XML appartenenti a quei tipi. Oltre ad includere una quarantina di tipi elementari incorporati, XML Schema fornisce al progettista lo stesso potere degli attuali linguaggi di programmazione ad oggetti. In sostanza XML Schema è un completo *Data Definition Language* (DDL) basato su XML che incoraggia, anche se non impone, il classico modo di precedere dei progettisti software: partire dalla *definizione* di tipi di dati per eseguire una successiva *dichiarazione* di elementi XML appartenenti a quei tipi.

Un documento che rispetta la sintassi generica di XML, senza riferirsi a una specifica DTD o ad uno Schema, è detto *ben formato*. Un documento invece congruente a una DTD o ad uno Schema è detto *valido*.

Il controllo della validità di un documento è una garanzia di congruenza formale, necessaria per applicare con successo al documento stesso procedure automatiche di elaborazione.

4.5. Parsing e validazione dei documenti: i formati SAX e DOM

Il risultato della *validazione* di un documento XML rispetto a una DTD o ad uno Schema, operazione spesso chiamata *parsing*, è una rappresentazione dell'Infoset del documento in un formato più adatto all'elaborazione rispetto al formato di serializzazione.

Le due rappresentazioni più comuni sono il DOM (Document Object Model) e il SAX (Simple API for XML). Entrambe queste tecniche si basano sulla traduzione delle astrazioni di Infoset in un modello ad oggetti che evita ai programmatori di dover manipolare direttamente i caratteri del formato di serializzazione di XML.

Sia SAX che DOM definiscono un insieme di interfaccie che permettono ai programmi di accedere all'insieme delle informazioni XML, ma differiscono in alcuni aspetti fondamentali. SAX traduce l'albero di un documento in una sequenza lineare di eventi. Leggendo un file XML, un parser SAX genera un evento ogni volta che incontra uno degli oggetti infoset. Un parser DOM traduce l'Infoset del file in un albero di oggetti.

DOM è particolarmente adatto alle applicazioni che devono rappresentare interamente in memoria un documento XML, mentre le applicazioni basate su SAX non hanno bisogno di tenere il memoria l'intera rappresentazione Infoset.

Una seconda non trascurabile differenza deriva dal fatto che DOM è una Raccomandazione del W3C e ha dietro di sé il peso istituzionale di questa organizzazione, mentre SAX è uno standard di fatto.

4.6. I linguaggi di stile, cenni

La modalità di rappresentazione grafica di ciascun elemento logico del testo, associato ad una coppia di marcatori, è definita a parte da appositi linguaggi, detti *fogli di stile*: diventa così possibile utilizzare lo stesso documento XML per diverse modalità di pubblicazione semplicemente cambiando il foglio di stile associato.
Il linguaggio XSL (eXtensible Stylesheet Language) è lo standard per la presentazione dei documenti XML.
Non ci addentriamo nella trattazione in quanto nel progetto TeX non si fa uso di fogli di stile.

4.7. XML e i dati temporali

Dai precedenti paragrafi si può evincere che la scelta di XML per la rappresentazione dei testi normativi è motivata principalmente da due aspetti, il primo è la possibilità di produrre documenti che contengano la struttura del testo oltre al testo stesso, il secondo, non meno importante, è la sempre maggior espansione di XML come standard per il passaggio di dati tra applicazioni diverse.
Resta da analizzare l'efficacia di XML in presenza di dati che, per essere correttamente interpretati, abbiano la necessità di un inquadramento temporale.
La strada che si è intrapresa consiste nell'arricchire i documenti che contengono i testi normativi, con meta-dati temporali, cioè con informazioni aggiuntive sulla valenza nel tempo dei testi o di loro porzioni, in altre parole, di dati sui dati.
XML non permette una distinzione tra dati e meta-dati a livello sintattico, i meta-dati sono trattati semplicemente come informazioni addizionali; è perciò necessaria la definizione di uno schema che permetta la corretta interpretazione degli elementi che formano un documento.
La potenza espressiva e la flessibilità dello standard XML Schema si adatta bene all'esigenza contrassegnare i dati contenuti in un documento, o parte di esso, con

etichette che ne rappresentino le proprietà. Queste proprietà, intese come meta-dati, possono essere di qualsiasi tipo, come ad esempio il nome dell'autore di una porzione di testo, o il livello di sicurezza necessario per accedere a una informazione. Nel nostro caso specifico è importante associare i dati a dimensioni temporali che li collochino in un preciso contesto cronologico.

4.7.1. XML temporale, stato dell'arte

La letteratura scientifica ha preso in considerazione molto di frequente le modifiche, il versionamento e anche esplicitamente gli aspetti temporali nella gestione dei dati semistrutturati e di XML, spesso applicando strumenti e tecniche sviluppati nella ricerca sui database temporali.

L'obiettivo principale di alcuni di questi approcci era la rappresentazione e la gestione delle modifiche, dove nuove versioni dei dati sono prodotte dagli aggiornamenti. In questi casi, gli attributi temporali sono spesso utilizzati per etichettare le versioni registrate e rappresentano il tempo in cui la modifica ha avuto effetto. Essi hanno quindi la semantica (implicita) del transaction time in relazione al sistema in cui le modifiche sono avvenute.[9,10]

Altri approcci hanno considerato la nozione classica di valid time, ad esempio il lavoro "Valid Web"[11,12] è una infrastruttura XML/XSL progettata per rappresentare e gestire documenti Web temporali (documenti etichettati esplicitamente dagli autori al fine di assegnare una validità alle informazioni contenute). Questi documenti temporali possono essere acceduti selettivamente, in accordo con dei periodi temporali di interesse forniti dall'utente.

Alcuni lavori considerano un modello dati bitemporale, che supporta sia il valid che il transaction time[13,14], mentre in altri viene anche presa in considerazione una dimensione temporale specifica del Web: il navigation time, che riguarda l'interazione degli utenti durante la navigazione dei siti Web[15,16].

Un approccio che ha parecchi punti in comune con il nostro lavoro è presentato nel paragrafo successivo.

4.7.2. Multidimensional XML

Un'estensione a XML per rappresentare in modo elegante e conciso informazioni dipendenti dal contesto è il Multidimensional XML[17]. In questo paragrafo descriveremo una proposta di come il MXML può essere usato nel caso di dati dipendenti dal tempo[18].
MXML permette di creare elementi o attributi che abbiano diverse versioni, dipendentemente da un set di dimensioni.
La sintassi XML è estesa per incorporare le dimensioni, in particolare un elemento multidimensionale ha la seguente forma:

```
<@nome_elemento attributi>
  [contesto_1]
    <nome_elemento attributi_1>
       contenuto_elemento_1
    </nome_elemento>
  [/]
. . .
  [contesto_N]
    <nome_elemento attributi_N>
       contenuto_elemento_N
    </nome_elemento>
  [/]
<@nome_elemento>
```

dove `contenuto_elemento_i`, con $1 \leq i \leq N$ è il contenuto dell'elemento multidimensionale specificato da `[contesto_i]`. Il medesimo nome elemento è usato per tutti i contesti, ogni gruppo è infatti incluso in uno speciale elemento, che

al nome fa precedere il simbolo “@”. La stessa cosa vale per gli attributi, anch’essi possono dipendere dal contesto.

I contesti specificano i valori delle diverse dimensioni utilizzati in cui il loro contenuto è valido e devono necessariamente essere mutuamente esclusivi. In altre parole, per ogni set di coordinate, non più di un contesto deve essere valido.

Un altro degli aspetti fondamentali è la propagazione del contesto, in generale, i sottoelementi ereditano il contesto del nodo padre e possono al più specializzarlo ulteriormente.

MXML permette di specificare i contesti attraverso qualsiasi tipo e numero di dimensioni, è quindi sufficiente considerare le dimensioni temporali descritte nel capitolo 3, per avere un modello per dati dipendenti dal tempo.

Per una più dettagliata analisi della sintassi per esprimente i contesti e la granularità delle dimensioni, si rimanda a [18].

C a p i t o l o 5

ORACLE 9i - XML DB e INTERMEDIA

Oracle[19] è il DBMS che negli ultimi anni ha meglio saputo anticipare le necessità dei progettisti e dei programmatori di basi di dati. Oltre a un collaudatissimo ambiente per la gestione dei dati relazionali attraverso SQL, propone sempre nuove estensioni che si specializzano sugli aspetti più attuali del mondo informatico.
Tutto ciò, unito alle alte prestazioni offerte e alla buona presenza sul mercato, ci ha spinto a scegliere Oracle come DBMS per il nostro lavoro.
In questo capitolo vedremo le caratteristiche dei pacchetti Oracle 9i XML DB[20,21] e *inter*Media[22], affrontandone prima una breve descrizione delle funzionalità per poi concentrare l'attenzione sugli aspetti più sfruttati nel progetto TeX.
Questa parte della tesi sarà volutamente sintetica nel contenuto per non appesantire eccessivamente il lettore con dati tecnici. E' comunque necessario introdurre due importanti strumenti utilizzati nel nostro lavoro.
Per una descrizione più dettagliata di Oracle XML DB e *inter*Media si rimanda ai riferimenti bibliografici.

5.1. Oracle XML DB

Oracle XML DB è un pacchetto, disponibile con Oracle 9i, che implementa un sistema ad alte prestazioni per l'archiviazione e l'interrogazione di dati XML. XML DB estende il database relazionale proponendo funzionalità tipiche di un database

XML nativo, offrendo un'infrastruttura per la gestione e la registrazione di documenti XML, indipendente dal contenuto e dai linguaggi di programmazione.
Da sottolineare è l'introduzione di un repository XML nativo accessibile dai più comuni protocolli e interrogabile con SQL.
In figura 5.1 illustriamo la struttura di XML DB; nei paragrafi successivi passeremo in rassegna le principali caratteristiche del prodotto.

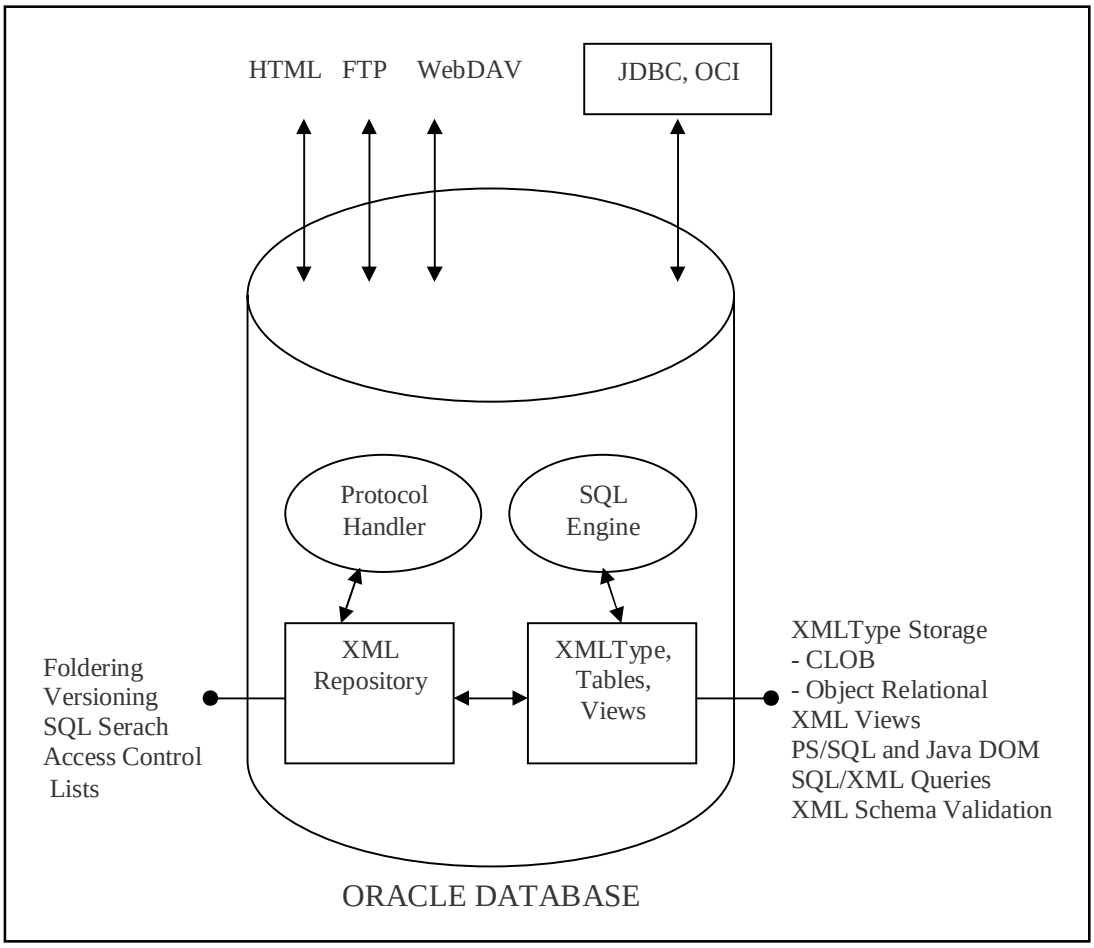


Fig. 5.1 – Struttura di Oracle XML DB

5.1.1. Dualità XML/SQL

Una gran parte delle informazioni rilevanti all'interno di una organizzazione è conservata sotto forma di dati semi-strutturati o non strutturati, tipicamente memorizzati in files su un file server. Questi file sono tipicamente in formati proprietari, accessibili solo tramite specifici strumenti, come word processors o fogli elettronici. Le ricerche su queste informazioni sono spesso limitate a ricerche testuali all'interno dei documenti.

I dati strutturati, tipicamente conservati in un database relazionale, non hanno queste limitazioni, sono accessibili attraverso SQL da una varietà di strumenti software.

Uno dei motivi che più spingono verso l'adozione di XML, è che ciò permetterebbe una migliore condivisione delle informazioni non strutturate o semi-strutturate, il cui contenuto potrebbe essere descritto accuratamente attraverso gli XML Schema.

Con la promessa della dualità XML/SQL Oracle XML DB abbatte le tradizionali barriere tra applicazioni che trattano dati strutturati e quelle che lavorano su dati non o semi-strutturati; le metafore relazionale e XML diventano interscambiabili.

Questa dualità significa che gli stessi dati possono essere trattati come righe in una tabella e manipolati con SQL, oppure trattati come nodi di un documento XML e gestiti con tecniche come il DOM o trasformazioni XSL. L'accesso ai dati e le tecniche di processing sono del tutto indipendenti dal formato di memorizzazione sottostante. Ciò permette la massima flessibilità, consentendo l'utilizzo dello strumento più adeguato per la soluzione di ogni particolare problema

XML DB fornisce nuovi e semplici soluzioni a numerosi problemi comuni:

- i dati relazionali possono essere facilmente convertiti in pagine HTML, grazie a nuovi operatori SQL che generano XML direttamente da interrogazioni SQL. I dati XML possono quindi essere trasformati in altri formati, come appunto HTML;

- i documenti XML possono essere archiviati e gestiti senza dover eseguire dispendiose operazioni di conversione tra diversi formati.

5.1.2. XMLType

XMLType è un tipo di dato nativo che permette al database di interpretare il contenuto di una colonna come dati XML. Questo nuovo tipo di dato si usa esattamente come tutti gli altri tipi nativi, quindi, ad esempio, può essere usato durante la creazione di una colonna in una tabella relazionale, oppure nella dichiarazione di variabili per una procedura PL/SQL. Vediamo un esempio:

```
CREATE TABLE warehouses(
  warehouse_id NUMBER(3),
  warehouse_spec SYS.XMLTYPE,
  warehouse_name VARCHAR2(35),
  location_id NUMBER(4));
```

XMLType può essere usato anche nella definizione di viste, permettendo ad Oracle di visualizzare dati relazionali sotto forma di documenti XML.

Ogni documento XML ben formato può essere contenuto in una colonna XMLType, esso sarà memorizzato sotto forma di testo, usando il tipo CLOB. Ciò permette la massima flessibilità in termini di strutture XML che possono essere immagazzinate in una singola colonna. Inoltre è possibile vincolare una colonna XMLType a un XML Schema. Ciò offre la possibilità di usare un diverso metodo per la memorizzazione, chiamato Object Relational:

In figura 5.2 sono rappresentate le possibili modalità di memorizzazione, esse verranno discusse più in dettaglio nel paragrafo 5.1.5.

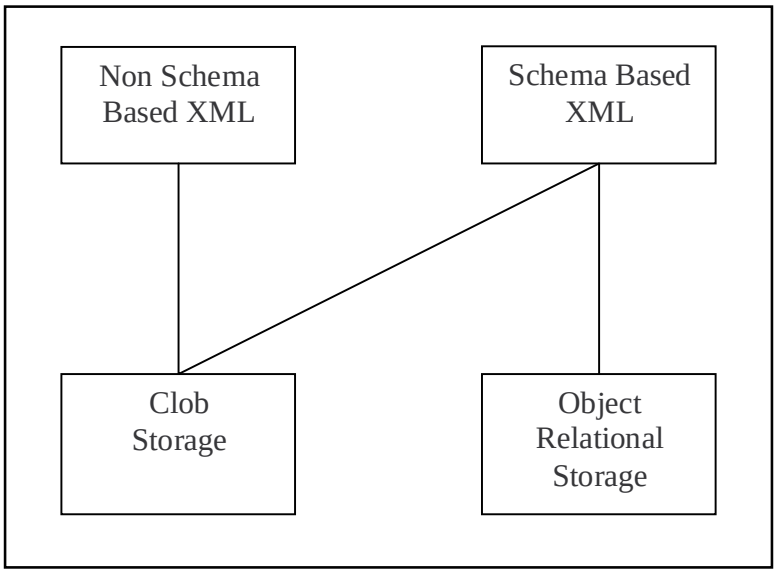


Fig 5.2 – Modalità di memorizzazione XMLType

5.1.3. Le funzioni di XMLType

Oracle 9i introduce tre nuove funzioni: `CreateXML`, `Extract` e `ExistsNode`, che operano su dati XMLType.

CreateXML: questa funzione serve per creare istanze di tipo XMLType partendo da stringhe; il documento viene generato solo le i dati sono XML ben formato. Vediamone un esempio:

```
INSERT INTO legge
values (xmlcolumn, sys.XMLType.createXML(<legge> testo </legge>))
```

Extract: per estrarre porzioni di documenti XML dal database è stata fornita la funzione Extract; attraverso un'espressione XPath, Oracle scansiona il documento alla ricerca dei frammenti che soddisfino le richieste, tali frammenti sono poi

restituiti nella forma specificata grazie all'uso di altre funzioni come ad esempio getStringVal() che trasforma il risultato in una stringa

ExistsNode: questa funzione controlla se una espressione XPath, fornita come argomento, restituisca o meno nodi validi. Vediamo come `ExistsNode` può essere usata insieme ad `Extract`:

```
select L.xmlcolumn.extract('/legge/natura/text()').getStringVal()
from legge L
where L.xmlcolumn.existsNode('/legge/natura') != 0;
```

Questa interrogazione estrae il contenuto dell'elemento natura dei documenti legge che hanno tale elemento , restituendolo sotto forma di stringa.

5.1.4. XML Schema

Il supporto allo standard W3C XML Schema è uno dei punti fondamentali di XML DB, il tipo di dato XMLType fornisce i metodi `isSchemaValid` e `schemaValidate` per la validazione di documenti XML.

Prima di poter usare un XML Schema all'interno del database è necessario registrarlo chiamando la procedura PL/SQL `dbms_xmlschema.register_schema()`. Lo schema viene così registrato con una URL interna che verrà usata come chiave unica per identificare lo schema.

I vantaggi derivanti dalla registrazione di uno schema e dalla validazione sono molteplici:

- il database assicurerà che solo i documenti validi rispetto allo schema saranno archiviati;
- i dati contenuti nella colonna vincolata sono conformi a una struttura ben precisa, saranno quindi possibili query mirate e trattamenti automatici dei dati;

- vincolare XMLType a uno schema offre l'opportunità di memorizzare i documenti usando una tecnica strutturata, decomponendo cioè il documento in frammenti che verranno immagazzinati come oggetti SQL, invece che usare un semplice oggetto CLOB, tale metodo di memorizzazione è chiamato Object Relational.

5.1.5. XML DB Repository

Il modello relazionale, con la sua forte metafora tabella-riga-colonna, non è molto efficace nella gestione dei dati semi-strutturati o non strutturati; un libro non si adatta molto ad essere rappresentato come un set di righe in una tabella, è decisamente più naturale concepirlo come una gerarchia: libro – capitolo – paragrafo, rappresentando tale gerarchia come un set di cartelle e sottocartelle.

Oracle XML DB introduce il concetto di un XML Repository gerarchico e interrogabile tramite query. Utilizzandolo è possibile rappresentare documenti XML come gerarchie. Inoltre Grazie a un nuovo indice gerarchico, il cui lavoro è del tutto trasparente all'utente, il Repository permette un accesso ai dati attraverso foldering e path, con prestazioni comparabili a quelle dei file system convenzionali.

Uno dei maggiori vantaggi offerti dal Repository XML sta nel supportare comuni protocolli come HTTP, FTP e WebDAV, permettendo quindi un accesso diretto ai dati XML.

Vediamo ora in che modo il Repository permette di memorizzare i documenti XML. Come già accennato sono disponibili due diverse modalità: Object Relational e CLOB; quale tra queste utilizzare è una delle decisioni fondamentali che si devono essere prese durante l'uso di XML DB, nella tabella 5.1 è riportato un dettagliato confronto tra le due tecniche.

	Memorizzazione Non Strutturata	**Memorizzazione Strutturata**
Throughput	Throughput massimo per l'inserimento e il recupero di un intero documento XML.	Il processo di decomposizione porta a un leggero calo di prestazioni per l'inserimento e il recupero di un intero documento XML.
Flessibilità	Offre la massima flessibilità in termini della struttura dei documenti XML che sono memorizzabili in una colonna XMLType.	Flessibilità limitata. Solo di documenti conformi allo schema possono essere inseriti. Un'eventuale cambiamento dello schema può richiedere che i dati debbano essere ricaricati.
Fedeltà a XML	Fedeltà al documento: mantiene il contenuto XML originale byte per byte; ciò può essere importate per alcune applicazioni.	Fedeltà al DOM: un DOM creato a partire da un documento XML memorizzato nel database è uguale al DOM creato dal documento originale. I caratteri di ritorno a capo, gli spazi bianchi tra i tag e alcuni dati di formattazione possono andare perduti.

Operazioni di modifica	Non sono possibili ottimizzazioni sulle operazioni di modifica, quando una porzione del documento deve essere modificata, tutto il documento deve essere riscritto.	La maggior parte delle operazioni di modifica possono essere ottimizzate, saranno quindi modificate soltanto alcune parti del documento. Ciò comporta una notevole riduzione dei tempi di risposta.
Query XPath	Le espressioni XPath sono valutate costruendo il DOM dal CLOB che contiene il documento. Ciò può essere dispendioso quando si eseguono ricerche su grandi collezioni di documenti.	Quando possibile, le operazioni XPath sono svolte dopo un procedimento di riscrittura delle query; il miglioramento di prestazioni è particolarmente sensibile con una grande collezione di documenti.
Indicizzazione	Text indexes e Functional indexes ammessi.	Oltre ai Text indexes e ai Functional indexes sono utilizzabili anche i B-Tree.
Management della memoria	Le operazioni XML sul documento richiedono la creazione del DOM.	Le operazioni XML possono essere ottimizzate per ridurre i requisiti di memoria.

Tab. 5.1 – Confronto tra i metodi di memorizzazione.

5.2. Oracle *inter*Media

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

*Inter*Media Text è una estensione di Oracle che permette l'indicizzazione di documenti di testo e l'esecuzione di interrogazioni, come ad esempio la ricerca dei documenti che contengono una certa parola, utilizzando semplicemente i costrutti di SQL standard. Si ha quindi una integrazione tra la gestione dei testi e le più tradizionali operazioni svolte sulle basi di dati relazionali.

5.2.1. La creazione dell'indice.

Il primo passo nell'utilizzo di *inter*Media è la creazione dell'indice di testo. Occorre dire, prima di tutto, che a differenza delle normali interrogazioni in un DBMS che sono solo più lente senza la presenza di indici, le interrogazioni che sfruttino la primitiva `contains`, caratteristica peculiare di *inter*Media Text sono del tutto impossibili senza la presenza di un indice sulla colonna contenente il testo da interrogare.

Il metodo di indicizzazione usato da *inter*Media è quello dell'inverted index (ved. Appendice C). Questo metodo è basato sulla costruzione di una struttura che associa ad ogni parola i documenti che la contengono oltre alla posizione all'interno di essi.

Un tale indice viene creato attraverso il comando:

```
CREATE INDEX my_index ON my_table (my_column)
     INDEXTYPE IS ctxsys.context
     PARAMETERS
      ( 'datastore           my_datastore
        filter         my_filt     er
        section group  my_section_group
        lexer          my_lexer
```

```
    wordlist       my_wordlist
    stoplist       my_stoplist
    storage        my_storage'
);
```

All'interno della clausola PARAMETERS viene inserita una lista di parametri che permettono di costruire un indice che sia adatto alle esigenze di ogni applicazione e ai diversi tipi di documenti. Questi parametri saranno analizzati in dettaglio successivamente. Ora, per meglio capire in che fase della indicizzazione essi interverranno, vediamo una descrizione semplificata dell'operazione.
L'indice è creato attraverso una sequenza di passi denominata *Indexing Pipeline*, se ne riporta una rappresentazione grafica in figura 5.1.

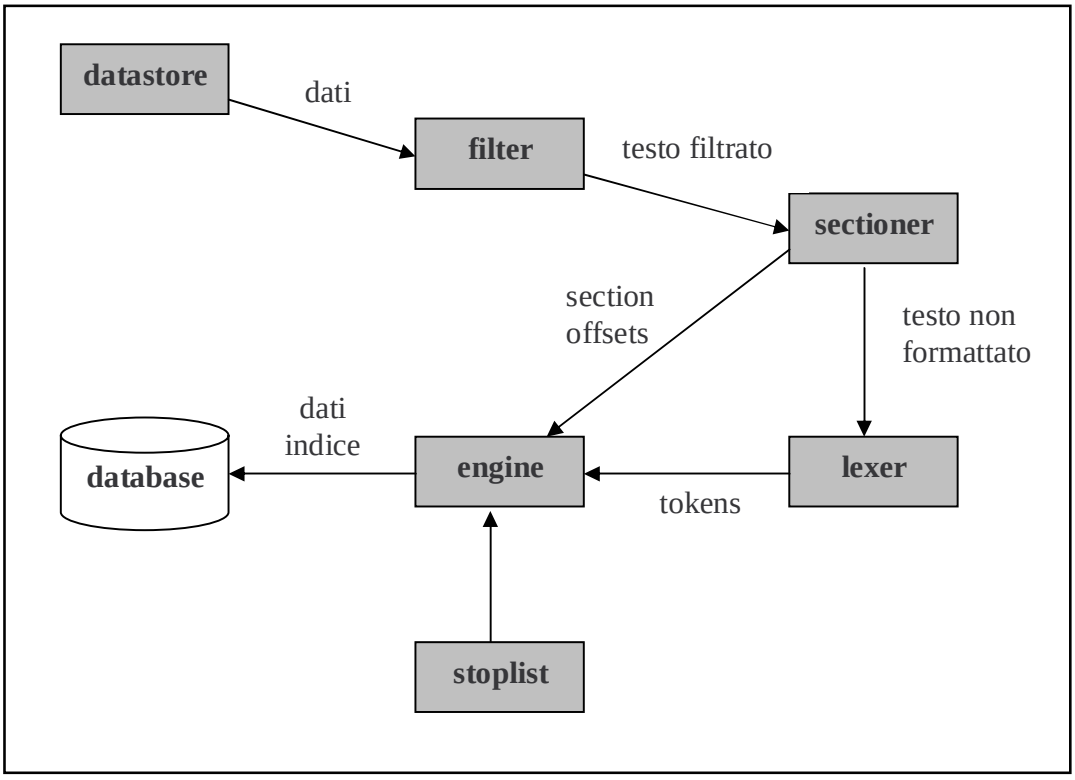


Fig. 5.3 – La Indexing Pipeline per la creazione di un indice

Diamo ora una breve descrizione del significato di ogni passo:

- *datastore*: in questo stadio della pipeline viene eseguito un ciclo di lettura sulla colonna indicizzata e sono prodotti come output i dati del documento. Questi dati possono essere testo vero e proprio oppure, per esempio, puntatori che, a loro volta, permetteranno di accedere a testo;
- *filter*: vengono assunti in ingresso i dati provenienti dal datastore i quali vengono "tradotti" in testo leggibile;
- *sectioner*: converte in testo senza formattazioni, o plain text, l'output del filter. Questa conversione include la ricerca di etichette che delimitano particolari sezioni nel testo e la loro rimozione. Il sectioner darà come output anche gli eventuali *section offsets* che verranno utilizzati per la ricognizione delle sezioni presenti sul documento;
- *lexer*: viene analizzato il testo proveniente dal sectioner e viene separato in termini, detti *tokens*,. La segmentazione viene eseguita tenendo presenti le regole di delimitazione delle parole nei vari linguaggi gestiti. Nelle lingue asiatiche, per esempio, la segmentazione è piuttosto complessa.

Al termine di questa sequenza di passi viene di fatto costruito l'indice. Per compiere questa operazione vengono presi in esame i termini provenienti dal lexer, i section offsets generati dal sectioner e la *stoplist*, cioè la lista di parole considerate di scarso significato e che quindi non saranno inserite nell'indice.

Ogni passo della pipeline è influenzato dal modo in cui sono stati impostati i parametri elencati nell'istruzione `create index` vista in precedenza. Nei paragrafi successivi li descriveremo più in dettaglio, riferendoci ad essi come *classi*, secondo la terminologia fornita da Oracle.

5.2.1.1. La classe Datastore

Questo parametro indica il modo saranno interpretati in cui i dati estratti dalla colonna su cui si sta costruendo l'indice. Esistono cinque opzioni per specificare come il testo è stato memorizzato.

Direct_Datastore: questo è il caso più semplice, nel senso che il contenuto dei documenti di testo si considera immagazzinato proprio nella colonna indicizzata, un documento per riga. Questo valore è assunto come dafault, può pertanto essere omesso nell'istruzione `create index`.

File_Datastore: in questo caso i dati letti saranno considerati nomi di file. Verranno perciò aperti questi files attraverso il file system locale; il loro contenuto sarà quanto restituito.

URL_Datastore: si utilizza per testo memorizzato in files su World Wide Web, accessibili mediante protocollo *http* o *ftp*, oppure in files locali, accessibili attraverso il protocollo *file*.

Detail_Datastore: questo oggetto si utilizza nel caso in cui il testo è memorizzato direttamente nella base di dati, ma in sottotabelle, dette *detail tables*, le quali saranno referenziate, attraverso una chiave d'accesso, da una tabella principale, o *master table*.

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

5.2.1.2. La classe Filter

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

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

Charset_Filter: permette di convertire i documenti aventi un set di caratteri diverso da quello gestito dalla nostra base di dati.

Inso_Filter: vengono utilizzati strumenti di filtraggio forniti dalla Inso Corporation. Questi strumenti permettono di gestire la maggior parte dei formati esistenti, come ad esempio quelli di Microsoft Word o Acrobat/PDF.

Null_Filter: si usa quando il documento da indicizzare à in formato plain text, HTML o XML e non sono necessarie operazioni di filtraggio.

User_Filter: permette di inserire un filtro definito dall'utente. Questo filtro consiste in un programma che, invocato dall'*indexing engine*, si occuperà di effettuare le operazioni di filtraggio su ogni documento da indicizzare. Si potrà inserire, per esempio, uno *script* che converta interamente il testo in lettere minuscole.

5.2.1.3. La classe Section_group

Questa classe permette di individuare e gestire sezioni in un documento per poter poi eseguire ricerche attraverso la clausola *within*, il cui uso approfondiremo in seguito. Questo passo, come indicato in figura 5.1, fornisce due risultati: plain text, scorporato da eventuali etichette, verso il lexer e i section offsets. Per riconoscere le sezioni presenti, verso l'indexing engine. Per poter creare un section group occorre specificare uno dei seguenti tipi di oggetti.

Null_Section_Group: è il valore di default. Non vengono considerate sezioni nel documento.

Basic_Section_Group: viene usato nel caso siano presenti nel documento sezioni contrassegnate da etichette come <Label> e </Label>.

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

XML_Section_Group: come il caso precedente, ma per documenti XML.

News_Section_Group: questo tipo di gruppo manipola documenti aventi le sezioni tipiche dei messaggi di newsgroup.

5.2.1.4. La classe Lexer

La classe lexer viene utilizzata per specificare il linguaggio del testo da indicizzare. Questa scelta determinerà il modo in cui il testo viene diviso in termini, detti tokens. Per gestire questa preferenza si sceglierà uno degli oggetti riportati di seguito.

Basic_Lexer: usato per estrarre i termini da testi scritti in linguaggi la cui codifica viene eseguita utilizzando un byte per carattere. Quasi tutte le lingue occidentali ricadono in questo caso.

Chinese_Vgram_Lexer, Japanese_Vgram_Lexer, Korean_Lexer: usati per identificare termini rispettivamente in cinese, giapponese e coreano.

5.2.1.5. La classe Stoplist

La stoplist è un elenco di parole che sono ritenute poco significative, e che quindi non si desidera inserire nell'indice, queste parole sono dette *stopwords* o *noise words*. Fanno spesso parte della stoplist termini come gli articoli o le congiunzioni, solitamente non vale la pena utilizzare spazio per indicizzarle.

L'indexing engine si occuperà di filtrare le stopwords in modo da non iserirle nell'indice.

*inter*Media fornisce stoplists standard per molti linguaggi, è inoltre possibile definirne di nuove.

5.2.1.6. La classe Wordlist

La classe Wordlist permette di impostare i parametri che serviranno per eseguire ricerche complesse sul testo.

L'unico oggetto fornito per questa classe è la Basic_Wordlist, i cui parametri sono:

- stemmer: specifica in quale linguaggio eseguire l'eventuale *stemming* delle parole. L'operazione di stemming mette in forma base le parole, ad esempio mette i nomi al singolare e i verbi all'infinito;
- fuzzy_match: specifica le routines da eseguire per trovare parole "simili"; serve per risolvere, per esempio, casi di errori di battitura;
- fuzzy_score: indica "quanto simili" devono essere le parole rintracciare rispetto all'oggetto della ricerca;
- fuzzy_numresults: serve per limitare il risultato della ricerca a un certo numero di parole;
- prefix_index: questa opzione permette l'indicizzazione dei prefissi, aumenta la dimensione dell'indice, ma permette migliori prestazioni nel caso si vogliano cercare termini con prefisso cumune;
- prefix_min_length: lunghezza minima dei prefissi da indicizzare;
- prefix_max_length: lunghezza massima dei prefissi da indicizzare.

5.2.2. Metodi di ricerca

Presentiamo ora i tipi di ricerca utilizzati durante il progetto TeX, per ciascuno di essi illustreremo le modalità che *inter*Media utilizza.

5.2.2.1. Interrogazioni "Direct Match"

Il più semplice scenario di ricerca è quello in cui l'utente conosce il termine esatto da cercare. Questo tipo di interrogazioni sono eseguite scorrendo l'indice alla ricerca dei documenti pertinenti.

Oracle *inter*Media fornisce la primitiva `contains` che viene inserita in istruzioni SQL come nel seguente esempio:

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

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

Nel caso si ricerchino più parole contigue lo strumento disponibile consiste sempre nella primitiva `contains`, se invece è sufficiente che le parole siano "vicine", l'operatore `near`, permette di impostare il livello di "vicinanza". Vediamo un esempio:

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

In questo caso saranno restituiti I documenti che contengono le parole cane e gatto, intervallate al più da 10 termini.

5.2.2.2. Interrogazioni "Indirect Match"

Le tecniche analizzate nel paragrafo precedente sono spesso insufficienti, frequentemente infatti, l'utente non può conoscere con esattezza il termine esatto da cercare, termine che può apparire in diverse forme e coniugazioni all'interno dei

documenti, oppure l'utente può volere cercare un argomento, non una particolare parola.

Lo strumento principale fornito da interMedia in queste situazioni è il *Wildcard Match*. Questo tipo di ricerca utilizza wildcards, cioè indicatori che sostituiscono qualsiasi sequenza di caratteri, come ad esempio "*" in MS-DOS. La corrispondenza che si va a ricercare con i termini dell'indice deve essere solo parziale. La primitiva `contains` supporta questa wildcard search con l'operatore "%". Vediamone un esempio:

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

Questa query restituirà "cane", ma anche parole come "canestro".

Esistono altri tipi di ricerca attraverso Indirect Match, non li analizzeremo perché non sono stati usati nel progetto TeX.

5.2.2.3. Interrogazioni composte

Capita spesso che gli utenti vogliano combinare termini multipli in interrogazioni composte per esprimere in maniera più precisa ciò che essi intendono cercare. Per esempio può essere necessario introdurre espressioni del tipo "gatto and felino". InterMedia permette di eseguire interrogazioni come questa ancora una volta attraverso la primitiva `contains`.

I metodi per comporre le interrogazioni sono diversi:

- Operatori Booleani: gli operatori booleani (AND, OR, NOT) possono essere usati per costruire un'espressione logica all'interno di una query. *inter*Media permette di costruire espressioni di questo tipo all'interno della primitiva `contains`. Vediamone un esempio:

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

- Interrogazioni strutturate: spesso le applicazioni richiedono che le ricerche testuali siano combinate con altri criteri di ricerca all'interno della base di dati. Poiché la primitiva `contains` che consente le ricerche testuali è integrata con le istruzioni di SQL standard, può venire usata, all'interno della clausola where, come qualsiasi altro criterio di ricerca;
- Ricerca all'interno di sezioni: questo tipo di ricerca considera la struttura interna dei documenti. Per esempio l'utente può voler cercare un nome nella sezione "address" di un messaggio di posta elettronica, oppure una parola nella sezione "titolo" di un testo normativo. Ad esempio:

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

Ogni espressione in una interrogazione composta può essere di uno qualsiasi dei tipi visti nei paragrafi precedenti.

5.2.3. La gestione delle istruzioni INSERT, UPDATE E DELETE

L'esecuzione delle istruzioni di inserimento, modifica e cancellazione di documenti indicizzati sono gestite come descritto di seguito.

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

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

Possiamo dire che *inter*Media gestisce l'eliminazione di dati in maniera sincrona e l'aggiunta in maniera asincrona. Vediamo alcuni esempi:

se eseguiamo l'istruzione

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

una successiva interrogazione

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

non restituirà alcuna tupla. L'effetto della cancellazione è immediato.

Se invece eseguiamo l'istruzione

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

una successiva interrogazione

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

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

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

Il metodo sync consiste nell'esecuzione dell'istruzione

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

La specifica `online` è importante in quanto senza di essa l'esecuzione di interrogazioni non sarebbe possibile durante l'operazione di sync.

Il metodo background prevede l'avvio di un processo attraverso il comando

```
Ctxsrv –user ctxsys/
```

Una volta avviato, il processo analista periodicamente la coda dr$pendine ed esegue automaticamente in background le operazioni di aggiornamento dell'indice.
In entrambi i metodi, comunque, l'aggiornamento dell'indice viene effettuato aggiungendo una lista invertita anche per itermini già presenti. A lungo andare questa frammentazione dell'indice può portare a cali di prestazioni, perché esistono più liste per lo stesso termine. Per questo motivo è importante eseguire un'ottimizzazione.

5.2.4. L'ottimizzazione

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

```
Word1 doc1 doc2
…
Word1 doc8
```

la deframmentazione produrrà la nuova lista per il termine Word1

```
Word1 doc1 doc2 doc8.
```

L'eliminazione fisica dei riferimenti ai documenti cancellati comporta un'operazione di *garbage collection*. Attraverso di essa, per ogni termine contenuto nei documenti cancellati, viene eliminata dalla corrispondente lista invertita l'occorrenza legata al documento non più valido.
Le operazioni di deframmentazione e di garbage collection riducono il numero di righe nell'indice migliorando quindi i tempi di scansione.
Per eseguire queste operazioni *inter*Media fornisce due modalità di ottimizzazione: FAST e FULL.

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

```
Alter index myindex rebulid online parameters('optimize fast');
```

L'ottimizzazione FULL esegue sia la deframmentazione che la garbage collection e viene impostata con l'istruzione

```
Alter index myindex rebulid online parameters('optimize full');
```

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

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

si più indicare che l'esecuzione abbia durata massima di 5 minuti, La successiva occasione in cui verrà avviata, l'ottimizzazione riprenderà dal punto in cui si è interrotta.

C a p i t o l o 6

IL PROGETTO TeX

Lo scopo del nostro lavoro consiste nel fornire uno strumento che sia di aiuto nell'interpretazione dei testi normativi, offrendo, attraverso un processo di ricostruzione cronologica, le versioni che soddisfino diversi vincoli temporali e che sia in grado di gestire in modo automatico ed efficiente gli aspetti temporali legati all'inserimento e all'aggiornamento delle disposizioni normative.

In questo capitolo sono descritte le fasi di progettazione e sviluppo del sistema TeX, è messo in pratica quanto introdotto nei capitoli precedenti e vegono usati gli strumenti che sono stati descritti.

La fase di modellazione è stata la prima ad essere affrontata, e ha prodotto uno schema per i testi normativi in grado di catturarne le modifiche e rappresentarne gli aspetti temporali. Sul modello è stata introdotta una serie di operatori che si occupa dell'effettiva gestione delle attività di ricostruzione e modifica. Abbiamo poi sviluppato la nostra proposta per quanto riguarda una possibile architettura di un sistema che si basi sul modello e gli operatori.

In seguito abbiamo progettato un prototipo che implementa quanto precedentemente discusso, abbiamo confrontato le diverse alternative prese in considerazione e analizzato le principali classi di oggetti utilizzando il formalismo UML[23].

Per la realizzazione del prototipo si è deciso di usare il linguaggio di programmazione Java "on top" a una base di dati Oracle, unendo così la portabilità

di uno strumento di sempre maggior successo come Java, alla solidità ed efficienza di uno dei migliori DBMS relazionali.

6.1. Modellazione dei testi normativi

La prima difficoltà nella realizzazione del progetto TeX è stata la produzione di un modello per i testi normativi che fosse contemporaneamente di semplice interpretazione e di grande potenza espressiva.

Una importante fonte di ispirazione durante la fase di modellazione è senza dubbio stata una DTD pubblicata dal progetto *Norma in Rete*[24]. Questo progetto è sviluppato da un gruppo di lavoro promosso dall'AIPA e dal ministero di Giustizia e si propone di fornire il collegamento mancante tra la pubblica amministrazione e l'informatizzazione delle banche dati di testi normativi. Uno dei punti centrali di Norma in Rete è il tentativo di ridurre la frammentazione che affligge la reperibilità online di norme, principalmente attraverso l'introduzione di uno standard, basato su XML, per la pubblicazione e lo scambio di testi normativi in rete.

Pur avendo attinto dalle esperienze pregresse di Norma in Rete, abbiamo deciso di non adottare la DTD già disponibile, che, a nostro parere, non sembra adatta a una gestione efficiente delle versioni. Ciò è principalmente dovuto alla complessità nella rappresentazione delle versioni attraverso più livelli di indirettezza.

Si è deciso di basare il modello su una organizzazione gerarchica del tipo *articolato – capo – articolo – comma*, particolarmente adatta a una codifica tramite XML, arricchendo questa struttura con metadati allo scopo di gestire gli aspetti temporali delle modifiche dei testi normativi e il loro versionamento.

Le dimensioni temporali che abbiamo ritenuto essenziali per la modellazione dei testi normativi sono le seguenti:

- Publication time: è il tempo in cui il testo normativo viene pubblicato sulla gazzetta ufficiale, ha la stessa semantica sia dell'event time che dell'availability time (in questo contesto le due dimensioni coincidono).
- Validity 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 ed è di durata indeterminata, a meno che diversamente disposto da un successivo atto. La semantica è quella del valid time in quanto 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 abrogata continui ad essere applicabile in alcuni casi, oppure che una norma venga 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. In questo caso il richiamo alla semantica del tradizionale transaction time è ovvio.

Le dimensioni temporali appena descritte, sono tra loro "ortogonali" e possono quindi essere trattate in maniera del tutto indipendente.

6.1.1. Il modello XML originario e la sua evoluzione

Come già accennato, XML è in grado di rappresentare con efficacia strutture gerarchiche, anche in relazione con metadati che associno agli elementi proprietà di vario genere. Tenuto conto di tutto ciò e considerato il tipo di documenti che dovevamo trattare, la scelta dello standard XML per la modellazione dei testi normativi è stata una conseguenza inevitabile, soprattutto se teniamo presente che

un altro degli aspetti fondamentali del nostro progetto è lo scambio di documenti tra diverse organizzazioni, probabilmente dotate di differenti sistemi software.
Il nostro modello rappresenta quindi un testo normativo attraverso uno schema XML e collega le disposizioni normative alle dimensioni temporali precedentemente introdotte. Una rappresentazione grafica è data in figura 6.1 (per la codifica si rimanda all'appendice B).
L'elemento *Legge* funge da radice, ed include, oltre a un attributo che identifica l'intero documento attraverso un numero, anche tre elementi: Natura, Titolo e Articolato. Quest'ultimo contiene il testo vero e proprio, strutturato secondo la gerarchia citata; un set di attributi specifica le coordinate temporali riepilogative dell'intero documento.
Ogni livello della gerarchia permette di rappresentare versioni multiple, ognuna caratterizzata dai propri attributi temporali, o timestamps. In accordo con la struttura gerarchica, gli aspetti temporali supportano l'ereditarietà, quindi i timestamps di un nodo N sono ereditati dai nodi figli a meno che non vengano raffinati. La raffinazione consiste soltanto nella riduzione dell'ambito di applicabilità temporale rispetto a quello del livello superiore e in alte parole non è possiblile che un elemento abbia coordinate temporali più ampie di quelle del nodo padre. L'unico attributo temporale che non viene ereditato è *pubblicazione*, esso rappresenta infatti una proprietà dell'intero documento, essendo la data di pubblicazione del testo originario, di conseguenza è associato soltanto all'elemento più esterno.
E' importante dire che grazie al prcoesso di ereditarietà e raffinazione, le diverse versioni di una disposizione normativa non possono sovrapporsi temporalmente.
Da notare anche la presenza dell'attributo *Rif*, esso specifica a quale disposizione normativa si deve la presenza della versione a cui è associato. La versione appartiene al testo originario se l'attributo Rif non è presente.

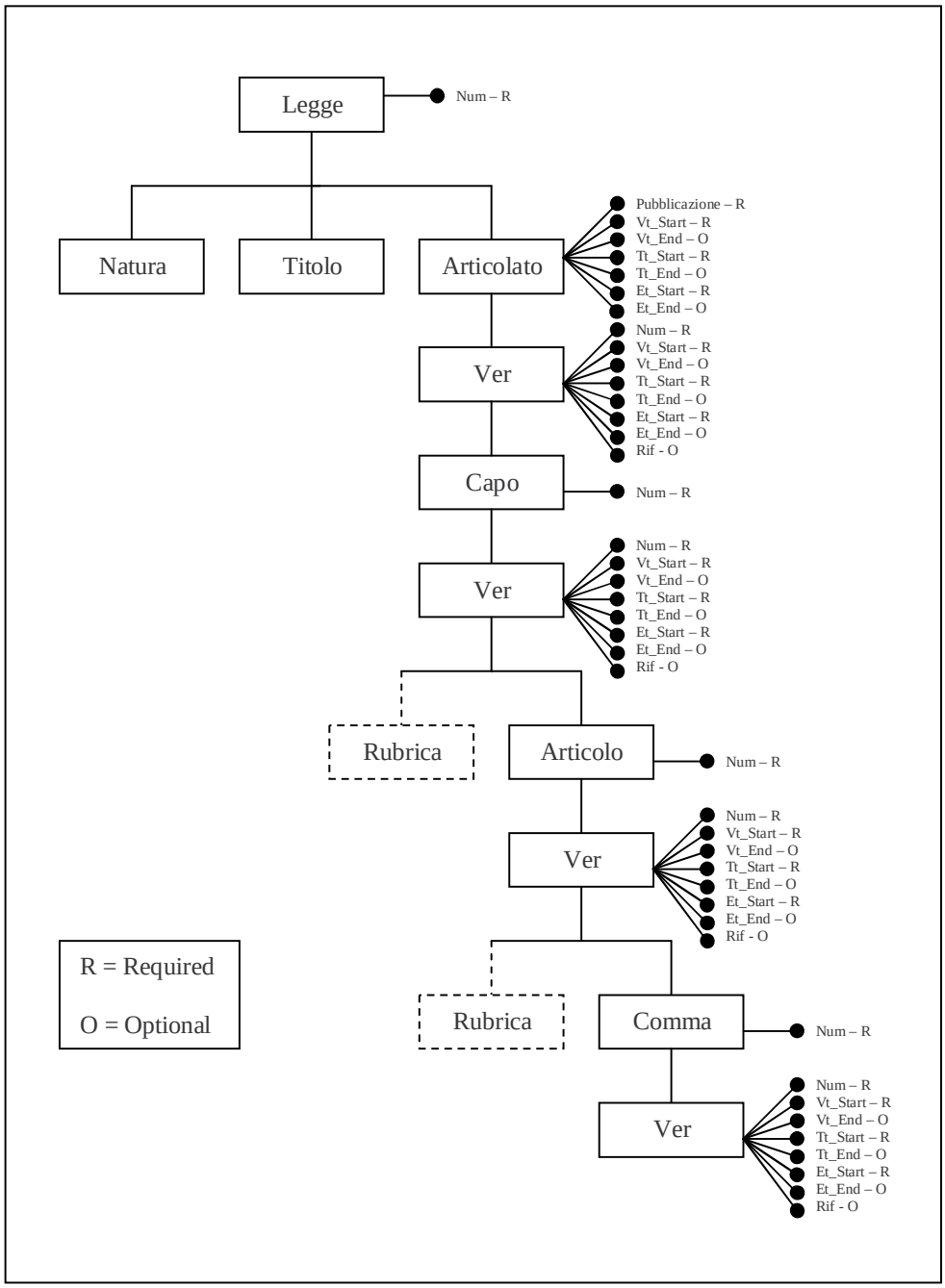


Fig. 6.1 – Modello di un testo normativo

Il modello appena descritto è stato modificato, durante la realizzazione del Sistema TeX, al fine di supportare gli *elementi temporali*. Si è deciso pertanto di associare a ogni versione più set di timestamps.

Questo accorgimento permette una gestione più efficiente e una sensibile riduzione dello spazio necessario per la memorizzazione di un testo normativo che abbia subito modifiche.

Ad esempio, nel caso in cui un articolo di una certa disposizione, in vigore dal 1/1/1990, venga sospeso per l'anno 2001, è sufficiente associare alla versione attiva l'opportuno elemento temporale, che conterrà due periodi, uno dal 1/1/1990 al 31/12/2000 e l'altro che partirà dal 1/1/2002. Col modello originale sarebbe invece stato necessario generare nuove versioni, con identico testo, una per ogni periodo temporale disgiunto, con un inevitabile spreco di memoria.

La figura 6.2 mostra come il modello viene modificato per supportare gli elementi temporali. Per la codifica XML si rimanda nuovamente all'appendice B.

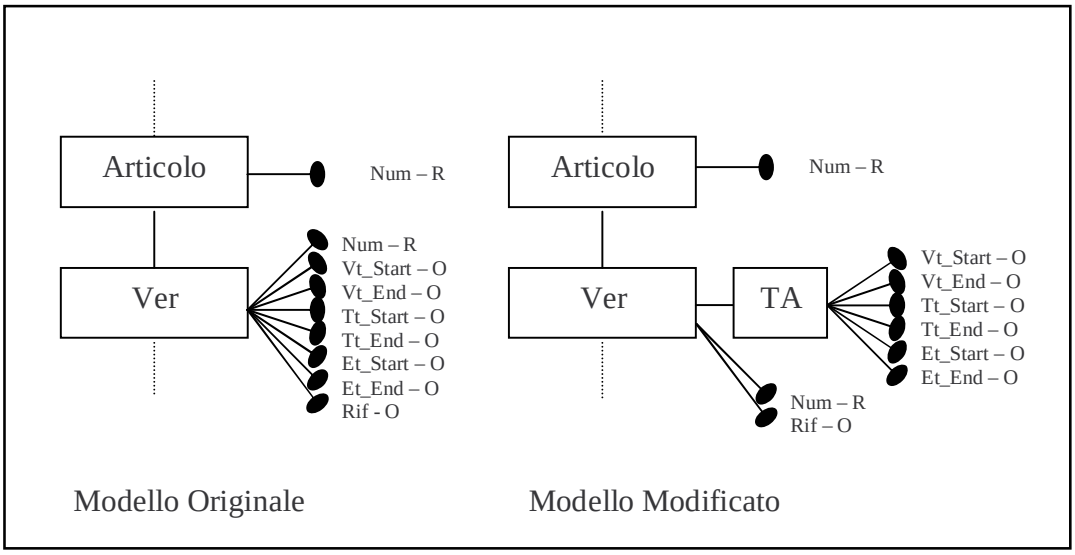


Fig 6.2 – Modelli a confronto

Ogni versione ha ora dei sottoelementi TA (temporal attributes), ognuno di questi identifica un periodo temporale disgiunto. Un modesto aumento della complessità del modello porta a un notevole miglioramento dell'efficienza.
La rappresentazione grafica di un documento conforme al modello modificato è riportata in figura 6.3.

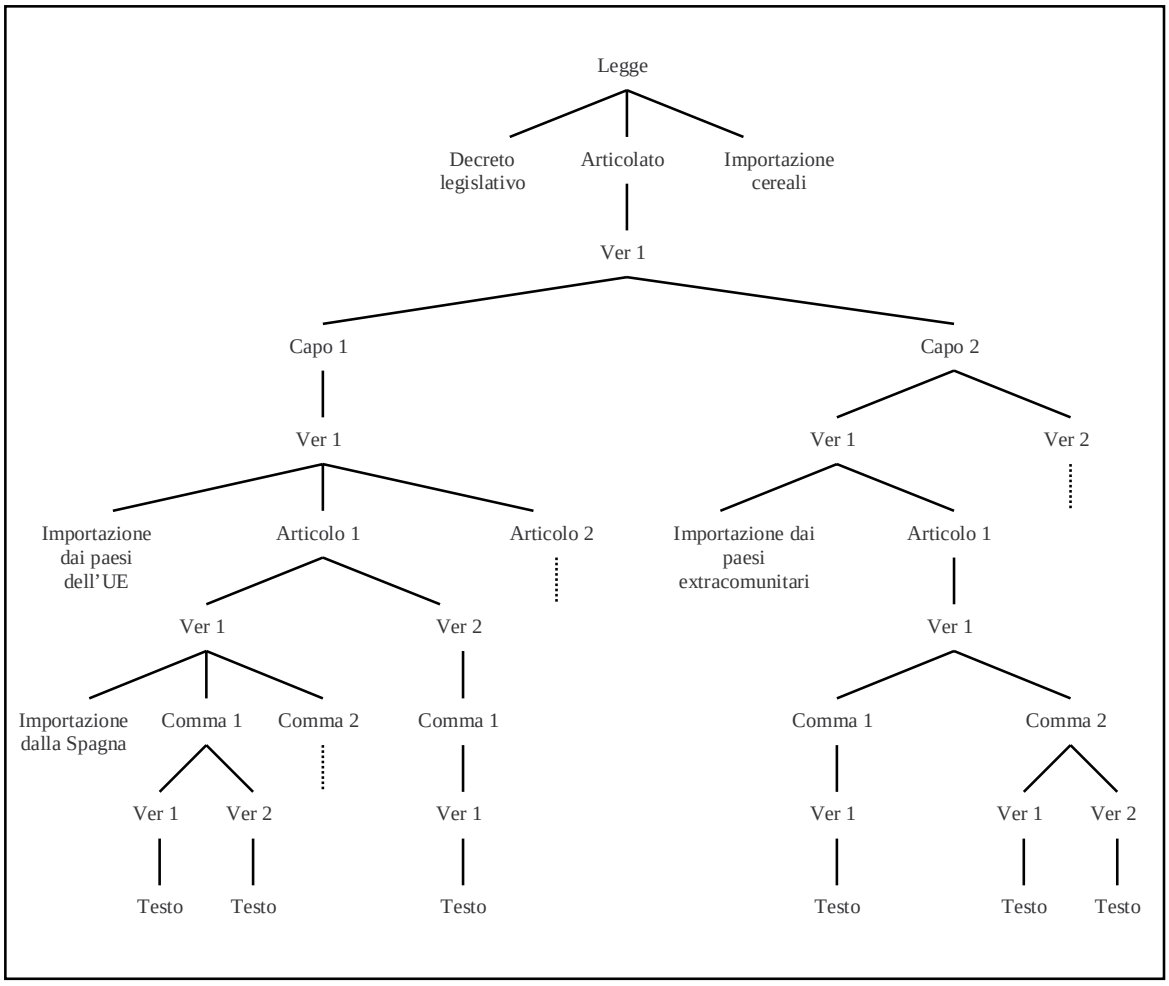


Fig. 6.3 – Un documento conforme al modello

6.1.2. Operatori

Sul modello sono previsti tre operatori per lo svolgimento delle attività di aggiornamento e consultazione dei testi normativi. La loro modellazione è stata uno dei punti cruciali del nostro lavoro, è stato infatti necessario raggiungere alcuni compromessi tra semplicità semantica degli operatori ed efficienza.

Gli aspetti gestiti dagli operatori sono la ricostruzione temporale dei testi normativi, le modifiche testuali e le modifiche temporali. Andiamo ora a descrivere in dettaglio queste operazioni.

6.1.2.1. Operatore Ricostruzione

L'operatore ricostruzione ha la forma:

Ricostruzione(en, vt_s, vt_e, et_s, et_e, pt_s, pt_e, tt, vt_{oper}, et_{oper}, p_{toper})

Il documento da ricostruire è identificato grazie al paramentro en, i parametri vt_s, vt_e, et_s, et_e, pt_s, pt_e, indicano invece l'inizio e la fine dei periodi considerati per le dimensioni temporali valid, efficacy e publication time, mentre tt specifica la data alla quale si vuole considerare la query, data che può essere passata, presente o futura.

Ad esempio porre il parametro tt al valore "1/1/2000" significa risolvere la ricostruzione di un documento selezionando le versioni correnti a quella data, che ovviamente possono differire da quelle attuali.

Infine i parametri vt_{oper}, et_{oper}, p_{toper}, specificano le operazioni da svolgere sulle dimensioni temporali corrispondenti.

Se non specificati, l'efficacy time viene preso uguale al valid time e per il transaction time viene considerata la data odierna.

Da notare che il valore dei parametri vt_e, et_e, pt_e, è preso in considerazione solo nel caso di operazioni che richiedano un periodo di tempo per la selezione delle versioni, negli altri casi non sarà svolto nessun controllo sul loro contenuto.

La ricostruzione consiste nel navigare attraverso l'albero che rappresenta il documento, alla ricerca delle versioni che rispettino i parametri. Per ogni disposizione normativa vengono infatti controllate tutte le versioni e soltanto quelle soddisfano le condizioni poste saranno restituite.

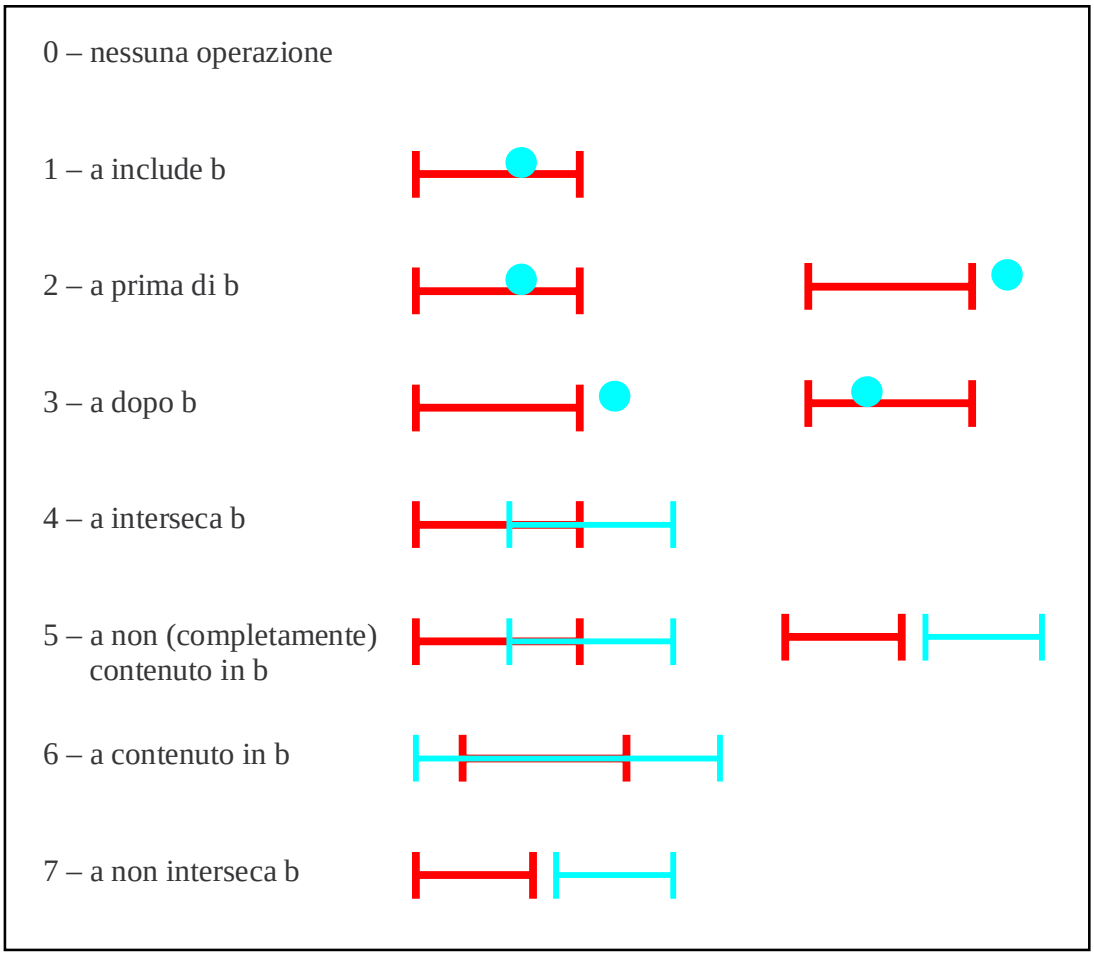


Fig. 6.4 – Operazioni previste per l'operatore Ricostruzione

Le operazioni[25] che si è deciso di supportare sono presentate in figura 6.4, in rosso sono raffigurate le coordinate temporali di versioni che soddisfano la condizione richiesta (a), in azzurro invece gli istanti o i periodi identificati dai parametri (b).
Quando vengono svolte sul publication time, le operazioni sono leggermente differenti, ciò e dovuto al fatto che tale dimensione supporta soltanto gli istanti temporali. Le coordinate delle versioni (in rosso in figura 6.5) si riducono quindi a punti.

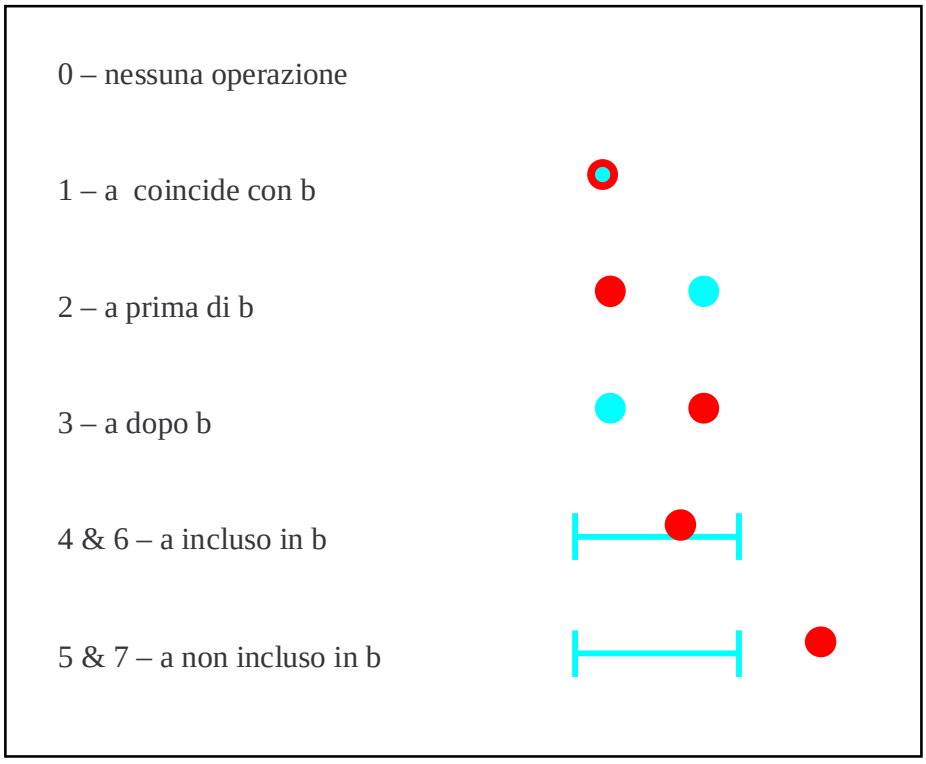


Fig. 6.5 – Operazioni sul publication time

6.1.2.2. Operatore Modifica_Testi

L'operatore `Modifica_Testi(en, vt`$_s$`, vt`$_e$`, et`$_s$`, et`$_e$`, txt, an)` agisce sulla disposizione normativa identificata dal parametro en, inserendone una nuova versione. Il testo e le coordinate temporali di questa nuova versione sono anch'essi ottenuti dai parametri, txt per il testo, vt_s, vt_e per il valid time e et_s, et_e per l'efficacy time, il transaction time è gestito automaticamente dal sistema. Viene inoltre aggiunto il riferimento alla norma attiva che ha provocato la modifica (parametro an).

La prima cosa da fare è quindi identificare la disposizione che deve essere modificata, il parametro en fornisce tutte le informazioni necessarie a questo scopo, si tratta infatti di un parametro composto che contiene, oltre all'identificativo del documento, anche il livello della gerarchia interessato dalla modifica (capo, articolo o comma) e una serie di indici che specificano la strada da percorrere all'interno della struttura ad albero del documento.

Una volta individuato l'elemento da modificare, si ottengono tutte le sue versioni attive (non chiuse rispetto al transaction time) e si verifica per ognuna di esse se i timestamps intersecano i tempi forniti dai parametri. Se viene rilevata intersezione la nuova versione può essere inserita, i suoi attributi temporali sono infatti calcolati dall'intersezione tra i parametri e gli attributi delle versioni esistenti.

L'aggiunta delle nuova versione non è però l'unica azione da svolgere, devono infatti essere aggiornati gli attributi temporali delle versioni modificate, e di tutti gli elementi che da esse discendono, in modo da mantenere la coerenza temporale dell'intero testo normativo, non devono mai presentarsi sovrapposizioni di tempi.

E' da sottolineare che l'operazione di modifica testuale non può ampliare o restringere le coordinate temporali complessive della disposizione normativa su cui agisce. L'ambito di applicabilità temporale deve rimanere invariato dopo la modifica, il frazionamento dovuto all'introduzione della nuova versione, non deve

pertanto produrne un ampliamento, ne tantomeno lasciarne porzioni “scoperte”, sprovviste cioè di un testo applicabile.
E’ importante ricordare che una modifica testuale, con testo nullo, equivale a una abrogazione della disposizione normativa su cui va ad agire.
Il diagramma a blocchi in figura 6.6 mostra la semantica dell’operatore Modifica_Testi in modo grafico. Di seguito sono riportati anche alcuni esempi allo scopo di chiarire definitivamente la dinamica delle modifiche testuali, per la semplice rappresentazione in due dimensioni delle versioni, si è deciso di considerare l’efficay time sempre uguale al valid time, le figure ritrarranno perciò solamente due dimensioni temporali.

Consideriamo il seguente esempio:
la legge n°134 del 12 febbraio 1996, viene inserita nel database il giorno successivo e entra in vigore il 27 dello stesso mese. In data 1 dicembre 2000 viene deliberata una modifica temporale all’articolo 2 del primo capo. Il suo contenuto deve essere completamente sostituito da un nuovo testo che entrerà in vigore il 1 gennaio 2001 e ne uscirà alla fine dello stesso anno. L’operazione è svolta sul database il 3 dicembre 2000. Vediamo come evolvono le coordinate temporali delle versioni in gioco.
Dalla figura 6.7 si vede che, dopo la modifica, l’ambito di applicabilità temporale dell’articolo non è stato modificato nel suo complesso, è però stato frazionato dalla introduzione della nuova versione. Il database è ora in grado di fornire il vecchio o il nuovo testo in relazione alle coordinate temporali richieste dalle interrogazioni. Supponendo, ad esempio, che la disposizione normativa non abbia più subito modifiche, se interroghiamo il database in data 19 giugno 2003 chiedendo la versione attualmente in vigore otterremo il testo originario, se invece chiediamo

cosa era in vigore il 19 giugno 2001, avremo il testo della nuova versione come risultato.

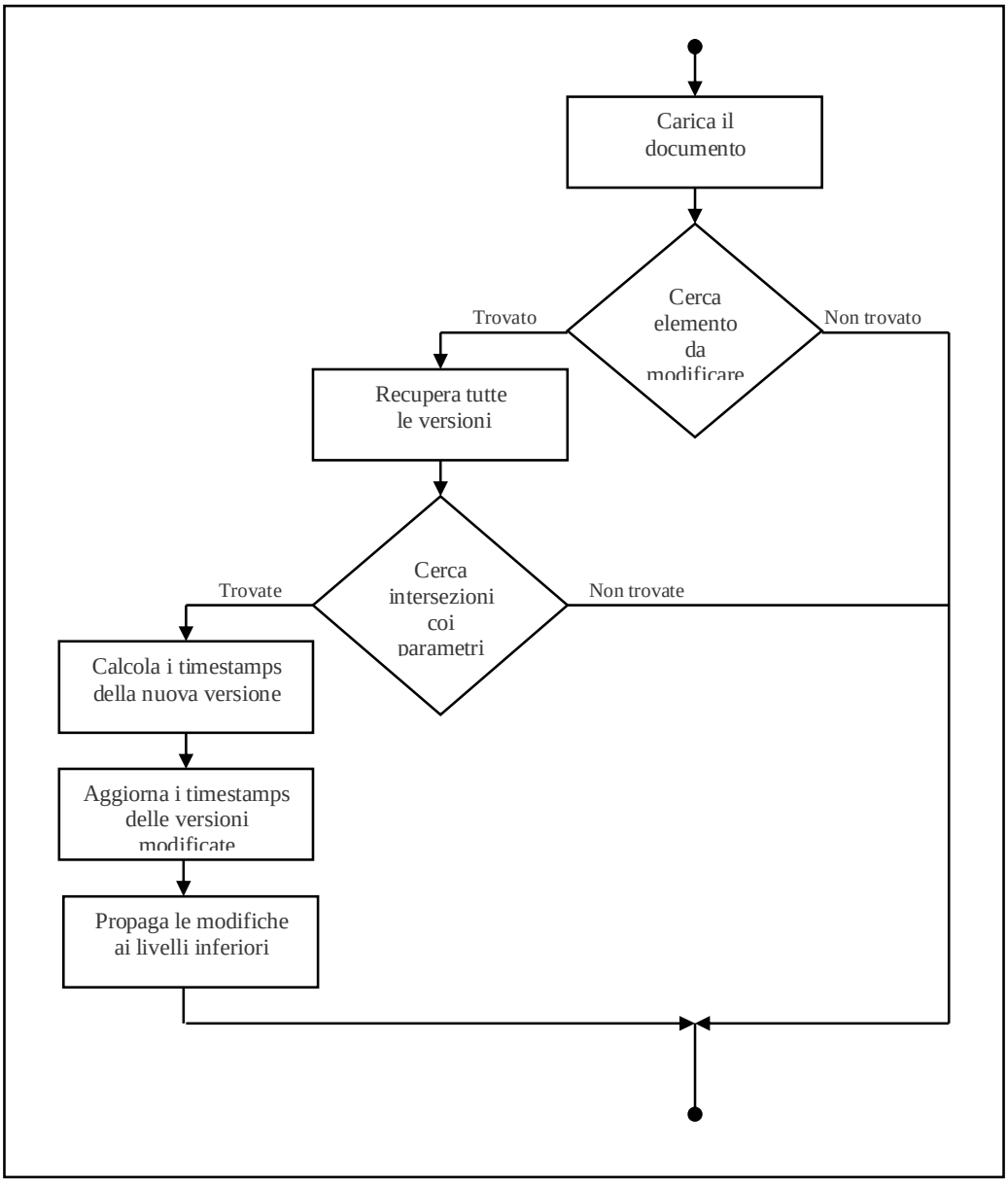


Fig. 6.6 – Dinamica delle modifiche testuali

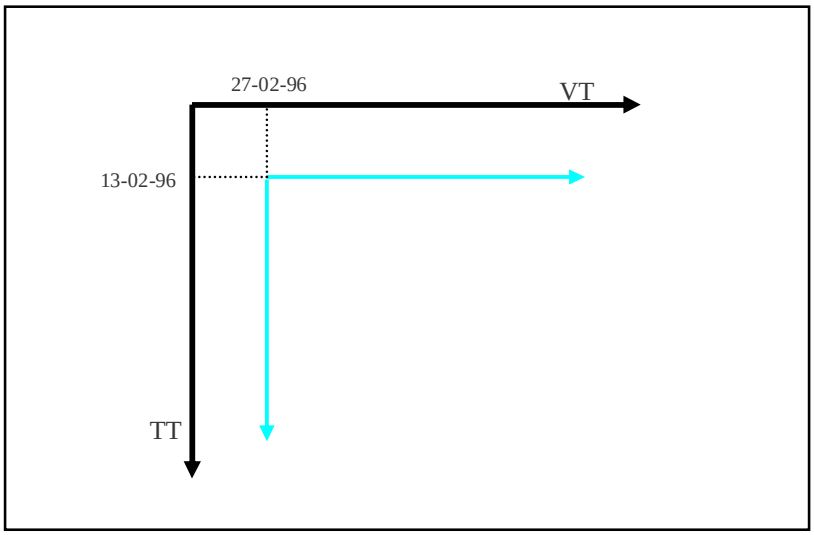


Fig. 6.7a – Ambito di applicabilità temporale dell'articolo 2 prima della modifica

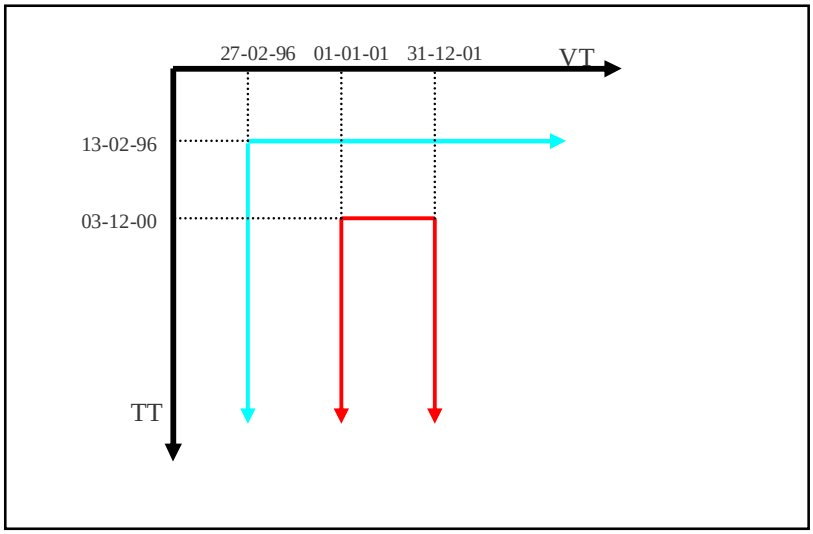


Fig. 6.7b – Ambito di applicabilità temporale dell'articolo 2 dopo la modifica

Vediamo un secondo esempio per chiarire definitivamente la dinamica delle modifiche testuali:

Una disposizione normativa, inserita nel database il 14 febbraio 2000 ha ambito di applicabilità temporale come in figura 6.8a. In data 7 luglio 2000 viene approvata una modifica testuale retroattiva al 1 marzo 2000. La disposizione deve quindi essere completamente sostituita da una nuova versione. La registrazione avviene il 10 luglio 2000.

Come si può notare in figura 6.8b, la modifica ha sostituito completamente la vecchia versione, ma il periodo tra il 1 marzo 2001 e il 1 marzo 2002 è rimasto escluso dall'ambito di applicabilità temporale della disposizione normativa considerata. Le interrogazioni che richiedano la versione in vigore in quel periodo non restituiranno nulla, indipendentemente dal transaction time.

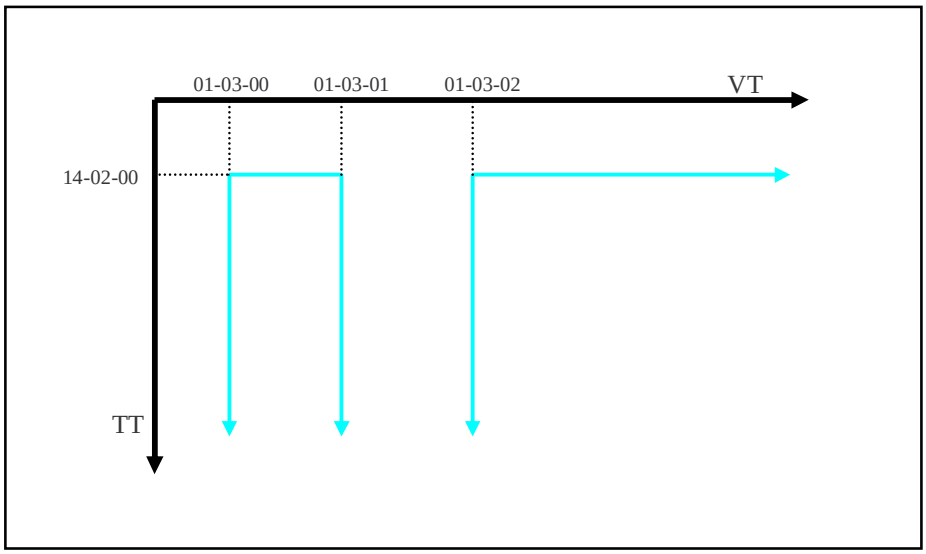


Fig. 6.8a – Ambito di applicabilità temporale prima della modifica

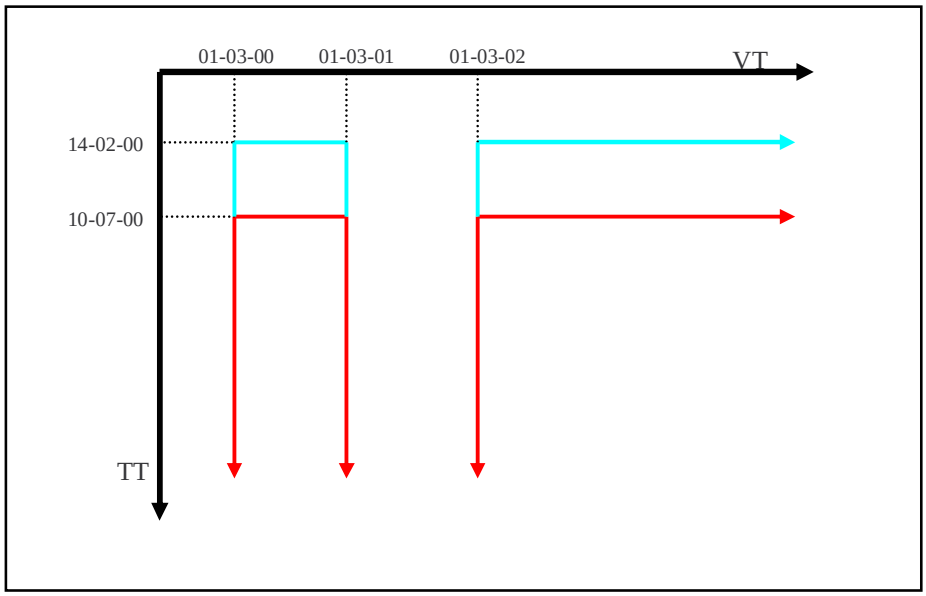


Fig. 6.8b – Ambito di applicabilità temporale dopo la modifica

6.1.2.3. Operatore Modifica_Tempi

L'operatore `Modifica_Tempi (en, vt, et, vt`$_s$`, vt`$_e$`, et`$_s$`, et`$_e$`, an)` è predisposto per la modifica delle coordinate temporali di disposizioni normative già esistenti. I parametri en e an hanno lo stesso significato di quelli dell'operatore Modifica_Testi e vt_s, vt_e, et_s, et_e sono le nuove coordinate temporali che saranno associate la versione dell'elemento da modificare identificata attraverso i parametri vt ed et.

In questi casi l'ambito di applicabilità temporale della disposizione dopo la modifica può chiaramente essere diverso da quello originale.

Diversamente dalle modifiche testuali, dopo aver trovato l'elemento oggetto dell'operazione, non se ne cosiderano tutte le versioni attive, ma solo quella che

comprende i valori specificati nei parametri vt ed et e transaction time equivalente a *now*, si tratterà sempre di una sola versione, non essendo possibili sovrapposizioni.
A questo punto viene aggiunto alla versione individuata un nuovo set di attributi temporali i cui valori sono ottenuti dai parametri dell'operatore, gli altri set di timestamps della versione sono adattati per evitare sovrapposizioni.
La conservazione della coerenza temporale dell'intero testo è molto più complessa rispetto alle modifiche testuali. Non solo i discendenti dovranno essere aggiornati, ma anche le altre versioni dell'elemento colpito eventualmente presenti dovranno subire le modifiche del caso, al fine di non introdurre sovrapposizioni temporali. Nel caso poi che la modifica vada a cambiare l'ambito di applicabilità temporale totale dell'elemento, la propagazione avrà effetti anche sui livelli superiori per rispettare il concetto di ereditarietà – raffinazione degli attributi temporali.
Per un riepilogo conciso di quanto appena descritto, si veda il diagramma a blocchi in figura 6.9.

Consideriamo il seguente esempio:
in data 12 novembre 2001 viene approvata una modifica sull'articolo 2 della legge n°134 del 12 febbraio 1996, il testo al momento in vigore deve essere prorogato per tutto l'anno 2002. La registrazione del fatto avviene il 15 novembre 2001. L'articolo, che ha già subito precedentemente una modifica testuale, ha ambito di applicabilità temporale come in figura 6.10a.
Come si può notare dalla figura 6.10b, non è soltanto la nuova versione (in rosso in figura) a veder cambiare le proprie coordinate temporali, ma anche la versione originale deve adattarsi alla modifica e infatti nell'anno 2002, prima coperto dal testo originario, è ora in vigore la nuova disposizione.
Tutti i discendenti di entrambe le versioni dovranno essere controllati ed eventualmente aggiornati, il livello superiore in questo caso non subisce nessun

cambiamento in quanto l'ambito di applicabilità temporale dell'articolo 2 nel suo complesso rimane invariato.

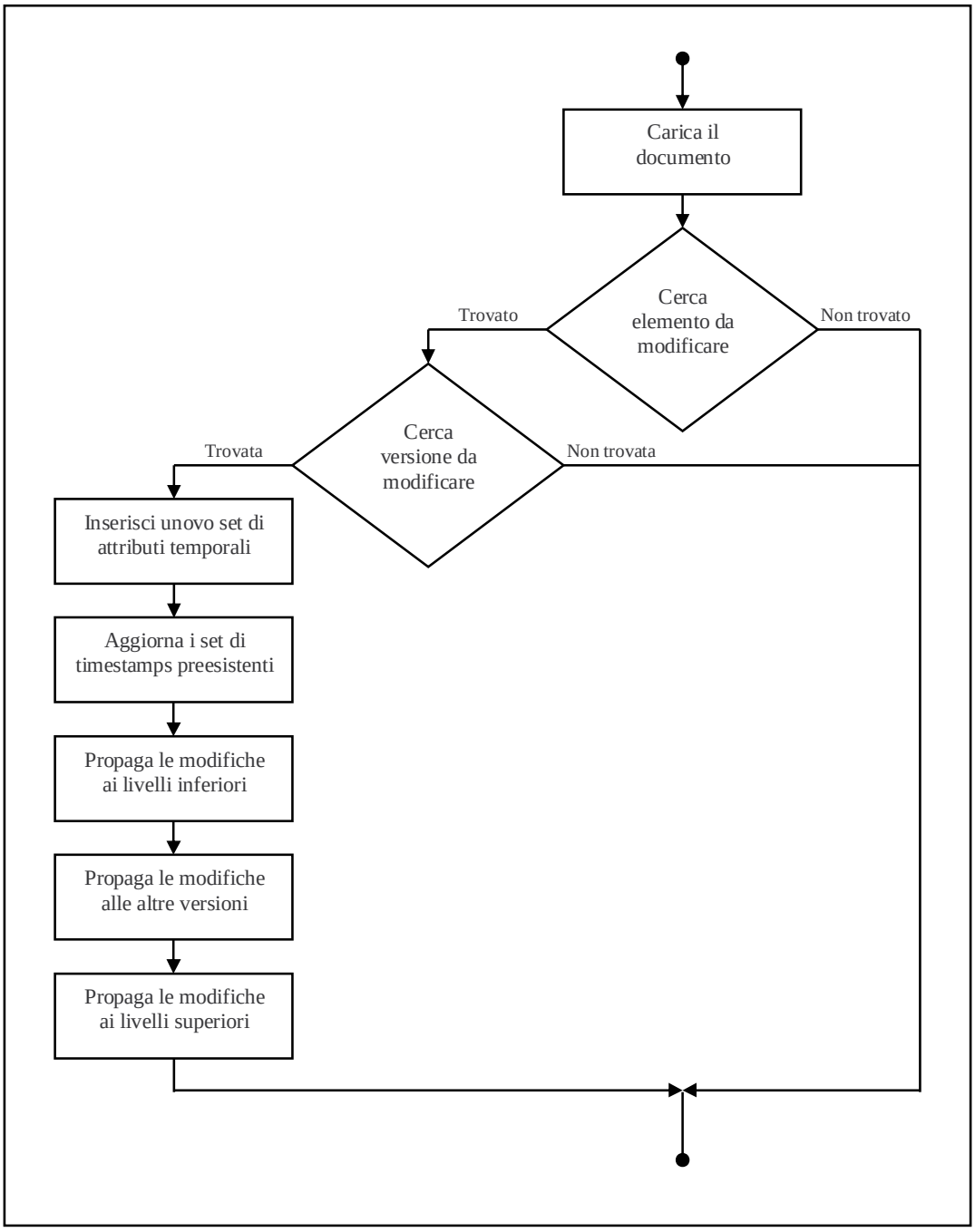


Fig. 6.9 – Dinamica delle modifiche temporali

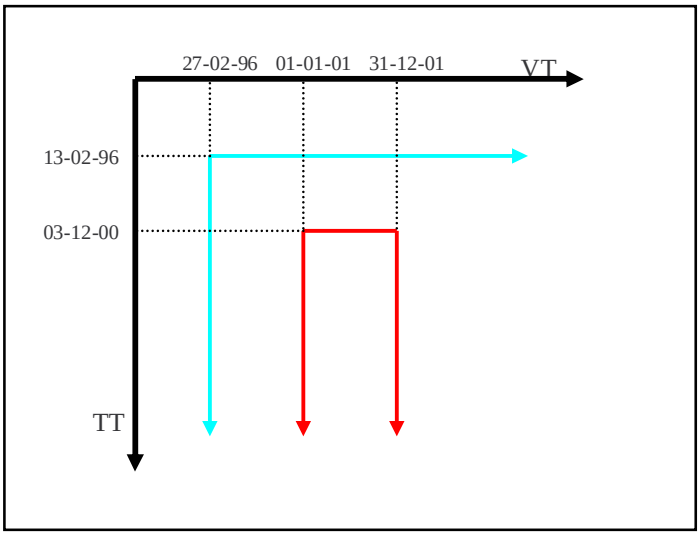


Fig. 6.10a – Ambito di applicabilità temporale dell'articolo 2 prima della proroga

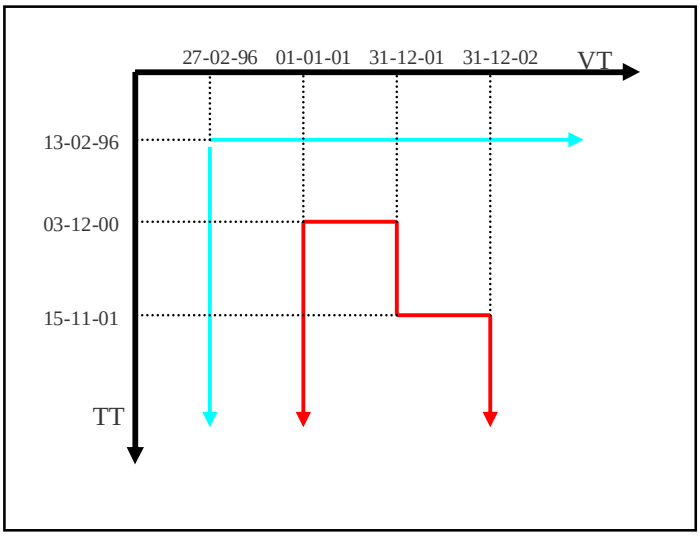


Fig. 6.10b – Ambito di applicabilità temporale dell'articolo 2 dopo la proroga

Uno degli aspetti fondamentali da tenere in considerazione è che in entrambi gli operatori di modifica, il transaction time è gestito in modo del tutto trasparente all'utente. Infatti le coordinate del transaction time per le nuove versioni o per i nuovi set di attributi da inserire sono calcolate automaticamente dal sistema, tt_s viene posto uguale al tempo in cui l'operatore è eseguito, (*now*), e tt_e rimane indefinito (U.C.).

6.2. Architettura del sistema

In questo paragrafo descriviamo l'architettura che è stata adottata per l'implementazione del nostro modello. Il sistema per la gestione dei testi normativi deve poter memorizzare i documenti in formato XML ed avere metodi di accesso efficienti attraverso interrogazioni che supportino vincoli temporali. Inoltre il sistema deve poter essere raggiungibile con facilità in rete attraverso una interfaccia web.

La tecnologia attuale fornisce due alternative per la gestione di grandi quantità di documenti XML: un server nativo XML e un DBMS dotato di strumenti per la gestione di dati XML. La prima opzione si basa su tecniche di accesso e memorizzazione ad-hoc per la gestione di documenti; anche se probabilmente in futuro potrà essere una valida alternativa, lo stato attuale della tecnologia non offre prodotti della stessa affidabilità e potenza dei collaudati sistemi relazionali. La seconda scelta considera i DBMS più largamente utilizzati, come Oracle, DB2 e SQL Server. Il sistema che abbiamo ritenuto adeguato è Oracle 9i, che con il pacchetto XML DB offre un esaustivo supporto a XML. Per un'analisi degli strumenti offerti da Oracle si rimanda al capitolo 5.

La scelta di utilizzare un comune DBMS relazionale è motivata anche dalla probabilità che una organizzazione sia già in possesso di un sistema del genere per altre attività, e quindi non sia costretta a rendere operativo un diverso sistema per la gestione dei soli testi normativi.
In figura 6.11 e descritta l'architettura del sistema TeX.

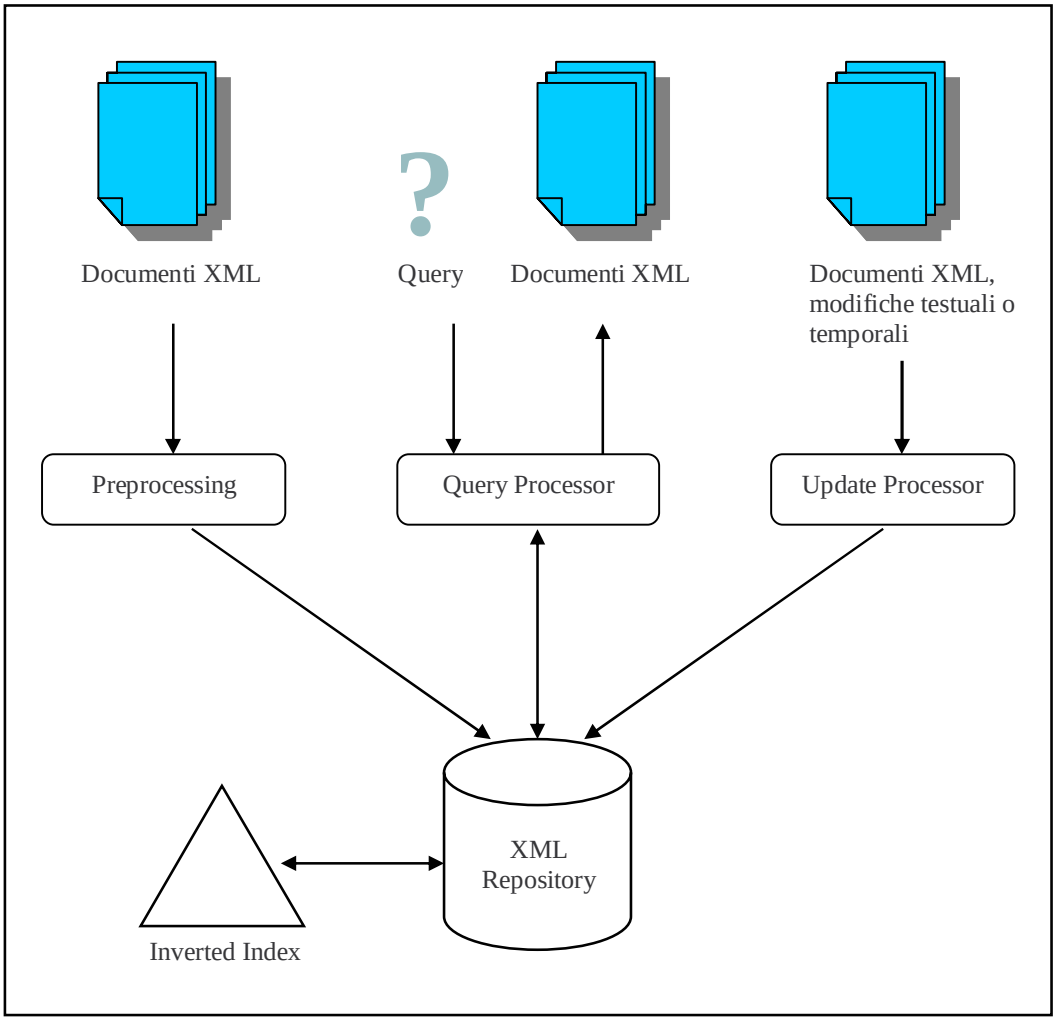


Fig. 6.11 – Struttura del sistema TeX

Tra i metodi di memorizzazione forniti da Oracle è stato scelto quello non strutturato (CLOB) Si è ritenuto infatti di dare fondamentale importanza alle prestazioni nel caso in cui si vogliano recuperare i documenti per intero.
I dati XML sono conservati in una colonna XMLType all'interno di una tabella, una riga per ogni documento. Per la struttura della tabella sono state considerate due alternative, la prima, alla quale ci riferiremo come *tabella XML*, considera soltanto una colonna dedicata a un codice univoco per ogni testo normativo e la colonna XMLType già nominata. La seconda invece, chiamata *tabella ibrida,* introduce colonne aggiuntive per la memorizzazione di metadati temporali al fine di velocizzare la ricerca dei documenti nelle interrogazioni temporali. Questi metadati non sono altro che la copia dei timestamps dell'elemento articolato, ovvero gli attributi temporali riepilogativi dell'intero documento.
Per le istruzioni per la creazione delle tabelle e la descrizione degli indici su esse costruiti si rimanda al paragrafo 6.3.3.

Allo scopo di migliorare ulteriormente le prestazioni del trattamento dei dati XML, in particolare nella fase di ricostruzione, si è inoltre deciso di esplicitare tutti i timestamps a ogni livello della gerarchia, anche quelli che possono essere derivati dai livelli superiori per ereditarietà.
Sia l'estrazione dei metadati per il modello ibrido, sia l'attività di esplicitazione dei timestamps sono eseguite dal modulo di *preprocessing* mostrato in figura 6.6.
Le interrogazioni sono invece eseguite dal modulo di *query processing*, che trasforma la richiesta dell'utente in una query SQL che filtra i documenti non compatibili con le condizioni richieste, condizioni che possono essere temporali oppure di inclusione di parole o di tipo di atto normativo.

In ogni caso, per ogni documento considerato ammissibile, si procede a una fase di ricostruzione per l'estrazione delle solo parti che soddisfino i vincoli temporali.
Dell'esecuzione delle modifiche, su testi normativi esistenti, si occupa il modulo denominato *update processor*.

6.2.1. La vita di un testo normativo nel sistema TeX

Con l'ausilio del formalismo degli activity diagrams UML, che ben si presta a rappresentare ad alto livello il lavoro svolto, puntando l'attenzione sulle attività più significative, andiamo ora ad analizzare come un testo normativo viene trattato dal sistema, a partire dal giorno in cui viene inserito fino alla data di abrogazione.

Il diagramma che si è deciso di utilizzare (Figura 6.12) è nella speciale forma *a corsie*, dette anche *swim lanes*, ciò permette di identificare i soggetti che svolgono le varie attività.
Le attività di consultazione e modifica del testo, hanno una durata non definibile a priori e, come indicato in figura, procedono parallelamente fino al momento in cui una particolare modifica, l'abrogazione totale, interrompe la vita del provvedimento. Da quel momento in avanti, il testo normativo sarà ancora consultabile, ma non potrà più essere modificato in quanto non più in vigore.
Una importante eccezione alla dinamica appena descritta si ha quando sulla disposizione normativa viene ad agire una proroga impropria (riviviscenza o richiamo in vigore), che ne ripristini il vigore. In questo caso, lo stato del documento torna ad essere quello precedente all'abrogazione.
Va inoltre ricordato che, anche dopo essere stato abrogato, un provvedimento potrebbe essere ancora efficace in alcune particolari circostanze, e quindi ancora applicabile al caso concreto.

Nei paragrafi successivi considereremo più nel dettaglio le funzionalità dei moduli, in relazione anche con gli operatori precedentemente introdotti. Utilizzeremo ancora gli activity diagrams, esplodendo le attività del diagramma in figura 6.12.

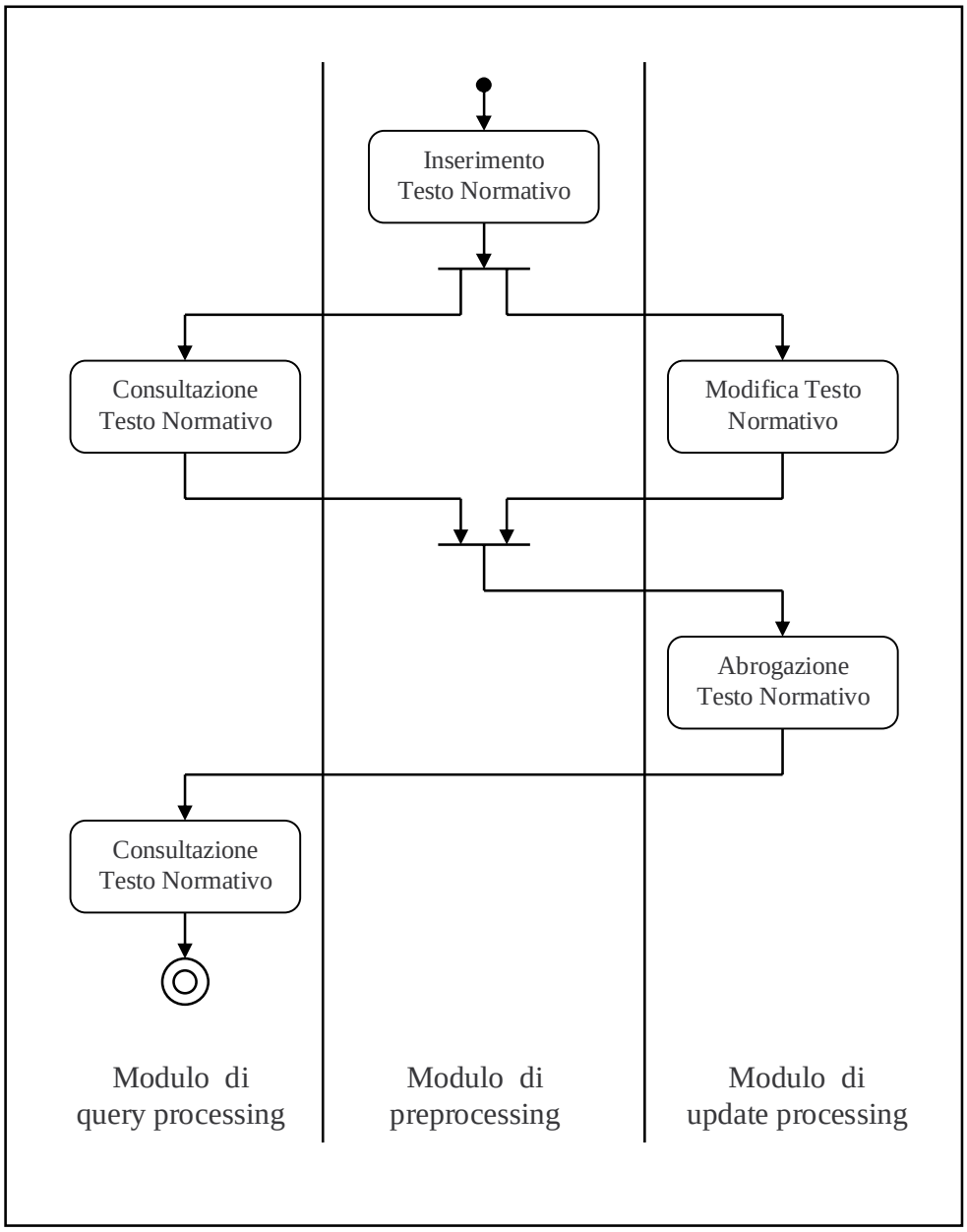


Fig 6.12 – Activity diagram: trattamento di un testo normativo

6.2.2. Il modulo di preprocessing

Come già accennato in precedenza il modulo di preprocessing è preposto al trattamento dei testi normativi prima dell'inserimento effettivo nel database.

In figura 6.13 mostriamo un semplice activity diagram che esplode, a un maggior livello di dettaglio, l'attività "Inserimento Testo Normativo" del diagramma precedente.

L'attività di esplicitazione dei timestamp ricopia gli attributi temporali dal livello N al livello N+1 se quest'ultimo non possiede già almeno un set di timestamps che raffinano quelli del nodo padre, in altre parole, esplicita a ogni livello della gerarchia, i timestamps che vengono ereditati dal livello superiore.

L'attività di estrazione dei metadati, che si occupa di copiare i timestamp dell'elemento articolato nelle colonne aggiuntive, viene ovviamente eseguita solo nel caso del modello ibrido.

6.2.3. Il modulo di query processing

Questo modulo gestisce gli aspetti che riguardano la consultazione dei documenti XML, si occupa quindi di ricevere le richieste dall'utente per poi preparare e inviare le query a Oracle.

L'attività di ricerca dei documenti conformi a quanto richiesto (Trova Documenti Compatibili) è svolta da Oracle, tutte le altre sono competenza del modulo di query processing.

In figura 6.14 vediamo nel dettaglio come si svolge l'attività di consultazione dei testi normativi.

L'attività *Ricostruzione*, utilizza l'operatore omonimo; in questa fase dell'analisi si è deciso di arricchire tale operatore, aggiungendo dei parametri che specifichino con quale criterio vengono scelte le versioni da restituire rispetto ad ogni dimensione temporale, oltre alla possibilità di porre condizioni anche sulla data di pubblicazione.

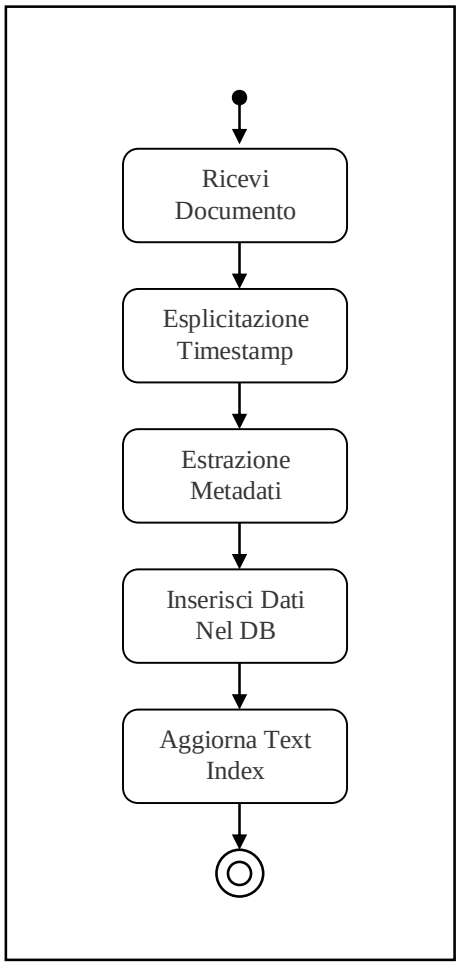


Fig 6.13 – Activity diagram: inserimento di un testo normativo

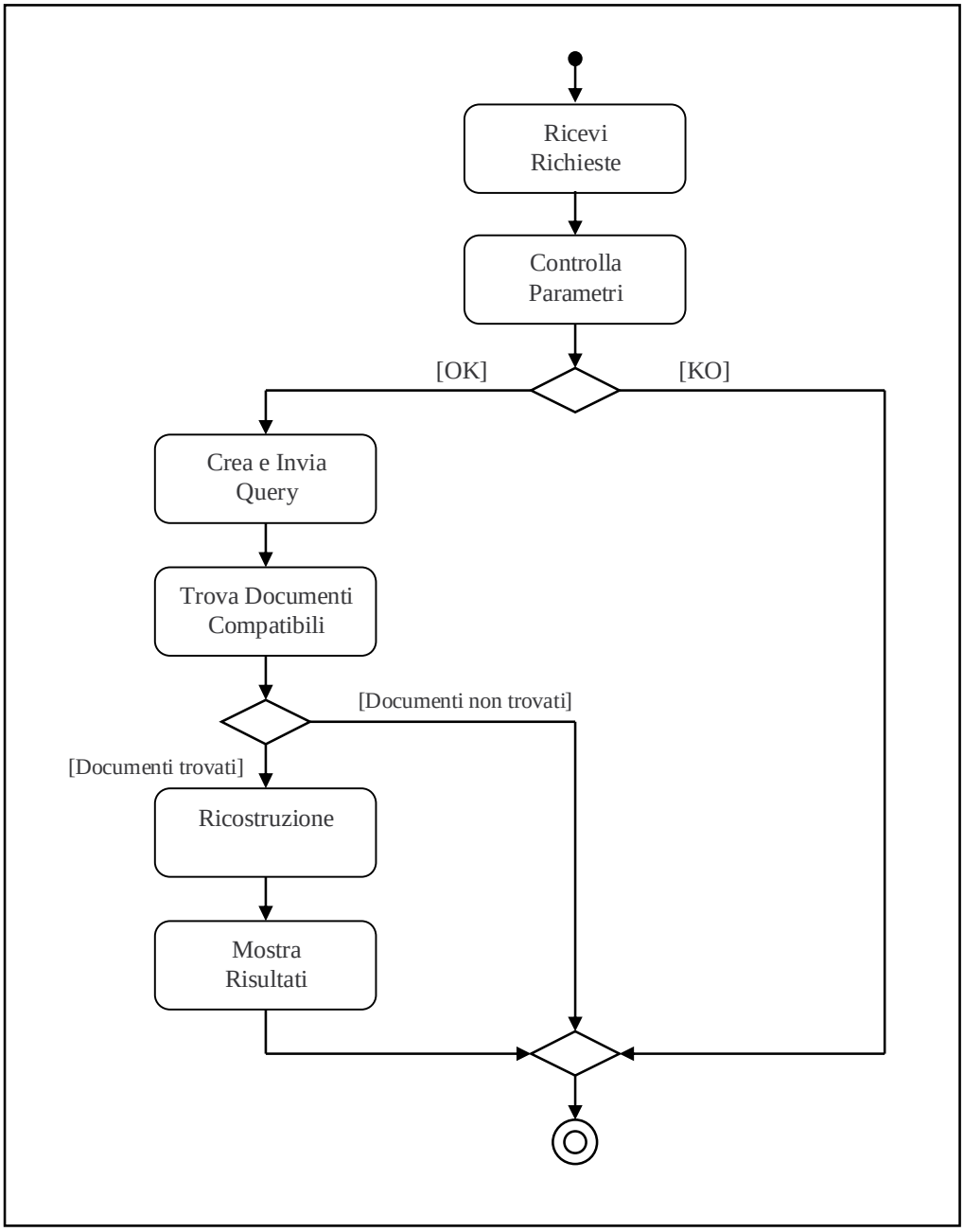

Ricevi
Richieste
Controlla
Parametri
[OK]
[KO]
Crea e Invia
Query
Trova Documenti
Compatibili
[Documenti non trovati]
[Documenti trovati]
Ricostruzione
Mostra
Risultati

Fig 6.14 – Activity diagram: consultazione di testi normativi

6.2.4. Il modulo di update processing

Le operazioni di modifica dei testi normativi sono svolte dal modulo di update processing.

Per l'attività di modifica dei testi normativi non presentiamo un activity diagram, in quanto sostanzialmente uguale a quello in figura 6.14.

Gli operatori di modifica dei testi e dei tempi sono utilizzati in questo modulo, attraverso il loro uso combinato è possibile effettuare ogni tipo di variazione sui testi normativi.

6.3. Il prototipo software

Lo scopo finale del nostro lavoro è stata la produzione di un prototipo che implementi il modello e gli operatori discussi. Come già accennato, si è deciso di sfruttare le potenzialità di Oracle 9i XML DB per quanto riguarda l'archiviazione e interrogazione dei testi XML e di costruire, al di sopra della base di dati, uno strato software che si occupi della gestione delle dimensioni temporali e del versionamento dei testi normativi.

Le scelte implementative sono quindi state fatte principalmente nell'ottica della portabilità e dell'utilizzo su Web, sia Java che XML sono infatti standard multipiattaforma ben integrati col mondo Internet. Per quanto riguarda le basi di dati relazionali e SQL, i prodotti attualmente sul mercato offrono alte prestazioni e semplici sistemi per collegarsi a Java e trattare dati semistrutturati.

Il prototipo è stato oggetto di due stesure di codice, la prima versione supportava solamente i periodi temporali. Durante il collaudo ci si è resi conto che ciò avrebbe

danneggiato le prestazioni del sistema in modo sensibile, perché i testi modificati avrebbero avuto un notevole grado di ridondanza di informazioni. Una seconda stesura del codice ha allora tenuto conto della necessità di supportare un tipo di dati temporale più espressivo: gli elementi temporali. Alcuni metodi hanno subito notevoli modifiche, anche se da un punto di vista di progettazione le variazioni non sono state degne di nota.

Procederemo ora all'analisi della fase di progettazione del prototipo attraverso un approccio ad oggetti; useremo il formalismo UML per schematizzare i concetti, presentando il modello statico, in cui elencheremo le classi che sono state create e ne daremo una dettagliata descrizione, e il modello funzionale, che servirà a chiarire quali siano appunto le funzioni che il prototipo fornisce all'utente.

6.3.1. Modello statico

Le principali componenti, o classi per usare la terminologia Java, che compongono il prototipo sono le seguenti:

- *Rebuild*: si occupa della ricostruzione delle norme secondo i parametri temporali richiesti; implementa quindi l'operatore Ricostruzione.
- *Modify*: classe che gestisce le modifiche a testi normativi esistenti, implementa gli operatori Modifica_Testi e Modifica_Tempi.
- *DBConnect*: tutte i metodi che interagiscono con Oracle sono in questa classe, che si occupa perciò delle connessioni al database, dell'inserimento e del recupero dei documenti e della gestione delle tabelle e degli indici.
- *Utility*: questa classe contiene i metodi di utilità generica maggiormente utilizzati.

Nei paragrafi successivi le analizzeremo una ad una descrivendone i campi e il funzionamento dei metodi.

Esitono poi alcune altre classi di importanza secondaria per le quali è sufficiente una breve descrizione:

- *AddAttributes*: classe utilizzata per l'esplicitazione dell'ereditarietà degli attributi temporali a ogni livello della gerarchia
- *MyException*: dichiara un nuovo tipo di eccezione. Verrà usata nella modifica dei testi normativi.
- *Result*: classe che definisce un oggetto specifico per la restituzione dei risultati, prevede sempre la restituzione di una durata di tempo oltre ad altri dati.

L'interfaccia grafica è realizzata attraverso i seguenti frames:

- *MainFrame*: è la finestra iniziale del programma, permette di accedere agli altri frames.
- *FrameQuery*: da questa finestra è possibile generare le query da inviare al database, una volta eseguita un'interrogazione, i risultati sono riportati in un apposito pannello, dal quale è possibile scegliere i documenti da ricostruire.
- *FrameInsert*: frame che permette di iserire nuove leggi, decidendo quale tipo di ottimizzazione fare sugli indici.
- *FrameRebuild*: piccolo frame che permette di iserire il nome del file su cui salvare il documento di cui si è richiesta la ricostruzione temporale.
- *FrameError*: finestra per la comunicazione degli errori.

6.3.1.1. La classe Rebuild

La classe Rebuild implementa il ricostruttore Ricostruzione. Le costanti che la classe offre, specificano i valori interi associati alle operazioni che possono essere eseguite sulle dimensioni temporali. In questo modo nell'invocazione dei metodi

possono essere usati nomi simbolici che renderanno il codice considerevolmente più leggibile.

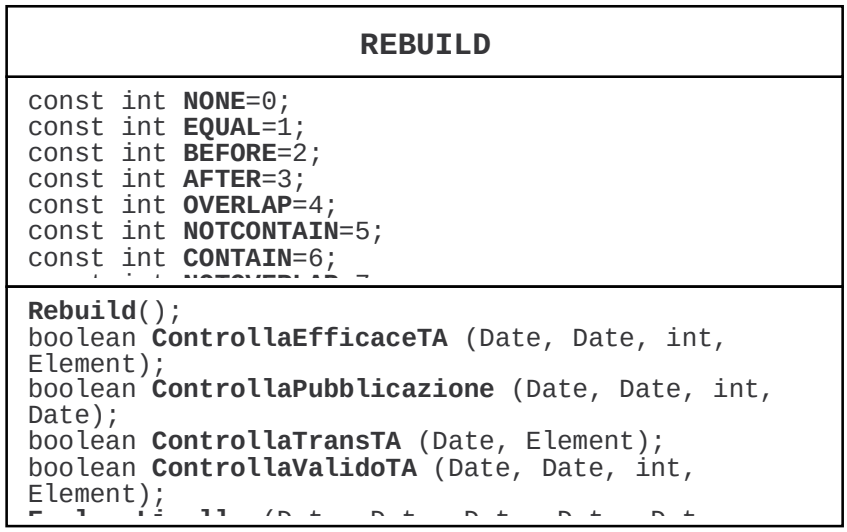


Fig. 6.15 – La classe Rebuild

Descriviamo i metodi che la classe implementa :

- **Rebuild**(): è il costruttore, rende possibile dichiarare oggetti di tipo Rebuild, sui quali eseguire gli altri metodi.
- boolean **ControllaEfficaceTA**(Date, Date, int, Element): questo metodo controlla se l'efficacy time dell'elemento passato come parametro è compatibile con le date e l'operazione specificate. Restituisce *true* come risultato se il controllo ha avuto esito positivo.
- boolean **ControllaPubblicazione** (Date, Date, int, Date): confronta una data passata come parametro con le altre due, sempre secondo il criterio fornito

dall'operazione specificata. Viene usato per controllare il publication time. Il valore restituito è *true* se il controllo è andato a buon fine.

- boolean **ControllaTransTA** (Date, Element): per quanto riguarda il controllo sul transaction time, eseguito da questo metodo, non è necessario specificare un'operazione, in quanto questa dimensione è interrogata in un solo modo. Il valore restituito ha lo stesso significato di quello dei metodi precedenti.
- boolean **ControllaValidoTA** (Date, Date, int, Element): è un altro metodo di controllo sugli attributi temporali, funziona esattamente come ControllaEfficaceTA, ma agisce sul valid time.
- **EsploraLivello** (Date, Date, Date, Date, Date, Date, Date, int, int, int, Element, int) : questo metodo è il cuore della procedura di ricostruzione temporale, esplora ricorsivamente la struttura gerarchica del documento alla ricerca degli elementi che non corrispondo a quanto richiesto. Il riconoscimento di tali elementi viene eseguito sfruttando i metodi di controllo sopra descritti, ogni elemento che non passa uno dei controlli, viene depennato dal documento originale.
- Result **Ricostruisci** (Document, Date, Date, Date, Date, Date, Date, Date, int, int, int): questo metodo si occupa di alcuni controlli preliminari sui parametri e poi avvia la procedura ricorsiva di ricostruzione. Al termine di essa prepara i risultati che saranno restituiti al chiamante attraverso l'oggetto Result.

La prima stesura del codice non prevedeva il supporto degli elementi temporali. Il metodo EsploraLivello ha infatti subito notevoli modifiche nella seconda versione del programma, infatti per depennare un elemento è ora necessario controllare tutti i

set di timestamps di cui dispone. Nella prima stesura ogni elemento aveva un solo set di attributi temporali.

I metodi di controllo invece, non hanno subito alcuna variazione

.

6.3.1.2. La classe Modify

La classe Modify si occupa di modificare testi normativi già esistenti, implementa quindi gli operatori Modifica_Testi e Modifica_Tempi. Da un punto di vista di complessità di programmazione, questa classe ha rappresentato una gran parte del lavoro svolto, ha anche subito notevoli modifiche in molti dei suoi metodi rspetto all prima stesura del codice, quando si è deciso di supportare gli elementi temporali. Anche in questo caso si è deciso di fornire delle costanti che permettano di usare nomi simbolici al posto di anonimi valori numerici.

MODIFY

const int **ARTICOLATO**=0;
const int **CAPO**=1;
const int **ARTICOLO**=2;
const int **COMMA**=3;

Modify(); boolean **CercaVersioni_Tempo**(Element, int, int, int[], Date, Date, Date, Date, Date); boolean **CercaVersioni_Testo**(Element, int, int, int[], Element, Date, Date, Date, Date); boolean **Contenuto**(Element, Date, Date, Date, Date); boolean **ControllaNuovoTa**(Element Ta); boolean **Intersezione** (Element, Date, Date, Date, Date); boolean **IntersezioneTa** (Element, Date, Date, Date, Date); Document **ModificaTempo** (int, int, int[], Date, Date, Date, Date, Date); Document **ModificaTesto** (int, int, int[], Element, Date, Date, Date, Date); Element **NuoviTa** (Element, Element, Date, Date, Date,

Fig. 6.16 – La classe Modify

I metodi della classe modify sono:

- **Modify**(): è il costruttore, rende possibile dichiarare oggetti di tipo Modify, sui quali eseguire gli altri metodi.
- boolean **CercaVersioni_Tempo**(Element, int, int, int[], Date, Date, Date, Date, Date): metodo fondamentale per le modifiche dei tempi, scansiona la strutture del documento in modo ricorsivo alla ricerca della porzione da modificare, dopo gli opportuni controlli esegue la modifica e ne propaga gli effetti.
- boolean **CercaVersioni_Testo**(Element, int, int, int[], Element, Date, Date, Date, Date): svolge le stesse funzioni del metodo precedente, ma per quanto riguarda le modifiche testuali.
- boolean **Contenuto**(Element, Date, Date, Date, Date): questo metodo controlla se un set di attributi è contenuto nelle date che riceve come parametri.
- boolean **ControllaNuovoTa**(Element Ta): questo metodo controlla che un nuovo set di attributi temporali creato da una modifica contenga dati coerenti, se così non è, il set viene eliminato. Può capitare infatti che venga generato un set che ha inizio e fine coincidenti, secondo una certa dimensione temporale, da un punto di vista logico tale set è corretto, ma non ha nessuna influenza pratica ed appesantirebbe il testo con dati inutili. Set del genere vengono perciò eliminati.

- boolean **Intersezione** (Element, Date, Date, Date, Date): controlla tutti i set di timestamps di un nodo cercandone almeno uno che intersechi i tempi forniti come parametri. Le dimensioni su cui lavora sono valid ed efficacy time.
- boolean **IntersezioneTa** (Element, Date, Date, Date, Date): come il metodo precedente, ma controlla un solo set di attributi temporali.
- Document **ModificaTempo** (int, int, int[], Date, Date, Date, Date, Date): questo esegue i preparativi per eseguire una modifica temporale ed avvia il processo ricorsivo.
- Document **ModificaTesto** (int, int, int[], Element, Date, Date, Date, Date): come il metodo precedente ma per le modifiche testuali.
- Element **NuoviTa** (Element, Element, Date, Date, Date, Date, Date): questo metodo si occupa della generazione del nuovo set di attributi temporali da assegnare a un elemento che abbia subito una modifica temporale. Una volta fatto questo, procede ad adattare gli altri set dello stesso elemento al fine di non avere sovrapposizioni.
- **PropagaGiu** (Element, int, Date, Date, Date, Date, Date): la propagazione verso i livelli inferiori delle modifiche è gestita da questo metodo.
- **PropagaLato** (Element, int, Element, Date, Date, Date, Date, Date): la propagazione delle modifiche alle versioni di un elemento che abbia subito una modifica temporale è compito di questo metodo.
- **PropagaSu** (Element, int, Date, Date, Date, Date, Date): metodo che si occupa di propagare le modifiche ai livelli superiori se l'ambito di validità temporale di un elemento ha subito un inceremento in seguito a una modifica temporale.

- boolean **Selezione** (Element Nodo, java.sql.Date TempoSel): cotrolla se almeno un set di timestamps del nodo passato come parametro contiene la data specificata.
- boolean **SelezioneTa** (Element Ta, java.sql.Date TempoSel): come il metodo precedente, ma lavora su un solo set di attributi temporali.

6.3.1.3. La classe DBConnect

Come già accennato, questa classe contiene tutti i metodi che interagiscono con il database. Le costanti qui definite sono usate nelle invocazioni dei metodi per la creazione degli indici.

I metodi che la classe DBConnect implementa sono i seguenti:

- **DBConnect**(): è il costruttore della classe, avvia la connessione ad oracle.
- Result **Cerca** (String, boolean, boolean): questo metodo invia una query ad Oracle e prepara i risultati richiesti, restituendoli attraverso l'oggetto Result.
- long **CreateIndexes**(int): la creazione degli indici sulle tabelle che contengono i testi normativi è gestita da questo metodo, il parametro indica su quale tabella andare ad agire. Torna come risultato il tempo dell'operazione.

DBConnect

const int **ALL**=0;
const int **XMLONLY**=1;
const int **IBRIDONLY**=2;

DBConnect(); Result **Cerca** (String, boolean, boolean); long **CreateIndexes**(int); **CreateTableLegge**(); **CreateTableLegge_I**(); **Disconnect**(); long **DropIndexes**(int); **DropTable** (String); long **GetMaxCod** (String); long **InsertLex** (String,int); long **InsertLex_I** (String,int); Document **LeggiLex**(int, String); long **OptimizeIndex**(int);

Fig. 6.17 – La classe DBConnect

- **CreateTableLegge**(): crea la tabbella del modello XML.
- **CreateTableLegge_I**(): crea la tabella del modello ibrido.
- **Disconnect**(): questo metodo effettua la disconnessione da Oracle, è essenziale per inviare al database il commit delle operazioni svolte.
- long **DropIndexes**(int): cancella gli indici.
- **DropTable** (String): cancella la tabella il cui nome è fornito come parametro.
- long **GetMaxCod** (String): questo metodo ottiene e restituisce il massimo codice inserito in una tabella, viene usato per inserire nuovi testi.
- long **InsertLex** (String, int): l'inserimento di un nuovo testo nella tabella del modello XML è eseguito da questo metodo. Restituisce la durata dell'operazione.
- long **InsertLex_I** (String, int): come per il metodo precedente, ma agisce sulla tabella del modello ibrido. Si occupa, perciò, anche di espicitare i dati temporali del livello più esterno, inserendoli nelle colonne relazionali.
- Document **LeggiLex**(int, String): legge un documento dalla tabella specificata e lo restituisce.

- long **OptimizeIndex**(int Quali): questo metodo è usato per l'ottimizzazione degli indici, il parametro specifica su quali indici andrà ad agire. Restituisce la durata dell'operazione.
- long **RebuildIndex(**int Quali): come il metodo precedente, ma esegue la ricostruzione degli indici.
- long **UpdateIndex**(int Quali): come per il metodo precedente, ma esegue l'aggiornamento degli indici.

6.3.1.4. La classe Utility

I metodi di utilià generica usati con maggior frequenza sono contenuti in questa classe. Sono tutti metodi statici, possono quindi essere invocati senza che venga creato nessun oggetto Utility.

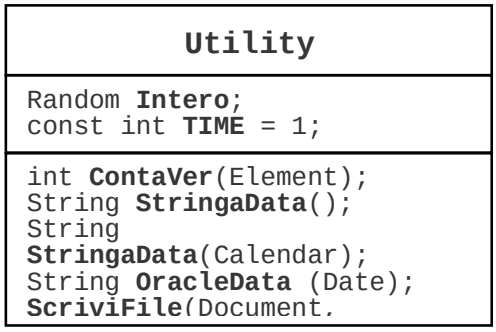


Fig. 6.18 – La classe Utility

La classe Utility mette a disposizione anche un oggetto e una costante: l'oggetto *Intero* è stato usato per la generazione di numeri casuali interi durante la creazione

di testi normativi di prova ; la costante, di nome *TIME*, viene uilizzata in tutto il progetto per scegliere l'ordine di grandezza in cui le durate delle elaborazioni devono essere restituite. E' stata usata principalmente nel collaudo del prototipo.

I metodi di questa classe sono:

- int **ContaVer**(Element): questo metodo conta il numero delle versioni dell'elemento passato come parametro.
- String **StringaData**(): restituisce, in formato stringa, la data odierna.
- String **StringaData**(Calendar): come il metodo precedente, ma restituisce la data ricevuta come parametro.
- String **OracleData** (Date): traduce una data nel formato di Oracle; restituisce il risultato in formato stringa.
- **ScriviFile**(Document, String): questo metodo scrive sul disco locale un file che contiene il documento ricevuto come parametro. E' possibile specificare il nome del file.
- Element **TrovaNum**(Element, int): questo metodo trova la versione identificata dal numero specificato dell'elemento ricevuto come parametro.

6.3.1.5. L'interfaccia grafica

Per non appesantire troppo la trattazione del modello statico, abbiamo deciso di non riportare descrizioni dettagliate della progettazione e della realizzazione dell'interfaccia grafica. Essa è stata creata principalemente per velocizzare la fase di test del sistema (di cui discuteremo nel prossimo capitolo). I frames principali sono infatti disegnati per consentire la creazione rapida di query da inviare al database e per l'inserimento in sequenza di nuovi testi normativi. Le figure 6.19 e 6.20 riportano rispettivamente i frames FrameInsert e FrameQuery.

Fig. 6.19 – FrameInsert

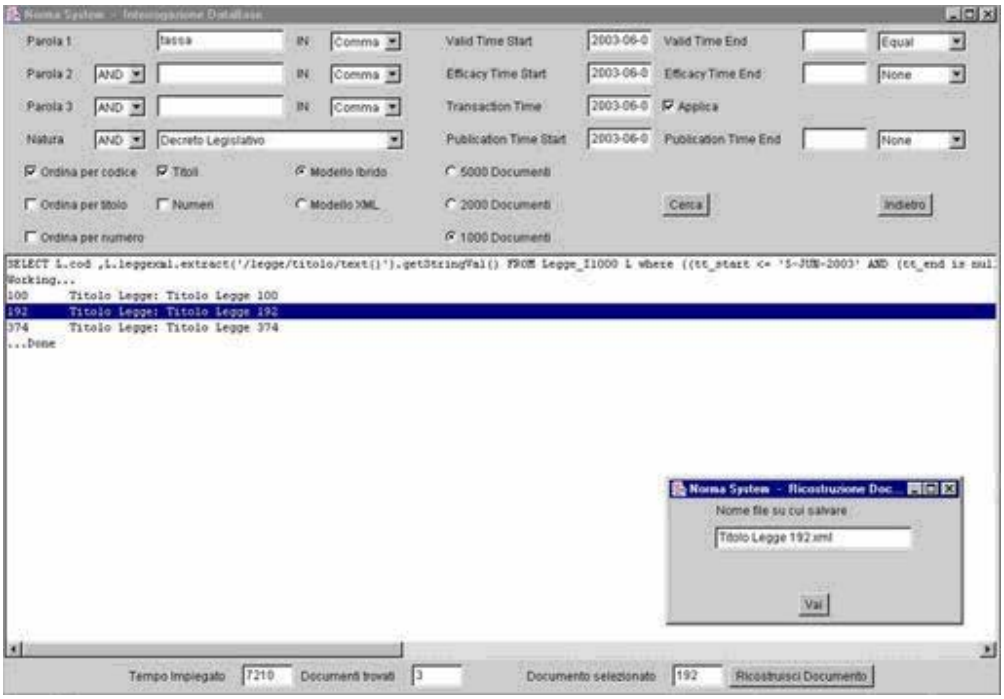


Fig. 6.20 - FrameQuery

6.3.2. Il Modello funzionale

Questo paragrafo spiega gli aspetti funzionali del prototipo software, in particolare analizzeremo come alcuni metodi fondamentali interagiscono con il database e con l'utente.

Ogni funzione richiede, infatti, dati in ingeresso e produce dati in uscita, queste informazioni possono essere scambiate con un agente esterno o con un deposito dati; nel nostro caso, si è scelto di illustrare queste dinamiche attraverso i DFD (*Data Flow Diagrams*), anche se questa tipoligia di diagrammi non rientra nello standard UML (erano contenuti nel modello OMT[26], predecessore di UML).

Le funzioni che vale la pena di analizzare sono l'inserimento di una nuova legge nella base di dati, la modifica di una disposizione (in questo contesto non fa nessuna differenza che si tratti di una modifica testuale o temporale) e la ricostruzione temporale di un testo normativo.

Le figure 6.21, 6.22 e 6.23 riportano i DFD delle tre funzioni.

I processi, contenuti negli ovali, corrispondono ai nomi dei metodi descritti nel modello statico, con una eccezione: l'interfaccia grafica è trattata come se fosse un unico metodo per rendere il diagramma più leggibile, anche se essa è in realtà formata da una collezione di frames e metodi.

Due aspetti da chiarire riguardo i diagrammi presentati sono le richieste che l'utente può inviare all'interfaccia e la presenza del file system come sorgente dati nel diagramma in figura 6.21.

L'utente può richiedere diverse operazioni attraverso l'interfaccia, in particolare durante la ricostruzione dei testi normativi si richiede di ottenere i documenti che rispono a certe caratteristiche, successivamente si ordina la ricostruzione di uno o più documenti tra quelli ottenuti dall'interrogazione.

Il motivo per cui la funzione di inserimento di un nuovo testo normativo sfrutta come sorgente dati il file system locale è che prima di entrare nel database i documenti sono sotto la forma di file XML sul disco locale o in arrivo dalla rete.
E' necessario ricordare che ne per la visualizzazione di un documento ricostruito, ne per l'esecuzione delle modifiche è stata prevista una interfaccia grafica.
I metodi Ricostruisci() e ModifcaTesto() usano, come specificato nella descrizione delle classi, molti altri metodi per ottenere i risultati, tali passaggi non sono stati inseriti nei DFD per migliorarne la leggibilità. Entrambi gli aspetti potrebbero essere analizzati a un livello di dettaglio superiore. Preferiamo però tralasciare tale passo perchè una volta avviati i processi non c'è più interazione ne con la base di dati ne con l'utente, il DFD risultante sarebbe, perciò, privo di significato.

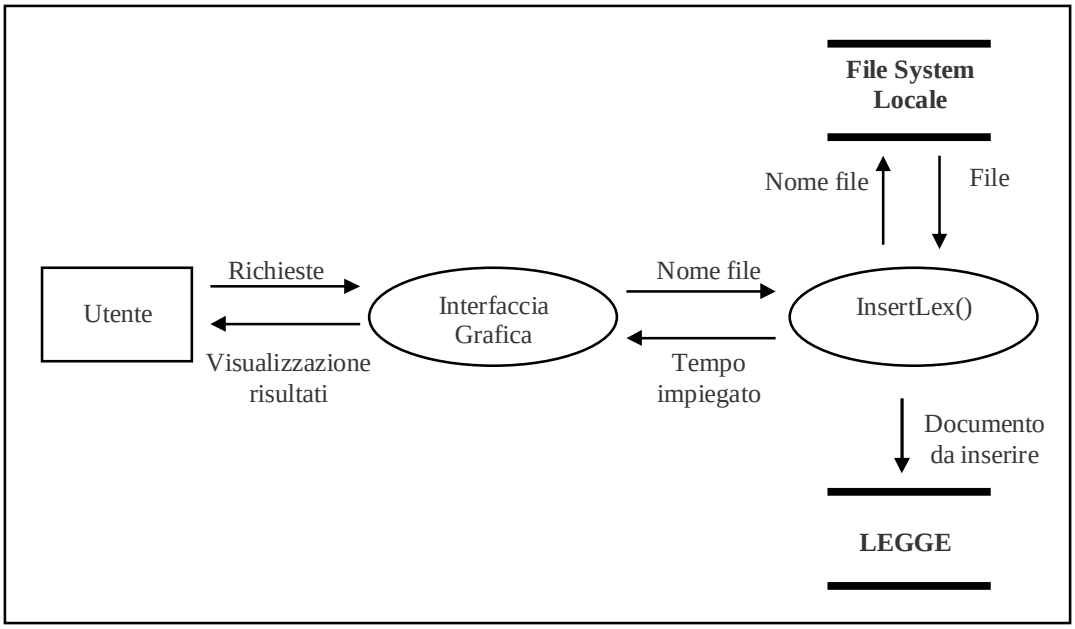


Fig. 6.21 – Data Flow Diagram: inserimento di un nuovo testo normativo

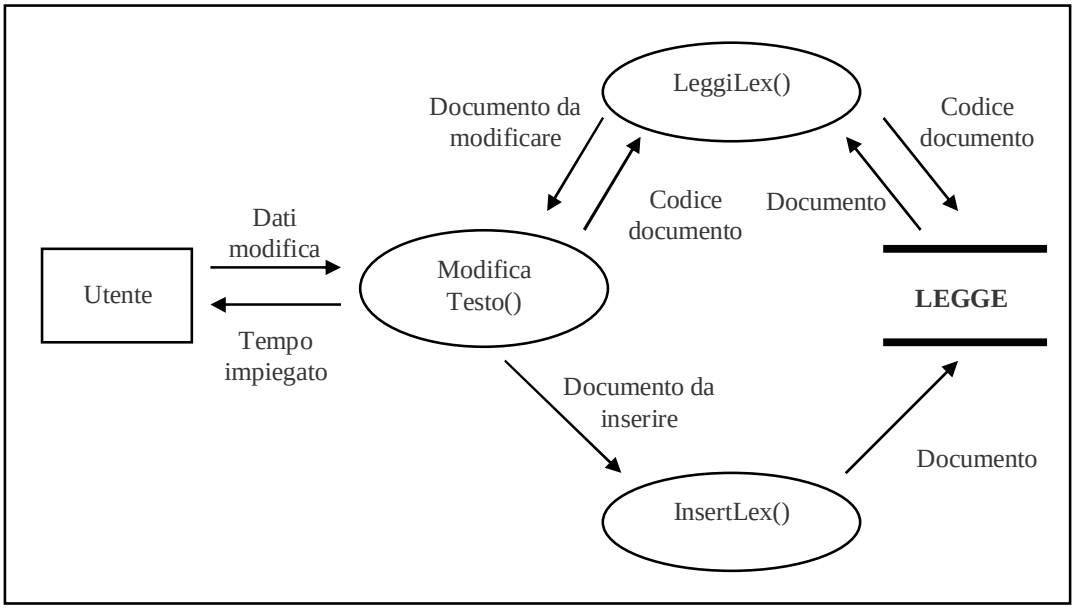
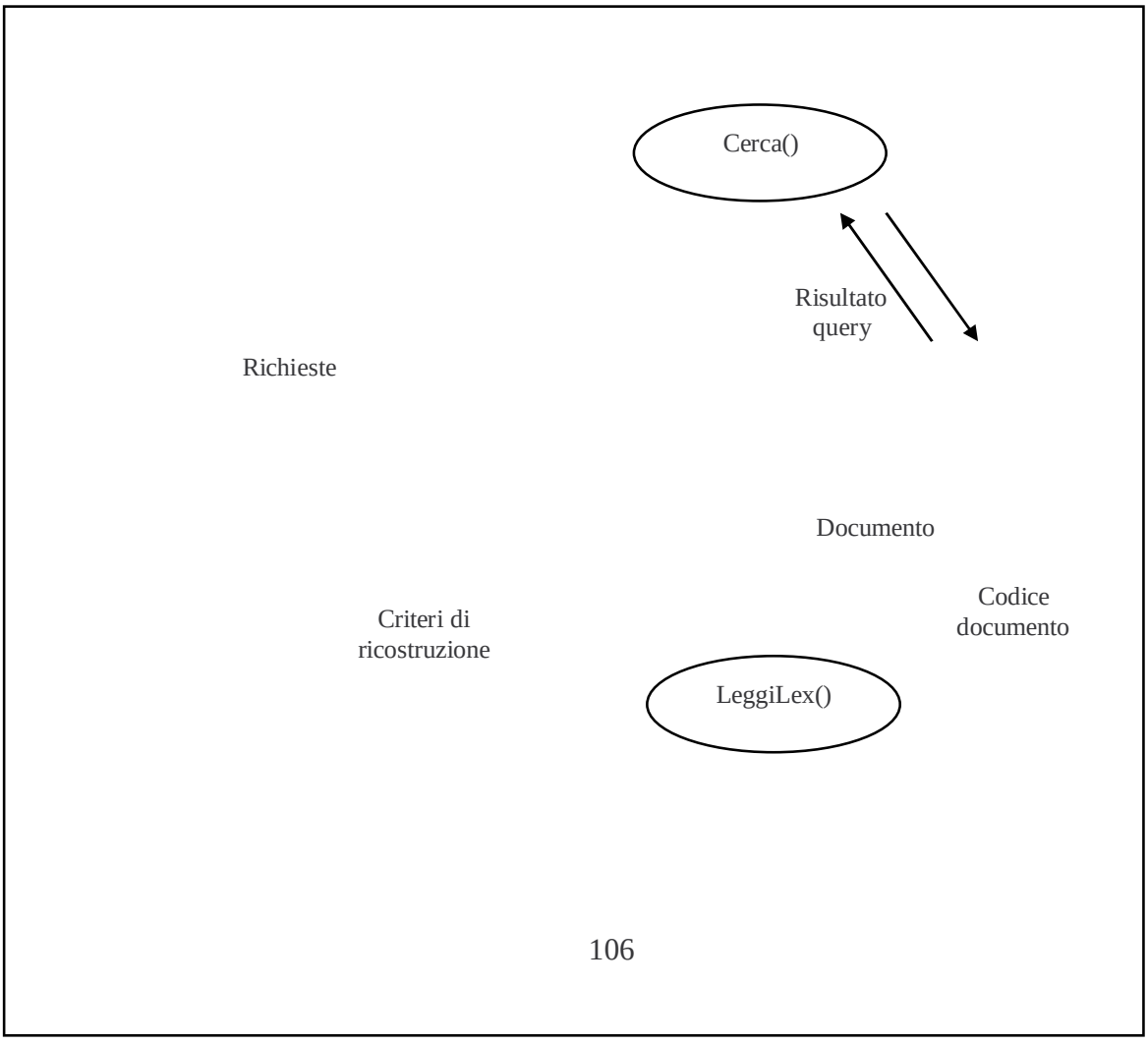


Fig. 6.22 – Data Flow Diagram: modifica di un testo normativo



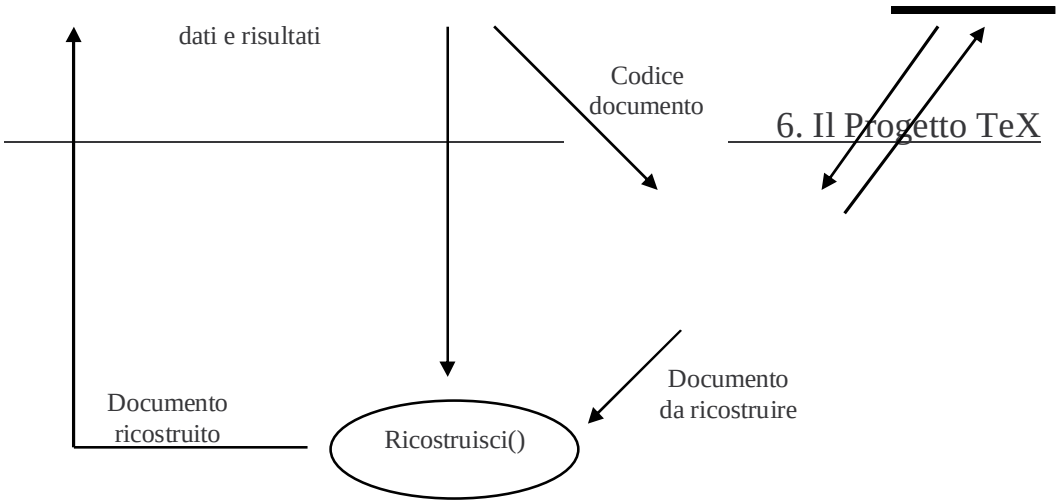


Fig. 6.23 – Data Flow Diagram: ricostruzione di un testo normativo

6.3.3. Le tabelle e gli indici utilizzati

In questo paragrafo sono presentate le tabelle utilizzate dal prototipo e i metodi di indicizzazione applicati.

Le tabelle con cui la classe DBConnect lavora sono essenzialmente due. Abbiamo già accennato che la prima, detta tabella XML, contiene soltanto un codice univoco e la colonna dati XML. La sua creazione viene eseguita con la seguente istruzione SQL:

```
CREATE TABLE LEGGE (
cod NUMBER (6) PRIMARY KEY,
leggeXML SYS.XMLTYPE);
```

La seconda tabella invece, alla quale ci riferiamo come tabella ibrida, dispone di alcune colonne relazionali che esplicitano i timestamps del livello più esterno del documento XML, oltre alla natura del testo normativo. La parola “ibrida” sta ad indicare che la nostra idea è di associare le qualità di XML alle prestazioni dell’ambiente relazionale. Come dimostreremo nel prossimo capitolo, questo accorgimento porta a un notevole risparimio in termini di tempo per la ricerca di documenti compatibili alle richieste dell’utente, in cambio solamente di un piccolo overhead in fase di iserimento dei testi normativi.

La seguente istruzione SQL crea la tabella ibrida.

```
CREATE TABLE LEGGE_I (
cod NUMBER (6) PRIMARY KEY,
leggeXML SYS.XMLTYPE,
pubblicazione DATE NOT NULL,
vt_start DATE NOT NULL,
vt_end DATE,
tt_start DATE NOT NULL,
tt_end DATE,
et_start DATE NOT NULL,
et_end DATE,
natura VARCHAR (100) NOT NULL);
```

Per quanto riguarda l'indicizzazione delle tabelle, gran parte del lavoro è svolto grazie a Oracle *inter*Media. Sulle colonne XML di entrambe le tabelle viene creato un *text index* con i parametri opportuni. Andiamo ad analizzare l'istruzione SQL che permetta la creazione di tale indice per la tabella ibirida:

```
CREATE INDEX ilegge_I ON legge_I (leggeXML)
 INDEXTYPE IS ctxsys.context
 PARAMETERS ( 'DATASTORE CTXSYS.DEFAULT_DATASTORE
 FILTER CTXSYS.NULL_FILTER
 SECTION GROUP xmlpath_I
 WORDLIST myword
 STOPLIST mystop');
```

Come si nota dal parametro Datastore, viene usata l'opzione di default per la memorizzazione dei dati XML, essi sono quindi registrati direttamente nella colonna indicizzata.

Il parametro Section_Group fa uso del gruppo di sezioni XML `xmlpath_I`. In questo gruppo sono considerate due sezioni, titolo e comma, in modo da poter mirare le ricerche dei termini al testo contenuto in una di esse e non all'intero documento. L'istruzione SQL che segue mostra come è possibile dichiarare un Section Group XML:

```
BEGIN
CTX_DDL.CREATE_SECTION_GROUP ('xmlpath_I','XML_SECTION_GROUP');
CTX DDL.ADD FIELD SECTION ('xmlpath I', 'titolo', 'titolo', TRUE);
CTX_DDL.ADD_ZONE_SECTION ('xmlpath_I', 'comma', 'comma');
END;
```

Per la tabella XML sarà prevista anche una sezione per la natura del documento.

L'unico altro parametro che richiede una breve discussione è Wordlist: attraverso esso è possibile specificare opzioni che consentono ricerce complesse sul testo. In

particolare si è deciso di abilitare sia lo stemmer che il fuzzy match e di indicizzare anche i prefissi dei termini. L'istruzione che imposta queste opzioni sia per la tabela ibrida che per la tabella XML è:

```
BEGIN
CTX_DDL.CREATE_PREFERENCE('myword','BASIC_WORDLIST');
CTX_DDL.SET_ATTRIBUTE('myword','STEMMER','ITALIAN');
CTX_DDL.SET_ATTRIBUTE('myword','FUZZY_MATCH','ITALIAN');
CTX_DDL.SET_ATTRIBUTE('myword','PREFIX_INDEX','YES');
CTX_DDL.SET_ATTRIBUTE('myword','PREFIX_MIN_LENGTH',3);
CTX_DDL.SET_ATTRIBUTE('myword','PREFIX_MAX_LENGTH',4);
END;
```

Per quanto riguarda la definizione della Stoplist, si rimanda ai listati del codice Java.

Gli unici altri indici previsti sono normali B-Tree sulle colonne `tt_start`, `tt_end` e `pubblicazione` della tabella ibrida.

C a p i t o l o 7

PROVE SPERIMENTALI E CONCLUSIONI

In quest'ultimo capitolo presentiamo i risultati sperimentali del collaudo del nostro prototipo; confronteremo le diverse alternative prese in esame in termini di prestazioni ed occupazione di memoria. In particolare, dimostreremo come il modello che supporta gli elementi temporali risulti essere una soluzione migliore rispetto al modello originale e che l'uso della tabella ibrida offre una maggiore efficienza globale.

Tutte le prove sono state eseguite su un personal comupter dotato di processore x86 Family 6 Model 7 con 256MB di RAM, sistema operativo Windows NT 4 Service Pack 6 e disco rigido SCSI; il DBMS usato è Oracle 9i Release 1.

Dopo l'analisi critica dei risultati, nell'ultimo paragrafo discuteremo le possibilità di evoluzioni future del nostro lavoro, anche in relazione con il costante sviluppo degli stumenti software presenti sul mercato.

7.1. Progetto degli esperimenti

Gli aspetti da indagare per poter valutare le differenze tra le alternative considerate durante la sviluppo del progetto TeX sono molteplici, si è percò deciso di suddividerli in tre categorie principali:

- selezione di testi normativi già memorizzati nel database;
- inserimento di nuovi testi normativi;

- ricostruzione/modifica di testi normativi.

Nelle prime due categorie si sono confrontate le prestazioni delle due tabelle proposte, in quanto, per il tipo operazioni svolte, la scelta del modello è ininfluente. Dove invece tale scelta ha il suo peso è nella terza categoria, oltre che nell'occupazione di memoria totale dei dati.

Per ogni categoria si è eseguito un notevole numero di prove per ogni alternativa, al fine di rendere il confronto il più obiettivo possibile. I dati presentati nei prossimi paragrafi sono tutti dati medi.

Per l'esecuzione delle prove sono stati generati in modo casuale dei testi normativi nei quali la coerenza temporale è sempre rispettata. Per quanto riguarda i testi dei commi si è deciso di crearli attraverso una procedura che inserisce casualmente parole significative. La dimensione media di questi documenti è di circa 22KB.

7.1.1. Selezione di testi normativi

Il primo aspetto che si è deciso di prendere in considerazione è l'efficienza nel recupero di testi normativi, che abbiano o meno subito variazioni, già presenti nella base di dati.

Da un punto di vista concettuale, l'interrogazione del databse normativo, avviene soltanto attraverso parole chiave. Si eseguono cioè query che selezionano i documenti che contengono certi termini; successivamente, sui testi restituiti verrà eventualmente eseguita la ricostruzione temporale che potrebbe far scompaire dal testo risultate le parole cercate. Al fine di velocizzare questo procedimento e di consentire all'utente di selezionare i documenti da ricostruire da un insieme più significativo, si è deciso di permettere l'inserimento nelle interrogazioni anche condizioni temporali, che saranno valutate considerando i timestamps del livello più esterno dei documenti. Ogni testo non compatibile con tali condizioni, se ricostruito temporalmente, non presenterebbe nessuna disposizione normativa applicabile, si

preferisce, perciò, scartarlo preventivamente, evitando così di portarlo in memoria centrale per la ricostruzione vera e propria.

In pratica quindi, ogni interrogazione che presenti condizioni temporali, esegue, oltre alla selezione, anche una fase preliminare di ricostruzione dei testi normativi

E' proprio quando questa ricostruzione preliminare è presente che l'uso della tabella ibirida consente prestazioni considerevolmente migliori rispetto a quelle ottenibili con la tabella XML.

I grafici in figura 7.1 mostrano i risultati dell'interrogazione del database attraverso entrambi i metodi su collezioni di 1000, 2000 e 5000 documenti; l'asse Y rappresenta il tempo in millisecondi.

Il confronto è stato effettuato considerando 6 diversi tipi di query:

- Le query 1 e 2 non contengono condizioni temporali.
- La query 3 presenta invece solo condizioni temporali.
- Le query da 4 a 6 hanno sia parole chiave che condizioni temporali.

Come si può vedere dalla figura 7.1, soltanto nelle query prive di condizioni temporali le prestazioni dei due metodi considerati sono paragonabili. Anche nella query 2, però, i risultati della tabella ibrida sono migliori, ciò è dovuto al fatto che in essa viene posta una condizione sulla natura del documento. Questa condizione è valutata per la tabella XML attraverso l'uso del text index, mentre per la tabella ibrida, che conserva tale dato in una colonna relazionale, mediante una ricerca in un indice B-Tree

Per rendere i grafici confrontabili si è deciso di tagliarli a 30 secondi. Infatti tutte le query con condizioni temporali, impiegano tempi anche superiori al minuto per essere eseguite sulla tabella XML. In particolare la query 3 viene svolta mediamente in 110 secondi nel caso di 2000 documenti e non si riesce a completare su 5000 per un errore di memoria generato da Oracle..

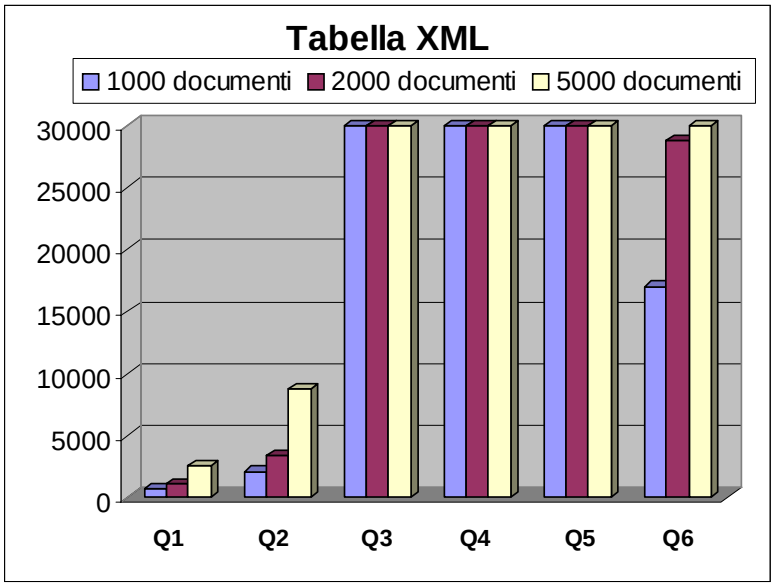


Fig. 7.1a – Selezione di testi normativi, tabella XML

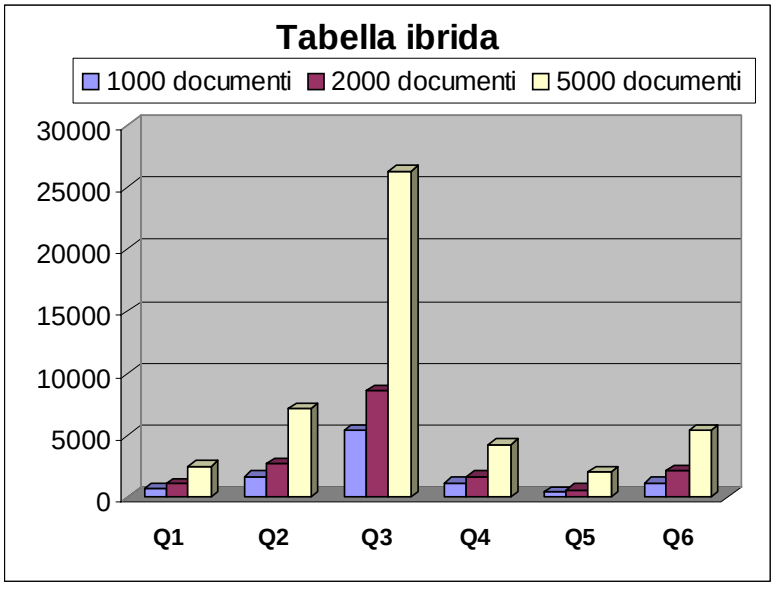


Fig. 7.1b – Selezione di testi normativi, tabella ibrida

Le prestazioni della tabella ibrida sono quindi mediamente molto migliori e, in alcuni casi, lo sono di più di un ordine di grandezza.

In tutte le interrogazioni è stato richiesto di ordinare il risultato per codice e di estrarre i titoli delle leggi selezionate per permettere all'utente di decidere quali andare a ricostruire. Quest'ultima operazione può diventare preponderante, in termini di tempo, rispetto all'effettiva selezione, soprattutto quando i documenti restituiti sono molti. Il grafico in figura 7.2 mostra, per la tabella ibrida nel caso di 5000 documenti, quanto tempo per ogni query è stato dedicato all'estrazione dei titoli. I documenti mediamente ritornati dalle query 2,3 e 6 sono rispettivamente 160, 133 e 1202, soltanto da 2 a 10 documenti invece per le altre interrogazioni.

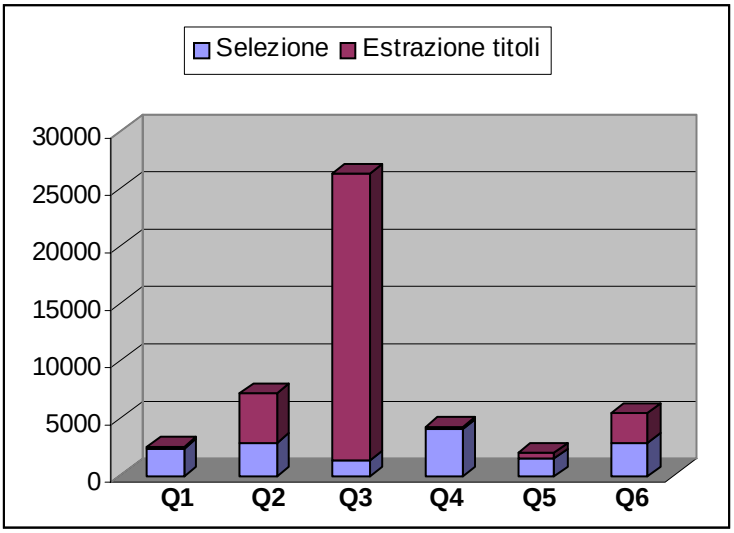


Fig. 7.2 – Tempo impiegato per l'estrazione dei titoli

Un altro importante dato che scaturisce dai risultati sperimentali è che le query eseguite più di frequente godono di una notevole diminuzione del tempo impiegato,

mediamente nell'ordine del 40-50%, con punte fino al 75%. Ciò è dovuto alle ottimizzazioni che Oracle gestisce automaticamente.

7.1.2. Inserimento di testi normativi

In questo paragrafo vediamo come la scelta della tabella infuisce sull'inserimento dei nuovi testi normativi. I risultati presentati in figura 7.3 mostrano anche il tempo necessario per l'aggiornamento del text index, senza il quale i nuovi documenti non sarebbero restituiti dalle interrogazioni. L'asse Y è anche in questo caso il tempo in millisecondi.

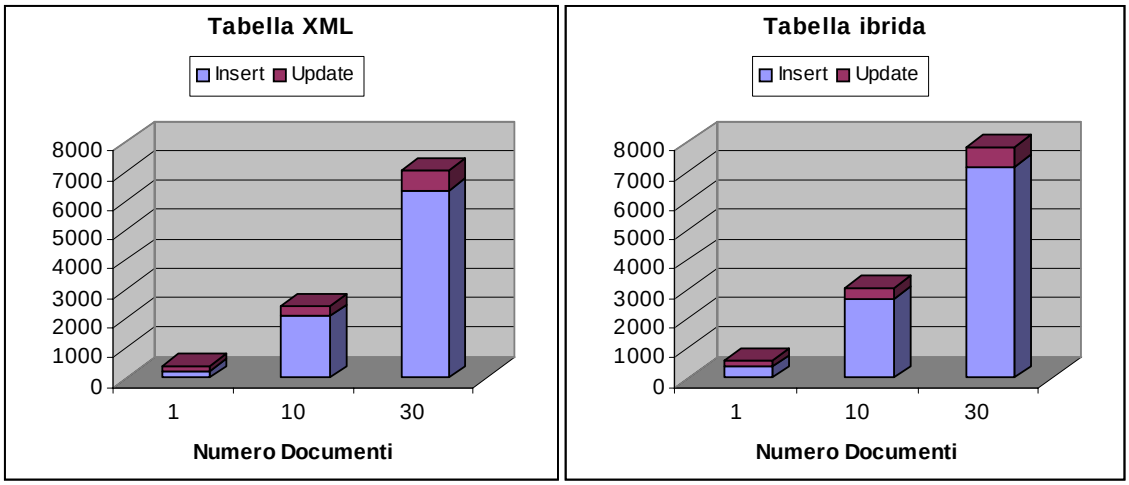


Fig. 7.3 – Inserimento nuovi testi e aggiornamento indici

Gli esperimenti sono stati svolti per gruppi di 1, 10 e 30 documenti che vengono inseriti in sequenza prima di aggiornare l'indice, la dimensione media è di 22KB per documento, gli stessi testi sono stati usati per entrambe le tabelle.
Si può osservare che il costo per l'aggiornamento è sempre meno influente più il numero di testi inseriti è grande e che non dipende dalla tabella utilizzata.

L'overhead per l'estrazione dei dati da inserire nelle colonne relazionali della tabella ibrida provoca un lieve incremento dei tempi per l'inserimento, che in percentuale si attesta attorno al 35%, nel caso più sfavorevole dell'inserimento di una sola legge alla volta e che cala fino al 12% inserendo 30 nuovi testi normativi. Comunque, la maggior frequenza delle operazioni di selezione rispetto agli inserimenti di nuovi testi rende questo dato più che accettabile.
Dopo un certo numero di inserimenti, il text index deve essere ottimizzato per evitare che le interrogazioni risentano della frammentazione provocata dalle operazioni di aggiornamento, i risultati sperimentali hanno dimostrato che l'ottimizzazione non è influenzata dalla tabella utilizzata e che nel caso di 5000 documenti richiede mediamente 11 secondi. La ricostruzione totale dell'indice, nelle stesse condizioni, richiede invece 1 minuto e 15 secondi.

7.1.3. Ricostruzione e modifica di testi normativi

L'ultimo aspetto che deve essere valutato è il tempo necessario alla riscostruzione e alla modifica di un testo normativo, in questo caso, la tabella utilizzata non influisce sulla durata delle operazioni, se non in minima parte, quando una modifica temporale propaga i suoi effetti fino al livello più esterno, provocando di conseguenza una variazione anche nei dati registrati nelle colonne relazionali della tabella ibrida.
La scelta del modello è invece di grande importanza, un testo normativo che abbia sostenuto diverse modifiche è risultato essere mediamente 3 volte più grande, in termini di occupazione di memoria, se conforme al modello originale rispetto allo stesso testo formattato secondo i canoni del secondo modello. Ciò è dovuto al fatto che il modello originale, che non supporta gli elementi temporali, provoca una certa ridondanza all'interno del documento, ogni modifica che va a frazionare il dominio temporale di una versione, provoca l'inserimento di altre versioni che contengono

tutte lo stesso testo ma con tempi diversi. Se invece gli elementi temporali sono supportati, il frazionamento dei tempi si risolve aggiungendo nuovi set di timestamps a una versione, che non viene così duplicata.

Anche se lo spazio occupato su memoria di massa non è un dato di particolare interesse, le diverse dimensioni dei documenti redatti in base ai due modelli ha altre conseguenze che possono influire sull'efficienza del sistema.

Durante le prove sperimentali la differenza di velocità di trattamento di testi basati sui due modelli, non è stata particolarmente apprezzabile, ad esempio i tempi medi per la ricostruzione del modello originale sono dell'ordine di 1,2 secondi, mentre per il modello modificato sono di circa 1 secondo; in entrambi i casi la varianza è molto grande, perché il tempo impiegato dipende sia dalle dimensioni del documento che dalle condizioni imposte per la ricostruzione.

E' però importante considerare l'occupazione di memoria centrale durante l'operazione, tenedo anche conto che la gestione di un documento XML in Java attraverso JDOM richiede anche fino a 3,5 – 4 volte la dimensione del documento. Il trattamento di testi di 250KB, comuni per documenti conformi al modello originario, necessita quindi di quasi 1MB di memoria centrale. Questo è un dato di grande importanza, perché un server adibito alle operazioni di ricostruzione e modifica, potrebbe vedere la sua memoria centrale in larga parte occupata dalle rappresentazioni JDOM dei documenti che tratta, e se tale memoria dovesse risultare insufficiente per tutte le operazioni, l'inevitabile ricorso alla memoria virtuale potrebbe portare a una notevole degenerazione delle prestazioni.

7.2. Conclusioni

Il prototipo presenta risultati incoraggianti. Soprattutto nella configurazione modello modificato – tabella ibrida gli incrementi dei tempi per le varie operazioni sono infatti sublineari. I valori assoluti di tali tempi sono accettabili, tenedo conto

che il computer su cui le prove sono state eseguite, una macchina non adatta ad agire come server di database, risentiva parecchio in termini di risorse utilizzate, dell'esecuzione di un DBMS complesso come Oracle 9i.

I possibili sviluppi futuri di un progetto come questo sono molteplici. Ad esempio una importante alternativa che si potrebbe considerare riguarda lo sfruttamento della seconda modalità di memorizzazione dei dati XML (modalità Object Relational), che permette di suddividere un documento in diverse tabelle seguendo uno schema XML. In questo caso, potermmo assitere probabilmente ad un calo nelle prestazioni degli inserimenti e del recupero degli interi documenti, ma la ricostruzione temporale, che avverebbe grazie all'uso di join strutturali, potrebbe rivelarsi più efficiente di quanto non sia la routine Java che ora si occupa di tale compito.

E' inoltre presumibile che nel prossimo futuro, i maggiori DBMS offrano estensioni per la gestione di alcuni aspetti temporali che sicuramente presenteranno migliori prestazioni di qualsiasi strato software aggiuntivo costruito "on top" a un DBMS. Sarà quindi possibile gestire almeno alcune delle dimensioni temporali di interesse senza bisogno di passare attraverso Java.

Un altro aspetto di interesse è l'eventuale sviluppo di un'interfaccia Web che permetta l'uso del sistema su Internet. Un software di questo tipo, infatti, che sfrutta una base di dati di dimensioni potenzialemente enormi, vede nel Web il suo sbocco naturale.

Il progetto TeX rappresenta quindi, a nostro avviso, un punto di partenza per lo sviluppo di un sistema in grado di aiutare concretamente nell'interpretazione dei testi normativi attraverso l'individuazione certa della versione temporalmente applicabile.

Appendice A

LISTATI JAVA

I listati del prototipo realizzato sono riportati di seguito. Per via della lunghezza del codice, si è deciso di selezionare soltanto le parti più significative, escludendo in particolare le classi create per l'interfaccia grafica.

```
package Principale;

import java.io.*;
import java.sql.*;
import java.util.*;
import org.jdom.*;
import org.jdom.input.*;
import oracle.jdbc.*;
import oracle.sqlj.runtime.*;

/**
     //\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\
                        DBConnect
     //\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\
         Classe che contiene i metodi che usano il
         database
     //\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\
*/


public class DBConnect
 {

   final int ALL=0;
   final int XMLONLY=1;
   final int IBRIDONLY=2;

   /**
      Costruttore che si connette a Oracle
   */

   DBConnect() throws SQLException
    {
     Oracle.connect(getClass(), "connect.properties");
    }


   /**
      Disconnessione da Oracle
   */
```

```
public static void Disconnect() throws SQLException
 {
   Oracle.close();
 }

public static void DropTable (String tabella) throws SQLException
  {
    // cancella la tabella il cui nome è passato come parametro
    // Crea una connessione JDBC dal DefaultContext SQLJ corrente
    Connection conn =
    sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

    Statement stmt = conn.createStatement ();
    String elimina = "DROP TABLE " + tabella;

    stmt.execute (elimina);        // Elimina la tabella
    stmt.close();
  }

public static void CreateTableLegge () throws SQLException
  {
    // crea la tabella "legge"  --  modello XML
    // Crea una connessione JDBC dal DefaultContext SQLJ corrente
    Connection conn =
    sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

    Statement stmt = conn.createStatement ();
    String crea = "CREATE TABLE LEGGE "
           +"(cod NUMBER (6) PRIMARY KEY,"
           +"leggeXML SYS.XMLTYPE"
           +")";

    stmt.execute (crea);        // Crea la tabella
    stmt.close();
  }

  public static void CreateTableLegge_I () throws SQLException
  {
    // crea la tabella "legge_I"  --  modello ibrido
    // Crea una connessione JDBC dal DefaultContext SQLJ corrente
    Connection conn =
    sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

    Statement stmt = conn.createStatement ();
    String crea = "CREATE TABLE LEGGE_I "
           +"(cod NUMBER (6) PRIMARY KEY,"
           +"leggeXML SYS.XMLTYPE,"
           +"pubblicazione DATE NOT NULL,"
           +"vt_start DATE NOT NULL,"
           +"vt_end DATE,"
           +"tt_start DATE NOT NULL,"
           +"tt_end DATE,"
           +"et_start DATE NOT NULL,"
           +"et_end DATE,"
           +"natura VARCHAR (100) NOT NULL"
           +")";

    stmt.execute (crea);        // Crea la tabella
    stmt.close();
  }
```

```
public static long DropIndexes (int Quali)
  {
    // cancecella gli indici e le preferenze create, il parametro
    // permette di scegliere su quale modello agire
    java.util.Date DataInizio = new java.util.Date();
    long Inizio = DataInizio.getTime();

    // Crea una connessione JDBC dal DefaultContext SQLJ corrente
    Connection conn =
    sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

    try {
      Statement stmt = conn.createStatement();

      String Stoplist = "BEGIN "
                      + "CTX_DDL.DROP_STOPLIST ('mystop'); "
                      + "END;";

      String Wordlist = "BEGIN "
                      + "CTX_DDL.DROP_WORDLIST ('myword'); "
                      + "END;";

      String Preferenze = "BEGIN "
                        + "CTX_DDL.DROP_SECTION_GROUP ('xmlpath'); "
                        + "END;";

      String Preferenze_I = "BEGIN "
                          + "CTX_DDL.DROP_SECTION_GROUP ('xmlpath_I'); "
                          + "END;";

      String Indice = "DROP INDEX ilegge";

      String Indice_I = "DROP INDEX ilegge_I";
      String Indice_I_TTS = "DROP INDEX tts";
      String Indice_I_TTE = "DROP INDEX tte";
      String Indice_I_PUB = "DROP INDEX pubblicazione";

      stmt.execute(Stoplist);
      stmt.execute(Wordlist);

      if (Quali == 0 || Quali == 1)
        {
          stmt.execute(Preferenze);
          stmt.execute(Indice);
        }
       if (Quali == 0 || Quali == 2)
        {
          stmt.execute(Preferenze_I);
          stmt.execute(Indice_I);
          stmt.execute(Indice_I_TTS);
          stmt.execute(Indice_I_TTE);
          stmt.execute(Indice_I_PUB);
        }
      stmt.close();

    } catch (Exception e) {e.printStackTrace();}

    java.util.Date DataFine = new java.util.Date();  // fornisce la durata
                                                       // dell'operazione
    long Fine = DataFine.getTime();
    long Durata = (Fine - Inizio) / Utility.TIME;
  return Durata;
```

```
  }

public static long UpdateIndex (int Quali)
  {
    // aggiorna gli indici e le preferenze create, il parametro permette
    // di scegliere su quale modello agire
    java.util.Date DataInizio = new java.util.Date();
    long Inizio = DataInizio.getTime();

    // Crea una connessione JDBC dal DefaultContext SQLJ corrente
    Connection conn =
    sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

    try {
      Statement stmt = conn.createStatement ();

      String UpdateIndex = "ALTER INDEX ilegge REBUILD ONLINE
                                  PARAMETERS('sync')";
      String UpdateIndex_I = "ALTER INDEX ilegge_I REBUILD ONLINE
                                   PARAMETERS('sync')";

      if (Quali == 0 || Quali == 1)
        stmt.execute(UpdateIndex);
      if (Quali == 0 || Quali == 2)
        stmt.execute(UpdateIndex_I);

    } catch (Exception e) {e.printStackTrace();}

    java.util.Date DataFine = new java.util.Date();  // fornisce la
                                                     // durata dell'operazione
    long Fine = DataFine.getTime();
    long Durata = (Fine - Inizio) / Utility.TIME;
  return Durata;
  }

  public static long OptimizeIndex (int Quali)
    {
      // ottimizza gli indici e le preferenze create, il parametro
      // permette di scegliere su quale modello agire
      java.util.Date DataInizio = new java.util.Date();
      long Inizio = DataInizio.getTime();

      // Crea una connessione JDBC dal DefaultContext SQLJ corrente
      Connection conn =
     sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

      try {
        Statement stmt = conn.createStatement ();

        String UpdateIndex = "ALTER INDEX ilegge REBUILD ONLINE
                              PARAMETERS('optimize fast')";
        String UpdateIndex_I = "ALTER INDEX ilegge_I REBUILD ONLINE
                               PARAMETERS('optimize fast')";

        if (Quali == 0 || Quali == 1)
          stmt.execute(UpdateIndex);
        if (Quali == 0 || Quali == 2)
          stmt.execute(UpdateIndex_I);

      } catch (Exception e) {e.printStackTrace();}

      java.util.Date DataFine = new java.util.Date();  // fornisce la
```

```
                                                    // durata dell'operazione
      long Fine = DataFine.getTime();
      long Durata = (Fine - Inizio) / Utility.TIME;
    return Durata;
    }

    public static long RebuildIndex (int Quali)
      {
        // ricostruisce gli indici e le preferenze create, il parametro
        // permette di scegliere su quale modello agire
        java.util.Date DataInizio = new java.util.Date();
        long Inizio = DataInizio.getTime();

        // Crea una connessione JDBC dal DefaultContext SQLJ corrente
        Connection conn =
     sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

        try {
          Statement stmt = conn.createStatement ();

          String UpdateIndex = "ALTER INDEX ilegge REBUILD";
          String UpdateIndex_I = "ALTER INDEX ilegge_I REBUILD";

          if (Quali == 0 || Quali == 1)
            stmt.execute(UpdateIndex);
          if (Quali == 0 || Quali == 2)
            stmt.execute(UpdateIndex_I);

        } catch (Exception e) {e.printStackTrace();}


        java.util.Date DataFine = new java.util.Date(); // fornisce la
                                                          // durata dell'operazione
        long Fine = DataFine.getTime();
        long Durata = (Fine - Inizio) / Utility.TIME;
      return Durata;
      }


public static long CreateIndexes (int Quali)
   {
     // crea gli indici e le preferenze, il parametro permette di scegliere
     // su quale modello agire
     java.util.Date DataInizio = new java.util.Date();
     long Inizio = DataInizio.getTime();

     // Crea una connessione JDBC dal DefaultContext SQLJ corrente
     Connection conn =
     sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

     try {
       Statement stmt = conn.createStatement ();

       String Stoplist = "BEGIN "
                                  // Stoplist italiana
                     + "CTX_DDL.CREATE_STOPLIST ('mystop','BASIC_STOPLIST');"
                     + "CTX_DDL.ADD_STOPWORD ('mystop','a'); "
                     + "CTX_DDL.ADD_STOPWORD ('mystop','affinchè'); "
                     + "CTX_DDL.ADD_STOPWORD ('mystop','agl'''); "
                     + "CTX_DDL.ADD_STOPWORD ('mystop','agli'); "
                     + "CTX_DDL.ADD_STOPWORD ('mystop','ai'); "
                     + "CTX_DDL.ADD_STOPWORD ('mystop','al'); "
```

```
+ "CTX_DDL.ADD_STOPWORD ('mystop','all'''); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','alla'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','alle'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','allo'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','anzichè'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','avere'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','bensì'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','che'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','chi'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','cioè'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','come'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','comunque'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','con'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','contro'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','cosa'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','da'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','dachè'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','dagl''); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','dagli'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','dai'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','dal'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','dall''); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','dalla'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','dalle'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','dallo'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','degl''); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','degli'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','dei'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','del'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','dell''); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','delle'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','dello'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','di'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','dopo'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','dove'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','dunque'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','durante'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','e'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','egli'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','eppure'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','essere'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','essi'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','finché'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','fino'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','fra'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','giacchè'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','gl'''); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','gli'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','grazie'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','i'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','il'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','in'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','inoltre'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','io'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','l'''); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','la'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','le'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','lo'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','loro'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','ma'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','mentre'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','mio'); "
```

```
+ "CTX_DDL.ADD_STOPWORD ('mystop','ne'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','neanche'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','negl''); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','negli'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','nei'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','nel'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','nell''); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','nella'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','nelle'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','nello'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','nemmeno'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','neppure'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','noi'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','nonchè'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','nondimeno'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','nostro'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','o'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','onde'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','oppure'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','ossia'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','ovvero'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','per'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','perchè'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','perciò'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','però'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','poichè'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','prima'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','purchè'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','quand''anche'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','quando'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','quantunque'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','quasi'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','quindi'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','se'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','sebbene'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','sennonchè'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','senza'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','seppure'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','si'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','siccome'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','sopra'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','sotto'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','su'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','subito'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','sugl''); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','sugli'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','sui'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','sul'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','sull''); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','sulla'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','sulle'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','sullo'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','suo'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','talchè'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','tu'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','tuo'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','tuttavia'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','tutti'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','un'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','una'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','uno'); "
+ "CTX_DDL.ADD_STOPWORD ('mystop','voi'); "
```

```
                + "CTX_DDL.ADD_STOPWORD ('mystop','vostro'); "
                            // Stoplist specifica
                + "CTX_DDL.ADD_STOPWORD ('mystop','text'); "
                + "CTX_DDL.ADD_STOPWORD ('mystop','versione'); "
                + "CTX_DDL.ADD_STOPWORD ('mystop','ver'); "
                + "CTX_DDL.ADD_STOPWORD ('mystop','comma'); "
                + "END;";

String Wordlist = "BEGIN "
             + "CTX_DDL.CREATE_PREFERENCE('myword','BASIC_WORDLIST');"
             + "CTX_DDL.SET_ATTRIBUTE('myword','STEMMER','ITALIAN'); "
             + "CTX DDL.SET ATTRIBUTE('myword','FUZZY MATCH','ITALIAN');
             + "CTX_DDL.SET_ATTRIBUTE('myword','PREFIX_INDEX','YES'); "
             + "CTX_DDL.SET_ATTRIBUTE('myword','PREFIX_MIN_LENGTH',3); "
             + "CTX_DDL.SET_ATTRIBUTE('myword','PREFIX_MAX_LENGTH',4); "
             + "END;";

String Preferenze = "BEGIN "
                 + "CTX_DDL.CREATE_SECTION_GROUP ('xmlpath',
                   'XML_SECTION_GROUP'); "
                 + "CTX_DDL.ADD_FIELD_SECTION ('xmlpath', 'titolo',
                   'titolo', TRUE); " // crea sezione titolo
                 + "CTX_DDL.ADD_FIELD_SECTION ('xmlpath', 'natura',
                    'natura', TRUE); " // crea sezione natura
                 + "CTX_DDL.ADD_ZONE_SECTION ('xmlpath', 'comma',
                   'comma'); " // crea sezione comma
                 + "END;";

String Preferenze_I = "BEGIN "
                   + "CTX_DDL.CREATE_SECTION_GROUP ('xmlpath_I',
                     'XML_SECTION_GROUP'); "
                   + "CTX_DDL.ADD_FIELD_SECTION ('xmlpath_I', 'titolo',
                       'titolo', TRUE); " // crea sezione titolo
                   + "CTX_DDL.ADD_ZONE_SECTION ('xmlpath_I', 'comma',
                     'comma'); " // crea sezione comma
                   + "END;";

String Indice = "CREATE INDEX ilegge ON legge (leggeXML)"
             + "INDEXTYPE IS ctxsys.context "
             + "PARAMETERS ( 'DATASTORE CTXSYS.DEFAULT_DATASTORE "
             + "FILTER CTXSYS.NULL_FILTER "
             + "SECTION GROUP xmlpath "
             + "WORDLIST myword "
             + "STOPLIST mystop')";

String Indice_I = "CREATE INDEX ilegge_I ON legge_I (leggeXML)"
                + "INDEXTYPE IS ctxsys.context "
                + "PARAMETERS ( 'DATASTORE CTXSYS.DEFAULT_DATASTORE "
                + "FILTER CTXSYS.NULL_FILTER "
                + "SECTION GROUP xmlpath_I "
                + "WORDLIST myword "
                + "STOPLIST mystop')";

 // indici B-Tree sulle colonne relazionali del modello ibrido
String Indice_I_TTS = "CREATE INDEX tts ON legge_I (tt_start)";
String Indice_I_TTE = "CREATE INDEX tte ON legge_I (tt_end)";
String Indice_I_PUB = "CREATE INDEX pubblicazione ON legge_I
                       (pubblicazione)";

 stmt.execute(Stoplist);
 stmt.execute(Wordlist);
 if (Quali == 0 || Quali == 1)
```

```
          {
            stmt.execute(Preferenze);
            stmt.execute(Indice);
          }
        if (Quali == 0 || Quali == 2)
          {
            stmt.execute(Preferenze_I);
            stmt.execute(Indice_I);
            stmt.execute(Indice_I_TTS);
            stmt.execute(Indice_I_TTE);
            stmt.execute(Indice_I_PUB);
          }
        stmt.close();

      } catch (Exception e) {e.printStackTrace();}

      java.util.Date DataFine = new java.util.Date();  // fornisce la durata
                                                         // dell'operazione
      long Fine = DataFine.getTime();
      long Durata = (Fine - Inizio) / Utility.TIME;
    return Durata;
    }

  public static long InsertLex (String NomeFile,int Codice) throws SQLException
    {
       // inserisce una legge nella tabella del modello XML
       java.util.Date DataInizio = new java.util.Date();
       long Inizio = DataInizio.getTime();

       // Crea una connessione JDBC dal DefaultContext SQLJ corrente
       Connection conn =
       sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

      try {
        // recupera il file contenente la legge
        RandomAccessFile file = new RandomAccessFile(NomeFile,"r");
        String LeggeDaInserire = "";
        String Porzione = "";
        while (Porzione != null)
          {
            Porzione = file.readLine();
            if (Porzione != null) LeggeDaInserire = LeggeDaInserire + Porzione;
          }

         // prepara un oggetto clob che contiene la legge da inserire
         oracle.sql.CLOB clob =
oracle.sql.CLOB.createTemporary(conn,false,oracle.sql.CLOB.DURATION_SESSION);
          clob.open(oracle.sql.CLOB.MODE_READWRITE);
          clob.putString((long) 1, LeggeDaInserire);

        OraclePreparedStatement pstmt = (OraclePreparedStatement)
        conn.prepareStatement(
            "INSERT INTO legge VALUES (?, sys.XMLType.createXML(?))");
        pstmt.setInt(1, Codice);
        pstmt.setCLOB(2, clob);

        pstmt.execute();                          // Inserisce la legge
      } catch (Exception e) {e.printStackTrace();}

      java.util.Date DataFine = new java.util.Date();  // fornisce la durata
                                                         // dell'operazione
      long Fine = DataFine.getTime();
```

```
  long Durata = (Fine - Inizio) / Utility.TIME;
return Durata;
}

public static long InsertLex_I (String NomeFile,int Codice) throws
                                                          SQLException
  {
    // inserisce una legge nella tabella del modello ibrido
     java.util.Date DataInizio = new java.util.Date();
     long Inizio = DataInizio.getTime();

     // Crea una connessione JDBC dal DefaultContext SQLJ corrente
     Connection conn =
     sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

    try {
      // recupera il file contenente la legge
      RandomAccessFile file = new RandomAccessFile(NomeFile,"r");
      String LeggeDaInserire = "";
      String Porzione = "";
      while (Porzione != null)
        {
          Porzione = file.readLine();
          if (Porzione != null)
           LeggeDaInserire = LeggeDaInserire + Porzione;
        }

      StringReader lettore = new StringReader (LeggeDaInserire);
      Document legge = new SAXBuilder().build(lettore);

      // ottiene dalla legge da inserire i valori che verranno messi nelle
      // colonne relazionali
      Element articolato = legge.getRootElement().getChild("articolato");
      Element natura = legge.getRootElement().getChild("natura");
      java.sql.Date pubblicazione = null;
      java.sql.Date valido_start = null;
      java.sql.Date valido_end = null;
      java.sql.Date trans_start = null;
      java.sql.Date trans_end = null;
      java.sql.Date efficace_start = null;
      java.sql.Date efficace_end = null;

      String t = articolato.getAttributeValue("pubblicazione");
      if (t != null) { pubblicazione = java.sql.Date.valueOf(t); }
      t = articolato.getAttributeValue("vt_start");
      if (t != null) { valido_start = java.sql.Date.valueOf(t); }
      t = articolato.getAttributeValue("vt_end");
      if (t != null) { valido_end = java.sql.Date.valueOf(t); }
      t = articolato.getAttributeValue("tt_start");
      if (t != null) { trans_start = java.sql.Date.valueOf(t); }
      t = articolato.getAttributeValue("tt_end");
      if (t != null) { trans_end = java.sql.Date.valueOf(t); }
      t = articolato.getAttributeValue("et_start");
      if (t != null) { efficace_start = java.sql.Date.valueOf(t); }
      t = articolato.getAttributeValue("et_end");
      if (t != null) { efficace_end = java.sql.Date.valueOf(t); }

      // prepara un oggetto clob che contiene la legge da inserire
      oracle.sql.CLOB clob =
oracle.sql.CLOB.createTemporary(conn,false,oracle.sql.CLOB.DURATION_SESSION);
      clob.open(oracle.sql.CLOB.MODE_READWRITE);
      clob.putString((long) 1, LeggeDaInserire);
```

```
      OraclePreparedStatement pstmt = (OraclePreparedStatement)
      conn.prepareStatement(
        "INSERT INTO legge_I VALUES
          (?, sys.XMLType.createXML(?), ?, ?, ?, ?, ?, ?, ?, ? )");

      pstmt.setInt(1, Codice);
      pstmt.setCLOB(2, clob);
      pstmt.setDate(3, pubblicazione);
      pstmt.setDate(4, valido_start);
      pstmt.setDate(5, valido_end);
      pstmt.setDate(6, trans_start);
      pstmt.setDate(7, trans_end);
      pstmt.setDate(8, efficace_start);
      pstmt.setDate(9, efficace_end);
      pstmt.setString(10, natura.getText());
      pstmt.execute ();                          // Inserisce la legge
    } catch (Exception e) {e.printStackTrace();}

    java.util.Date DataFine = new java.util.Date();  // fornisce la durata
                                                       // dell'operazione
    long Fine = DataFine.getTime();
    long Durata = (Fine - Inizio) / Utility.TIME;
  return Durata;
  }


public static Document LeggiLex(int Cod, String Table) throws SQLException
  {
   // recupera dal db il documento il cui codice/tabella è passato
   // come  parametro
   Document legge=null;

   Connection conn =
   sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

   Statement st = conn.createStatement ();
   String query = "SELECT t.leggexml.extract('/legge').getClobVal() "
                + "FROM " + Table + " t where t.cod = "
                + Cod;
   ResultSet rs = st.executeQuery (query);
   while (rs.next())
     {
       try {
         Reader lettore = rs.getClob(1).getCharacterStream();
         legge = new SAXBuilder().build(lettore);
       } catch (Exception e) {e.printStackTrace();}
     }
   st.close();
   return (legge);

  }

  public static Result Cerca (String Query, boolean Titoli, boolean Numeri)
                                                          throws SQLException
   {
     // esegue una query di selezione e torna i risultati richiesti
     java.util.Date DataInizio = new java.util.Date();
     long Inizio = DataInizio.getTime();

     Vector Codes = new Vector();
     Vector Titles = new Vector();
```

```
  Vector Numbers = new Vector();
  // Crea una connessione JDBC dal DefaultContext SQLJ corrente
  Connection conn =
  sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

  try {
    Statement stmt = conn.createStatement ();

    ResultSet rs = stmt.executeQuery (Query);

    while (rs.next())
      {
        Codes.add(rs.getObject(1));                 // torna i codici
        if (Titoli) Titles.add(rs.getObject(2));    // torna i titoli
        if (Numeri)
          if (Titoli) Numbers.add(rs.getObject(3)); // torna i numeri
          else Numbers.add(rs.getObject(2));
      }

    stmt.close();

  } catch (Exception e) {throw new SQLException();}

  java.util.Date DataFine = new java.util.Date();  // fornisce la durata
                                                   // dell'operazione
  long Fine = DataFine.getTime();
  long Durata = (Fine - Inizio) / Utility.TIME;
return (new Result (Codes,Titles,Numbers,Durata));
}

public static long GetMaxCod (String Table)
 {
   // restituisce il massimo valore di cod presente nella tabella il cui
   // nome è passato come parametro
   long Max = 0;
   // Crea una connessione JDBC dal DefaultContext SQLJ corrente
   Connection conn =
   sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

   try {
     Statement stmt = conn.createStatement ();
     String Query = "SELECT MAX(cod) FROM " + Table;

     ResultSet rs = stmt.executeQuery (Query);

     while (rs.next())
       {
         Max = rs.getLong(1);
       }

     stmt.close();

   } catch (Exception e) {e.printStackTrace();}

 return Max;
 }
}
```

```
package Principale;

import java.util.*;
import org.jdom.*;

/**
     //\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\
                        Ricostruzione
     //\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\
          Classe adibita alla ricostruzione delle
          norme secondo specifici valori temporali
          e operatori su di essi
     //\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\
*/

 /* Operazioni:
    0 - nessuna operazione
    1 - Uguaglianza
    2 - prima (giorno specificato incluso)
    3 - dopo (giorno specificato incluso)
    4 - intersecante (Overlap) (estremi compresi)
    5 - escluso l'intervallo (estremi compresi)
    6 - contenuto nell'intervallo (Contain) (estremi esclusi)
    7 - fuori dell' intervallo (non intersecante)
  */


public class Rebuild
{
   // inizializzazione
   final int NONE=0;
   final int EQUAL=1;
   final int BEFORE=2;
   final int AFTER=3;
   final int OVERLAP=4;
   final int NOTCONTAIN=5;
   final int CONTAIN=6;
   final int NOTOVERLAP=7;

   Rebuild () { }

  public boolean ControllaValidoTA (java.sql.Date VTS, java.sql.Date VTE,
                                    int VToper, Element Ta)
  {
    // controlla il valid time di un TA secondo l'operatore inserito come
    // parametro
    boolean valido = true;

    java.sql.Date valido_start = null;
    String t = Ta.getAttributeValue("vt_start");
    if (t != null) valido_start = java.sql.Date.valueOf(t);
    java.sql.Date valido_end = null;
    t = Ta.getAttributeValue("vt_end");
    if (t != null) valido_end = java.sql.Date.valueOf(t);

    if (valido_end == null)    // la versione ha validità infinita
      switch (VToper) {
        case 1: if (VTS.before(valido_start)) { valido = false; } break;
        case 2: if (VTS.before(valido_start)) { valido = false; } break;
        case 4: if (VTE.before(valido_start)) { valido = false; } break;
        case 6: { valido = false; } break;
        case 7: if (VTE.after(valido_start)) { valido = false; } break;
```

```
    }
  else
    switch (VToper) {
      case 1: if ((VTS.before(valido_start)) || (VTS.after(valido_end)))
                { valido = false; } break;
      case 2: if (VTS.before(valido_start)) { valido = false; } break;
      case 3: if (VTS.after(valido_end)) { valido = false; } break;
      case 4: if ((VTS.after(valido_end)) || (VTE.before(valido_start)))
                { valido = false; } break;
      case 5: if ((VTS.before(valido_start)) && (VTE.after(valido_end)))
                { valido = false; } break;
      case 6: if (((VTS.before(valido_start)) &&
                  (VTE.after(valido_end))) == false)
                { valido = false; } break;
      case 7: if (((VTE.before(valido_start)) ||
                 (VTS.after(valido_end))) == false)
                 { valido = false; } break;
    }

return valido;
}

public boolean ControllaEfficaceTA (java.sql.Date ETS, java.sql.Date
                                    ETE, int EToper, Element Ta)
{
  // controlla l'efficacy time di un TA secondo l'operatore inserito come
  // parametro
  boolean efficace = true;

  java.sql.Date efficace_start = null;
  String t = Ta.getAttributeValue("et_start");
  if (t != null) efficace_start = java.sql.Date.valueOf(t);
  java.sql.Date efficace_end = null;
  t = Ta.getAttributeValue("et_end");
  if (t != null) efficace_end = java.sql.Date.valueOf(t);

  if (efficace_end == null)    // la versione ha efficacia infinita
    switch (EToper) {
      case 1: if (ETS.before(efficace_start)) { efficace = false; } break;
      case 2: if (ETS.before(efficace_start)) { efficace = false; } break;
      case 4: if (ETE.before(efficace_start)) { efficace = false; } break;
      case 6: if (ETS.after(efficace_start)) { efficace = false; } break;
      case 7: if (ETE.after(efficace_start)) { efficace = false; } break;
    }
  else
    switch (EToper) {
      case 1: if ((ETS.before(efficace_start)) || (ETS.after(efficace_end)))
                { efficace = false; } break;
      case 2: if (ETS.before(efficace_start)) { efficace = false; } break;
      case 3: if (ETS.after(efficace_end)) { efficace = false; } break;
      case 4: if ((ETS.after(efficace_end)) || (ETE.before(efficace_start)))
                { efficace = false; } break;
      case 5: if ((ETS.before(efficace_start)) && (ETE.after(efficace_end)))
                { efficace = false; } break;
      case 6: if (((ETS.before(efficace_start)) &&
                  (ETE.after(efficace_end))) == false)
                { efficace = false; } break;
      case 7: if (((ETE.before(efficace_start)) ||
                  (ETS.after(efficace_end))) == false) { efficace = false; } break;
    }

return efficace;
```

```
}

public boolean ControllaTransTA (java.sql.Date TT, Element Ta)
{
  // controlla il transaction time di un TA alla ricerca di TA ancora attivi
  boolean attiva = true;

  java.sql.Date trans_start = null;
  String t = Ta.getAttributeValue("tt_start");
  if (t != null) trans_start = java.sql.Date.valueOf(t);
  java.sql.Date trans_end = null;
  t = Ta.getAttributeValue("tt_end");
  if (t != null) trans_end = java.sql.Date.valueOf(t);

  if (TT.before(trans_start) == true) attiva = false; // versione non ancora
                                                      // esistente
  else if (trans_end != null)
    if (TT.after(trans_end) == true) attiva = false; // versione esistente ma
                                                       // non più attiva

return attiva;
}

public void EsploraLivello (java.sql.Date VTS, java.sql.Date VTE,
                            java.sql.Date TT, java.sql.Date ETS,
                            java.sql.Date ETE, java.sql.Date PTS,
                            java.sql.Date PTE, int VToper, int EToper,
                            int, PToper, Element nodo, int depth)
{
  // procedura ricorsiva che scorre l'intero documento depennando i componenti
  // non compatibili con quanto richiesto tramite i parametri
  int ChildNumber = 0;
  boolean Ok = true;

  Vector DaRimuovere = new Vector();
  java.util.List versions = nodo.getChildren(); // lista versioni
  Iterator iterator = versions.iterator();
  while (iterator.hasNext())
   {
    Element ver = (Element) iterator.next();
    if (depth % 2 == 1)
     {
       boolean OkVer = false;
       java.util.List Tas = ver.getChildren("ta"); // lista TA
       Iterator iterator2 = Tas.iterator();
       while (iterator2.hasNext() && OkVer == false)
       {
         Element Ta = (Element) iterator2.next();
         if ((ControllaTransTA (TT,Ta)) &&
             (ControllaValidoTA (VTS,VTE,VToper,Ta))
            && (ControllaEfficaceTA (ETS,ETE,EToper,Ta)))
           OkVer = true;
       }

       if (OkVer == false)
       { // componente da rimuovere
         DaRimuovere.addElement(new Integer(ChildNumber));
         Ok = false;
       }
     }
    ChildNumber++;
    if (Ok == true)
```

```
      EsploraLivello (VTS,VTE,TT,ETS,ETE,PTS,PTE,VToper,
                      EToper,PToper,ver,depth+1); // chiamata ricorsiva
   }

   for (int i = 0; i < DaRimuovere.size(); i++) // rimozione dei componenti
                                                // non compatibili
     versions.remove((Integer.parseInt
     (DaRimuovere.elementAt(i).toString()))-i);
}

public boolean ControllaPubblicazione (java.sql.Date PTS, java.sql.Date PTE,
                                       int PToper, java.sql.Date pubblicazione)
{
  // controlla che la data di pubblicazione del documento sia compatibile con
  // quanto richiesto
  boolean ritorno=true;
  switch (PToper) {
    case 1: if (PTS.compareTo(pubblicazione) != 0)
              { ritorno = false; }  break;
    case 2: if (PTS.before(pubblicazione)) { ritorno = false; } break;
    case 3: if (PTS.after(pubblicazione)) { ritorno = false; } break;
    case 4: if ((PTS.after(pubblicazione)) || (PTE.before(pubblicazione)))
              { ritorno = false; } break;
    case 5: if ((PTS.before(pubblicazione)) && (PTE.after(pubblicazione)))
              { ritorno = false; } break;
    case 6: if ((PTS.after(pubblicazione)) || (PTE.before(pubblicazione)))
              { ritorno = false; } break;
    case 7: if ((PTS.before(pubblicazione)) && (PTE.after(pubblicazione)))
              { ritorno = false; } break;
  }
return ritorno;
}

public boolean ControllaArticolato (java.sql.Date VTS, java.sql.Date
                                    VTE, java.sql.Date TT,
                                    java.sql.Date ETS, java.sql.Date
                                    ETE, java.sql.Date PTS,
                                    java.sql.Date PTE, int VToper, int
                                    EToper, int PToper, Element nodo)
{
   // controllo preliminare sugli attributi eterni riepilogativi
   boolean ritorno=true;
   java.sql.Date pubblicazione =
   java.sql.Date.valueOf(nodo.getAttributeValue("pubblicazione"));
   if (ControllaPubblicazione(PTS,PTE,PToper,pubblicazione) == false)
    { ritorno = false; }

   if ((ControllaTransTA(TT,nodo) == false) ||
      (ControllaValidoTA (VTS,VTE,VToper,nodo) == false) ||
      (ControllaEfficaceTA (ETS,ETE,EToper,nodo) == false))
     { ritorno = false; }

return ritorno;
}

public Result Ricostruisci (Document legge, java.sql.Date VTS, java.sql.Date
                            VTE, java.sql.Date TT, java.sql.Date ETS,
                            java.sql.Date ETE, java.sql.Date PTS,
                            java.sql.Date PTE, int VToper,
                            int EToper, int PToper)
{
  // avvia la procedura di ricostruzione
```

```
    java.util.Date DataInizio = new java.util.Date();
    long Inizio = DataInizio.getTime();

    Element root = legge.getRootElement();
    Element articolato = root.getChild("articolato");
    boolean controllo = true;
     try {
          // controllo degli attributi temporali e degli operatori usati nella
          // ricostruzione
          if ((VToper != 0) && (VToper != 1) && (VToper != 2) && (VToper != 3)
             && (VToper != 4) && (VToper != 5) && (VToper != 6) &&
             (VToper != 7))
             { controllo=false; }
          else if ((EToper != 0) && (EToper != 1) && (EToper != 2) &&
                  (EToper != 3) && (EToper != 4) && (EToper != 5) && (EToper !=
                  6) && (EToper != 7))
                { controllo=false; }
          else if ((PToper != 0) && (PToper != 1) && (PToper != 2) && (PToper !=
                  3) && (PToper != 4) && (PToper != 5) && (PToper != 6) &&
                  (PToper != 7))
                { controllo=false; }
          else if ((VToper != 0) && (VTS == null)) { controllo=false; }
          else if ((EToper != 0) && (ETS == null)) { controllo=false; }
          else if ((PToper != 0) && (PTS == null)) { controllo=false; }
          else if (((VToper == 4) || (VToper == 5) || (VToper == 6) ||
                  (VToper == 7)) && (VTE == null))
                { controllo=false; }
          else if (((EToper == 4) || (EToper == 5) || (EToper == 6) ||
                  (EToper == 7)) && (ETE == null))
                { controllo=false; }
          else if (((PToper == 4) || (PToper == 5) || (PToper == 6) ||
                  (PToper == 7)) && (PTE == null))
                { controllo=false; }

          if (TT == null) TT = java.sql.Date.valueOf(Utility.StringaData());

          if (controllo) // controllo preliminare
            if (ControllaArticolato
               (VTS,VTE,TT,ETS,ETE,PTS,PTE,VToper,EToper,PToper,articolato))
                EsploraLivello
                (VTS,VTE,TT,ETS,ETE,PTS,PTE,VToper,EToper,PToper,articolato,1);
                // avvia la procedura ricorsiva
            else { System.err.println("documento non compatibile"); }
          else { System.err.println("errore nei parametri"); }

        } catch (Exception e) { e.printStackTrace(); }

    java.util.Date DataFine = new java.util.Date();  // ritorna la durata
                                                       // dell'operazione
    long Fine = DataFine.getTime();
    long Durata = (Fine - Inizio) / Utility.TIME;
  return (new Result (legge,Durata));
  }
}



package Principale;

import java.sql.*;
import java.util.*;
```

```
import org.jdom.*;

/**
     //\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\
                        ModificaNorma
     //\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\
         Classe adibita all'esecuzione delle modifiche
         delle norme presenti nel db
     //\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\
*/


public class Modify {

  // inizializzazione
  final int ARTICOLATO=0;
  final int CAPO=1;
  final int ARTICOLO=2;
  final int COMMA=3;

  final static int CUT=4;
  final static int ADD=5;

  public Modify() {
  }


  public void PropagaGiu (Element Nodo, int Depth, java.util.Date Trans,
                          java.sql.Date NewVTS, java.sql.Date NewVTE,
                          java.sql.Date NewETS, java.sql.Date NewETE)
  {
    // propaga verso le foglie le modifiche di un documento legge
    java.util.List Elementi = Nodo.getChildren();
    Iterator iterator = Elementi.iterator();
    while (iterator.hasNext())
    {
      Element Elemento = (Element) iterator.next();
      java.util.List Versioni = Elemento.getChildren("ver"); // versioni del
                                                             // componente
      Iterator iterator2 = Versioni.iterator();
      while (iterator2.hasNext())
      {
        Element Ver = (Element) iterator2.next();
        java.util.List Tas = Ver.getChildren("ta"); // TA delle versioni
        Iterator iterator3 = Tas.iterator();
        while (iterator3.hasNext())
        {
          Element Ta = (Element) iterator3.next();
          if ((Ta.getAttribute("tt_end") == null) &&
             (IntersezioneTa (Ta,NewVTS,NewVTE,NewETS,NewETE)))
            {  // se la versione richiede modifiche esse vengono apportate
              NuoviTa(Ver, Ta, Trans, NewVTS, NewVTE, NewETS, NewETE, CUT);
              Ta.addAttribute("tt_end", Trans.toString());
            }
        }
        if (Depth < 3)
         PropagaGiu(Ver,Depth+1,Trans,NewVTS,NewVTE,NewETS,NewETE);
        // ricorsività
      }
    }
  }
```

```
public boolean ControllaNuovoTa (Element Ta, int Add)
{
  // controlla che il nuovo TA creato contenga dati temporali coerenti
  boolean ritorno = true;
  if (Ta.getAttribute("vt_end") != null)
    if (Ta.getAttributeValue("vt_start").compareTo
       (Ta.getAttributeValue("vt_end")) == 0)
        ritorno = false;

 if (ritorno)
   if (Ta.getAttribute("et_end") != null)
     if (Ta.getAttributeValue("et_start").compareTo
        (Ta.getAttributeValue("et_end")) == 0)
        ritorno = false;

if (ritorno)
  if (Ta.getAttribute("vt_end") != null)
    if (java.sql.Date.valueOf(Ta.getAttributeValue("vt_start")).
        after(java.sql.Date.valueOf(Ta.getAttributeValue("vt_end"))))
      if (Add == CUT)
        ritorno = false;
      else  // in modalità add i TA che hanno vt_start > vt_end subiscono uno
            // scambio tra i due valori
      {
        String vts = Ta.getAttributeValue("vt_end");
        String vte = Ta.getAttributeValue("vt_start");
        Ta.removeAttribute("vt_start");
        Ta.removeAttribute("vt_end");
        Ta.addAttribute("vt_start",vts);
        Ta.addAttribute("vt_end",vte);
      }

if (ritorno)
  if (Ta.getAttribute("et_end") != null)
    if (java.sql.Date.valueOf(Ta.getAttributeValue("et_start")).
        after(java.sql.Date.valueOf(Ta.getAttributeValue("et_end"))))
      if (Add == CUT)
        ritorno = false;
      else  // in modalità add i TA che hanno et_start > et_end subiscono uno
            // scambio tra i due valori
      {
        String ets = Ta.getAttributeValue("et_end");
        String ete = Ta.getAttributeValue("et_start");
        Ta.removeAttribute("et_start");
        Ta.removeAttribute("et_end");
        Ta.addAttribute("et_start",ets);
        Ta.addAttribute("et_end",ete);
      }

return ritorno;
}

public Element NuoviTa (Element VecchiaVer, Element OldTa,
                        java.util.Date Trans, java.sql.Date NewVTS,
                        java.sql.Date NewVTE, java.sql.Date NewETS,
                        java.sql.Date NewETE, int Add)
{
  // genera il nuovo TA in base a dati preesistenti e a quanto specificato nel
  // procedimento di modifica
  Element Ta = new Element ("ta");
  Ta.addAttribute("tt_start", Trans.toString());
  if (NewVTS != null)
```

```
  Ta.addAttribute("vt_start", NewVTS.toString());
else
  Ta.addAttribute("vt_start", OldTa.getAttributeValue("vt_start"));
if (NewETS != null)
  Ta.addAttribute("et_start", NewETS.toString());
else
  Ta.addAttribute("et_start", OldTa.getAttributeValue("et_start"));

if (OldTa.getAttribute("vt_end") == null)
{
  if (NewVTE != null)
    Ta.addAttribute("vt_end", NewVTE.toString());
}
else
  if (NewVTE == null)
    Ta.addAttribute("vt_end", OldTa.getAttributeValue("vt_end").toString());
  else
    if (NewVTE.before(java.sql.Date.valueOf(
        OldTa.getAttributeValue("vt_end").toString())))
      Ta.addAttribute("vt_end", NewVTE.toString());
    else
      Ta.addAttribute("vt_end",
      OldTa.getAttributeValue("vt_end").toString());

if (OldTa.getAttribute("et_end") == null)
{
  if (NewETE != null)
    Ta.addAttribute("et_end", NewETE.toString());
}
else
  if (NewETE == null)
    Ta.addAttribute("et_end", OldTa.getAttributeValue("et_end").toString());
  else
    if (NewETE.before(java.sql.Date.valueOf(
        OldTa.getAttributeValue("et_end").toString())))
      Ta.addAttribute("et_end", NewETE.toString());
    else
      Ta.addAttribute("et_end",
      OldTa.getAttributeValue("et_end").toString());

// crea i nuovi TA derivanti da una modifica spezzando in più parti un TA
// preesistente.
Element TaAdd1 = new Element ("ta"); // destra
Element TaAdd2 = new Element ("ta"); // sotto
Element TaAdd3 = new Element ("ta"); // sinistra
Element TaAdd4 = new Element ("ta"); // sopra

TaAdd1.addAttribute("tt_start", Ta.getAttributeValue("tt_start"));
if (Ta.getAttribute("tt_end") != null)
  TaAdd1.addAttribute("tt_end", Ta.getAttributeValue("tt_end"));
TaAdd1.addAttribute("vt_start", OldTa.getAttributeValue("vt_start"));
TaAdd1.addAttribute("vt_end", Ta.getAttributeValue("vt_start"));
TaAdd1.addAttribute("et_start", OldTa.getAttributeValue("et_start"));
if (OldTa.getAttribute("et_end") != null)
  TaAdd1.addAttribute("et_end", OldTa.getAttributeValue("et_end"));
if (ControllaNuovoTa(TaAdd1,Add)) // il TA viene aggiunto solo se contiene
                                  // dati coerenti
  VecchiaVer.addContent(TaAdd1);

TaAdd2.addAttribute("tt_start", Ta.getAttributeValue("tt_start"));
if (Ta.getAttribute("tt_end") != null)
  TaAdd2.addAttribute("tt_end", Ta.getAttributeValue("tt_end"));
```

```
  TaAdd2.addAttribute("vt_start", Ta.getAttributeValue("vt_start"));
  if (Ta.getAttribute("vt_end") != null)
    TaAdd2.addAttribute("vt_end", Ta.getAttributeValue("vt_end"));
  TaAdd2.addAttribute("et_start", OldTa.getAttributeValue("et_start"));
  TaAdd2.addAttribute("et_end", Ta.getAttributeValue("et_start"));
  if (ControllaNuovoTa(TaAdd2,Add)) // il TA viene aggiunto solo se contiene
                                    // dati coerenti
    VecchiaVer.addContent(TaAdd2);

  if (Ta.getAttribute("vt_end") != null)
  {
    TaAdd3.addAttribute("tt_start",Ta.getAttributeValue("tt_start"));
    if (Ta.getAttribute("tt_end") != null)
      TaAdd3.addAttribute("tt_end",Ta.getAttributeValue("tt_end"));
    TaAdd3.addAttribute("vt_start",Ta.getAttributeValue("vt_end"));
    if (OldTa.getAttribute("vt_end") != null)
      TaAdd3.addAttribute("vt_end",OldTa.getAttributeValue("vt_end"));
    TaAdd3.addAttribute("et_start",OldTa.getAttributeValue("et_start"));
    if (OldTa.getAttribute("et_end") != null)
      TaAdd3.addAttribute("et_end",OldTa.getAttributeValue("et_end"));
    if (ControllaNuovoTa(TaAdd3,Add)) // il TA viene aggiunto solo se contiene
                                      // dati coerenti
      VecchiaVer.addContent(TaAdd3);
  }

  if (Ta.getAttribute("et_end") != null)
  {
    TaAdd4.addAttribute("tt_start",Ta.getAttributeValue("tt_start"));
    if (Ta.getAttribute("tt_end") != null)
      TaAdd4.addAttribute("tt_end",Ta.getAttributeValue("tt_end"));
    TaAdd4.addAttribute("vt_start",Ta.getAttributeValue("vt_start"));
    if (Ta.getAttribute("vt_end") != null)
      TaAdd4.addAttribute("vt_end",Ta.getAttributeValue("vt_end"));
    TaAdd4.addAttribute("et_start",Ta.getAttributeValue("et_end"));
    if (OldTa.getAttribute("et_end") != null)
      TaAdd4.addAttribute("et_end",OldTa.getAttributeValue("et_end"));
    if (ControllaNuovoTa(TaAdd4,Add)) // il TA viene aggiunto solo se contiene
                                      // dati coerenti
      VecchiaVer.addContent(TaAdd4);
  }

return Ta;
}


  public boolean IntersezioneTa (Element Ta, java.sql.Date NewVTS,
                                 java.sql.Date NewVTE, java.sql.Date
                                 NewETS, java.sql.Date NewETE)
  {
    // verifica che un TA intersechi i valori temporali passati come parametri
    boolean ritorno = false;

    java.sql.Date valido_start =
    java.sql.Date.valueOf(Ta.getAttributeValue("vt_start"));
    java.sql.Date efficace_start =
    java.sql.Date.valueOf(Ta.getAttributeValue("et_start"));
    java.sql.Date valido_end = null;
    String t = Ta.getAttributeValue("vt_end");
    if (t != null) valido_end = java.sql.Date.valueOf(t);
    java.sql.Date efficace_end = null;
    t = Ta.getAttributeValue("et_end");
    if (t != null) efficace_end = java.sql.Date.valueOf(t);
```

```
  boolean valido = true;  // controllo valid time
  if (NewVTE == null)
  {
    if (valido_end != null)
      if (NewVTS.compareTo(valido_end) >= 0)
        valido = false;
  }
  else
    if (valido_end != null)
    {
      if ((NewVTE.compareTo(valido_start)) <= 0 ||
         (NewVTS.compareTo(valido_end)) >= 0)
        valido = false;
    }
    else
      if (NewVTE.compareTo(valido_start) <= 0)
        valido = false;

  boolean efficace = true;  // controllo efficacy time
  if (NewETE == null)
  {
    if (efficace_end != null)
      if (NewETS.compareTo(efficace_end) >= 0)
        efficace = false;
  }
  else
    if (efficace_end != null)
    {
      if ((NewETE.compareTo(efficace_start)) <= 0 ||
         (NewETS.compareTo(efficace_end)) >= 0)
        efficace = false;
    }
    else
      if (NewETE.compareTo(efficace_start) <= 0)
        efficace = false;

  ritorno = valido && efficace;  // ritorno vero se valido e efficace veri

return ritorno;
}

public boolean Intersezione (Element Nodo, java.sql.Date NewVTS,
                             java.sql.Date NewVTE, java.sql.Date
                             NewETS, java.sql.Date NewETE)
  {
    // Controlla tutti gli elementi che contengono attributi temporali
    // cercando quelli compatibili con i valori passati come parametri.
    boolean ritorno = false;

    java.util.List Temporal = Nodo.getChildren("ta");
    Iterator iterator = Temporal.iterator();

    while (iterator.hasNext() && ritorno == false)
    {
      Element Ta = (Element) iterator.next();

      java.sql.Date valido_start =
      java.sql.Date.valueOf(Ta.getAttributeValue("vt_start"));
      java.sql.Date efficace_start =
      java.sql.Date.valueOf(Ta.getAttributeValue("et_start"));
      java.sql.Date valido_end = null;
```

```
        String t = Ta.getAttributeValue("vt_end");
        if (t != null) valido_end = java.sql.Date.valueOf(t);
        java.sql.Date efficace_end = null;
        t = Ta.getAttributeValue("et_end");
        if (t != null) efficace_end = java.sql.Date.valueOf(t);

        boolean valido = true;  // controllo valid time
        if (NewVTE == null)
        {
          if (valido_end != null)
            if (NewVTS.compareTo(valido_end) >= 0)
              valido = false;
        }
        else
          if (valido_end != null)
          {
            if ((NewVTE.compareTo(valido_start)) <= 0 ||
                (NewVTS.compareTo(valido_end)) >= 0)
              valido = false;
          }
          else
            if (NewVTE.compareTo(valido_start) <= 0)
              valido = false;

        boolean efficace = true;  // controllo efficacy time
        if (NewETE == null)
        {
          if (efficace_end != null)
            if (NewETS.compareTo(efficace_end) >= 0)
              efficace = false;
        }
        else
          if (efficace_end != null)
          {
            if ((NewETE.compareTo(efficace_start)) <= 0 ||
                (NewETS.compareTo(efficace_end)) >= 0)
              efficace = false;
          }
          else
            if (NewETE.compareTo(efficace_start) <= 0)
              efficace = false;

        ritorno = valido && efficace;  // ritorno vero se valido e efficace
                                       // veri

        if (Ta.getAttribute("tt_end") != null)  // controlla se la versione è
                                                // modificabile
          ritorno = false;
      }

     return ritorno;
     }

public boolean SelezioneTa (Element Ta, java.sql.Date TempoSel)
 {
   // controlla che il parametro sia contenuto nelgi intervalli di validità ed
   // efficacia del TA
   boolean ritorno = false;
   java.sql.Date valido_start =
   java.sql.Date.valueOf(Ta.getAttributeValue("vt_start"));
   java.sql.Date efficace_start =
   java.sql.Date.valueOf(Ta.getAttributeValue("et_start"));
```

```
   java.sql.Date valido_end = null;
   String t = Ta.getAttributeValue("vt_end");
   if (t != null) valido_end = java.sql.Date.valueOf(t);
   java.sql.Date efficace_end = null;
   t = Ta.getAttributeValue("et_end");
   if (t != null) efficace_end = java.sql.Date.valueOf(t);

   boolean valido = false;
   if (valido_end == null)  // controllo sulla validità
   {
     if (TempoSel.after(valido_start))
       valido = true;
   }
   else
   {
     if ((TempoSel.after(valido_start)) && (TempoSel.before(valido_end)))
       valido = true;
   }

   boolean efficace = false;
   if (efficace_end == null)  // controllo sull'efficacia
   {
     if (TempoSel.after(efficace_start))
       efficace = true;
   }
   else
   {
     if ((TempoSel.after(efficace_start)) && (TempoSel.before(efficace_end)))
       efficace = true;
   }

   ritorno = valido && efficace;  // ritorno vero se valido e efficace veri

 return ritorno;
 }

public boolean Selezione (Element Nodo, java.sql.Date TempoSel)
  {
    // Controlla tutti gli elementi che contengono attributi temporali
    // cercando quelli compatibili con il valore passati come parametro.
    boolean ritorno = false;

    java.util.List Temporal = Nodo.getChildren("ta");
    Iterator iterator = Temporal.iterator();

    while (iterator.hasNext() && ritorno == false)
    {
      Element Ta = (Element) iterator.next();

      java.sql.Date valido_start =
      java.sql.Date.valueOf(Ta.getAttributeValue("vt_start"));
      java.sql.Date efficace_start =
      java.sql.Date.valueOf(Ta.getAttributeValue("et_start"));
      java.sql.Date valido_end = null;
      String t = Ta.getAttributeValue("vt_end");
      if (t != null) valido_end = java.sql.Date.valueOf(t);
      java.sql.Date efficace_end = null;
      t = Ta.getAttributeValue("et_end");
      if (t != null) efficace_end = java.sql.Date.valueOf(t);

      boolean valido = false;  // controllo sulla validità
      if (valido_end == null)
```

```
      {
        if (TempoSel.after(valido_start))
          valido = true;
      }
      else
      {
        if ((TempoSel.after(valido_start)) && (TempoSel.before(valido_end)))
          valido = true;
      }

      boolean efficace = false;  // controllo sull'efficacia
      if (efficace_end == null)
      {
        if (TempoSel.after(efficace_start))
          efficace = true;
      }
      else
      {
        if ((TempoSel.after(efficace_start)) &&
            (TempoSel.before(efficace_end)))
          efficace = true;
      }

      ritorno = valido && efficace;  // ritorno vero se valido e efficace veri

      if (Ta.getAttribute("tt_end") != null)   // controlla se la versione è
                                               // modificabile
        ritorno = false;
    }

  return ritorno;
  }

public boolean CercaVersioni_Testo(Element Nodo, int Livello,int Depth,
                                   int[] Indici, Element NewElement,
                                   java.sql.Date NewVTS, java.sql.Date
                                   NewVTE, java.sql.Date NewETS,
                                   java.sql.Date NewETE)
 {
   // metodo centrale per le modifiche testuali
   boolean ritorno=true;
   Vector DaModificare = new Vector();
   java.util.List versioni = Nodo.getChildren();
   Iterator iterator = versioni.iterator();

   try {
    // procedura ricorsiva che perlustra il documento alla ricerca del
    // percorso richiesto dalla modifica
    while (iterator.hasNext())
      {
        Element ver = (Element) iterator.next();

        if (Intersezione(ver,NewVTS,NewVTE,NewETS,NewETE)) // controlla la
                              // compatibilità tra dati esistenti e modifica
          if (Depth < Livello)
          {
            ver = Utility.TrovaNum (ver, Indici[Depth]); // prosegue la
                      // ricerca dentro al componente richiesto dalla modifica
            CercaVersioni_Testo(ver,Livello,Depth+1,Indici,
                                NewElement,NewVTS,NewVTE,NewETS,NewETE);
          }
          else // versione da modificare individuata
```

```
            DaModificare.addElement(new Integer (ver.getAttributeValue("num")));
        }
      } catch (MyException e) { System.err.println("Elemento da modificare non
                                trovato"); ritorno=false;}

       if (ritorno) //se è stato individuato qualcosa da modificare prosegue qui
       try {
         java.sql.Date Trans = java.sql.Date.valueOf(Utility.StringaData());
         for (int i = 0; i < DaModificare.size(); i++) {
           Element VecchiaVer = Utility.TrovaNum(Nodo,
           Integer.parseInt(DaModificare.elementAt(i).toString()));
           Element Aggiunta = null;

           // se il nuovo componente è nullo non si aggiunge, ma si procede
           // comunque con la modifica, in questo modo in pratica si ottiene una
           // abrogazione
           if (NewElement != null)
           { // aggiunge in nuovo componente originato dalla modifica
             Aggiunta = NewElement;
             int NumeroVer = Utility.ContaVer(Nodo) + 1;
             String s = new String("" + NumeroVer);
             Aggiunta.addAttribute("num", s);
             Aggiunta.addAttribute("rif","" +
             (Utility.Intero.nextInt(1000) + 5000));
             // il riferimento alla norma origine della modifca
             // è generato casualmente qui
           }

           java.util.List OldTas = VecchiaVer.getChildren("ta");
           Iterator iterator2 = OldTas.iterator();
           while (iterator2.hasNext())
           {
             // ristrutturazione dei TA e propagazione verso il basso della
             // modifica
             Element OldTa = (Element) iterator2.next();
             if ( (OldTa.getAttribute("tt_end") == null) &&
                (IntersezioneTa(OldTa, NewVTS, NewVTE, NewETS, NewETE)))
             {
               Element Ta = NuoviTa(VecchiaVer, OldTa, Trans, NewVTS,
                                    NewVTE, NewETS, NewETE, CUT);
               if (NewElement != null) Aggiunta.addContent(Ta);
               OldTa.addAttribute("tt_end", Trans.toString());
               PropagaGiu(VecchiaVer, Depth, Trans, NewVTS, NewVTE,
                          NewETS,NewETE);
             }
           }
           if (NewElement != null) Nodo.addContent(Aggiunta);
         }
       } catch (MyException e) {e.printStackTrace();}

     return ritorno;
     }


public Document ModificaTesto (int Cod, int Livello, int[] Indici,
                               Element NewElement, java.sql.Date
                               NewVTS, java.sql.Date NewVTE,
                               java.sql.Date NewETS, java.sql.Date
                               NewETE) throws SQLException
     {
      // procedura iniziale per la modifca del testo, controlla alcuni parametri
      // e avvia la procedura ricorsiva
```

```
    Document legge = null;
    try {
          DBConnect db = new DBConnect();
          legge=db.LeggiLex(Cod,"Legge_I");
        }
    catch (SQLException e) { throw new SQLException("SQL error:
                                                    "+e.getMessage()); }
    finally {DBConnect.Disconnect();}

    Element Root = legge.getRootElement();
    Element Nodo = Root.getChild("articolato");

    if (NewVTS != null)
    {
      if (NewETS == null)
        NewETS = NewVTS;  // se non specificato il nuovo et_start è posto
                          // uguale a vt_start
      CercaVersioni_Testo(Nodo, Livello, 0, Indici, NewElement, NewVTS,
                          NewVTE,NewETS, NewETE);
    }
    else System.err.println("Il nuovo vt_start deve essere specificato");

    AddAttributes aa = new AddAttributes(); // aggiunge i TA se dopo la
                                            // modifca ne mancano
    aa.AggiungiDoc(legge);

   return legge;
   }

public boolean Contenuto(Element Ta, java.sql.Date VerVTS,java.sql.Date
                         VerVTE, java.sql.Date VerETS, java.sql.Date
                         VerETE)
{
   // controlla se un TA è contenuto negli attributi temporali passati come
   // parametri
   boolean ritorno = true;

   if (Ta.getAttribute("tt_end") != null)
     ritorno = false;

   if (ritorno)
     if (VerVTS.before(java.sql.Date.valueOf
        (Ta.getAttributeValue("vt_start"))))
       ritorno = false;
   if (ritorno)
     if (VerETS.before(java.sql.Date.valueOf
        (Ta.getAttributeValue("et_start"))))
       ritorno = false;
   if (ritorno)
     if (Ta.getAttribute("vt_end") != null)
       if (VerVTE == null)
         ritorno = false;
       else if (VerVTE.after(java.sql.Date.valueOf
                (Ta.getAttributeValue("vt_end"))))
         ritorno = false;
   if (ritorno)
     if (Ta.getAttribute("et_end") != null)
       if (VerETE == null)
         ritorno = false;
       else if (VerETE.after(java.sql.Date.valueOf
                (Ta.getAttributeValue("et_end"))))
         ritorno = false;
```

```
return ritorno;
}

public void PropagaSu (Element Nodo, int Depth, java.sql.Date VerVTS,
                       java.sql.Date VerVTE, java.sql.Date VerETS,
                       java.sql.Date VerETE, java.sql.Date Trans)
{


  if (Depth == 0)    // modifica gli attributi di riepilogo di articolato
  {
    // amplia se necessario il valid time start
    if (VerVTS.before(java.sql.Date.valueOf
       (Nodo.getAttributeValue("vt_start"))))
    {
      Nodo.removeAttribute("vt_start");
      Nodo.addAttribute("vt_start",VerVTS.toString());
    }
    // amplia se necessario l' efficacy time start
    if (VerETS.before(java.sql.Date.valueOf
       (Nodo.getAttributeValue("et_start"))))
    {
      Nodo.removeAttribute("et_start");
      Nodo.addAttribute("et_start",VerETS.toString());
    }
    // amplia se necessario il valid time end
    if (Nodo.getAttribute("vt_end") != null)
      if (VerVTE != null)
      {
        if (VerVTE.after(java.sql.Date.valueOf
           (Nodo.getAttributeValue("vt_end"))))
        {
          Nodo.removeAttribute("vt_end");
          Nodo.addAttribute("vt_end", VerVTE.toString());
        }
      }
      else Nodo.removeAttribute("vt_end");
    // amplia se necessario l' efficacy time end
    if (Nodo.getAttribute("et_end") != null)
      if (VerETE != null)
      {
        if (VerETE.after(java.sql.Date.valueOf
           (Nodo.getAttributeValue("et_end"))))
        {
          Nodo.removeAttribute("et_end");
          Nodo.addAttribute("et_end", VerETE.toString());
        }
      }
      else Nodo.removeAttribute("et_end");
  }
  else    // modifica gli attributi temporali dei livelli intermedi
  {
    Element Ver = Nodo.getParent();
    java.util.List OldTas = Ver.getChildren("ta");
    Iterator iterator = OldTas.iterator();
    boolean contenuto = false;
    /*
       controllo se l'intervallo temporale da propagare è o meno già contenuto
       in un altro intervallo.
    */
    while (iterator.hasNext() && contenuto == false)
```

```
    {
      Element OldTa = (Element) iterator.next();
      contenuto = Contenuto(OldTa,VerVTS, VerVTE, VerETS, VerETE);
    }

    if (!contenuto)   // intervallo non contenuto
    {
      iterator = OldTas.iterator();
      while (iterator.hasNext())
      {
        Element OldTa = (Element) iterator.next();
        if ((OldTa.getAttribute("tt_end") == null) &&
           (IntersezioneTa(OldTa, VerVTS, VerVTE, VerETS, VerETE)))
        {
          Element Ta = NuoviTa(Ver, OldTa, Trans, VerVTS, VerVTE, VerETS,
                               VerETE, CUT);
          OldTa.addAttribute("tt_end",Trans.toString());
          Ver.addContent(Ta);
        }
      }
    }
    PropagaSu (Ver.getParent(), Depth-1, VerVTS, VerVTE, VerETS,VerETE,
               Trans); // propagazione verso l'alto
  }
}

public void PropagaLato (Element Nodo, int Depth, Element VecchiaVer,
                         java.sql.Date VerVTS, java.sql.Date VerVTE,
                         java.sql.Date VerETS, java.sql.Date VerETE,
                         java.sql.Date Trans)
{
  // le versioni adiacenti a quella colpita dalla modifica temporale sono qui
  // opportunamente trattate andando a togliere le eventuali parti ora
  // contenute nella versione modificata.
  java.util.List versioni3 = Nodo.getChildren();
  Iterator iterator3 = versioni3.iterator();

  while (iterator3.hasNext()) {
    Element ver = (Element) iterator3.next();
    if (ver.getAttributeValue("num").compareTo
       (VecchiaVer.getAttributeValue("num"))
        != 0)
    {
      java.util.List Tas = ver.getChildren("ta");
      Iterator iterator4 = Tas.iterator();
      while (iterator4.hasNext()) {
        Element Taa = (Element) iterator4.next();
        if ((Taa.getAttribute("tt_end") == null) &&
           (IntersezioneTa(Taa, VerVTS, VerVTE, VerETS, VerETE)))
        {
          NuoviTa(ver, Taa, Trans, VerVTS, VerVTE, VerETS, VerETE, CUT);
          Taa.addAttribute("tt_end", Trans.toString());
          /*
             ogni variazione viene propagata ai livelli sottostanti.
          */
          PropagaGiu(ver, Depth, Trans, VerVTS, VerVTE, VerETS, VerETE);
        }
      }
    }
  }
}
```

```
public boolean CercaVersioni_Tempo (Element Nodo, int Livello, int
                                    Depth, int[] Indici, java.sql.Date
                                    TempoSel, java.sql.Date NewVTS,
                                    java.sql.Date NewVTE, java.sql.Date
                                    NewETS, java.sql.Date NewETE)
{
  // metodo fondamentale per le modifiche dei tempi, scansiona la strutture
  // del documento alla ricerca della porzione da modificare, dopo alcuni
  // controlli esegue la modifica e ne propaga gli effetti.
  boolean ritorno = true;
  Vector DaModificare = new Vector();
  java.util.List versioni = Nodo.getChildren();
  Iterator iterator = versioni.iterator();

  try {
   while (iterator.hasNext())
     {
       // esplorazione ricorsiva della struttura ad albero del documento.
       Element ver = (Element) iterator.next();

        if (Selezione(ver,TempoSel))
         if (Depth < Livello)
          {
            ver = Utility.TrovaNum (ver, Indici[Depth]);
            CercaVersioni_Tempo(ver,Livello,Depth+1,Indici,TempoSel,
            NewVTS,NewVTE,NewETS,NewETE);
          }
         else
           // metto in DaModificare tutte le versioni colpite dalla modifica.
           DaModificare.addElement(new Integer
                                   (ver.getAttributeValue("num")));
    }
   } catch (MyException e) { System.err.println("Elemento da modificare non
                             trovato"); ritorno=false;}

   if (ritorno)  // se almeno uno da modificare è stato trovato proseguo con
                 // la modifica.
   try {
     java.sql.Date Trans = java.sql.Date.valueOf(Utility.StringaData());
     for (int i = 0; i < DaModificare.size(); i++) {
       Element VecchiaVer =
       Utility.TrovaNum(Nodo,Integer.parseInt
                       (DaModificare.elementAt(i).toString()));

       java.util.List OldTas = VecchiaVer.getChildren("ta");
       Iterator iterator2 = OldTas.iterator();
       while (iterator2.hasNext())
       {
         boolean Errore = false;
         Element OldTa = (Element) iterator2.next();

         if ((OldTa.getAttribute("tt_end") == null) &&
            (SelezioneTa (OldTa,TempoSel)))
         {
           Element Ta = new Element("ta");
           Ta.addAttribute("tt_start", Trans.toString());

           // controlli che garantiscono la natura di solo ampliamento delle
           // modifiche dei tempi.
           if (NewVTS != null)
           {
             if (NewVTS.after(java.sql.Date.valueOf(
```

```
        OldTa.getAttributeValue("vt_start"))))
    {
      Errore = true;
      System.err.println("NewVTS non può essere successivo al
                          vt_start della versione  -  le modifiche
                          dei tempi sono solo di ampliamento");
    }
    else
      Ta.addAttribute("vt_start", NewVTS.toString());
  }
  else
    Ta.addAttribute("vt_start",
    OldTa.getAttributeValue("vt_start"));
  if (NewETS != null)
  {
    if (NewETS.after(java.sql.Date.valueOf(
        OldTa.getAttributeValue("et_start"))))
    {
      Errore = true;
      System.err.println("NewETS non può essere successivo al
                          et_start della versione  -  le modifiche
                          dei tempi sono solo di ampliamento");
    }
    else
      Ta.addAttribute("et_start", NewETS.toString());
  }
  else
    Ta.addAttribute("et_start",
    OldTa.getAttributeValue("et_start"));
  if (NewVTE != null)
    if (OldTa.getAttribute("vt_end") != null)
      if (NewVTE.before(java.sql.Date.valueOf(
          OldTa.getAttributeValue("vt_end"))))
      {
        Errore = true;
        System.err.println("NewVTE non può essere precedente al
                            vt_end della versione  -  le modifiche
                            dei tempi sono solo di ampliamento");
      }
      else
        Ta.addAttribute("vt_end", NewVTE.toString());
    else
    {
      Errore = true;
     System.err.println("NewVTE non può essere precedente al vt_end
                         della versione  -  le modifiche dei tempi
                         sono solo di ampliamento");
    }

  if (NewETE != null)
    if (OldTa.getAttribute("et_end") != null)
      if (NewETE.before(java.sql.Date.valueOf(
          OldTa.getAttributeValue("et_end"))))
      {
        Errore = true;
        System.err.println("NewETE non può essere precedente al
                            et_end della versione  -  le modifiche
                            dei tempi sono solo di ampliamento");
      }
      else
        Ta.addAttribute("et_end", NewETE.toString());
    else
```

```
  {
    Errore = true;
    System.err.println("NewETE non può essere precedente al et_end
                        della versione  -  le modifiche dei tempi
                        sono solo di ampliamento");
  }

if (!Errore)  // se i controlli sono andati a buon fine proseguo.
{
  OldTa.addAttribute("tt_end", Trans.toString());
  VecchiaVer.addContent(Ta);

  java.sql.Date VerVTS = null;
  java.sql.Date VerVTE = null;
  java.sql.Date VerETS = null;
  java.sql.Date VerETE = null;
  boolean allargaRG = false;

  // qui vengono valutate le "direzioni" in cui la modifica
  // provoca un ampliamento delle dimensioni temporali, ai fini
  // della propagazione "laterale" degli effetti della modifica.
  // La prima direzione analizzata è "Destra"
  if (OldTa.getAttribute("vt_end") != null)
  {
    if (NewVTE == null) allargaRG = true;
    else if (java.sql.Date.valueOf(
             OldTa.getAttributeValue("vt_end")).before(NewVTE))
      allargaRG = true;

    if (allargaRG)
    {
      VerVTS = java.sql.Date.valueOf(
      OldTa.getAttributeValue("vt_end"));
      VerETS =
      java.sql.Date.valueOf(Ta.getAttributeValue("et_start"));
      if (Ta.getAttribute("vt_end") != null)
        VerVTE =
        java.sql.Date.valueOf(Ta.getAttributeValue("vt_end"));
      if (Ta.getAttribute("et_end") != null)
        VerETE =
        java.sql.Date.valueOf(Ta.getAttributeValue("et_end"));

      PropagaLato(Nodo, Depth, VecchiaVer, VerVTS, VerVTE,
                  VerETS,VerETE, Trans);
    }
  }

  VerVTS = null; // giu
  VerVTE = null;
  VerETS = null;
  VerETE = null;
  boolean allargaDW = false;

  if (OldTa.getAttribute("et_end") != null)
  {
    if (NewETE == null) allargaDW = true;
    else if (java.sql.Date.valueOf(
             OldTa.getAttributeValue("et_end")).before(NewETE))
      allargaDW = true;

    if (allargaDW)
    {
```

```
    VerVTS = java.sql.Date.valueOf(
    OldTa.getAttributeValue("vt_start"));
    VerETS = java.sql.Date.valueOf(
    OldTa.getAttributeValue("et_end"));
    if (OldTa.getAttribute("vt_end") != null)
      VerVTE = java.sql.Date.valueOf(
      OldTa.getAttributeValue("vt_end"));
    if (Ta.getAttribute("et_end") != null)
      VerETE =
      java.sql.Date.valueOf(Ta.getAttributeValue("et_end"));

    PropagaLato(Nodo, Depth, VecchiaVer, VerVTS, VerVTE,
                VerETS,VerETE, Trans);
  }
}

VerVTS = null;  // sinistra
VerVTE = null;
VerETS = null;
VerETE = null;
boolean allargaLF = false;

if (NewVTS != null)
  if (java.sql.Date.valueOf(
      OldTa.getAttributeValue("vt_start")).after(NewVTS))
    allargaLF = true;

if (allargaLF)
{
  VerVTS =
  java.sql.Date.valueOf(Ta.getAttributeValue("vt_start"));
  VerETS =
  java.sql.Date.valueOf(Ta.getAttributeValue("et_start"));
  VerVTE = java.sql.Date.valueOf(
  OldTa.getAttributeValue("vt_start"));
  if (Ta.getAttribute("et_end") != null)
    VerETE =
    java.sql.Date.valueOf(Ta.getAttributeValue("et_end"));

  PropagaLato(Nodo, Depth, VecchiaVer, VerVTS, VerVTE, VerETS,
              VerETE, Trans);
}

VerVTS = null; // su
VerVTE = null;
VerETS = null;
VerETE = null;
boolean allargaUP = false;

if (NewETS != null)
 if (java.sql.Date.valueOf(
     OldTa.getAttributeValue("et_start")).after(NewETS))
  allargaUP = true;

if (allargaUP)
{
  VerVTS = java.sql.Date.valueOf(
  OldTa.getAttributeValue("vt_start"));
  VerETS =
  java.sql.Date.valueOf(Ta.getAttributeValue("et_start"));
  if (OldTa.getAttribute("vt_end") != null)
    VerVTE = java.sql.Date.valueOf(
```

```
                    OldTa.getAttributeValue("vt_end"));
                  VerETE = java.sql.Date.valueOf(
                  OldTa.getAttributeValue("et_start"));

                  PropagaLato(Nodo, Depth, VecchiaVer, VerVTS, VerVTE, VerETS,
                              VerETE, Trans);
                }

                // propagazione verso i livelli superiori.
                PropagaSu(Nodo, Depth, NewVTS,NewVTE,NewETS,NewETE, Trans);
              }
            }
          }
        }
      } catch (MyException e) {e.printStackTrace();}

  return ritorno;
  }

  public Document ModificaTempo (int Cod, int Livello, int[] Indici,
                                 java.sql.Date TempoSel, java.sql.Date
                                 NewVTS, java.sql.Date NewVTE,
                                 java.sql.Date NewETS, java.sql.Date
                                 NewETE) throws SQLException
    {
      // metodo iniziale per la modifica dei tempi, controlla alcuni parametri e
      // avvia la procedura ricorsiva
      Document legge = null;
      try
      {
        DBConnect db = new DBConnect();
        legge=db.LeggiLex(Cod,"Legge_I");
      }
      catch (SQLException e)
       { throw new SQLException("SQL error: "+e.getMessage()); }
      finally {DBConnect.Disconnect();}
      Element Nodo = null;
      Element Root = legge.getRootElement();
      Nodo = Root.getChild("articolato");

      if (TempoSel != null)
      {
        if (NewETS == null)
          NewETS = NewVTS; // se non specificato il nuovo et_start è posto
                           // uguale a vt_start
        CercaVersioni_Tempo(Nodo, Livello, 0, Indici, TempoSel, NewVTS,
                            NewVTE,NewETS, NewETE);
      }
      else System.err.println("Il tempo di selezione deve essere specificato");

    return legge;
    }
}



package Principale;

import java.sql.*;
import java.awt.*;

/**
```

```
     //\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\
                            Main
     //\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\
                       Classe main
     //\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\\
*/


public class Start {

  public static void main(String[] args) throws SQLException
   {
     MainFrame f = new MainFrame();

     // Center the frame
     Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
     Dimension frameSize = f.getSize();
     if (frameSize.height > screenSize.height)
     frameSize.height = screenSize.height;
     if (frameSize.width > screenSize.width)
     frameSize.width = screenSize.width;
     f.setLocation((screenSize.width - frameSize.width) / 2,
     (screenSize.height - frameSize.height) / 2);

     f.setVisible(true);
   }
}
```

Appendice B

SCHEMI XML E DOCUMENTI DI ESEMPIO

Schema originale, non supporta gli elementi temporali

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSPY v5 rel. 2 U -->
<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>
          <xs:attribute name="num" type="xs:integer" use="required"/>
```

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

Schema modificato, supporta gli elementi temporali

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSPY v5 rel. 2 U -->
<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>
          <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="required"/>
```

```
      <xs:attribute name="tt_start" type="transaction_time" use="requird"/>
      <xs:attribute name="tt_end" type="transaction_time" use="required"/>
      <xs:attribute name="et_start" type="ud_time" use="required"/>
      <xs:attribute name="et_end" type="ud_time" use="required"/>
      </xs:complexType>
     </xs:element>
     </xs:sequence>
     <xs:attribute name="num" type="xs:integer" use="required"/>
     <xs:attribute name="rif" 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="required"/>
    <xs:attribute name="tt_start" type="transaction_time" use="required"/>
    <xs:attribute name="tt_end" type="transaction_time" use="required"/>
    <xs:attribute name="et_start" type="ud_time" use="required"/>
    <xs:attribute name="et_end" type="ud_time" use="required"/>
    </xs:complexType>
   </xs:element>
   </xs:sequence>
   <xs:attribute name="num" type="xs:integer" use="required"/>
   <xs:attribute name="rif" 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="required"/>
  <xs:attribute name="tt_start" type="transaction_time" use="required"/>
  <xs:attribute name="tt_end" type="transaction_time" use="required"/>
  <xs:attribute name="et_start" type="ud_time" use="required"/>
  <xs:attribute name="et_end" type="ud_time" use="required"/>
  </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="required"/>
 <xs:attribute name="tt_start" type="transaction_time" use="required"/>
 <xs:attribute name="tt_end" type="transaction_time" use="required"/>
 <xs:attribute name="et_start" type="ud_time" use="required"/>
 <xs:attribute name="et_end" type="ud_time" use="required"/>
 </xs:complexType>
```

```
    </xs:element>
    </xs:sequence>
    <xs:attribute name="num" type="xs:integer" use="required"/>
    <xs:attribute name="rif" 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"/>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Un documento conforme al secondo schema: esempio 1

```
<legge num="2">
 <titolo>Titolo Legge 2</titolo>
 <natura>Delibera</natura>
 <articolato pubblicazione="1962-03-04" vt_start="1962-03-19" et_start=
  "1962-03-19" tt_start="1962-03-12">
  <ver num="1">
  <capo num="1">
   <ver num="1">
   <articolo num="1">
    <ver num="1">
    <comma num="1">
     <ver num="1">
      Testo della Versione 1 del C.a 1 (Versione 1
      A. 1, Versione 1 C. 1, Versione 1 dell'art.)
      <ta tt_start="1962-03-12" vt_start="1962-03-19" et_start="1962-03-19"
       tt_end="2003-04-14" />
      <ta tt_start="2003-04-14" vt_start="1962-03-19" vt_end="1990-01-01"
       et_start="1962-03-19" />
      <ta tt_start="2003-04-14" vt_start="1990-01-01" vt_end="1995-02-21"
       et_start="1962-03-19" et_end="1990-01-01" />
      <ta tt_start="2003-04-14" vt_start="1995-02-21" et_start=
       "1962-03-19" /></ver>
      <ver num="2" rif="5790">
       nuovo comma
      <ta tt_start="2003-04-14" vt_start="1990-01-01" et_start="1990-01-01"
       vt_end="1995-02-21" />
     </ver>
    </comma>
    <ta tt_start="1962-03-12" vt_start="1962-03-19" et_start="1962-03-19" />
    </ver>
   </articolo>
   <ta tt_start="1962-03-12" vt_start="1962-03-19" et_start="1962-03-19" />
   </ver>
  </capo>
  <ta tt_start="1962-03-12" vt_start="1962-03-19" et_start="1962-03-19" />
  </ver>
 </articolato>
```

```
</legge>
```

Un documento conforme al secondo schema: esempio 2

```
<legge num="2">
 <titolo>Titolo Legge 2</titolo>
 <natura>Delibera</natura>
 <articolato pubblicazione="1962-03-04" vt_start="1962-03-19" et_start=
  "1962-03-19" tt_start="1962-03-12">
  <ver num="1">
  <capo num="1">
   <ver num="1">
   <articolo num="1">
    <ver num="1">
    <comma num="1">
     <ver num="1">
      Testo della Versione 1 del C.a 1 (Versione
      1 A. 1, Versione 1 C. 1, Versione 1 dell'art.)
     <ta tt_start="1962-03-12" vt_start="1962-03-19" et_start="1962-03-19"
      tt_end="2003-04-14" />
     <ta tt_start="2003-04-14" vt_start="1962-03-19" vt_end="1990-01-01"
      et_start="1962-03-19" />
     <ta tt_start="2003-04-14" vt_start="1990-01-01" vt_end="1995-02-21"
      et_start="1962-03-19" et_end="1990-01-01" />
     <ta tt_start="2003-04-14" vt_start="1995-02-21" et_start="1962-03-19"
      tt_end="2003-04-23" />
     <ta tt_start="2003-04-23" vt_start="1995-02-21" vt_end="2000-02-21"
      et_start="1962-03-19" et_end="1990-01-01" />
     <ta tt_start="2003-04-23" vt_start="2000-02-21"et_start="1962-03-19" />
    </ver>
    <ver num="2" rif="5790">
     nuovo comma
     <ta tt_start="2003-04-14" vt_start="1990-01-01" et_start="1990-01-01"
      vt_end="1995-02-21" tt_end="2003-04-23" />
     <ta tt_start="2003-04-23" vt_start="1990-01-01" et_start="1990-01-01"
      vt_end="2000-02-21" />
     </ver>
    </comma>
    <ta tt_start="1962-03-12" vt_start="1962-03-19" et_start="1962-03-19" />
    </ver>
   </articolo>
   <ta tt_start="1962-03-12" vt_start="1962-03-19" et_start="1962-03-19" />
   </ver>
  </capo>
  <ta tt_start="1962-03-12" vt_start="1962-03-19" et_start="1962-03-19" />
  </ver>
 </articolato>
</legge>
```

Appendice C

INVERTED INDEX

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

Parola	Frequenza
Termine 1	Frequenza 1
…	…
Termine i	Frequenza i
…	…
Termine n	Frequenza n

Tab. C.1

che associa ogni parola al rispettivo numero di occorrenze nel testo, viene individuata una *soglia di frequenza* al di sopra della quale si devono eliminare le parole. Solitamente la soglia è del 25% - 30%: una parola viene eliminata

dall'indice se è ripetuta più del 25% di volte sul totale delle parole presenti nel documento. In questa maniera si crea una lista di termini (estratti dal documento principale) ripulita da tutte quelle parole comuni come possono essere gli articoli, gli avverbi, le congiunzioni, ecc. Un altro metodo per effettuare la "scrematura" dalle *noise words* consiste nell'utilizzare quella che viene definita *Stop List*: si tratta di una lista che contiene tutte quelle parole che non si vogliono indicizzare perché o troppo comuni o di scarso significato ai fini del recupero.

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

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

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

Parola	Frequenza	Locazione
Animale	Frequenza 1	Documento1.txt, Prova42.txt, Documento78.txt
…	…	…
Cane	Frequenza i	Prova12.txt, Documento1.txt
…	…	…
Uomo	Frequenza n	Documento4.txt

Tab. C.2

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

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

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

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

Appendice D

INTRODUZIONE AL LINGUAGGIO JAVA E ALLA CONNESSIONE AD UNA BASE DI DATI ORACLE

Per rendere più comprensibile l'analisi del prototipo realizzato, riportiamo una breve introduzione agli aspetti salienti del linguaggio Java[27], in particolare presenteremo gli strumenti per la connessione a un database Oracle.

D.1. Caratteristiche del linguaggio Java

Java è un linguaggio di programmazione orientato agli oggetti sviluppato dalla Sun Microsystems, particolarmente adatto all'uso su Internet. Si tratta in parte di un'evoluzione del C++ dal quale Java eredita i concetti orientati ad oggetti e ridefinisce altri aspetti, come ad esempio la gestione della memoria, che diventa automatica.

La programmazione orientata agli oggetti organizza un programma come una serie di componenti chiamati *oggetti*, separati l'uno dall'altro, che rispettano regole precise per la comunicazione tra loro.

Uno dei motivi del grande successo di Java è l'indipendenza dalla piattaforma, la capacità, in altre parole, di poter eseguire un programma su sistemi differenti. I tipi di variabili di Java hanno la stessa dimensione su ogni piattaforma, perciò i bytes che rappresentano un certo tipo sono gli stessi, indipendentemente dal sistema usato per la compilazione. Un file con estensione .class di Java, composto da istruzioni in bytecode, può essere eseguito su qualsiasi piattaforma senza alterazioni (i programmi Java sono compilati in una serie di istruzioni dette appunto bytecode).

Un altro importante punto di forza del linguaggio sta nella sicurezza riguardo all'esecuzione di un programma Java, detto applet, effettuata da una pagina Web: quando un browser compatibile con Java riscontra un'applet, la preleva insieme al testo e alle immagini della pagina, e quindi la esegue sul computer locale. Ciò può essere pericoloso, perché l'esecuzione di un programma può portare virus o altri problemi. Al fine di ridurre tali insidie, Java predispone diversi livelli di sicurezza: in primo luogo, per merito della mancanza dei puntatori, Java risulta essere un linguaggio molto più sicuro dei suoi predecessori, inoltre dispone di un verificatore di bytecode, che controlla, prima dell'esecuzione, ogni bytecode per garantirne l'integrità. Oltre a queste misure generiche, ve ne sono poi di specifiche per le applet, secondo l'impostazione di default ad esempio, le applet non possono leggere o scrivere files sul sitema dell'utente, riducendo così le possibilità di danneggiamento dei dischi locali.
Queste caratteristiche hanno fatto di Java un linguaggio che ben si adatta alle problematiche di una rete eterogenea e mutevole come Internet.

D.2. Presentazione del linguaggio Java

In questo paragrafo presenteremo gli elementi specifici del linguaggio Java che forniscono supporto alla programmazione ad oggetti: le classi, i package e le interfacce.

D.2.1. Le classi

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

Nella stesura di un programma Java, si progetta e si costruisce una famiglia di classi. Durante l'esecuzione del programma, invece, si creano e rimuovono istanze di tali classi.
Il compito del programmatore consiste nel creare, o utilizzare, un gruppo di classi che siano adeguate agli scopi del programma.
Una classe è composta essenzialmente da due parti: gli attributi (variabili istanza e di classe) e il comportamento (metodi istanza e di classe).

D.2.1.1. Variabili istanza

Le variabili istanza definiscono gli attributi di un oggetto. La classe definisce il tipo dell'attributo, ogni istanza contiene il proprio valore dell'attributo.
Una variabile si considera d'istanza se dichiarata al di fuori dalle definizioni dei metodi, essa può essere impostata al momento della creazione dell'oggetto oppure usata come ogni altra variabile e modificata a piacere durante l'esecuzione del programma.

D.2.1.2. Variabili di classe

Le variabili di classe sono visibili a tutta una classe e a tutte le sue istanze; possono essere considerate più generali delle variabili istanza.
Le variabili di classe sono spesso usate per mantenere traccia di uno stato globale, relativo ad una famiglia di oggetti, oppure per le comunicazioni tra istanze della stessa classe.
La dichiarazione di variabili di classe avviene premettendo la parola *static* alla normale dichiarazione.

D.2.1.3. Metodi istanza

Un metodo istanza è una funzione definita all'interno di una classe, che può agire sulle istanze della classe. Gli oggetti comunicano fra loro attraverso i metodi, una classe può chiamare i metodi di un'altra per comunicarle cambiamenti avvenuti nell'ambiente, o per chiederle di modificare il proprio stato.

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

Nel caso un metodo ritorni un valore, è necessario usare l'istruzione *return*, che provvede alla restituzione del risultato.

Il passaggio dei parametri avviene per valore per i tipi primitivi, gli oggetti sono invece passati per riferimento, le operazioni svolte su di esse nel metodo che li riceve hanno quindi effetto sugli originali.

D.2.1.4. Metodi di classe

I metodi di classe sono disponibili per ogni istanza della classe e possono essere messi a disposizione di altre classi, possono perciò essere richiamati indipendentemente dall'esistenza di un'istanza della classe relativa.

Come per la variabili, un metodo di classe si definisce attraverso l'uso della clausola *static*.

Come regola generale, i metodi che operano su un oggetto specifico dovrebbero essere definiti come metodi di istanza, mentre i metodi che forniscono funzioni di utilità generica, ma non agiscono direttamente sulle istanze, possono ricoprire il ruolo di metodi di classe.

D.2.1.5. Metodi costruttori

La definizione di una classe può contenere, oltre ai metodi normali, dei metodi definiti *costruttori*. Tali metodi non possono essere chiamati direttamente, come per i metodi normali, ma vengono richiamati automaticamente da Java.

Quando, mediante l'operatore *new*, si crea una nuova istanza di una classe, java compie diversi passi, fra i quali invocare il metodo costruttore. Nel caso di una classe che non disponga di costruttori, l'oggetto viene comunque creato, ma può essere necessario impostare le variabili istanza, oppure invocare altri metodi che inizializzino l'oggetto.

Grazie ai metodi costruttori è possibile impostare i valori delle variabili istanza e inizializzare l'oggetto in base a quei valori. Per una classe, si possono definire più costruttori omonimi, come per i normali metodi, che si differenziano per il numero e il tipo dei parametri.

La differenza fondamentale tra i costruttori e gli altri metodi è che il costruttore ha sempre lo stesso nome della classe e non restituisce un risultato.

D.2.1.6. Metodi conclusivi

Il metodo conclusivo è invocato immediatamente prima che un oggetto venga eliminato e la sua memoria liberata. La classe *Object* definisce un metodo conclusivo che non esegue nessuna operazione. Per definire un metodo conclusivo per una qualsiasi classe bisogna ridefinire il metodo *finalize()*.

Questo metodo può essere chiamato esplicitamente, come ogni altro metodo, la sua invocazione non provoca la rimozione dell'oggetto.

I metodi conclusivi sono normalmente sfruttati per ottimizzare la rimozione di un oggetto: ad esempio per liberare risorse esterne o per cancellare riferimenti ad altri oggetti.

D.2.1.7. Variabili locali

Nelle definizioni dei metodi è possibile dichiarare variabili rilevanti solo all'interno di essi, come ad esempio contatori per i cicli. Le variabili locali si possono utilizzare anche all'interno dei blocchi. In ogni caso, al termine dell'esecuzione del metodo o del blocco, le sue variabili locali vengono eliminate e di conseguenza i loro valori perduti.

D.2.1.8. Costanti

Per assegnare nomi descrittivi a valori invariabili e renderli disponibili a tutti i metodi è possibile definire costanti, mediante l'uso della parola chiave *final* inserita prima della dichiarazione del tipo.

In Java si possono creare costanti solo per le variabili istanza e di classe, non per quelle locali.

D.2.2. Le interfacce

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

I vantaggi nell'uso delle intefacce sono gli stessi offerti dall'utilizzo delle classi astratte. Le interfacce costituiscono un mezzo per definire protocolli per una classe, svincolandosi completamente dai dettagli dell'implementazione.

Una classe può implementare diverse interfacce, simulando così il concetto di ereditarietà multipla, con questo sistema però è possibile ereditare soltanto le descrizioni dei metodi, non la loro implementazione. Quando una classe implementa un'interfaccia, deve fornire tutte le funzionalità per i metodi definiti in questa.

D.2.3. I package

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

E' possibile strutturare ulteriormente i package in una gerarchia simile, sotto certi aspetti, alla gerarchia ereditaria, dove ogni livelli rappresenta un gruppo più ristretto e specifico di classi. Anche la libreria di classi di Java ha questo tipo di organizzazione.

Grazie all'utilizzo dei package si possono risolvere conflitti tra nomi di classi create da diversi soggetti in tempi diversi.

Ogni volta che nel proprio codice Java si fa riferimento a una classe attraverso il suo nome, si utilizza un package. Spesso non ci si rende conto di questo fatto, perché molte delle classi più comunemente usate, si trovano in un package che il compilatore importa automaticamente, chiamato java.lang. Per le classi non appartenenti a questo package bisogna far precedere al loro nome quello del package che le contiene, formando così un riferimento unico. Tenuto conto che tale procedimento può rivelarsi scomodo, soprattutto nel caso di package con nomi molto lunghi, è consentito "importare" i nomi delle classi desiderate nel proprio programma, dopo questa operazione è possibile riferirsi ad esse senza alcun prefisso.

Tutte le istruzioni di *import* devono trovarsi dopo *package*, ma prima di ogni definizione di classe. L'importazione avviene attraverso l'istruzione:

`import <nome package>.<nome classe>`.

Spesso al posto di `<nome classe>` viene posto l'asterisco (*) per consentire l'utilizzo di tutte le classi del package, ma non quelle dei sottopackage.

D.2.4. Alcune parole chiave

extends: è usata nelle definizioni delle classi per indicare se la classe che si sta generando sia sottoclasse di un'altra.

new: quando si crea un nuovo oggetto, si usa *new*, seguito dal nome dalla classe di cui si vuole creare un'istanza e dalle parentesi tonde. Le parentesi non possono essere omesse, anche nel caso siano vuote. All'interno di esse è possibile specificare parametri che saranno trattati dai metodi costruttori.

Ogni volta si usa la *new*, Java esegue una sequenza di tre operazioni:

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

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

L'omissione di *this* per le variabili istanza è possibile se non esistono variabili locali con lo stesso nome. Dato che *this* è un riferimento all'istanza corrente di una classe, lo si può utilizzare solo nel corpo di un metodo istanza. I metodi di classe non possono usare *this*.

D.3. La classe "Vector"

Alcuni tipi di strutture dati, come gli elenchi a collegamento dinamico e le code richiedono, nella loro implementazione, l'utilizzo dei puntatori. In Java non è possibile crearle direttamente perché i puntatori non esistono. Per sopperire a questa

mancanza è fornita la classe Vector, essa può gestire le occasioni in cui è necessario memorizzare dinamicamente gli oggetti.
Rispetto all'uso diretto dei puntatori, la classe Vector presenta vantaggi e svantaggi. Ad esempio contribuisce a mantenere semplice il linguaggio e leggibile il codice, limita però i programmatori nell'utilizzo dei programmi più sofisticati.
Per ottimizzarne l'utilizzo, l'oggetto Vector dispone di diverse proprietà che si possono impostare. Per esempio si può decidere la capacità prima di inerire un gran numero di oggetti, riducendo così la necessità di riallocare l'elenco per incrementarlo. Anche la capacità di memorizzazione iniziale e l'incremento della capacità possono essere specificate nel costruttore. La capacità è aumentata in modo automatico, è però possibile aumentarla di un numero minimo di elementi per volta.
Al Vector possono essere aggiunti elementi grazie a due metodi: *addElement(),* che aggiunge l'elemento alla fine del Vector, e *insertElementAt()*, che invece inserisce l'elemento in una posizione specificata.
E' possibile accedere agli elementi di un Vector attraverso vari metodi, *elementAt()* ad esempio si mette sull'elemento la cui posizione è indicata come parametro, *firstElement()* si posiziona sul primo elemento del Vector e *lastElement()* sull'ultimo.
Un altro metodo di accesso si ottiene con l'uso dell'oggetto *Enumeration*. Esso è un'interfaccia che specifica una serie di metodi usati per enumerare un elenco. Un oggetto che implementa questa interfaccia può essere usato per iterare un elenco una sola volta, in quanto l'oggetto Enumeration si consuma nell'utilizzo.
L'interfaccia Enumeration specifica due soli metodi: *hasMoreElements(),* che restituisce *true* se l'oggetto ha altri elementi, e *nextElement()* che restituisce il prossimo elemento dell'oggetto Enumeration.
Il seguente codice:

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

vediamo come l'oggetto Enumeration viene usato per accedere agli elementi di un Vector(v). L'oggetto Enumeration viene creato dal metodo *elements()* di Vector.

D.4. Modificatori di accesso

Il modificatore di accesso di default specifica che soltanto le classi all'interno dello stesso package abbiano accesso alle variabili e ai metodi di una classe. In altre parole i membri di una classe con accesso predefinito hanno visibilità limitata alle altre classi dello stesso package.
In assenza di altri modificatori di accesso viene applicato automaticamente quello predefinito, non esistono parole chiave per indicarlo.

D.4.1. Public

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

D.4.2. Protected

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

D.4.3. Private

Il modificatore di accesso più restrittivo è *private*, esso specifica che i membri di una classe sono accessibili solamente alla classe in cui sono definiti. In questo caso perciò, nessuna altra classe, sottoclassi incluse, ha accesso ai membri *private* di una classe.

D.4.4. Synchronized

Questo modificatore viene usato per specificare che un metodo è a *thread protetto*, vale a dire che in un metodo *synchronized* è concesso un solo percorso di esecuzione alla volta. In un ambiente multithreading come quello di Java, è normalmente possibile che nello stesso codice vengano eseguiti in contemporanea diversi percorsi di esecuzione. Attraverso il modificatore *synchronized* si evade a questa regola, permettendo ad un solo therad di accedere al metodo in un dato momento, tutti gli altri thread dovranno attendere il loro turno.

D.4.5. Native

Si usa questo modificatore per identificare i metodi con implementazione nativa. Esso comunica al compilatore che l'implementazione di un metodo si trova in un file esterno C. Le dichiarazioni di metodi *native*, sono per questo motivo diverse dalle altre: non hanno corpo.

La dichiarazione di un metodo *native* termina semplicemente con un punto e virgola, senza codice Java racchiuso tra parentesi graffe, ciò perché i metodi nativi sono implementati nel codice C, che risiede sin file sorgenti C esterni.

D.5. Garbage Collector

Come già accennato, Java non offre al programmatore la possibilità di deallocare gli oggetti in maniera esplicita. Quando non esistono più riferimenti a un oggetto, cioè quando non ci sono più oggetti che lo utilizzano, lo si può distruggere.

Il *garbage collector* è una speciale routine di sistema che scandisce periodicamente la zona di memoria che contiene gli oggetti (Java Heap), alla ricerca di oggetti non più referenziati, allo scopo di eliminarli.

Uno svantaggio derivante dall'uso di questa routine è la riduzione delle prestazioni, ciò è provocato dal fatto che il garbage collector è a tutti gli effetti un thread che

viene eseguito in parallelo a ogni altro programma. Anche se la sua priorità è minima, e quindi viene avviato solo quando non ci sono altri thread attivi, è comunque da considerarsi in termini di tempo di CPU. Inoltre la continua allocazione e deallocazione aumenta la frammentazione della memoria.
Il garbage collector è considerato un processo *demon*, in quanto non gestito dal programmatore e sempre presente in background.
Ogni algoritmo di garbage collection deve fare essenzialmente tre cose:

- determinare l'oggetto che deve essere eliminato, fase detta anche *garbage detection*;
- liberare la corrispondente zona di memoria e renderla disponibile per altri oggetti;
- mantenere la frammentazione dello heap al minimo possibile.

D.5.1. Garbage detection

L'attività di garbage detection deve determinare quali oggetti sono referenziati, per questo motivo si definiscono le *root*, o radici, che sono gli oggetti persistenti dell'applicazione, in altre parole, quelli sempre presenti durante l'intera esecuzione del programma. Nel caso di un'applicazione Java, è senz'altro una root l'oggetto a cui appartiene il metodo *main()*. Quali altri oggetti saranno ritenuti root dipende dall'implementazione dell'algoritmo di garbage collection, solitamente si considerano root tutti gli oggetti che abbiano uno *scope* coincidente con il corpo del main.
Agli altri oggetti viene assegnata una proprietà detta *reachability* (raggiungibilità) dalla root. Si dice che un oggetto è raggiungibile se esiste un percorso di riferimenti che, partendo dalla root, permetta di indirizzalo. E' evidente che gli oggetti raggiungibili sono referenziabili e quindi vivi, mentre gli altri possono essere rimossi. Oggetti raggiungibili da oggetti vivi sono anch'essi vivi.

Gli algoritmi più comunemente utilizzati per la determinazione della raggiungibilità sono due: *reference counting* e *tracing*.

Reference counting: ad ogni nuovo oggetto creato, viene associato un contatore inizializzato a 1.In ogni istante il contatore tiene conto del numero dei riferimenti all'oggetto a cui è collegato, tutte le volte che viene creato un nuovo riferimento ad un oggetto, il relativo contatore viene incrementato, mentre se al riferimento viene associato un nuovo valore o l'oggetto esce dal suo scope, il contatore viene ridotto. Il garbage collector dovrà quindi semplicemente verificare se il contatore relativo a un oggetto è nullo, in questo caso l'oggetto non è referenziato, e quindi si può eliminare.

Tracing: questo algoritmo consiste nel prendere come riferimento gli oggetti root. Partendo da essi si marcano tutti gli oggetti collegati tra loro da un riferimento. Dopo questa operazione, gli oggetti marcati saranno quelli raggiungibili, gli altri posso essere rimossi.

D.5.2. Tecniche di deframmentazione

Un importante compito del garbage collector è quello di mantenere entro limiti accettabili la frammentazione dello heap, conseguenza del processo di continuo alternarsi di operazioni di allocazione e deallocazione di memoria durante l'esecuzione di un programma. La frammentazione potrebbe portare a gravi conseguenze se non tenuta sotto controllo, ad esempio un oggetto potrebbe non essere allocato perché non è disponibile un gruppo locazioni contigue in grado di contenerlo.

Le tecniche di deframmentazione più comuni sono il *compacting* e il *copying*. Entrambe eseguono opportuni spostamenti degli oggetti.

Compacting: questo metodo sposta tutti gli oggetti validi verso un estremo dello heap in modo che l'altro lato venga ad essere composto dalla memoria libera per

l'allocazione di nuovi oggetti. La conseguenza principale è che tutti i riferimenti agli oggetti rilocati devono essere aggiornati, ciò può essere un problema per l'efficienza. Viene perciò aggiunto un livello di indirizzamento indiretto per semplificare il procedimento.

Copying: tutti gli oggetti validi vengono copiati in una nuova area di memoria, dove sono inseriti uno di seguito all'altro. La vecchia area di memoria viene così lasciata libera. Un vantaggio di questo sistema sta nella perfetta compatibilità con il metodo di *tracing*, infatti, quando un oggetto viene marcato, lo si può immediatamente copiare nella nuova area. Gli oggetti che rimarranno indietro saranno quelli non marcati, che verranno automaticamente eliminati.

D.6. Gestione delle eccezioni

Le eccezioni sono uno strumento avanzato per la gestione degli errori, permettono di intercettarli e di specificare il modo per gestirli.

Un metodo che debba intercettare le eccezioni generate dai metodi che invoca, deve inserire le chiamate all'interno di un blocco *try*, quando una eccezione viene generata viene associato ad essa il modo per gestirla attraverso un blocco *catch*. Più blocchi *catch* possono essere associati a un blocco *try*, permettendo così la gestione di diversi tipi di eccezioni.

Ogni volta che un metodo contenuto in un blocco *try* genera una qualsiasi eccezione, l'esecuzione viene interrotta e il controllo passa al relativo blocco *catch*. Se questo è in grado di gestire l'eccezione prosegue l'esecuzione, altrimenti passa l'eccezione a chi ha invocato il metodo. Questo processo continua fino a che un blocco catch in grado di gestirla intercetta l'eccezione, o finché questa arriva al metodo *main()* senza essere stata intercettata. In questo caso l'applicazione viene terminata.

Una importante caratteristica delle eccezioni consiste nel fatto che è consentito crearne di proprie. Queste eccezioni si gestiranno come quelle normali.

D.6.1. Blocchi catch multipli

Può accadere che un metodo debba intercettare diversi tipi di eccezioni. In Java è possibile utilizzare più blocchi *catch*, uno per ogni tipo di eccezione che si vuole gestire.

Ogni volta che una eccezione è sollevata in un blocco try, essa viene intercetta e gestita dal primo *catch* del tipo appropriato. Per ogni serie di blocchi *catch* ne viene eseguito sempre solo uno. Da notare che i blocchi *catch* assomigliano alle dichiarazione dei metodi.

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

D.6.2. La clausola finally

Java introduce un nuovo concetto per la gestione delle eccezioni: la clausola *finally*. Essa contrassegna un blocco di codice che verrà comunque eseguito, che venga o meno generata una eccezione.

Il blocco *finally* viene eseguito subito dopo il blocco *try*, oppure, nel caso di eccezioni, dopo che il *catch* ha avuto modo di gestirle.

D.7. Integrazione tra Java e Oracle 9i - JDBC

JDBC[28] è la tecnologia Java per l'accesso ai database relazionali, si tratta di un API, Application Program Interface, quindi un insieme di classi e interfacce Java finalizzate a interagire con i server di database locali e remoti.

La funzione principale di JDBC è quella di consentire allo sviluppatore Java di accedere a ogni tipo di informazione rappresentata in forma tabellare, permetendo di inviare richieste SQL a un server di database e di ricevere e gestire i dati ottenuti. Per arrivare a questo risultato sono necessari almeno tre passi fondamentali:

- Stabilire una connessione con la fonte dei dati.
- Preparare e inviare la query.
- Gestire il risultato.

Nei paragrafi successivi analizzeremo brevemente i principali oggetti usati nel prototipo.

D.7.1. Driver e DriverManager

L'interfaccia Driver rappresenta il punto di partenza per ottenere una connessione con un DBMS. I produttori di driver JDBC implementano l'interfaccia Driver affinchè possa funzionare con un tipo particolare di DBMS.

La classe DriverManager facilita la gestone degli oggetti di tipo Driver e consente la connessione con il database sottostante. Nel momento in cui un Driver è istanziato, sarà automaticamente registrato nella classe DriverManager. Ogni applicazione può registrare uno o più driver JDBC diversi tra loro. Il metodo statico `getConnection()` della classe DriverManager può stabilire la connessione con il database usando il river opportuno tra quelli registrati.

D.7.2. Connection

Un oggetto di tipo Connection rappresenta una connessione attiva con il database. Il metodo `getConnection()` descitto precedentemente, se non fallisce, restituisce un oggetto Connection. L'interfaccia mete a disposizione svariati metodi, tra cui quelli necessari a inviare query SQL alla base di dati e ottenere i risultati.

Nel nostro caso, la connessione a Oracle avviene grazie alle seguenti istruzioni:

```
Oracle.connect(getClass(), "connect.properties");
Connection conn =
sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection()
;
```

D.7.3. Statement SQL

Per l'invio di query SQL al database sono disponibili diversi metodi, che vengono usati in base al tipo di richiesta che deve essere fatta al database.

Gli oggetti Statement si utilizzano per query "semplici", cioè senza parametri, come ad esempio la creazone di una tabella.

Per le query che invece richiedano dei parametri è previsto l'oggetto PreparedStatement che consente la specificazione di parametri di input. L'interfaccia prevede, infatti, dei metodi che consentono la sostituzione dei paramtri con i valori effettivi con cui si intende realizzare la query.

Quando è necessario gestire anche parametri di output, l'oggetto da usare è CallableStatement, attraverso esse è possibile eseguire anche le Stored Procedure memorizzate nel server di database.

D.7.4. ResultSet

L'oggetto ResultSet è il risultato di una query di selezione e per questo motivo rappresenta di fatto una tabella composta da righe (gli elementi selezionati) e colonne (gli attributi richiesti).

E' possibile accedere agli oggetti ResultSet attraverso diversi metodi che consentono un accesso casuale alle righe, oltre alla possibilità di cancellarle e modificarle senza bisogno di estrarle dal ResultSet.

Il seguente codice mostra un breve esempio di come si utilizzano gli oggetti Statement e ResultSet.

```
Statement st = conn.createStatement ();
 String query = "SELECT t.leggexml.extract('/legge').getClobVal() "
                + "FROM " + Table + " t where t.cod = "
                + Cod;
 ResultSet rs = st.executeQuery (query);
```

```
while (rs.next())
 {
   try {
     Reader lettore = rs.getClob(1).getCharacterStream();
     legge = new SAXBuilder().build(lettore);
   } catch (Exception e) {e.printStackTrace();}
 }
st.close();
```

D.8. JDOM

Negli ultimi tempi, la necessità dei programmatori Java di avere a disposizione uno strumento più potente del SAX e più pratico del DOM, si è fatta sempre più pressante in seguito al grande successo di XML. JDOM[29] è una libreria "open source" per la manipolazione di dati XML in Java e ha contribuito ad abbattere molte barriere tra Java e XML aumentandone la interoperabilità.

L'obiettivo di JDOM è di nascondere il più possibile al programmatore Java la complessità di XML, integrandosi completamente con il modo di lavorare di Java, è infatti stato creato in Java e per Java e utilizza, perciò, molte delle sue convenzioni; ad esempio, tutte le classi di JDOM possiedono metodi come `equals()` o `toString()`.

Il modello JDOM, pur essendo simile al DOM di XML, non è costruito su di esso o modellato su di esso. Come DOM rappresenta un documento XML come un albero composto da elementi, attributi, commenti e così via. E' possibile accedere in ogni momento ad ogni parte dell'albero.

JDOM non include un suo parser, dipende dal parser SAX per creare il modello JDOM da documenti XML; è inoltre possibile convertire automaticamente documenti DOM in oggetti JDOM. Quando però i dati XML arrivano da uno strema da disco o da rete, è preferibile usare il parser SAX, evitando così di costruire l'albero del documento in memoria due volte per le due diverse rappresentazioni.

Una delle più importanti caratteristiche di JDOM, comune a DOM, è la possibilità di costruire in memoria un nuovo albero XML partendo da dati non XML, come ad esempio dati relazionali o risultati di elaborazioni.
Una volta che un documento XML è stato caricato in memoria, lo si può modificare in ogni modo compatibile con lo standard XML, si può, ad esempio, aggiungere un attributo a un elemento, ma non a un commento.
Terminata la manipolazione, JDOM offre numerose opzioni di serializzazione per converire l'albero in memoria in un formato adatto ad essere scritto su disco o spedito via rete. E' anche possibile generare un output in formato SAX o DOM.
Vediamo ora quali sono le classi più importanti offerte da JDOM.
Document: la classe Document rappresenta un intero documento XML ben formato e obbliga a rispettare tutte le regole a cui un documento XML è sottoposto.
Element: la struttura di un documento XML è costruita sui suoi elementi, non stupisce quindi che questa classe sia una dell più estese ed importanti di JDOM. La Classe Element è il modo fondamentale attraverso il quale è possibile navigare un oggetto JDOM.
Attibute: la classe Attribute rappresenta un attributo di un elemento.
Text: JDOM usa questa classe internamente per rappresentare i nodi di testo. Nell'uso normale il programmatore non interagisce direttamente con questa classe, usa direttamente oggetti stringa.
Per i metodi di queste ed altre classi di JDOM si rimanda ai riferimenti bibliografici.

Appendice E

LISTATI DELLE QUERY

Tabella XML

Query 1

```
SELECT L.cod ,L.leggexml.extract('/legge/titolo/text()').getStringVal() FROM
Legge_I1000 L where (((CONTAINS (L.leggeXML, 'soccorso WITHIN Comma') > 0)
AND (CONTAINS (L.leggeXML, 'volontariato WITHIN Comma') > 0) AND (CONTAINS
(L.leggeXML, 'yyyyyyyy WITHIN Comma') > 0) ))
```

Query 2

```
SELECT L.cod ,L.leggexml.extract('/legge/titolo/text()').getStringVal() FROM
Legge1000 L where (((CONTAINS (L.leggeXML, 'Veneto WITHIN Comma') > 0) OR
(CONTAINS (L.leggeXML, 'Sicilia WITHIN Comma') > 0) AND (CONTAINS
(L.leggeXML, 'Delibera WITHIN Natura') > 0) )) ORDER BY L.cod
```

Query 3

```
SELECT L.cod ,L.leggexml.extract('/legge/titolo/text()').getStringVal() FROM
Legge1000 L where
((L.leggexml.extract('/legge/articolato/@tt_start').getStringVal() <= '2003-
05-14' AND (L.leggexml.extract('/legge/articolato/@tt_end').getStringVal()
IS NULL OR L.leggexml.extract('/legge/articolato/@tt_end').getStringVal() >
'2003-05-14')) AND
((L.leggexml.extract('/legge/articolato/@vt_end').getStringVal() IS NULL AND
L.leggexml.extract('/legge/articolato/@vt_start').getStringVal() <= '2003-
05-01') OR (L.leggexml.extract('/legge/articolato/@vt_start').getStringVal()
<= '2003-05-01' AND
L.leggexml.extract('/legge/articolato/@vt_end').getStringVal() >= '1973-05-
14')) AND
(L.leggexml.extract('/legge/articolato/@pubblicazione').getStringVal() >=
'1973-05-14') ) ORDER BY L.cod
```

Query 4

```
SELECT L.cod ,L.leggexml.extract('/legge/titolo/text()').getStringVal() FROM
Legge1000 L where
((L.leggexml.extract('/legge/articolato/@tt_start').getStringVal() <= '2003-
05-14' AND (L.leggexml.extract('/legge/articolato/@tt_end').getStringVal()
IS NULL OR L.leggexml.extract('/legge/articolato/@tt_end').getStringVal() >
'2003-05-14')) AND
(L.leggexml.extract('/legge/articolato/@vt_start').getStringVal() <= '2003-
05-14' AND (L.leggexml.extract('/legge/articolato/@vt_end').getStringVal()
IS NULL OR L.leggexml.extract('/legge/articolato/@vt_end').getStringVal() >=
'2003-05-14')) AND ((CONTAINS (L.leggeXML, 'trasporti WITHIN Comma') > 0)
AND (CONTAINS (L.leggeXML, 'ceramica WITHIN Comma') > 0) )) ORDER BY L.cod
```

Query 5

```
SELECT L.cod ,L.leggexml.extract('/legge/titolo/text()').getStringVal() FROM
Legge1000 L where
((L.leggexml.extract('/legge/articolato/@tt_start').getStringVal() <= '2003-
05-14' AND (L.leggexml.extract('/legge/articolato/@tt_end').getStringVal()
IS NULL OR L.leggexml.extract('/legge/articolato/@tt_end').getStringVal() >
'2003-05-14')) AND
(L.leggexml.extract('/legge/articolato/@pubblicazione').getStringVal() <=
'2003-05-14') AND ((CONTAINS (L.leggeXML, 'latte WITHIN Comma') > 0) AND
(CONTAINS (L.leggeXML, 'Decreto del Presidente della Repubblica WITHIN
Natura') > 0) )) ORDER BY L.cod
```

Query 6

```
SELECT L.cod ,L.leggexml.extract('/legge/titolo/text()').getStringVal() FROM
Legge1000 L where
((L.leggexml.extract('/legge/articolato/@et_start').getStringVal() <= '2003-
05-14' AND (L.leggexml.extract('/legge/articolato/@et_end').getStringVal()
IS NULL OR L.leggexml.extract('/legge/articolato/@et_end').getStringVal() >=
'2003-05-14')) AND ((CONTAINS (L.leggeXML, 'oro WITHIN Comma') > 0) OR
(CONTAINS (L.leggeXML, 'diamante WITHIN Titolo') > 0) AND (CONTAINS
(L.leggeXML, 'Direttiva WITHIN Natura') > 0) )) ORDER BY L.cod
```

Tabella ibrida

Query 1

```
SELECT L.cod ,L.leggexml.extract('/legge/titolo/text()').getStringVal() FROM
Legge1000 L where (((CONTAINS (L.leggeXML, 'soccorso WITHIN Comma') > 0) AND
(CONTAINS (L.leggeXML, 'volontariato WITHIN Comma') > 0) AND (CONTAINS
(L.leggeXML, 'yyyyyyyy WITHIN Comma') > 0) ))
```

Query 2

```
SELECT L.cod ,L.leggexml.extract('/legge/titolo/text()').getStringVal() FROM
Legge_I1000 L where (((CONTAINS (L.leggeXML, 'Veneto WITHIN Comma') > 0) OR
(CONTAINS (L.leggeXML, 'Sicilia WITHIN Comma') > 0) AND (natura =
'Delibera') )) ORDER BY L.cod
```

Query 3

```
SELECT L.cod ,L.leggexml.extract('/legge/titolo/text()').getStringVal() FROM
Legge_I2000 L where ((tt_start <= '15-MAY-2003' AND (tt_end is null OR
tt_end > '15-MAY-2003')) AND ((vt_end IS NULL AND vt_start <= '14-MAY-2003')
OR (vt_start <= '14-MAY-2003' AND vt_end >= '15-MAY-1973')) AND
(pubblicazione <= '15-MAY-1983') ) ORDER BY L.cod
```

Query 4

```
SELECT L.cod ,L.leggexml.extract('/legge/titolo/text()').getStringVal() FROM
Legge_I1000 L where ((tt_start <= '14-MAY-2003' AND (tt_end is null OR
tt_end > '14-MAY-2003')) AND (vt_start <= '14-MAY-2003' AND (vt_end IS NULL
OR vt_end >= '14-MAY-2003')) AND ((CONTAINS (L.leggeXML, 'trasporti WITHIN
Comma') > 0) AND (CONTAINS (L.leggeXML, 'ceramica WITHIN Comma') > 0) ))
ORDER BY L.cod
```

Query 5

```
SELECT L.cod ,L.leggexml.extract('/legge/titolo/text()').getStringVal() FROM
Legge_I1000 L where ((tt_start <= '14-MAY-2003' AND (tt_end is null OR
tt_end > '14-MAY-2003')) AND (pubblicazione <= '14-MAY-2003') AND ((CONTAINS
(L.leggeXML, 'latte WITHIN Comma') > 0) AND (natura = 'Decreto del
Presidente della Repubblica') )) ORDER BY L.cod
```

Query 6

```
SELECT L.cod ,L.leggexml.extract('/legge/titolo/text()').getStringVal() FROM
Legge_I1000 L where ((et_start <= '14-MAY-2003' AND (et_end IS NULL OR
et_end >= '14-MAY-2003')) AND ((CONTAINS (L.leggeXML, 'oro WITHIN Comma') >
0) OR (CONTAINS (L.leggeXML, 'diamante WITHIN Titolo') > 0) AND (natura =
'Direttiva') )) ORDER BY L.cod
```

RIFERIMENTI BIBLIOGRAFICI

[1] G. Sartor. *Riferimenti normativi e dinamica dei testi normativi*. Università di Bologna.

[2] F. Mandreoli. *Database Temporali*. Dipartimento di Scienze dell'Ingegneria. Università di Modena e Reggio Emilia.

[3] G. Ozsoyoglu and R. T. Snodgrass. *Temporal and Real Time Databases: A Survey*. IEEE TKDE vol.7 No.4, pp. 513-532 (1995).

[4] C. Combi, A. Montanari. *Data Models vith Multiple Temporal Dimensions: Completing the Picture*. Dipartimento di matematica e informatica. Università di Udine. CaiSE 2001, pp. 187-202.

[5] E. Damiani, P. Fezzi. *XML: Uno Standard In Espansione*. Mondo Digitale n.2, pp. 50-65.

[6] http://www.w3.org

[7] http://www.w3.org/TR/xml-infoset/

[8] http://www.w3.org/XML/Schema

[9] S.S. Chawathe, S. Abiteboul, J. Widom. *Managing historical semistructured data*. Theory and Practice of Object Systems, 5(3):143-162, 1999.

[10] T. Amagasa, M. Yoshikawa, S. Uemura. *A Data Model for Temporal XML* Documents. Lecture Notes in Computer Science 1873, pp. 334-344. Springer- Verlag 2000.

[11] F. Grandi, F. Mandreoli. *The valid web: it's time to go…* Technical Report TR-46, TimeCenter, www.cs.auc.dk/researc/DP/tdb/TimeCenter/, 1999.

[12] F. Grandi, F. Mandreoli. *The valid web: an XML/XSL infrastructure for temporal management of web document.* In Proc. of the Intl'Conf. On

Advances in Information Systems (ADVIS 2000), Izmir, Turkey, October 2000. LNCS 1909.

[13] C.E. Dyreson, M. H. Bohlen, C. S. Jensen. *Capturing and querying multiple aspects of semistructured data*. In Proc. of 25th Intl'Conf. On Very Large Data Bases (VLDB '99), Edinburgh, Scotland, September 1999.

[14] L.A. Kalinchenko, M.G. Manukyan. *Temporal XML*. In Proc. of 5th East European Conf. on Advances in Databases and Inforamtion Systems (ADBIS '01) – Vol.1, Research Communications, Vilnius, Lithuania, September 2001.

[15] J.Andersen, A. Giversen, A.H. Jensen, R.S. Larsen, T.B. Pedersen, J. Skyt. *Analizing clickstreams using subsessions*. In Proc. of 3rd ACM Intl'Workshop on Data Warehousing and OLAP (DOLAP 2000), Washington, DC, November 2000.

[16] E. Damiani, B. Oliboni, E. Quintarelli, L. Tanca. Modeling users' navigation history. In Proc. of Intl'Workshop of Intelligent Techniques for Web Personalisation (in conj. With IJCAI-01), Seattle, WA, August 2001.

[17] Y. Stavrakas, M.Gergatsoulis, P. Rondogiannis. Multidimensional XML. Lecture Notes in Computer Science 1830, pp. 100-109. Springer-Verlag 2000.

[18] T. Mitakos, M. Gergatsoulis, Y. Stavrakas, E. V. Ioannidis. *Representing Time-Dependent Informatio in Multidimensional XML*. ECDL 2000, pp. 368-371.

[19] http://www.Oracle.com

[20] http://www.Oracle.com/ip/deploy/database/Oracle9i/

[21] Oracle XML DB. An Oracle Technical White Paper. January 2003.

[22] Oracle Text Reference. Release 9.0.1. June 2002

[23] Scott Fowler. *UML Distilled*. Addison Wesley

[24] http://www.normeinrete.it

[25] Richard Thomas Snodgrass. *The TSQL2 temporal query language*. Kluwers Academic Publishers

[26] J.Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, N.J., 1992

[27] Bruce Eckel. *Thinking in Java*. Apogeo

[28] Giuseppe Naccarato – *Java database e programmazione client/server*. Apogeo.

[29] http://www.jdom.org/downloads/docs.html