Introduzione 4

Capitolo 1 - Linguaggi di interrogazione per documenti XML 10

1.1 Descrizione di un documento XML 10

1.2 Requisiti di un linguaggio di interrogazione per file XML 16

1.3 Linguaggi esistenti 17

1.3.1 YAT_L 18

1.3.2 Lorel 20

1.3.3 XML-QL 24

1.3.4 TQL 27

1.3.5 CXQuery 32

1.3.6 XQL 34

1.3.7 XQuery 36

1.4 La nostra proposta: XCDEv2.0 40

Capitolo 2 - XCDEv2.0: Query Language 47

2.1 Esempi di interrogazioni 47

2.1.1 Interrogazioni su parole 49

2.1.1.1 Esatta, per Prefisso, per Suffisso, per Sottostringa 49

2.1.1.2 Per espressioni regolari 50

2.1.1.3 Con errori 50

2.1.2 Interrogazioni su elementi 51

2.1.2.1 Interrogazioni sul contenuto di un elemento 51

2.1.2.2 Interrogazioni sul valore degli attributi associati ad un elemento 52

2.1.2.3 Interrogazioni sugli elementi e le loro profondità 54

2.1.3 Interrogazioni sui valori degli attributi senza specificare l'elemento 54

2.1.3.1 Esatta, per prefisso, per suffisso, per sottostringa 55

2.1.3.2 Con errori 55

2.1.4 Interrogazioni per proximity 56

2.1.5 Interrogazioni con operatori booleani 58

2.1.5.1 AND 58

2.1.5.2 OR 58

2.1.5.3 NOT 61

2.1.6 Altri esempi 62

2.1.6.1 Interrogazioni su elementi con restituzione di elementi differenti 62

2.1.6.2 Interrogazioni su parole ignorando gli elementi 63
2.1.6.3 Interrogazioni su entità 64
2.1.7 Formattazione del risultato **65**
2.1.7.1 Dimensione e direzione dello snippet visualizzato 65
2.1.7.2 Formato dello snippet visualizzato 67
2.1.8 Restituzione di un sottoinsieme dei risultati **68**
2.2 Struttura della Query 69
2.2.1 La grammatica **69**
2.2.2 Clausola SELECT **71**
2.2.2.1 Elementi speciali 71
2.2.2.2 Attributi speciali 73
2.2.3 Clausola FROM **74**
2.2.4 Clausola RETURN **75**
2.2.4.1 Elementi speciali 75
2.2.4.2 Attributi speciali 75
2.2.5 Clausola RANGE **76**
Capitolo 3 - XCDEv2.0: Query Engine 77
3.1 Strutture dati del Query Engine 77
3.1.1 Pila_parsing **77**
3.1.2 Query_tree_node **78**
3.1.3 Tabella hash **80**
3.1.4 Tabella dei risultati **81**
3.2 Parsing dell'interrogazione e costruzione dell'albero associato 81
3.3 Esecuzione dell'interrogazione 82
3.3.1 Visita bottom-up **84**
3.3.2 Visita top-down **86**
3.4 Costruzione dei risultati 86
3.4.1 Memorizzazione dei risultati nel file system **88**
3.4.2 Recupero dei risultati dal file system **89**
3.5 Visualizzazione dei risultati 90
Capitolo 4 - Le funzioni di XCDEv2.0 91
4.1 La 'console query' 91
4.1.1 Struttura della console query **91**
4.2 L'API 92

4.2.1 Funzioni ad alto livello **93**
4.2.2 Funzioni per l'esecuzione di un'interrogazione **93**
4.2.3 Funzioni per operare sui risultati di un'interrogazione **93**
Capitolo 5 - Conclusioni 95
Bibliografia 97
Indice delle figure 101
APPENDICE A – L'API di XCDEv2.0 102
APPENDICE B – I file della libreria XCDE 108
APPENDICE C – Documento XML 121

Introduzione

L'XML (*Extensible Markup Language)* è attualmente uno dei linguaggi più promettenti per la memorizzazione ed il trasferimento di informazioni in rete. Infatti esso possiede una grande flessibilità sintattica che permette di descrivere praticamente qualsiasi tipo di oggetto, sia esso una pagina *web* o una complessa base di dati. Inoltre un documento XML può essere visualizzato mediante un *browser*, allegandogli un foglio di stile o anche includendolo in una normale pagina *web*. Di contro il ben noto linguaggio HTML (*HyperText Markup Language)* è sicuramente il più diffuso per la creazione di pagine *web*, ma il suo utilizzo risulta limitato essenzialmente alla visualizzazione delle stesse.
Sin da quando è stata completata la specifica dell'XML da parte del *World Wide Web Consortium* (W3C) nel febbraio 1998 **[W3C XML Query]**, il suo utilizzo ed interesse nel mondo dell'industria e in quello accademico sono aumentati in maniera costante, fino a divenire uno standard *de facto* per la pubblicazione e lo scambio di dati semi-strutturati. L'XML fornisce quindi la soluzione ideale per gestire grandi quantità di informazioni sempre più complesse, come quelle disponibili su rete e non solo. Esso è un potente linguaggio descrittivo che consente di creare i propri elementi sulla base della natura stessa dell'informazione da descrivere, e di etichettare quest'ultima in accordo con essi. Questa proprietà facilita la comprensione dei documenti, il loro interscambio tra varie applicazioni, così come consente di cercare, ordinare, filtrare e manipolare gli elementi di un documento in maniera altamente flessibile e automatica.

Un'*interrogazione* realizzata su un documento XML può dunque utilizzare la sua struttura per risultare più efficiente ed efficace. Queste caratteristiche del linguaggio hanno spinto alcuni ricercatori a parlare recentemente di *"ricerche semantiche"* sui documenti XML, e più in generale di "*semantic web"* per sottolineare le potenzialità del linguaggio XML nell'interscambio ed elaborazione di dati sulla rete e tra diverse applicazioni **[Applications]**. Questa visione è, attualmente, abbastanza utopistica anche se un elevato numero di persone e organizzazioni stanno lavorando affinché diventi ben presto una realtà.

Diversi linguaggi per l'interrogazione di documenti XML sono stati proposti e realizzati in questi ultimi anni *(Qexo, Galax, XML-QL, XQL, YAT_L, QUILT, Lorel, TQL, CXQuery, XDuce, XML-GL ecc.)*. In base alle loro caratteristiche questi linguaggi possono essere suddivisi in due gruppi:

- I linguaggi orientati alle *basi di dati* effettuano ricerche su documenti XML memorizzati all'interno di basi di dati e possono sfruttare le tecniche di ottimizzazione tipiche di queste, ma sono vincolati dalla rigidità degli schemi.

- I linguaggi orientati all'*Information Retrieval* interrogano i documenti XML per effettuare ricerche *Full-Text*; questi linguaggi sfruttano la principale caratteristica di XML di rappresentare dati non strutturati.

Fra i linguaggi citati precedentemente: *XML_QL, XQL, YAT*$_L$ **[YAT$_L$]**, *TQL, Lorel* **[Lorel]**, appartengono alla classe dei linguaggi orientati alle basi di dati e verranno approfonditi nel capitolo 1, e anche se non sono saranno analizzati dettagliatamente nel seguito, di questa classe fanno ragionevolmente parte anche*, Qexo, Galax, XML-GL* **[XML-GL]**, *QUILT*, *XML-SQL* **[XML-SQL],** *Xduce* **[XDuce]**. Di contro, l'unico linguaggio orientato all'Information Retrieval è *XQL* **[XQL]**.
Una domanda interessante è perché non adattare SQL o OQL per interrogare documenti XML? La risposta è che i dati XML sono fondamentalmente diversi dai dati memorizzati nelle basi di dati relazionali e orientate agli oggetti, e quindi nè SQL **[SQL]** nè OQL **[OQL]** sono adatti per XML. La distinzione chiave tra i dati XML e i dati nei modelli tradizionali è che XML non è rigidamente strutturato. Nei modelli relazionali e orientati agli oggetti, ogni istanza di dato ha uno *schema*, che è indipendente dal dato stesso. Per esempio, nel modello relazionale, uno schema può definire la relazione `person` con i nomi di attributi `name` e `address`, e.g., `person(name, address)`. Un'istanza di questo schema dovrebbe contenere tuple come`("Smith", "Philadelphia")`. La relazione e i nomi di attributi sono separati dai dati e solitamente sono memorizzati in un catalogo separato dalla base di dati.
In XML, lo schema è memorizzato insieme al dato. I valori che descrivono la struttura sono chiamati *elementi*. I nomi di elementi, sono chiamati *tag*s, e gli elementi possono avere anche *attributi* i cui valori sono atomici.
Per esempio,

```
<person><name>Smith</name><address>Philadelphia</address></person>
```

è un'espressione XML *ben formata*. Quindi, il dato XML è *auto-descrittivo* e può naturalmente modellare irregolarità che non possono essere modellate dai dati relazionali o orientati agli oggetti.
Per esempio, un'espressione XML può avere elementi mancanti o occorrenze multiple dello stesso elemento; gli elementi possono avere valori atomici in alcune espressioni e valori strutturati in altre; ed elementi diversi appartenenti alla stessa collezione possono avere strutture eterogenee.
I dati relazionali sono spesso *densi,* e i sistemi relazionali rappresentano le informazioni mancanti attraverso uno speciale valore "NULL". Al contrario, i dati XML sono spesso *sparsi* e gli elementi mancanti si rappresentano semplicemente omettendoli.
Infine, i documenti XML spesso contengono molti livelli di elementi innestati mentre, i dati relazionali sono *piatti*.
Per queste ed altre ragioni, i linguaggi di interrogazione esistenti orientati alle basi di dati non sono adatti ad interrogare dati XML. Così, nell'ottobre del 1999, il World Wide Consortium ha riunito un gruppo di lavoro per progettare un linguaggio di interrogazione per sorgenti di dati XML. Il risultato di questo progetto è stato il linguaggio di interrogazione chiamato *XQuery*. Il gruppo di lavoro ha pubblicato una lista di requisiti, unitamente al modello dei dati che descrive il linguaggio, ad una descrizione formale della semantica, ad un insieme di funzioni e operatori e ad una collezione di esempi che illustrano le applicazioni del linguaggio. Ognuno di questi documenti mostra come il linguaggio ha subito e subisce tuttora continui aggiornamenti. Il linguaggio, orientato ad interrogazioni su basi di dati, è un linguaggio funzionale basato sul sistema dei tipi di *XML Schema* ed è stato implementato per essere compatibile con altri standard XML. Consiste

di diversi tipi di funzioni ed espressioni, tra cui *path expressions*, costruttori di elementi, chiamate di funzioni, espressioni aritmetiche, espressioni condizionali e quantificate. Il gruppo di lavoro recentemente ha discusso e proposto un'estensione per trattare ricerche Full-Text. Il motivo di questa scelta è dovuto al fatto che la parte testuale nei documenti XML può essere preponderante e quindi risulta necessario avere anche funzionalità di Information Retrieval. XQuery, ormai uno standard per ricerche di informazioni da documenti XML, ha contribuito a far affermare XML come il linguaggio per la rappresentazione universale di informazione.
Analizzando questi sistemi e la letteratura internazionale si può osservare come attualmente manchi una libreria *semplice, efficiente* e *di pubblico dominio* che consenta a un qualunque programmatore di sviluppare facilmente una applicazione che preveda, l'interrogazione, l'elaborazione o più in generale il *mining* di documenti XML. In questa tesi si affronta per la prima volta il problema di studiare, progettare e realizzare una libreria che risponda a questi requisiti, offrendo una collezione di algoritmi e strutture dati per un'efficiente *elaborazione* ed *interrogazione* di documenti XML. La libreria, chiamata XCDEv2.0 (*Xml Compressed Document Engine version 2.0*), è in grado di implementare quel sottoinsieme delle funzionalità di XQuery orientato alle *stringhe*, supportando anche ricerche con funzionalità avanzate di *Information Retrieval*.
Questa scelta è motivata dal nostro interesse di operare su documenti letterari codificati con XML-TEI in ambito umanistico (Dipartimento di Italianistica, Università di Pisa).
Per realizzare questo obiettivo siamo partiti dalla libreria XCDEv1.0 (*Xml Compressed Document Engine version 1.0*) **[XCDEv1.0]**, che offre un ottimo kernel per comprimere ed indicizzare documenti XML, sviluppata da Andrea Mastroianni in un precedente lavoro di tesi. La libreria è scritta in linguaggio C e supporta la memorizzazione dei documenti XML in modo *nativo*, cioè operando direttamente al livello del *file system.* Le sue principali caratteristiche sono:

- algoritmi e strutture dati che rappresentano *lo stato dell'arte* per quanto riguarda l'indicizzazione e l'elaborazione dei documenti;

- tecniche di compressione per la memorizzazione dei documenti in spazio ridotto;

- strutture dati *innovative* per la gestione della struttura gerarchica dei documenti XML, cioè dei suoi *tag* e *attributi*.

Rispetto ad altri sistemi, XCDEv1.0 consente di ottimizzare lo spazio di memorizzazione delle informazioni e garantisce al tempo stesso il supporto efficiente a complesse interrogazioni. Prevede diverse strutture dati e indici, come le liste di proximity, che la rendono particolarmente indicata ad operare su documenti con ampie parti testuali. Questi documenti possono avere una struttura con numerosi *tag* innestati su più livelli, e valori dei loro attributi formati da stringhe alfanumeriche arbitrariamente complesse. XCDEv1.0 contiene inoltre, un prototipo di motore di ricerca che permette di formulare interrogazioni che sono semplificazioni di quelle formulabili su XML mediante altri linguaggi, come ad esempio XQuery. Le interrogazioni rispondono alle richieste dell'utente umanista e sono essenzialmente su *paths*.
Viste le limitate potenzialità del motore di ricerca di XCDEv1.0, in questa tesi si propone un linguaggio completo per formulare interrogazioni, *XCDE Query Language*. Il linguaggio è XML-based così da essere facilmente accessibile ad utenti non esperti (utenti umanisti).
Infatti un esempio di interrogazione con *XCDE Query Language* può essere il seguente:

```
SELECT    <xml_error xml_maxerr = '1' xml_var = '$parola'>
                campana
          </xml_error>
FROM      esempio.xml
RETURN    $parola
```

Con questa interrogazione si cercano tutte le occorrenze della parola `campana`, con al più un errore, all'interno del documento `esempio.xml`. L'interrogazione viene formulata con una sintassi a clausole, **`SELECT-FROM-RETURN-RANGE`** dove **`RANGE`** è opzionale.
Le clausole **`SELECT`** e **`RETURN`** contengono espressioni XML ben-formate, in modo che la sintassi risulti semplice e intuitiva, consentendo altresì agli utenti di poter effettuare interrogazioni, anche complesse. La clausola **`FROM`** contiene i nomi dei file sui quali effettuare la ricerca.
Al risultato dell'interrogazione (*snippet*) si può imporre una formattazione attraverso la clausola **`RETURN`** e l'attributo speciale `xml_var` (chiamato *Pivot*). L'attributo deve essere inserito all'interno degli elementi contenuti nell'espresssione della clausola **`RETURN`** con lo scopo di identificarli come "punti interessanti" della ricerca. I *Pivot* devono poi essere inseriti nella clausola **`RETURN`** per specificare il formato secondo il quale questi "punti interessanti" devono essere visualizzati. Ovviamente è possibile specificare nell'espressione più di un *Pivot*, in questo modo più "punti interessanti" possono essere simultaneamente identificati senza dover usare complicate combinazioni di espressioni basate su cammini semplici. Nell'esempio è stato marcato come *Pivot* la parola `campana`, le cui occorrenze verranno quindi restituite nel risultato. Un'altra importante funzionalità del linguaggio è il processo di estrazione dello *snippet* che permette all'utente di definire la dimensione della porzione di testo da estrarre, la presenza o no di tag in esse e la possibilità di restituire tutti gli elementi che racchiudono lo snippet. La clausola **`RETURN`** può essere seguita dalla clausola **`RANGE`** che permette di specificare l'intervallo di risultati da restituire.
Il linguaggio può inoltre essere utilizzato per effettuare ricerche *Full-Text* così come ricerche per *espressioni regolari*, ricerche sul contenuto testuale dei tag o sul valore dei loro attributi.
Più precisamente l'utente può eseguire l'interrogazione attraverso il comando:

```
xcde_search2 [-p] [-f  Filename] query_expression
```

in cui l'opzione "–f filename" indica al motore di caricare l'interrogazione dal file 'filename'. E "-p" indica al motore che devono essere restituite le posizioni nel documento compresso delle occorrenze che soddisfano l'interrogazione e non gli snippet.
La `query_expression` corrisponde ad un'espressione XML *ben formata* composta da elementi e attributi che appartengono al documento interrogato più alcuni *elementi speciali* che permettono di specificare le modalità di ricerca e di estrazione degli snippet.
Esistono *elementi speciali* per specificare le modalità di ricerca. Questi si classificano in:

- *elementi speciali* semplici: specificano il tipo di ricerca da effettuare sulla parola contenuta al loro interno come, `<xml_exact>`, `<xml_prefix>`, `<xml_suffix>`, `<xml_contained>`, `<xml_regexp>` e `<xml_error>`. Tutti questi elementi possono contenere anche l'attributo speciale `xml_case` che indica se la ricerca deve essere *case sensitive*.

- *elementi speciali* complessi: utili a definire una ricerca complessa. Si possono effettuare ricerche in OR su almeno due operandi con `<xml_or>`, ricerche su un insieme di parole ad una data distanza specificata dall'attributo `xml_maxprox` con

`<xml_proximity>`, ricerche di qualsiasi elemento che contiene gli attributi indicati in questo elemento speciale con `<xml_anyvalue>` e ricerche con il NOT su un operando con `<xml_not>`.

- *elementi speciali* sul valore degli attributi: per specificare il tipo di ricerca sul valore degli attributi degli elementi come `exi prefi sufi conti` per ricerche *case insensitive* e `exs prefs sufs conts err` rispettivamente per ricerche *case sensitive*.

- *L'elemento speciale* `<xml_context>`: per gestire la formattazione degli snippet nel risultato. All'interno si possono specificare uno o più Pivot e la formattazione viene applicata ad ognuno in modo indipendente. Permette di definire la dimensione della porzione di testo da estrarre oltre allo snippet tramite l'attributo `xml_size`, di specificare se la parzione deve essere estratta prima e/o dopo lo snippet associato al Pivot con `xml_view` e di scegliere se visualizzare o meno i tag con `xml_format`.

Infine la libreria fornisce un API con un insieme di funzioni che operano sulla collezione di strutture dati ed algoritmi. La libreria XCDEv2.0 è scritta in linguaggio C, e per il sorgente, la documentazione e alcuni esempi si suggerisce di consultare la *home page* http://butirro.di.unipi.it/~ferrax/xcde/xcdelib.html. In questa pagina è possibile anche scaricare per scopi accademici o comunque non commerciali il sorgente della libreria (spazio occupato circa 400 kb).

La tesi risulta composta dai seguenti capitoli:

Nel **capitolo 1** viene fornita una panoramica sui linguaggi esistenti di interrogazione di documenti XML. Viene fornita inoltre una descrizione generale della libreria XCDEv2.0, che sarà oggetto di studio e sperimentazione più approfondita nei capitoli seguenti.

Nella prima parte del **capitolo 2** si mettono in risalto le funzionalità principali del linguaggio proposto attraverso una carrellata di esempi. Nella seconda parte, viene descritta la struttura dell'interrogazione illustrando le varie parti che la compongono.

Nel **capitolo 3** è descritto l'*XCDEv2.0 Query Engine*, il motore di ricerca per i documenti compressi. Nel corso del capitolo sarà data giustificazione di alcune scelte sede di progettazione e implementazione.

Nel **capitolo 4** sono descritte in maniera approfondita le funzioni offerte dalla libreria XCDEv2.0, la loro implementazione e le API in linguaggio C per il loro utilizzo.

Il **capitolo 5** contiene alcune considerazioni conclusive e indica diversi possibili spunti di studio e sperimentazione che meritano sicuramente un maggiore approfondimento futuro.

L'**appendice A** contiene una descrizione dettagliata delle singole funzioni che costituiscono l'API della libreria XCDEv2.0 con la loro dichiarazione in linguaggio C.

L'**appendice B** contiene una descrizione della procedura di installazione della libreria XCDEv2.0 e dei *file* che la compongono.

L’**appendice C** contiene il file di esempio “esempio.xml” utilizzato negli esempi di interrogazione del capitolo 2 ed in tutti i punti in cui sarà necessario avere un file di riferimento.

Capitolo 1

Linguaggi di interrogazione per documenti XML

Grandi quantità di informazioni vengono oggi elaborate, memorizzate e scambiate in XML, per cui l'abilità di interrogare in modo efficace ed efficiente sorgenti di dati XML è diventata oggi sempre più importante. Il modello di dati di XML è semi-strutturato e quindi differente dal modello di dati delle basi di dati tradizionali, cosicché i linguaggi per basi di dati non sono pienamente soddisfacenti per interrogare efficientemente documenti XML. Poiché XML può rappresentare informazioni provenienti da diverse sorgenti di dati, strutturate in diversi modi, un linguaggio di interrogazione per documenti XML deve riuscire a gestire questa varietà di informazioni dando la possibilità di formulare opportune interrogazioni.

1.1 Descrizione di un documento XML

La figura 1.1 riporta un esempio di documento XML che verrà utilizzato nella presente sezione per descrivere le strutture, le proprietà e le caratteristiche del linguaggio XML e il progetto dei suoi documenti. Si tratta di una collezione di libri: l'elemento radice è `COLLEZIONE_LIBRI`; gli elementi annidati sono `TITOLO`, `AUTORE`, `PAGINE`, e `COPERTINA`. Alcuni di essi possono essere omessi. Gli elementi (detti anche tag) indicano la struttura logica del documento e contengono le informazioni vere e proprie. Ad un documento deve essere associato sempre un *elemento radice*, che contiene tutti gli altri. Un elemento che include altri elementi è detto *elemento padre*, mentre un elemento contenuto in un elemento padre è chiamato *elemento figlio*. Ad ogni elemento possono essere associati un contenuto e delle informazioni aggiuntive, cioè degli *attributi*. Entrambi sono, in generale, opzionali. Un tipico elemento consiste di un'etichetta iniziale, un contenuto ed un'etichetta finale:

```
<TITOLO>Per chi suona la campana</TITOLO>
```

```
<?xml version="1.0"?>
<!-- esempio di una collezione di libri -->

<!DOCTYPE COLLEZIONE_LIBRI SYSTEM "iso_latin.dtd"
[ <!ELEMENT COLLEZIONE_LIBRI (LIBRO)+ >
  <!ELEMENT LIBRO (TITOLO, AUTORE, (PAGINE)?,(COPERTINA)?)>
  <!ELEMENT TITOLO (#PCDATA)>
  <!ELEMENT AUTORE (#PCDATA)>
  <!ELEMENT PAGINE (#PCDATA)>
  <!ELEMENT COPERTINA EMPTY>
  <!ATTLIST AUTORE data_nascita CDATA #IMPLIED>
  <!ATTLIST TITOLO venditori CDATA #IMPLIED>
  <!NOTATION JPG SYSTEM "Joint Picture Group">
  <!ATTLIST COPERTINA image CDATA #REQUIRED
                      image_type NOTATION (JPG) #REQUIRED
                      didascalia CDATA #IMPLIED>
 <!ENTITY ernie "Ernest Hemingway">
] >
<COLLEZIONE_LIBRI>
 <LIBRO>
   <TITOLO>Per chi suona la campana</TITOLO>
   <AUTORE data_nascita="1899">&ernie;</AUTORE>
   <PAGINE>220</PAGINE>
   <COPERTINA image="campana.jpg"
              image_type="JPG" didascalia="foto di &ernie;" />
 </LIBRO>
 <LIBRO>
   <TITOLO venditori="Barnes&amp;Noble, Bol">Senilit&agrave;</TITOLO>
   <AUTORE data_nascita='1861'>Italo Svevo</AUTORE>
 </LIBRO>
 <LIBRO>
   <TITOLO>G&#246;del, Escher, Bach: un'eterna ghirlanda brillante</TITOLO>
   <AUTORE data_nascita='1945'>Douglas F. Hofstadter</AUTORE>
 </LIBRO>
 <LIBRO>
   <TITOLO venditori="bol">The <![CDATA[J&J]]> italian basketball register 2001 </TITOLO>
   <AUTORE>Giampiero Hruby</AUTORE>
 </LIBRO>
</COLLEZIONE_LIBRI>
```

Figura 1.1 - Esempio di documento XML

Il contenuto può essere formato da una sequenza di caratteri, da elementi innestati o da una combinazioni di questi. In XML, al contrario dell'HTML, si richiede sempre che si includano l'etichetta iniziale e quella finale di un elemento; la sola eccezione è per gli elementi senza contenuto (o *vuoti*), per i quali si può usare una sola etichetta con il simbolo di chiusura:

```
<COPERTINA ... />
```

Esistono anche elementi a cui sono associati, oltre al contenuto, delle informazioni aggiuntive chiamate *attributi*:

```
<AUTORE data_nascita='1861'>Italo Svevo</AUTORE>
```

Il documento dell'esempio di figura 1.1, come tutti i documenti XML, si divide in due parti: *prologo* e *corpo* (chiamato anche *elemento radice*). Il prologo è formato dalla parte del documento che si trova prima dell'elemento radice. Esso ha funzionalità descrittive e molte sue componenti sono opzionali. La prima linea del documento è:

```
<?xml version="1.0" ?>
```

si tratta della *dichiarazione XML*, la quale specifica che si tratta di un documento XML e

fornisce il numero di versione. Essa è opzionale, anche se dovrebbe essere sempre inclusa. Quando è presente deve apparire all'inizio del documento. La versione del documento può essere delimitata da apici singoli o doppi. In generale tutte le stringhe all'interno dei documenti XML possono essere delimitate in questi due modi. Quindi la dichiarazione potrebbe anche essere scritta nel seguente modo:

```
<?xml version='1.0' ?>
```

La dichiarazione XML può anche includere la dichiarazione di documento *standalone*, nel modo seguente:

```
<?xml version="1.0" standalone="yes"?>
```

Questa dichiarazione può essere utilizzata in alcuni documenti per semplificarne l'elaborazione. In pratica attraverso di essa si certifica che il documento non contiene riferimenti ad altri documenti.
La seconda linea del prologo contiene un commento. Inserire dei commenti in un documento XML è opzionale, ma ne migliora chiaramente la leggibilità. Un commento inizia con i caratteri **<!--** e termina con i caratteri **-->**. Esso può contenere qualsiasi sequenza, escluso la doppia linea **--**, che ovviamente svolge la funzione di terminatore. Il contenuto del commento è ignorato dal *processore* XML ed ha come unico scopo quello di costituire un'annotazione per i lettori del documento sorgente.
La terza linea del sorgente di figura 1.1 è una linea vuota. Per migliorare la leggibilità del documento si possono aggiungere delle linee vuote, che sono ignorate dal *processore* XML.
Il prologo può contenere anche i seguenti elementi:

- una *Document Type Declaration (DTD)*: esso definisce il tipo e la struttura del documento. Se presente (come in figura 1.1), la DTD deve essere dopo la dichiarazione XML.

- una o più direttive per il processore *(processing instructions)*: queste forniscono informazioni che il *processore* XML passerà ad eventuali applicazioni.

La DTD della figura 1.1 è data dall'unione fra una *DTD interna* al documento ed una *DTD esterna,* contenuta in un file differente. Le due DTD non sono mutuamente esclusive. Se la DTD è esterna la dichiarazione sarà:

```
<!DOCTYPE COLLEZIONE_LIBRI SYSTEM "nomefile">
```

mentre se è interna sarà:

```
<!DOCTYPE COLLEZIONE_LIBRI [ ... ]>
```

dove `COLLEZIONE_LIBRI` è l'elemento radice. Nell'esempio la dichiarazione contiene sia la parola chiave `SYSTEM` che le dichiarazioni fra parentesi quadre. L'ordine è importante e cambiarlo provocherebbe un errore sintattico. La presenza della DTD consente di avere due livelli di controllo sintattico: i documenti possono essere *ben formati*, se rispettano le regole sintattiche base di XML, e *validi*, se sono ben formati e rispettano la struttura definita nella DTD. Tutti i documenti XML devono essere ben formati, mentre possono essere validi solo i documenti che includono la DTD. La DTD può essere complicata e contenere moltissime componenti, le principali sono le definizioni *degli elementi*, degli

attributi e delle *entità* (le entità rappresentano elementi predefiniti nel linguaggio o definiti nella DTD che possono essere usati all'interno dei documenti).
Analizziamo ora la DTD della figura 1.1. La prima definizione contenuta è:

```
<!ELEMENT COLLEZIONE_LIBRI (LIBRO)+ >
```

si tratta della definizione dell'elemento radice. Attraverso di essa si definiscono il *nome* ed il *tipo* dell'elemento. Un elemento può avere un nome qualsiasi, scelto fra quelli che rispettano le seguenti regole sintattiche:

- il nome deve iniziare con una lettera o con un underscore (_), seguita da zero o più lettere, cifre, punti (.), linee (-) o underscore;
- i nomi degli elementi che iniziano con il prefisso "xml" (in ogni combinazione di lettere maiuscole o minuscole) sono "riservati per lo standard";
- i due punti (:) sono riservati per la gestione di un meccanismo del linguaggio chiamato spazio dei nomi;

Figura 1.2 - regole sintattiche per i nomi di elementi XML

Il tipo dichiarato è una sequenza non vuota di elementi `LIBRO`. Per le dichiarazioni si utilizza un meccanismo basato sulle espressioni regolari. Ad esempio la dichiarazione:

```
<!ELEMENT LIBRO (TITOLO, AUTORE, (PAGINE)?,(COPERTINA)?)>
```

Definisce `LIBRO` come una sequenza di `TITOLO`, `AUTORE`, `PAGINE` e `COPERTINA`. Questi ultimi due elementi possono anche non essere presenti. Gli operatori usati (come +,? e la virgola) si riferiscono all'elemento che li precedente; le parentesi possono essere usate per raggruppare gli elementi. Altri operatori sono il simbolo asterisco *, che indica una sequenza di lunghezza arbitraria, e il simbolo | indicante una scelta esclusiva fra gli elementi. Un'altro esempio di dichiarazione è il seguente:

```
<!ELEMENT TITOLO (#PCDATA)>
```

Gli elementi `TITOLO` contengono una sequenza legale di caratteri (lo stesso tipo di `AUTORE` e `PAGINE`). Una sequenza legale è formata da un qualsiasi carattere ad esclusione delle parentesi angolata aperta < e dell'*ampersand* & quando non rappresentano degli elementi del linguaggio. Ad esempio l'elemento:

```
<TITOLO>Per chi suona la campana</TITOLO>
```

contiene una sequenza legale di caratteri, così come l'elemento:

```
<AUTORE data_nascita="1899">&ernie;</AUTORE>
```

In quest'ultimo caso la sequenza è legale perché il carattere & rappresenta l'inizio di un elemento del linguaggio chiamato *riferimento ad entità*. Si tratta di un meccanismo attraverso il quale si inseriscono nel documento delle componenti che sono state precedentemente definite dall'utente o che sono predefinite nel linguaggio. Il *processore* XML si occupa di sostituire i riferimenti ad entità con la componente corrispondente, si tratta quindi di meccanismi di abbreviazione. Quindi essendo presente nella DTD la dichiarazione di entità:

```
<!ENTITY ernie "Ernest Hemingway">
```

l'elemento AUTORE precedente deve essere logicamente interpretato come

```
<AUTORE data_nascita="1899">Ernest Hemingway</AUTORE>
```

Esistono anche altri meccanismi che sono usati per l'inserimento di caratteri all'interno dei documenti. Ad esempio i *riferimenti a caratteri*, come in:

```
<TITOLO>G&#246;del, Escher, Bach: un'eterna ghirlanda brillante /TITOLO>
```

Questi riferimenti rappresentano un modo per inserire qualsiasi carattere all'interno di un documento, sia i caratteri che non sono presenti nella tastiera (come il carattere ä), che quelli per cui sarebbe illegale, secondo la specifica del linguaggio, l'inserimento in quel particolare punto del documento (per esempio < e & all'interno di un elemento).
Esistono due modi per inserire un riferimento ai caratteri. La prima forma è:

&#d;

dove d è formato da una o più cifre decimali (da 0 a 9) che rappresentano un codice nell'insieme dei caratteri ISO/IEC 10646. La seconda forma è:

&#xh;

dove h rappresenta una o più cifre esadecimali (da 0 a f) che esprimono lo stesso riferimento al codice ISO/IEC.
Per esempio sia A che A rappresentano lettera maiuscola A (il cui codice è 65 in decimale e 41 in esadecimale).
L'elemento TITOLO dell'esempio precedente corrisponde logicamente a:

```
<TITOLO>Gödel, Escher, Bach: un'eterna ghirlanda brillante</TITOLO>
```

Il codice **ISO/IEC 10646** è un codice internazionale per la rappresentazioni di caratteri appartenenti in teoria a tutti i linguaggi scritti (ISO sta per *International Standard Organization* e IEC per *International Electrotechnical Commission*). I primi 128 caratteri del codice sono gli stessi del ben conosciuto codice ASCII.
Esistono in XML delle *entità predefinite*, cioè delle entità definite nel linguaggio a cui sono associati dei caratteri particolarmente importanti e frequentemente utilizzati. Ad esempio:

```
<TITOLO venditori="Barnes&amp;Noble, Bol">Senilit&agrave;</TITOLO>
```

che corrisponde a:

```
<TITOLO venditori="Barnes&Noble, Bol">Senilit&agrave;</TITOLO>
```

Le entità predefinite sono le seguenti:

Riferimento ad una entità generale predefinita	*Carattere inserito*	*Riferimento a carattere equivalente*
&amp;	&	&
<	<	<
>	>	>

Riferimento ad una entità generale predefinita	*Carattere inserito*	*Riferimento a carattere equivalente*
&apos;	'	&#39;
&quot ;	“	&#34;

Figura 1.3 - entità predefinite di XML

I riferimenti possono essere anche ad entità definite in una DTD esterna, dette *entità esterne*. Ad esempio l'elemento `TITOLO` contiene il riferimento ad una entità *agrave* che non è dichiarata all'interno della DTD contenuta nel documento. Essa è dichiarata nel file "iso_latin.dtd", contenente una DTD esterna. L'elemento precedente può essere quindi interpretato dal punto di vista logico come:

```
<TITOLO venditori="Barnes&Noble, Bol">Senilità</TITOLO>
```

Se si vuole inserire la sequenza di caratteri *&* all'interno del documento senza che questa sia interpretata come il carattere *ampersand* dovrei prima utilizzare una *sezione CDATA*. Una sezione CDATA è un blocco di testo nel quale possiamo inserire qualsiasi carattere liberamente, ad eccezione della sequenza]]>. Essa inizia con <![CDATA[e termina con]]>. Fra questi due delimitatori si può inserire qualsiasi sequenza, ad eccezione di]]> (che, naturalmente, verrebbe interpretato come il terminatore della sezione medesima). Ad esempio nell'elemento:

```
<TITOLO …>The <![CDATA[J&J]]> italian basketball register 2001 </TITOLO>
```

Viene inserito il carattere & senza che sia interpretato come l'inizio di un riferimento. La parola chiave CDATA, come tutte le parole riservate XML, deve essere scritta in caratteri maiuscoli. Una sezione CDATA deve essere inserita fra le etichette di apertura e di chiusura di un elemento in modo non ricorsivo, cioè non è possibile aprire una sezione CDATA all'interno di un'altra.
Ad un elemento possono essere associati uno o più attributi, cioè un insieme di coppie nome-valore. I nomi degli attributi devono rispettare le regole già esposte per gli elementi (cfr. figura 1.2*)*. I valori sono formati da una sequenza di caratteri delimitati con degli apici, in accordo con le regole seguenti:

- La sequenza non può contenere il carattere delimitatore;
- La sequenza non può contenere il carattere < (il parser lo riconoscerebbe come l'inizio di un elemento), mentre il carattere & può essere incluso solo nel caso in cui rappresenti un riferimento;

Figura 1.4 - regole sintattiche per i valori degli attributi XML

Gli attributi possono essere definiti all'interno della DTD così come avviene per gli elementi. L'istruzione:

```
<!ATTLIST AUTORE data_nascita CDATA #IMPLIED>
```

associa all'elemento `AUTORE` un attributo data_nascita avente come valore una sequenza legale di caratteri (specificata tramite la parola chiave CDATA). La parola chiave #IMPLIED indica che il valore dell'attributo può essere anche non specificato (in

alternativa #REQUIRED indica che l'attributo è richiesto). La dichiarazione dell'attributo può essere anche molto complicata. Ad esempio:

```
  <!ELEMENT COPERTINA EMPTY>
.....
  <!NOTATION JPG SYSTEM "Joint Picture Group">
  <!ATTLIST COPERTINA image CDATA #REQUIRED
                      image_type NOTATION (JPG) #REQUIRED
                      didascalia CDATA #IMPLIED>
```

Questa dichiarazione associa all'elemento `COPERTINA`, che è un elemento vuoto (EMPTY), tre attributi:

- *image*: che contiene una stringa, intuitivamente il nome del file contenente l'immagine. Questo attributo deve essere sempre presente;
- *image_type*: che contiene un attributo di tipo *notazione*, definito tramite l'istruzione NOTATION. Questo attributo può essere usato ad esempio per stabilire qual'è il formato dell'immagine associata. Anche questo attributo deve essere sempre presente;
- *didascalia*: che contiene una stringa, cioè la descrizione dell'immagine. Questo attributo può anche essere omesso.

Alla dichiarazione corrispondono degli elementi come il seguente:

```
<COPERTINA image="campana.jpg" image_type="JPG" didascalia="foto di
                              &ernie;" />
```

Il documento di figura 1.1 rappresenta quindi un documento *ben formato* e *valido*. Naturalmente per la descrizione di tutte le caratteristiche dell'XML rimandiamo al sito della W3C o ai vari libri sull'argomento (ad esempio **[Young]**). Utilizzeremo in seguito il documento di figura 1.1 come esempio per i diversi concetti introdotti.

1.2 Requisiti di un linguaggio di interrogazione per file XML

Per capire meglio quali devono essere i requisiti di un linguaggio di interrogazione per documenti XML accenniamo brevemente all'evoluzione di XML **[XML]**. Esso è una recente specifica del World Wide Web Consortium (W3C) di un meta-linguaggio di markup progettato per la memorizzazione e l'interscambio di informazioni sulla rete. Derivato da SGML **[SGML]**, XML permette ai produttori di documenti di definire e usare l'insieme di tag che meglio rispecchia la struttura e le proprietà concettuali delle informazioni che essi vogliono pubblicare.
L'uso di XML ha portato un grande cambiamento nella strutturazione delle informazioni sul Web, il quale sta diventando una collezione di oggetti semi-strutturati, i.e., dati per cui è disponibile una rappresentazione parziale della loro struttura (conosciuta come *schema* nel lessico delle basi di dati). Questa evoluzione porta alla necessità di avere nuovi linguaggi per l'estrazione di informazioni dalle sorgenti XML, così come i linguaggi di interrogazione tradizionali (SQL, OQL) sono nati per l'estrazione di informazione da sorgenti di dati strutturati, per esempio, basi di dati relazionali.
La definizione di un linguaggio di interrogazione per XML dovrebbe essere associata allo studio e alla produzione di strumenti per la rappresentazione di metadati. Infatti, un

documento XML può essere affiancato da una Document Type Definition (DTD), che specifica i tipi degli elementi che possono apparire nel documento, i loro attributi e le loro relazioni di contenimento. Se un documento XML è conforme ad una DTD, si dice che è *valido*. D'altra parte, i documenti XML non necessariamente devono essere accompagnati da una DTD; se questi non lo sono ma rispettano alcune regole sintattiche per il posizionamento dei tag si dicono *ben formati*. La nozione di DTD in XML si presta bene ad interpretare un documento XML come uno schema. La disponibilità di uno schema è molto utile per la formulazione di interrogazioni così come per la loro ottimizzazione. Elencheremo di seguito alcuni requisiti che caratterizzano sia i linguaggi di interrogazione sia quelli di rappresentazione di metadati:

- essere abbastanza flessibile da permettere la formulazione *di interrogazioni di documenti validi e/o ben-formati*.
- poter interrogare più documenti alla volta.
- dare la possibilità di accedere ai *metadati* così come ai dati, usando la stessa sintassi per le interrogazioni.
- supportare l'estrazione di uno snippet del documento XML a partire da un'interrogazione e/o rielaborare i risultati ottenuti specificando nuovi elementi. Questo significa offrire la possibilità di formattare il risultato introducendo nuovi elementi.
- prevedere *espressioni regolari* per specificare condizioni sui nodi. Questo aspetto è particolarmente utile nel caso di ricerche in cui non si conosca la struttura del documento XML sorgente.
- essere possibilmente semplice e intuitivo in modo tale da permetterne l'utilizzo anche a persone poco esperte e da rendere facile la lettura dell'interrogazione.

1.3 Linguaggi esistenti

Essendo XML uno standard emergente per lo scambio di informazioni nel World Wide Web, è chiaro che una quantità sempre maggiore di dati in internet saranno codificati in XML nel prossimo futuro. Un dato in XML è auto-descrittivo, cioè, la semantica del dato stesso è espressa dagli elementi XML. Di conseguenza, la sintassi di un'interrogazione non è influenzata dal fatto che documenti XML possano provenire da sorgenti eterogenee. Quindi, la capacità di interrogare documenti XML giocherà un ruolo chiave nelle applicazioni come commercio elettronico, e-newspaper, information retrieval, e così via.
Poiché il modello dei dati in XML è differente da quello delle basi di dati tradizionali, non si possono usare direttamente le tecniche progettate per esse. Infatti, il modello dei dati XML è basato su grafi, mentre i modelli delle basi di dati relazionali sono basati sul concetto di "tuple relazionali" e i modelli delle basi di dati orientati agli oggetti sulla "gerarchia di classi". Come memorizzare e interrogare un documento XML è un problema molto interessante così come l'estrazione, la trasformazione e l'integrazione di dati sono problemi per basi di dati ben conosciuti. Le loro soluzioni sono spesso affidate ai linguaggi di interrogazione relazionali (SQL) od orientati agli oggetti (OQL). Questi linguaggi non sono adatti per l'XML poiché il modello di dati relazionale è diverso dal modello di dati *semi-strutturato* di XML. Diversi linguaggi di interrogazione sono stati progettati e implementati per trattare dati semi-strutturati.
Nonostante la differenza nel modello dei dati, inizialmente venivano utilizzate le basi di dati tradizionali per memorizzare e interrogare documenti XML. In questo caso, le interrogazioni su dati semi-strutturati vengono convertite nel corrispondente linguaggio per basi di dati (SQL, OQL) e vengono usati i *query engines* dei DBMS per la loro

risoluzione. In questo contesto è importante creare lo *schema* della base di dati in modo appropriato. Cioè, un diverso disegno di schema influisce sull'efficienza della memorizzazione e sulla performance del processo di interrogazione.
Negli anni c'è stata poi un'evoluzione dalle basi di dati relazionali, attraverso le basi di dati orientate agli oggetti, fino alle basi di dati semi-strutturate, ma molti dei principi sono rimasti invariati. Sono emersi quindi importanti linguaggi, quali XML-QL, Lorel, YAT_L, QUILT, XDuce etc. Questi linguaggi operano su dati provenienti da sorgenti eterogenee, danno la possibilità di generare nuovi dati, e trasformano tali dati in formati scambiabili. Con il passare del tempo sono state studiate algebre e tecniche per rendere sempre più efficienti tali linguaggi. Nei prossimi paragrafi descriviamo in dettaglio alcuni di questi linguaggi.

1.3.1 YAT_L

YAT_L è un linguaggio funzionale, sviluppato da Sophie Cluet e Jérôme Siméon, che permette di effettuare operazioni su dati XML per interrogarli, convertirli e integrarli. A causa dell'elevato numero di operazioni richieste da ciascuna di queste funzionalità e della loro complessa interazione sono pochi i linguaggi che le supportano tutte; YAT_L è uno di essi. Dal punto di vista delle basi di dati YAT_L ha le principali caratteristiche di un linguaggio d'interrogazione, è dichiarativo e fornisce un bilanciamento tra espressività ed efficienza:

- *espressività*: fornisce operazioni elementari su alberi XML, primitive per confrontarli, per modificare la loro struttura e per combinare informazioni. Supporta anche una forma di ricorsione e ha la capacità di creare grafi (manipolando riferimenti).

- *ottimizzabilità*: fornisce un meccanismo per la conversione delle espressioni dichiarative in un'*algebra* per la valutazione. Quest'*algebra*, composta da un'insieme fisso di operazioni elementari, supporta varie tecniche di ottimizzazione. A livello logico l'ottimizzazione può trarre vantaggio dalle regole di riscrittura delle operazioni algebriche e dalle proprietà commutative di alcuni operatori. A livello fisico può scegliere il miglior algoritmo per ciascuna operazione (con o senza pipeline, con uso di indici, con nested-loop, merge-sort, giunzioni, etc.).

Da un altro punto di vista YAT_L è un linguaggio funzionale, ciò aiuta a risolvere i problemi tradizionali dei linguaggi per basi di dati come ad esempio la ricorsione, il trattamento dei riferimenti, i confronti e la gestione delle alternative. YAT_L è implementato nel linguaggio funzionale *Objective CAML* **[CAML]** diversamente da altri linguaggi d'interrogazione che, pur pensati in modo funzionale, sono stati implementati con linguaggi tradizionali.
Il modello dei dati di YAT_L astrae le informazioni contenute nel documento XML modellandole attraverso alberi con nodi etichettati. Le foglie di tali alberi contengono valori atomici, mentre i nodi interni corrispondono agli elementi XML. Tale modello permette di identificare elementi e successivamente riferirli.
Il linguaggio è costituito da un nucleo contenente alcune primitive di base per costruire, accedere o confrontare alberi, e per eseguire operazioni aritmetiche. Tale nucleo supporta anche la definizione di funzioni e la chiamata di funzioni, anche ricorsive.

Per creare un albero si usa una sintassi diversa da quella originale di XML. Ad esempio l'espressione YAT_L in figura 1.5a costruisce un albero equivalente a quello associato all'espressione XML di figura 1.5b.

```
painting[artist["Claude Monet"],
         title["Nympheas"],
         style["impressionist"],
         size[width[21],
              length[61]]];
```

Figura 1.5a – Espressione YAT_L per la creazione di un albero

```
<painting>
  <artist>Claude Monet</artist>
  <title>Nympheas</title>
  <style>impressionist</style>
  <size>
    <width>21</width>
    <length>61</length>
  </size>
</painting>
```

Figura 1.5b – Espressione XML

Vi è anche la possibilità di dare un nome agli alberi e creare riferimenti usando gli operatori ':=' e '&'.
Sul nucleo appena descritto si basano le funzionalità avanzate del linguaggio. Tali funzionalità fanno uso degli iteratori `match` e `case`. Il primo è molto simile alla **`SELECT-FROM-WHERE`** di SQL, mentre il secondo è stato introdotto per trattare le alternative.
Ad esempio l'interrogazione in figura 1.6 costruisce un albero con la radice etichettata `MonetPaintings`, ed un figlio per ogni oggetto della classe `artifact` il cui campo `creator` è "`Claude Monet`" nella foresta `allObjects`.

```
make MonetPaintings[*artifact[V]]
match allObjects with *object[@class["artifact"], $V]
where (V/tuple/creator) = "Claude Monet";
```

Figura 1.6 – Esempio di uso dell'iteratore match

L'iteratore `match` è formato da tre clausole, **`make`**, **`match`** e **`where`**. Le clausole **`make`** e **`where`** sono opzionali. La clausola **`match`** definisce un insieme di variabili legate che vengono usate dalle altre due clausole. I legami sono ottenuti dal confronto di un'espressione con un albero o con una foresta. Nell'esempio di figura 1.6 l'espressione `allObjects with *object[@class["artifact"], $V]` confronta tutti gli oggetti della classe `artifact` e lega la variavile `$V` al secondo figlio di ogni `object`. La clausola **`where`** filtra l'insieme dei legami rispetto ad una condizione booleana. Nell'esempio si mantengono solo i legami di `$V` in cui il `creator` è `"Claude Monet"`. La clausola **`make`** contiene un'espressione che viene applicata all'insieme di legami filtrato per costruire il risultato. Nell'esempio l'espressione contiene l'operatore `*` che è usato per raggruppare tutti i legami di `$V` sotto il nodo `MonetPaintings`. L'espressione `artifact[V]` che segue l'operatore viene applicata ad ogni elemento di questo gruppo.
L'iteratore `case` si ispira alla classica espressione *case* dei linguaggi di programmazione moderni ma esegue un confronto come l'iteratore `match`.
Ad esempio la funzione in figura 1.7 gestisce due schemi alternativi per il nome di una persona, restituendo il contenuto corrispondente nel nuovo elemento `name`.

```
define process_name ($E) =
case E of
| name[fn[$FN],
       ln[$LN]
  -> name[concat(FN,LN)]
| full_name[$N]
  -> name[N];
```

Figura 1.7 – Esempio di uso dell'operatore case

La valutazione delle alternative avviene nell'ordine specificato dal costrutto **`case`**.
Il linguaggio prevede anche la possibilità di esprimere la ricorsione che, insieme all'iteratore **`case`**, è essenziale per trattare le strutture eterogenee di XML e per effettuare le conversioni.

1.3.2 Lorel

Lorel è un linguaggio di interrogazione implementato sulla base del sistema *Lore*, creato all'università di Stanford per la gestione di dati semi-strutturati. Lo stile di tale linguaggio ricalca quello di SQL/OQL e ciò contribuisce a renderlo facilmente utilizzabile. Un interrogazione in Lorel ha la seguente forma:

```
select Guide.restaurant.address
where  Guide.restaurant.address.zipcode = 92310
```

questa interrogazione permette di trovare gli indirizzi di tutti i ristoranti che hanno il codice postale uguale a 92310. Nonostante la somiglianza nello stile con SQL/OQL, il modello dei dati sottostante richiede di avere nuove funzionalità:

- Un meccanismo di "coercion" per rendere l'utente libero dalla forte tipizzazione di OQL inappropriata per dati semi-strutturati;
- potenti "*path expressions*", che permettono una flessibile navigazione nella struttura dei dati e sono particolarmente adattabili quando i dettagli di tale struttura non sono completamente conosciuti dall'utente.

Prima di illustrare nel dettaglio queste funzionalità, introduciamo il modello seguito per la rappresentazione del documento XML, ovvero l'OEM, Object Exchange Model **[OEM]**, particolarmente utile per dati semi-strutturati. Il documento viene rappresentato come un grafo orientato ed etichettato, con gli oggetti ai vertici e le etichette sugli archi. Ogni oggetto ha un identificatore unico (oid). Gli oggetti che non hanno archi uscenti sono detti *atomici* e ad essi sono associati dei valori di tipo primitivo, come interi, reali, stringhe etc.; gli oggetti non atomici sono detti *complessi*. Le etichette degli archi sono stringhe.
La "coercion", prima innovazione di Lorel rispetto a SQL/OQL, pur agevolando l'utente costringe ad attuare il confronto tra oggetti e/o valori che possono avere tipi differenti. Facendo riferimento all'esempio precedentemente riportato non è necessario che l'utente sappia se il codice postale sia rappresentato come intero o come una stringa, infatti otterrà comunque il risultato atteso. Questa funzionalità è implementata estendendo i predicati di base (e.g., =) e le funzioni di base (e.g., +) di OQL e definendo un nuovo valore sull'operatore di uguaglianza.

In generale, predicati e funzioni si aspettano argomenti di tipi atomici particolari. A volte essi accettano più di un tipo; e.g., il predicato < si applica ad interi e reali. Nel contesto di dati semi-strutturati, si preferisce accettare condizioni come Z = 1.0 e Z > "0.9" come *vere* se Z è un oggetto con valore 1 o "1". In Lorel viene forzato il confronto tra oggetti atomici e valori, oggetti complessi e collezioni di oggetti. La tabella in figura 1.8 mostra le trasformazioni imposte sui tipi atomici stringa, interi, e reali, per gli operatori di confronto basilari (=, <). Vengono omessi i casi simmetrici.

arg2 arg1	string	Real	int
string	-	string -> real	entrambi ->real
real		-	int -> real
int			-

Figura 1.8 – "Coercion" per operatori di confronto

La "coercion" per operatori di confronto non è banale perché bisogna forzare entrambi i valori per confrontare tipi atomici. Per esempio, nel confronto "4.3" < 5, sia la stringa "4.3" che l'intero 5 devono essere "forzati" ad un reale per poter eseguire il confronto.
Un'altra novità di Lorel sono le potenti *path expressions*. L'idea è di specificare cammini nel grafo OEM basati su una sequenza di archi etichettati.
Ci sono due tipi di *path expressions*, le *simple path expression* e le *general path expression*.
Una *simple path expression* è una sequenza $Z.l_1….l_n$, dove $l_1….l_n$ sono etichette e Z è un nome di oggetto o una variabile che denota un oggetto. Le *path expresions* sono una funzionalità comoda e facilmente utilizzabile del linguaggio.
Descriviamo la semantica delle *simple path expressions* mostrando come le interrogazioni possono essere ridotte ad uno o più riferimenti ad *object-component* in stile OQL. Dato l'oggetto denominato `Guide` e la *simple path expression* `Guide.A.B.C`. questo cammino si può interpretare posizionandosi sull'oggetto `Guide`, seguendo un arco `A`, poi uno `B` e infine un arco `C`. Poiché ci potrebbero essere più archi etichettati `A`, `B` o `C`, l'espressione del cammino può corrispondere a più cammini nel grafo OEM. Il cammino può anche essere interpretato in OQL nel seguente modo: `Guide.A` denota un insieme di oggetti *R* che hanno un arco etichettato `A` da `Guide` a *R*, `Guide.A.B` denota gli oggetti *Z* tali che per qualche *R* in `Guide.A`, esiste un arco `B` da *R* a *Z*, e similmente per `Guide.A.B.C`.. L'interrogazione in Lorel riportata in figura 1.9a è uguale a quella in OQL in figura 1.9b con il cammino ridotto.

```
SELECT Z
FROM Guide.restaurant.zipcode Z
```

Figura 1.9a – Interrogazione in Lorel

```
SELECT Z
FROM   Guide.restaurant R,
       R.zipcode Z
```

Figura 1.9b –Interrogazione in OQL

Il tipo di riduzione di una *simple path expression* dipende da dove essa appare, se nella clausola **`FROM`**, **`SELECT`** o **`WHERE`**. Se l'espressione del cammino si trova nella clausola **`FROM`** il cammino viene ridotto inserendo una variabile dopo ogni etichetta. Se l'espressione si trova nella clausola **`SELECT`** occorre controllare se l'intera espressione si trova anche nella clausola **`FROM`** o no. Se l'espressione del cammino si trova sia nella clausola **`SELECT`** che

FROM allora traducendo la clausola **FROM** si ha già una variabile ed è sufficiente rimpiazzare nella **SELECT** l'espressione del cammino con tale variabile. Per esempio, l'interrogazione:

```
SELECT Guide.restaurant
FROM   Guide.restaurant.address.zipcode Z
WHERE  Z = 92310
```

viene tradotta in:

```
SELECT R
FROM   Guide.restaurant R, R.address A, A.zipcode Z
WHERE  Z = 92310
```

Consideriamo infine le *path expressions* che occorrono nella clausola **WHERE**. Come per la clausola **SELECT**, se l'espressione è un prefisso di qualche cammino presente nella clausola **FROM** si rimpiazza l'espressione con la variabile corrispondente presente nella clausola **FROM**.
Comunque, un'interrogazione **SELECT-FROM-WHERE** in Lorel ha la stessa semantica di una in SQL o OQL. In Lorel, il risultato è sempre una collezione di oggetti OEM. Come in SQL e OQL, per ogni oggetto legato alla variabile nella clausola **FROM** che soddisfa la condizione della clausola **WHERE**, viene generato un valore in accordo alle espressioni presenti nella clausola **SELECT**. Ognuno di questi valori è poi trasformato in un oggetto OEM. Questa trasformazione può comportare la creazione di nuovi oggetti e archi nel grafo OEM. Il risultato di un'interrogazione può essere riutilizzato in altre interrogazioni purché venga rinominato, in modo tale da non sovrascrivere la risposta precedente.
Le *simple path expressions* possono essere estese con una sintassi più potente diventando *general path espressions*. Una *general path expression* (*gpe*), come una *simple path expression*, inizia col nome di un oggetto o di una variabile. Le *gpe* permettono al nome dell'oggetto o della variabile di essere seguito da uno o più componenti *gpe*, piuttosto che da una sola sequenza di etichette come avviene nelle *simple path expressions*. La sintassi di una componente *gpe* è la seguente:

- Se *t* è un'etichetta, allora *.t* è una componente *gpe*.
- Se X è una variabile oggetto, allora .unquote(X) è una componente *gpe*.
- Se s_1 e s_2 sono componenti *gpe*, allora anche i seguenti sono componenti *gpe*:
  $s_1 s_2$ $s_1 \mid s_2$ (s_1) $(s_1)?$ $(s_1)+$ $(s_1)*$

Lorel può essere usato non solo per formulare ed eseguire le interrogazioni, ma anche per aggiornare la base di dati; esistono infatti al suo interno delle funzioni per creare ed eliminare i nomi delle basi di dati (*name*), per creare un nuovo oggetto atomico o complesso (*new_oem*), per modificare il valore di un oggetto atomico o complesso già esistente (*update*), e per caricare una base di dati OEM (*load*).

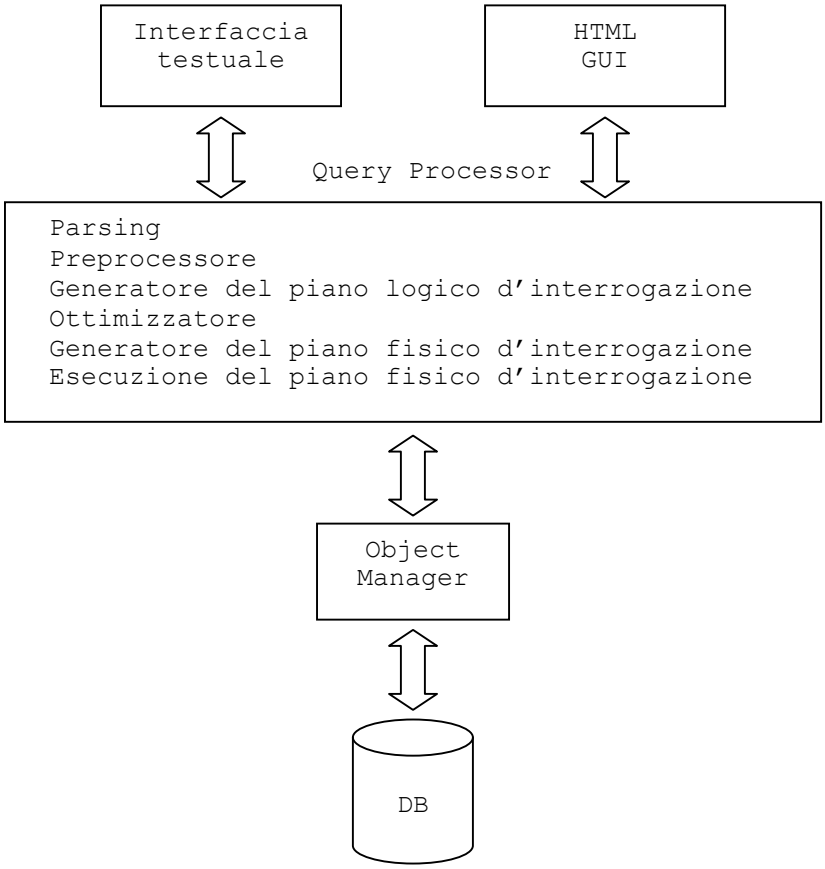


Figura 1.10 – Struttura del sistema LORE

Analizzando una schematizzazione della struttura del sistema *Lore* (figura 1.10), si oserva che la componente principale per l'esecuzione di un'interrogazione è il *Query Processor*, che si trova tra l'*User Interface* e l'*Object Manager*. Il *Query Processor* esegue i seguenti passi:

1. sottopone l'interrogazione al parsing,
2. analizza l'albero dell'interrogazione per tradurlo in formato OQL,
3. costruisce un piano logico per l'interrogazione,
4. ottimizza il piano logico dell'interrogazione,
5. traduce il piano logico ottimizzato in un piano fisico, ed
6. esegue il piano fisico.

Il *Query Processor* così come il resto del sistema è funzionale e robusto per un sottoinsieme del linguaggio *Lorel*. Esistono aspetti del linguaggio la cui implementazione è ancora in corso. Ad esempio il sistema mantiene strutture di indicizzazione che non sono ancora sfruttate dai piani di esecuzione delle interrogazioni e le operazioni di visita dell'albero sono costose e numerose.
Un server Lore con un certo numero di semplici basi di dati è disponibile per un uso pubblico all'indirizzo http://www-db.stanford.edu/lore. Gli utenti possono eseguire interrogazioni nel sottoinsieme del linguaggio Lorel correntemente implementato e sperimentare le funzionalità del linguaggio.

1.3.3 XML-QL

Presentiamo in questo paragrafo un linguaggio per interrogare file XML, chiamato XML-QL **[XML-QL]**. Il linguaggio ha una sintassi composta dalle clausole **`SELECT – WHERE`**, come SQL, e segue alcuni aspetti dei linguaggi di interrogazione per dati semi-strutturati sviluppati nell'ambito delle basi di dati. Presenta, comunque, alcune caratteristiche interessanti quali:

1. un modello dei dati ordinato;
2. una sintassi che combina elementi della sintassi XML con altri della sintassi dei linguaggi tradizionali;
3. una semantica che supporta l'ordine sia nei dati di input sia in quelli di output.

XML-QL è un linguaggio in grado di esprimere interrogazioni per estrarre dati da documenti XML, per trasformare documenti mappandoli da una DTD ad un'altra, e per integrare dati XML provenienti da sorgenti differenti. Il linguaggio viene presentato descrivendo le sue funzionalità e affiancando ad ognuna di esse un esempio illustrativo. Inoltre, contemporaneamente viene descritto a grandi linee il suo modello dei dati. Si parte da interrogazioni semplici che permettono, ad esempio, di cercare e restituire dati a partire da documenti XML e di costruire nuovi elementi, per arrivare a quelle più complesse che fanno uso ad esempio di *regular path expressions* o *variabili tag*.
La prima interrogazione mostra l'uso di espressioni per confrontare elementi in un documento, restituendo tutti gli elementi `<author>` dei `<book>` il cui `<publisher>` è `Addison-Wesley`:

```
WHERE <book>
        <publisher><name>Addison-Wesley</name></publisher>
        <title> $t</title>
        <author> $a</author>
      </book> IN "www.a.b.c/bib.xml"
CONSTRUCT $a
```

L'interrogazione cerca ogni elemento `<book>` presente nel documento `'www.a.b.c/bib.xml'` che ha almeno un elemento `<title>`, un `<author>` e un `<publisher>` il cui `<name>` è `Addison-Wesley`. Per ogni elemento `<title>` e `<author>` del risultato vengono legate le variabili `$t` ed `$a` rispettivamente. Il risultato restituito è la lista di tutti gli `<author>` legati ad `a`.
La prossima interrogazione illustra come si possono costruire elementi XML; l'esempio costruisce in particolar modo un nuovo elemento `<result>` nel risultato (che non esisteva nel documento), restituendo al suo interno entrambi gli elementi `<author>` e `<title>`:

```
WHERE <book>
        <publisher><name>Addison-Wesley</></>
        <title> $t</>
        <author> $a</>
      </> IN "www.a.b.c/bib.xml"
CONSTRUCT <result>
            <author> $a</>
            <title> $t</>
          </>
```

Una funzionalità interessante del linguaggo è la possibilità di legare il contenuto di un elemento e confrontare 'patterns' all'interno dell'elemento legato. Per mostrare tale funzionalità consideriamo un esempio in cui i risultati sono raggruppati in base ad un preciso elemento, in questo caso in base ai titoli dei libri, e viene usata un'interrogazione innestata che produce un risultato per ogni titolo, e in cui ogni titolo contiene una lista di tutti i suoi autori:

```
WHERE <book> $p</> IN "www.a.b.c/bib.xml",
      <title > $t</>,
      <publisher><name>Addison-Wesley</>> IN $p
CONSTRUCT <result>
             <title> $t </>
             WHERE  <author> $a </> IN $p
             CONSTRUCT <author> $a</>
          </>
```

Da notare che il 'pattern' che inizia con `<book>` è fuori dall'interrogazione innestata così da poter legare la variabile `$p` al contenuto dell'elemento. Una volta che il contenuto della variabile è stato legato, questo può apparire all'interno delle espressioni **IN** e **CONSTRUCT** innestate.
Poiché questa funzionalità è largamente usata, XML-QL ha introdotto la clausola **CONTENT_AS** preceduta da un'espressione e seguita da una variabile. Questa clausola lega il contenuto dell'elemento più esterno dell'espressione alla variabile. Ad esempio in:

```
WHERE <book>
        <title> $t </>
        <publisher><name>Addison-Wesley </> </>
      </> CONTENT_AS $p IN "www.a.b.c/bib.xml"
CONSTRUCT <result><title> $t </>
                  WHERE  <author> $a</> IN $p
                  CONSTRUCT <author> $a</>
          </>
```

la variabile `$p` viene legata al contenuto dell'elemento `<book>`.
XML-QL può esprimere giunzioni confrontando due o più elementi che contengono lo stesso valore. Per esempio per cercare tutti gli `<article>` che hanno almeno un `<author>` che ha scritto un `<book>` dal 1995 si scrive:

```
WHERE <article>
        <author>
          <firstname> $f </>
          <lastname> $l </>
        </>
      </> CONTENT_AS $a IN "www.a.b.c/bib.xml"

      <book year=$y>
        <author>
          <firstname> $f </> // giunzione sullo stesso elemento firstname $f
          <lastname> $l </>  // giunzione sullo stesso elemento lastname $l
        </>
      </> IN "www.a.b.c/bib.xml",
      y > 1995
CONSTRUCT <article> $a </>
```

Poiché la variabile `$a` è legata al contenuto dell'elemento `<article>`, per restituire anche tale elemento nel risultato lo si deve inserire esplicitamente nella clausola **CONSTRUCT**. Per evitare di creare nuovamente un elemento già esistente nel documento, il linguaggio

prevede la clausola **ELEMENT_AS** seguita da una variabile che viene legata all'intera espressione.
L'interrogazione precedente diverrebbe della forma:

```
WHERE <article>
        <author>
          <firstname> $f</> // firstname $f
          <lastname> $l</>  // lastname $l
        </>
      </> ELEMENT_AS $e IN "www.a.b.c/bib.xml"
...
CONSTRUCT $e
```

Aspetti avanzati di XML-QL permettono di effettuare interrogazioni che includono *variabili tag* e *regular path expressions*.
Le *variabili tag* pemettono ad una variabile di essere legata ad un elemento tag non conosciuto a priori, ad esempio in:

```
WHERE <$p>
        <title> $t </title>
        <year>1995</>
        <$e> Smith </>
      </> IN "www.a.b.c/bib.xml",
      $e IN {author, editor}
CONSTRUCT <$p>
            <title> $t </title>
            <$e> Smith </>
          </>
```

`$p` è la *variabile tag* che può essere legata all'elemento esterno `<book>` o `<article>` visto che la struttura di entrambi prevede un tag `<title>`, un tag `<year>` etc. Notiamo che anche `$e` è una *variabile tag* vincolata ad essere legata o ad un `<author>` o ad un `<editor>`.
Le *regular path expressions* permettono, invece, di specificare cammini arbitrari nel grafo XML. Il grafo XML è il risultato del modello dei dati di XML-QL che è simile a quello per i dati semi-strutturati. Qui un documento XML viene modellato attraverso un grafo i cui archi sono etichettati con gli elementi, i nodi con le coppie (attributo, valore) e le foglie con valori di tipo stringa. Un uso di *regular path expression* è illustrato nella seguente interrogazione che restituisce il nome di ogni elemento `<part>` che contiene un elemento `<brand>` uguale a `Ford`, senza curarsi del livello di innestamento al quale si trova la variabile `$r`:

```
WHERE <part*> <name>$r</> <brand>Ford</> </> IN "www.a.b.c/bib.xml"
CONSTRUCT <result>$r</>
```

Qui `<part*>` è una *path expression* che è soddisfatta da qualsiasi sequenza di archi, tutti etichettati `<part>`. Le *regular path expressions* di XML-QL forniscono gli operatori di concatenazione (.), disgiunzione (|) e chiusura di Kleene (*), simili a quelli usati nelle *path expressions*.
Con XML-QL è inoltre possibile *trasformare* dati XML, ad esempio generando dati conformi ad una nuova DTD, ed *interrogare* simultaneamente sorgenti diverse di dati dando quindi una visione globale di essi. Per esempio la seguente interrogazione calcola tutte le coppie (nome, reddito) dal documento `www.a.b.c/data.xml` e `www.irs.gov/taxpayers` rispettivamente, e le lega alla variabile `$ssn`, e restituisce un

risultato che contiene gli elementi che hanno sia un elemento <name> nel primo documento che uno <income> nel secondo documento:

```
WHERE <person>
        <name></> ELEMENT_AS $n
        <ssn> $ssn</>
      </> IN "www.a.b.c/data.xml",

      <taxpayer>
        <ssn> $ssn</>
        <income></> ELEMENT_AS $i
      </> IN "www.irs.gov/taxpayers.xml"
CONSTRUCT <result> $n $i </>
```

Tutti gli esempi visti finora hanno alla base un modello di dati non ordinato, in cui si ignora l'ordine in cui appaiono gli elementi. In realtà XML-QL supporta due modelli di dati distinti: uno ordinato e uno non ordinato. Un grafo XML ordinato è simile ad uno non ordinato, ma include per ogni nodo, un ordine totale sui suoi discendenti. Sul modello di dati ordinato XML-QL fornisce costrutti addizionali per supportare tale ordine. Il prezzo da pagare per un modello di dati ordinato è avere una semantica del linguaggio di interrogazione più complessa, e tempi di risposta maggiori. Sebbene la semantica di un modello di dati non ordinato sia più semplice, bisogna comunque scegliere un ordine quando si vuole trasformare un grafo XML non ordinato in un documento XML.

1.3.4 TQL

Il linguaggio di interrogazione Tree Query Language **[TQL]**, è stato realizzato presso il Dipartimento di Informatica dell'Università di Pisa nel 2002, al fine di gestire dati semi-strutturati basandosi su un ambiente logico. Il linguaggio è abbastanza espressivo: un'interrogazione TQL non è altro che un insieme di operazioni innestate, ognuna costruita su una formula logica.
La parte strutturale dell'ambiente logico è essenzialmente una logica progettata per descrivere proprietà di alberi etichettati. Questa è un buon fondamento per un linguaggio di interrogazione per dati semi-strutturati, allo stesso modo di come la logica del prim'ordine è un fondamento per i linguaggi di interrogazione relazionali.
Descriviamo TQL attraverso degli esempi, iniziando con le interrogazioni standard suggerite dalla W3C XMP Use Cases. Le interrogazioni in TQL vengono valutate in un ambiente globale, definito dal'utente, in cui alcune variabili sono legate a file XML locali o remoti. Illustriamo, nell'esempio che segue, la struttura di un documento (si veda `//tql.di.unipi.it/tql/pubb.xml`) contenente annotazioni bibliografiche su cui effettueremo alcune interrogazioni:

```
bib[
    book[year[1999]
      | title[DataOnTheWeb]
      | author[ first[Serge] | last[Abiteboul] ]
      | author[ first[Dan] | last[Suciu] ]
      | author[ first[Peter] | last[Buneman] ]
      | publisher[MorganKaufmann]
      | price[45]
    ]
    book[year[1995]
      | title[FoundationsDatabases]
      | author[ first[Serge] | last[Abiteboul] ]
      | author[ first[Richard] | last[Hull] ]
```

```
      | author[ first[Victor] | last[Vianu] ]
      | publisher[Addison]
      | price[60]
    ]
    | book[year[1999]
      | title[ProcICDT99]
      | editor[ first[Peter] | last[Buneman] ]
      | publisher[Springer]
      | price[12]
    ]
  ...
]
```

dove `bib[C]` indica l'elemento `bib` il cui contenuto è `C`, mentre `C1 | C2` è la concatenazione di due elementi o, in generale, di due insiemi di elementi.
La notazione TQL è differente da quella di XML perché TQL è stato progettato come un linguaggio per interrogare dati semi-strutturati, cioè alberi non ordinati i cui archi sono etichettati; XML è invece solo un modo per costruire tali alberi, usando elementi (e attributi).
L'interrogazione base del linguaggio è

```
FROM   Q |= A
SELECT Q'
```

Dove `Q` è l'insieme di dati sorgente che deve essere sottoposto al confronto con la formula `A`, e `Q'` è l'espressione risultato. Il confronto di `Q` con `A` restituisce l'insieme dei legami delle variabili che sono libere nella formula `A`. `Q'` viene valutata una volta per ognuno di questi legami, e la concatenazione dei risultati di tutte queste valutazioni è il risultato finale dell'interrogazione.
Per esempio, consideriamo la seguente interrogazione TQL, che restituisce i titoli di tutti i libri scritti nel 1999, e che viene valutata nell'ambiente dove la variabile `$Bib` è legata nel seguente modo:

```
FROM  $Bib |= .bib[.book[.year[1999]
                              And .title[$t]
                        ]
                  ]
SELECT title[$t]
```

La formula `.bib[.book[.year[1999]And.title[$t]]]` è una formula logica, che dovrebbe essere letta come "esiste un cammino `.bib[.book[]]` che raggiunge un nodo che soddisfa l'AND tra `.year[1999]` e `.title[$t]`, cioè un nodo che soddisfa il cammino `.year[]` che porta a 1999 e il cammino `.title[]` che porta all'oggetto legato alla variabile `$t`".
Poiché il documento XML preso in esame contiene due libri con l'anno 1999, uno con il titolo `DataOnTheWeb` e l'altro con titolo `ProcICDT99`, l'interrogazione prima calcola l'insieme dei legami:

```
{[$t = DataOnTheWeb]; [$t = ProcICDT99]}
```

e poi valuta la sotto-interrogazione `title[$t]`, una volta, per ognuno di tali legami, portando al risultato:

```
title[DataOnTheWeb] | title[ProcICDT99].
```

La formula `.t[A]` si legge "esiste un elemento `t` il cui contenuto soddisfa `A`"; questa è realmente definita in termini di tre operatori primitivi, `T` che è la formula sempre vera, la separazione orizzontale `A1 | A2` e l'elemento di confronto `t[A]`. La formula `t[A]` soddisfa solo un elemento, quindi per esempio mentre `.t[A]` soddisfa entrambi gli alberi `t[D]` e `t[D]|[E]|...` (sapendo che `A` soddisfa `D`) la formula `t[A]` soddisfa solo il primo. La formula `T` è soddisfatta da ogni albero. Infine le formula `A1 | A2` soddisfa `D` se e solo se `D` è uguale, a `D1 | D2` con `A1` che soddisfa `D1` e `A2` che soddisfa `D2`.
Mentre nell'esempio di sopra sono state confrontate variabili con alberi, una variabile TQL può anche essere confrontata con un tag. Per esempio, l'interrogazione seguente restituisce qualsiasi tag contenuto nell'elemento `<book>` che contiene `first[Serge]` (il risultato è `SergeTag[author]`):

```
FROM   $Bib |= .bib.book.$tag.first[Serge]
SELECT SergeTag[$tag]
```

In seguito, come convenzione, useremo iniziali minuscole per le variabili che sono legate a dei tag e iniziali maiuscole per variabili che sono legate a degli alberi.
La logica di TQL contiene sia operatori strutturali che operatori della logica del primo ordine. Gli operatori strutturali (`t[A], A1 | A2, ...`) possono essere usati per esprimere condizioni sui confronti, e gli altri possono essere usati per combinare tali condizioni e per quantificare variabili.
Per esempio, la condizione nell'interrogazione seguente richiede l'esistenza di un campo `title`, di una variabile `$x` che contiene `Springer`, e di un `author.last` o di un `editor.last` che raggiungono `Buneman`:

```
FROM $Bib |=
      .bib.book [.title[$t]
                 And Exists $x. .$x[Springer]
                 And (.author.last[Buneman] Or
                       .editor.last[Buneman])
                 ]
SELECT title[$t]
```

Il pattern `Exists $x. .$x[Springer]` può essere abbreviato con `.%[Springer]`, che useremo d'ora in poi e dove il simbolo '`%`' può essere letto come 'che soddisfa qualsiasi etichetta'.
Infine la logica di TQL include anche un operatore di ricorsione che può essere usato, per esempio, per definire un altro operatore per cammini, `.%*[Springer]`, che confronta un cammino di lunghezza arbitraria. Per esempio, l'interrogazione seguente cerca a qualsiasi profondità dell'albero, qualsiasi pubblicazione `$pub` dove `Suciu` ha un ruolo definito da `$role`. L'interrogazione restituisce il titolo della pubblicazione e il campo dove compare `Suciu`, restituendo anche i tag di entrambi.

```
FROM $Bib |= .%*.$pub[ .title[$T]
                        And .$role[.lastname[Suciu]]
                      ]
SELECT $pub[ title[$T] | $role[Dan Suciu] ]
```

Descriviamo ora come TQL può essere usato per controllare proprietà strutturali di dati semi-strutturati. Quando una formula chiusa `A` esprime una proprietà per noi interessante, la si può controllare eseguendo un'interrogazione del tipo

```
Q |= A SELECT success.
```

Questa interrogazione restituisce un arco etichettato `'success'` se `A` è valida per `Q`, un albero vuoto altrimenti.
Come primo esempio consideriamo un'interrogazione che verifica se il tag <title> è presente in tutti gli elementi <book> del documento `$Bib`:

```
FROM   $Bib |= bib[Not .book[Not .title[T]]]
SELECT title_is_mandatory
```

La formula `Not .book[Not .title[T]]` significa: non esiste un <book> che non contiene nessun <title>, cioè ogni <book> contiene un <title>. In realtà, TQL è caratterizzato da un operatore `!t[A]` definito come `Not .t[Not A]` che può essere usato direttamente come nel seguente esempio. Qui `!book.title[T]` è un'abbreviazione per `!book[.title[T]]`, e significa "per ogni libro c'è un titolo".

```
FROM   $Bib |= bib[ !book.title[T] ]
SELECT title_is_mandatory
```

La formula `!t[A]` è duale a `.t[A]` così come sono duali $\forall$`x.A` e $\exists$`x.A`, oppure $\wedge$ e $\vee$. In TQL ogni operatore primitivo ha il suo duale; questo implica che la negazione può essere sempre spinta all'interno di qualsiasi operatore. Perciò possiamo riscrivere qualsiasi interrogazione così che le uniche formule negate sono quelle atomiche. Infatti quando appare una negazione in un'interrogazione, nella maggior parte dei casi l'ottimizzatore TQL la spinge nelle formule atomiche, anche perché la negazione è abbastanza costosa.
Illustriamo ora un'interrogazione che controlla se `$Bib` contiene solo elementi etichettati <book>, vedendo che ogni tag all'interno del tag esterno <bib> sia uguale a <book>:

```
FROM   $Bib |= bib[foreach $x .$x[T] implies $x=book]
SELECT only_book_inside_bib
```

Questa interrogazione può anche essere riscritta usando operatori per cammini, come segue:

```
FROM   $Bib |= bib[Not (.Not book)[T]]
SELECT only_book_inside_bib
```

In questo caso `Not book` è un'espressione che sta per qualsiasi tag diverso da `book`. Per questo `(.Not book)[T]` significa: esiste un sotto-elemento `T` il cui tag è diverso da `book`. E inoltre `Not(.Not book)[T]` significa: non esiste nessun sotto-elemento il cui tag è diverso da `book`.
Anche in TQL è possibile imporre alcune proprietà sui tag. In un'interrogazione TQL una *variabile tag* può apparire in qualsiasi punto in cui può apparire un tag. Quindi, si può controllare se <title> è una chiave e sostituire <title> con `$k` nel seguente modo:

```
FROM  $Bib |=
        bib[!book[.$k[T]]
             And foreach $X. Not (.book.$k[$X] | .book.$k[$X])
            ]
SELECT key[$k]
```

Questa interrogazione è *ben formata*, e restituisce l'insieme di tutti di sotto-tag `book` il cui contenuto è una chiave per l'insieme dei libri. Per ogni interrogazione Q che controlla una

proprietà P di un tag t, se si sostituisce t con una variabile tag, si ottiene un'interrogazione che cerca l'insieme di tutti i tag che soddisfano la proprietà P.
La logica di TQL include anche due opertori di ricorsione (*rec* e *maxrec*), simili agli operatori di minino e massimo punto fisso della logica modale. Questi possono essere usati arbitrariamente per attraversare cammini in profondità, generalizzando l'operatore `.%*` visto precedentemente, e per esprimere proprietà ricorsive degli alberi. Consideriamo per esempio la seguente formula:

```
rec $Binary. 0 Or (%[$Binary] | %[$Binary])
```

La formula descrive un albero binario, definito o come albero vuoto o come albero con due figli entrambi `Binary`.
Tutte le interrogazioni viste finore sono state eseguite dal TQL query engine. Eseguire tali interrogazioni su documenti reali richiede abbastanza tempo. Questo non sorprende, poiché l'implementazione corrente è un prototipo con l'obiettivo di mostrare che un tale linguaggio può essere implementato.
Analizzando il modello dei dati di TQL, questo è essenzialmente una versione non ordinata del XML Query Data Model definito dalla W3C (W3C, 2002c). Una differenza principale è che il modello della W3C considera sette tipi differenti di nodi (elementi, attributi, testo, …), mentre TQL ne considera solo uno (elementi), e inoltre il modello della W3C assegna un'identità ad ogni nodo cosa che TQL non fa. L'identità di un nodo permette il confronto in modo da poter distinguere se essi sono stati costruiti da due differenti applicazioni a partire da uno stesso nodo costruttore. Il modello della W3C descrive dati come ad esempio, una foresta di nodi etichettati, mentre il modello TQL opera su alberi con archi etichettati. I due comunque sono perfettamente isomorfi. Infatti i dati TQL possono essere visti come foreste di nodi etichettati interpretando '0' come foresta vuota, F | F_0 come l'unione di foreste F e F_0, e t[F] come un albero con radice nel nodo t e i cui figli sono gli alberi della foresta F.
La versione implementata di TQL, però, ha un modello dei dati più ricco, poiché vengono presi in considerazione anche nodi di tipo testo.
Abbiamo già detto che TQL si basa su una logica completa. È risaputo anche che la disgiunzione, la negazione e la quantificazione universale creano problemi di 'sicurezza' nei linguaggi di interrogazione basati su una logica. Il problema appare anche in TQL. Per questo motivo, i progettisti hanno seguito una strada diversa, definendo un meccanismo di valutazione che opera su ogni formula, 'sicura' o 'non-sicura'. Il meccanismo è basato su una rappresentazione finita di ogni insieme di valori calcolato, finito o infinito. Questo meccanismo permette di valutare ogni legame così com'è, senza la necessità di scartare una formula 'non-sicura' o riscriverne altre in una forma più accettabile. Questo approccio non è nuovo nel campo delle basi di dati, di cui fa parte anche TQL. In questo modo, l'ottimizzatore è libero di riscrivere qualsiasi formula in un'altra, senza preoccuparsi delle condizioni riguardanti la 'sicurezza' questo perché nonostante i risultati intermedi possano essere infiniti il risultato finale della valutazione di una formula può essere finito o infinito. A questo punto, se l'insieme calcolato è infinito, si solleva un errore a run-time poiché gli sviluppatori di TQL non hanno considerato interessante definire una rappresentazione finita di un albero infinito. Perciò sebbene sia stato risolto il problema di valutare formule 'non sicure', si cerca ancora di identificare una classe di formule che garantiscano di non restituire mai un insieme infinito di valori. Questo permetterebbe di analizzare un'interrogazione staticamente e garantire al programmatore di non sollevare mai un'eccezione di 'risultato infinito'. Tuttavia questo test è abbastanza differente da quello sulla 'sicurezza' in ambito relazionale. In quest'ultimo, solo le formule che passano

il test vengono tradotte nell'algebra ed eseguite. Nel caso di TQL, ogni formula può essere tradotta ed eseguita, ma se non passa il test, questa potrebbe restituire un risultato infinito.

1.3.5 CXQuery

CXQuery (*Constraint XML Query Language*) è un linguaggio per interrogare documenti XML, proposto da Yi Chen e Pete Revesz **[CXQuery]**, con l'obiettivo di sviluppare un linguaggio dichiarativo, facile da imparare e con potenti funzionalità di interrogazione per dati semi-strutturati. Inoltre è stato progettato con lo scopo di superare le limitazioni poste dai linguaggi come XML-QL, Lorel, Quilt **[Quilt]** tra le quali elenchiamo:

1. lo schema del risultato di un'interrogazione che non è esplicitamente specificato;
2. il non utilizzo della DTD del documento sorgente nell'interrogazione che porta l'utente a dover scrivere nell'interrogazione un'espressione XML complicata.
3. l'uso di complicate espressioni per costruire il risultato, dovendo riportare l'espressione completa che rispetta lo schema originale.

CXQuery segue alcuni aspetti di SQL e di altri linguaggi e si basa sull'uso della DTD per specificare esplicitamente lo schema del risultato di un'interrogazione. Gli sviluppatori hanno seguito lo schema della clausola **`SELECT`** di SQL e fatto uso degli operatori di aggregazione, come **`AVG`** e **`SUM`**. Inoltre, viene usata la funzione `document()` di XQuery per specificare il documento XML da interrogare e si fa uso delle *regular expressions* di XPath **[XPath]**. CXQuery è simile a XPathLog **[XPathLog]** perché entrambi sono linguaggi basati su *regole*. In XPathLog una regola atomica è costruita su un'espressione XPath, mentre in CXQuery le regole atomiche vengono costruite sulla base di una DTD che offre una vista orizzontale dello schema facilmente frammentabile e ricostruibile.
L'input di una interrogazione con CXQuery è un insieme di documenti XML e l'output è ancora un documento XML.
Un'espressione di CXQuery contiene una regola principale ed una regola secondaria. Queste sono separate dal simbolo `":-"`. La regola secondaria contiene un insieme di predicati separati da virgole che si intendono legati dall'operatore logico "AND". L'espressione di CXQuery è semplificata utilizzando un sottoinsieme delle funzionalità di XPath usate per navigare nella struttura gerarchica dei documenti XML ed evitare conflitti nello spazio dei nomi. Quindi viene utilizzato il simbolo "/" per denotare il nodo radice e il figlio del nodo corrente. Viene inoltre utilizzata la funzione `document()` per denotare la radice del documento XML. Nell'esempio seguente Q1, Q2, Q3 sono tutte espressioni valide in CXQuery:

```
(Q1)  document("campus.xml")
      Building(name, dept, spatial);

(Q2)  document("campus.xml")
      //Building(name, dept, spatial);

(Q3)  document("campus.xml")
      /CampusBuilding
      /Building(name, dept, spatial);
```

Mostriamo ora l'uso della DTD nella regola principale per confrontare un'espressione XML e per generare lo schema del risultato. Per esempio supponiamo di avere uno schema che contiene informazioni sugli edifici di un Dipartimento di Informatica e si vuole trovare

dove sono allocati i dipartimenti e poi tutte le informazioni associate all'edificio chiamato `"Ferguson hall"`:

```
buildingview(name,dept,spatial):-
      document("campus.xml"),
      building(name,dept,spatial),
      dept/department="Computer Science";
CSBuilding(name,dept,spatial):-
      buildingview(name,dept,spatial),
      name = "Ferguson hall".
```

In questa interrogazione è stato creato un nuovo elemento nel risultato chiamato `'CSBuilding'`.
Quando un'interrogazione viene valutata, durante la fase di parsing viene estratta la DTD associata al documento per essere trasferita al DTD Matcher (vedi figura 1.11). Tutti i predicati presenti nell'interrogazione vengono confrontati dal DTD Matcher durante l'esecuzione. Lo schema del risultato dell'interrogazione viene poi unito con la DTD originale dal DTD Generator. Quando lo schema generato dal risultato è conforme alla DTD originale, l'interrogazione ha prodotto un aggiornamento XML.

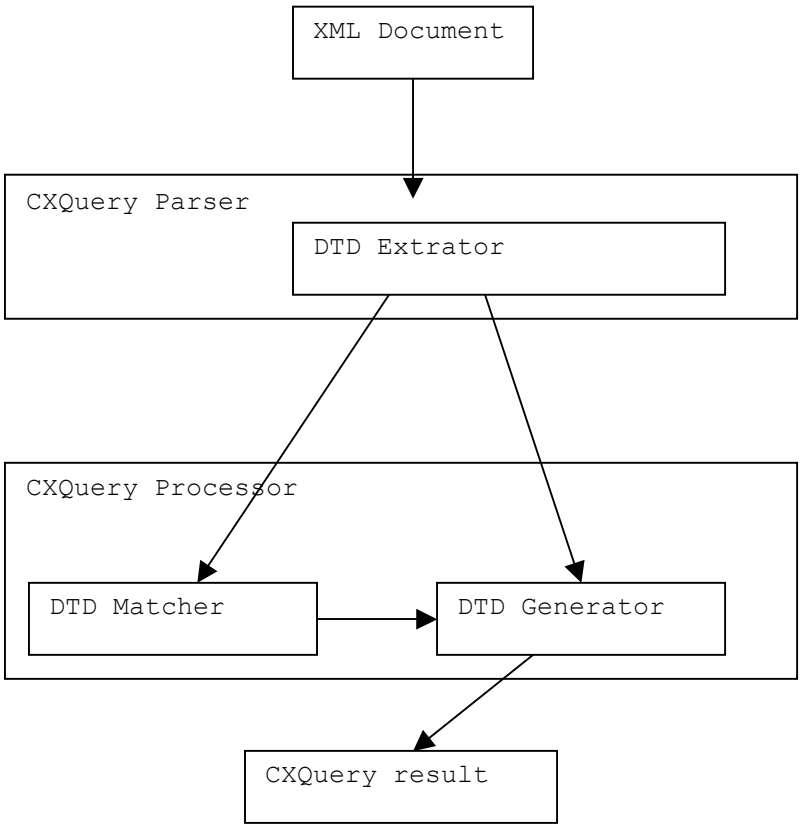


Figura 1.11 – DTD in CXQuery

CXQuery fornisce una libreria di funzioni utili nelle interrogazioni XML. In essa sono presenti funzioni di aggregazione tipiche di SQL come (`avg, sum, count, max e min`) e operatori spaziali e spazio-temporali, come area e lunghezza, per supportare interrogazioni su dati spaziali e spazio-temporali. Le funzioni proposte forniscono un'interfaccia per accedere alle informazioni spazio-temporali codificate in XML che aumentano il potere del linguaggio. Per esempio la seguente interrogazione permette di

rimpiazzare l'elemento `spatial` dell'elemento `building` con un elemento `area` che rappresenta l'area dell'edificio:

```
building(name,dept, buildingarea:area(spatial)):-
      document("campus.xml"), building(name, dept, spatial).
```

Come già osservato più volte ogni documento XML può essere modellato come un albero etichettato in cui le foglie sono tipi di dato primitivi, per esempio, interi o stringhe, ed i nodi sono tipi di dato complessi.
Anche in CXQuery è permesso legare variabili ai nodi. Cioè, il nome di una variabile legata ad uno specifico nodo (un tag nel documento) denota il sottoalbero che ha quel nodo come radice. La seguente interrogazione cerca tutti gli edifici nel documento `campus.xml` il cui nome è '`Ferguson Hall`':

```
document("campus.xml"),
building(name, dept, spatial),
name = "Ferguson Hall".
```

Infine CXQuery permette di manipolare dati per aggiornare i documenti XML. L'interrogazione seguente aggiorna il nome dell'edificio `Ferguson Hall` in `Ferguson Building`. Qui viene mostrato anche l'uso dell'operatore '`:`' come operatore di assegnamento.

```
building(name:"Ferguson Building",
                  dept,spatial) :-  document("campus.xml"),
                                    building(name, dept, spatial),
                                    name="Ferguson Hall"
```

1.3.6 XQL

XQL (*XML Query Language*) **[XQL]** è un linguaggio sviluppato nell'ambito dell'Information Retrieval. In tale ambito, i ricercatori si occupano di ricerche Full-Text, ad esempio effettuano interrogazioni su documenti strutturati o cercano di derivare multiple rappresentazioni da un singolo documento etc.. Illustriamo gli aspetti essenziali del linguaggio attraverso alcuni esempi. Comunque, come i linguaggi di interrogazione per basi di dati, anche XQL si basa su pattern e su condizioni, ma non prevede clausole per la costruzione dei risultati. XQL può applicare condizioni su elementi e attributi, così come su commenti ed entità. L'esecuzione dell'interrogazione restituisce sempre un albero XML ben-formato. Gli esempi vengono eseguiti sul documento `www.bn.com/bib.xml` che contiene dati di una bibliografia. Il primo esempio di interrogazione seleziona tutti i `<title>` dei `<book>` pubblicati da `Addison-Wesley` dopo il 1991. Inizialmente si assume che l'output di un'interrogazione non sia ordinato.

```
document ("http://www.bn.com")/bib {
      book[publisher/name = "Addison-Wesley" and @year>1991] {
            @year | title
      }
}
```

In questa interrogazione XQL, il cammino `(http://www.bn.com)/bib` seleziona tutti gli elementi `bib` dal documento in input e valuta l'espressione più interna per ognuno di tali elementi. L'espressione interna `book` seleziona gli elementi `book` che sono figli di un elemento `book` e che soddisfano la condizione racchiusa tra parentesi quadre. XQL, pur

non avendo una clausola esplicita, fa uso di un'espressione per la costruzione del risultato di un'interrogazione. In questo caso, il risultato è un elemento `bib` che contiene gli elementi `book` selezionati. L'espressione più interna permette di restituire l'attributo `year` del libro e l'elemento `title`.
In XQL il risultato di un'interrogazione ha necessariamente la struttura originale dei nodi del documento XML di input. La prossima interrogazione è simile alla precedente ma i risultati vengono raggruppati in base al `title`:

```
document ("http://www.bn.com")/bib->results {
      book->results {
            title | author
      }
}
```

Il raggruppamento in XQL è implicito, perché il risultato di un'interrogazione è sempre una proiezione del documento originale. Quindi, per ogni elemento `book`, l'esempio di sopra crea un elemento `results` usando l'operatore di ridenominazione `->`. I figli dell'elemento `results` sono tutti gli elementi `title` e `author` contenuti al suo interno.
Vediamo ora come con XQL si possono combinare informazioni provenienti da documenti XML differenti. Per questa interrogazione si suppone di avere un altro documento `www.amazon.com/reviews.xml` che contiene informazioni su riviste e prezzi. L'interrogazione restituisce tutti il `books` e i loro `price` da entrambi i documenti:

```
document ("www.bn.com/bib.xml")/books-with-prices {
      book->books-with-prices[$t:=title]{
            title | price->price-bn |
            document ("www.amazon.com/reviews.xml")/reviews
                                                /entry[title=$t] {
                  price->price-amazon
            }
      }
}
```

In XQL, la giunzione viene eseguita assegnando e usando variabili. Nell'esempio precedente l'assegnamento `$t:=title` lega la variabile `$t` ai titoli dei libri, e il predicato `title=$t` seleziona i titoli corrispondenti delle riviste.
Come già detto più volte gli elementi contenuti all'interno di un documento XML sono ordinati. In alcuni casi è importante mantenere questo ordine anche nei risultati di un'interrogazione, oppure imporne uno nuovo. L'esempio seguente illustra un'interrogazione che restituisce ogni book con i suoi `title` e il primo dei due `author`, e l'elemento `<et-al/>` se ci sono più di due autori:

```
document ("www.bn.com/bib.xml")/bib/book {
      title | author[1 to 2] | author[3]->et-al { }
}
```

XQL usa *subscript* per gestire l'ordine. Un *subscript* può contenere un numero, un intervallo, o una combinazione di essi. Ad esempio, l'espressione `author[1 to 2]` seleziona il primo di due `author`. Il terzo elemento `author` è ridenominato con l'elemeto vuoto `<et-al>`.
In alcune interrogazioni si possono voler specificare cammini attraverso l'albero XML, usando *regular path expressions*. XQL non supporta *regular path expressions* ma fa uso degli operatori / e // per raggiungere rispettivamente figli e discendenti di un nodo. La

seguente interrogazione seleziona il `title` dei `chapter` e delle `section` che contengono la stringa XML, ma non vincola l'elemento `section` ad essere contenuto nell'elemento `chapter`:

```
document ("books.xml")results {
      chapter[title contains "XML"] {title }
      ..//section[title contains "XML"] { title }
}
```

In conclusione si è visto come il linguaggio non supporta espliciti operatori di raggruppamento e di ordinamento. In più non è possibile in XQL effettuare interrogazioni senza conoscere a priori la struttura del documento, in quanto non supporta l'utilizzo delle variabili tag.

1.3.7 XQuery

Il World Wide Web Consortium ha riunito nell'ottobre del 1999 un gruppo di lavoro con lo scopo di progettare un linguaggio di interrogazione per sorgenti di dati XML. Questo nuovo linguaggio, chiamato XQuery, è ancora in evoluzione ed è stato presentato in diversi articoli pubblicati dal gruppo stesso che descrivono lo stato corrente del progetto. Tra questi, forse il più importante è **[XQuery 1.0]** che contiene una sintassi e una descrizione informale del linguaggio.
L'aggiornamento di questi documenti mette in evidenza come il linguaggio sia tuttora in evoluzione. Nonostante la continua evoluzione esistono implementazioni, anche open-source, del linguaggio. Tra queste segnaliamo Galax **[Galax]** e Qexo **[Qexo]**.
Il progetto di XQuery è stato soggetto a numerose influenze. Le più importanti si rifanno agli standard della W3C, quali Schema, XSLT, XPath e XML stesso **[W3C Req]**. XPath, in particolare è così importante che XQuery è stato definito come un suo sovrainsieme. Il progetto complessivo di XQuery, è comunque basato sul linguaggio QUILT **[Quilt]**.
Il progetto iniziale di XQuery è stato successivamente esteso con aspetti di Information Retrieval e in particolare per trattare ricerche 'Full-Text' (FTS). Le ricerche Full-Text forniscono un modo per interrogare un testo che viene visto come una sequenza di parole, separate da punteggiatura e da spazi. Questo formato del testo porta ad introdurre funzioni e operatori che lavorano sulle posizioni delle parole, ad esempio operatori di proximity.
Illustriamo ora il modello dei dati del linguaggio per poi proseguire con le sue funzionalità. Il modello dei dati di XQuery è basato sulla nozione di *sequenza*. Una *sequenza* è una collezione ordinata di zero o più *items*. Un *item* può essere un *nodo* o un *valore atomico*. Un *nodo* può rappresentare elementi, attributi, documenti, testo, etc. I documenti XML di input vengono trasformati in modo da rispettare tale modello dei dati, attraverso un processo chiamato *validation* che analizza il documento, lo valida secondo un particolare *Schema* e lo rappresenta attraverso una gerarchia di nodi e valori atomici etichettati con informazioni estratte dallo *Schema* stesso. A sua volta, il risultato di un'interrogazione può subire la trasformazione inversa, dal modello dei dati ad una visione XML, attraverso un processo chiamato *serialization*. Notiamo che il risultato di un'interrogazione non è sempre un documento XML ben-formato (può venire restituito ad esempio il solo valore 47 o sequenze di elementi disgiunte).
Per quanto riguarda le funzionalità, XQuery deve fornire per prima cosa il minimo insieme di ricerche Full-Text considerate importanti:

1. ricerche su singola parola;
2. ricerche su frasi;

3. ricerche per prefisso;
4. ricerche per suffisso;
5. ricerche per proximity;
6. ricerche per proximity specificando l'ordine;
7. ricerche in AND;
8. ricerche in OR;
9. ricerche in NOT;
10. ricerche che si basano su ranking e rilevanza.

Per la risoluzione di queste interrogazioni si prende spunto da altri linguaggi. Illustriamo le funzionalità del linguaggio indicando anche le somiglianze con altri linguaggi.
Come XML e XPath, XQuery è un linguaggio *case-sensitive*. È un linguaggio funzionale e in quanto tale, è costituito da espressioni che restituiscono valori e che non hanno effetti collaterali. Il più semplice tipo di espressione è *literal* utile a rappresentare un valore atomico. Valori atomici di tipi diversi da quelli primitivi possono essere creati a partire da costruttori. Un costruttore è una funzione che crea un valore di un particolare tipo da una stringa che contiene una rappresentazione lessicale del tipo voluto. Ad esempio per creare il valore del tipo `date` si scrive:

```
date ("2003-09-17")
```

Le espressioni in XQuery possono contenere variabili che rappresentano il valore ad esse legato. Un modo per legare una variabile è quello di usare il costrutto **`LET`**. Quest'ultimo lega una o più variabili valutando poi l'espressione contenuta nella clausola **`RETURN`**. L'esempio illustra l'uso del costrutto **`LET`** e restituisce la sequenza 1,2,3:

```
LET    $start := 1, $stop := 3
RETURN $start to $stop
```

Le *path expressions* in XQuery sono basate sulla sintassi di XPath.Una path expression in XPath consiste di una serie di passi separati da '/'. Il risultato di ogni passo è una sequenza di nodi. Le *path expressions* sono un meccanismo potente, ma hanno una limitazione, ossia possono selezionare solo nodi esistenti. Quindi un linguaggio di interrogazione completo deve poter costruire nuovi elementi e attributi specificando il loro contenuto e le loro relazioni. Questa funzionalità è fornita da XQuery grazie ad un tipo di espressione chiamata *element constructor*. Il tipo più semplice di *element constructor* per l'elemento da creare è dato direttamente dalla sintassi XML. Per esempio, l'espressione seguente costruisce un elemento chiamato `<highbid>` che contiene un attributo `status` e due elementi `<itemno>` e `<bid-amount>`:

```
<highbid status = "pending" >
<itemno>4871</itemno>
<bid-amount>250.00</bid-amount>
</highbid>
```

Tuttavia in alcuni casi è necessario creare un elemento o un attributo il cui valore deve essere il risultato della valutazione di un'espressione. In questo caso l'espressione deve essere racchiusa tra parentesi graffe ad indicare che l'espressione deve essere valutata.
In altri casi ancora, si vorrebbe costruire un elemento calcolando separatamente il nome e il contenuto. XQuery mette a disposizione il costruttore *computed element constructor* che consiste della parola chiave *Element* seguita da due espressioni, la prima calcola il nome dell'elemento e la seconda il contenuto:

```
Element
      { name ($e) }
      { $e/@* , data($e)*2 }
```

La stessa cosa vale per gli attributi, utilizzando il *computed attribute constructor* che calcola separatamente il nome e il valore di un nuovo attributo.
Un altro aspetto interessante di un linguaggio di interrogazione è quello di possedere funzioni iterative. XQuery permette di iterare su una sequenza di valori, legando una variabile ad ogni valore e valutando un'espressione per ognuna delle variabili legate. La forma più semplice di iterazione in XQuery consiste di una clausola **FOR** che crea la variabile e determina una sequenza di valori sui quali iterare, seguita da una clausola **RETURN** che contiene l'espressione da valutare per ogni legame effettuato. Facciamo un esempio:

```
FOR    $n in (2,3)
RETURN $n + 1
```

l'espressione iterativa restituisce `(3,4)`.
Abbiamo già visto diversi operatori aritmetici. Infatti XQuery fornisce gli usuali operatori `+, -, *, div e mod` così come le funzioni di aggregazione `sum, avg, count, max e min` che operano su sequenze di valori e restituiscono un risultato numerico.
Il comportamento di tutti gli altri operatori è quello classico, ma in presenza di sequenze vuote vediamo cosa succede. In XQuery, una sequenza vuota è usata per rappresentare informazione mancante o non conosciuta allo stesso modo di come il valore NULL è usato nelle basi di dati relazionali. Quindi, il `+, -, *, div e mod` sono ridefiniti in modo tale da restituire una sequenza vuota se qualche loro operando è una sequenza vuota.
Inoltre, XQuery mette a disposizione tre operatori che sono progettati specificatamente per combinare sequenze di nodi: `union`, `intersect`, `expect`. È importante ricordare che `intersect` ed `expect` non sono utili quando si combinano sequenze di nodi a partire da differenti documenti, poiché non è possibile che due nodi in documenti diversi possano avere la stessa identità.
È anche possibile far uso di espressioni condizionali. Un'espressione condizionale fornisce un modo per eseguire una delle due espressioni, in base al valore di una terza espressione. Il formato dell'espressione è:

```
IF..THEN..ELSE
```

In XQuery le tre clausole sono obbligatorie e l'espressione della clausola `IF` è racchiusa tra parentesi.
Per riassumere mostriamo un esempio di interrogazione che illustra la maggior parte degli aspetti di Xquery descritti finora; l'interrogazione genera un '`report`' che contiene lo stato dei `bid` per diversi `items`. Viene etichettato ogni `bid` con "`OK`", "`too small`" o "`too late`" e si racchiude il `report` in un elemento chiamato `<bid-status-report>`:

```
<bid-status-report>
      for $i in document ("items.xml")/*/item
      return
        <item>
            {
             $i/itemno,
             for $b in document ("bids.xml")/*/bid[itemno = $i/itemno]
             return
               <bid>
                  {
                   $b/bidder,
```

```
                    $b/bid-amount,
                    <status>
                         {
                          if ($b/bid-date > $i/end-date) then "too late"
                          else if ($b/bid-amount < $i/reserve-price)
                               then "too small"
                          else "OK"
                         }
                    </status>
                   }
               </bid>
            }
        </item>
</bid-status-report>
```

Infine XQuery può essere usato anche per definire nuove funzioni. Infatti il linguaggio ha alla base una libreria di funzioni, elencate nel documento **[Functions and Operators]**, ma permette anche agli utenti di definirne delle nuove. Lo scopo di tali definizioni è quello di semplificare la sintassi di un'interrogazione, ad esempio la seguente interrogazione illustra come un utente possa scrivere una funzione per fornire un valore di 'default' ad un elemento mancante. La funzione si chiama `defaulted` e prende in input due parametri: l'elemento nodo (possibilmente dimenticato) e il valore di default. Se l'elemento è presente e ha un valore diverso dal vuoto, la funzione restituisce il suo valore; ma se l'elemento non esiste o è vuoto, la funzione restituisce il valore di 'default':

```
define function defaulted
       (element? $e, anySimpleType $d)
       returns anySimpleType
       {
        if (empty($e)) then $d
        else if (empty($e/_)) then $d
        else data($e)
}
```

In questo modo la seguente interrogazione:

```
for $e in $emps
return
       <emp>
         {
           $e/name,
           <pay> {$e/salary + $e/commission
              + $e/bonus} </pay>
         }
       </emp>
sortby (pay)
```

si può riscrivere nel seguente modo dove `commission` o `bonus` mancanti o vuoti sono trattati come se avessero il valore zero

```
for $e in $emps
return
       <emp>
         {
           $e/name,
           <pay> { $e/salary
             + defaulted ($e/commission, 0)
             + defaulted ($e/bonus, 0) }
           </pay>
```

```
        }
      </emp>
sortby(pay)
```

Concludendo possiamo affermare che XQuery è ormai lo standard per ricerche di informazioni da documenti XML.

1.4 La nostra proposta: XCDEv2.0

XCDEv2.0 è una libreria per comprimere, indicizzare e interrogare documenti XML. È stata implementata sulla base della libreria XCDEv1.0 **[XCDEv1.0]** sviluppata da Andrea Mastroianni come lavoro di tesi nel 2002.
XCDEv1.0 è una libreria per la *compressione* e *l'indicizzazione* di documenti XML. Le sue caratteristiche fondamentali sono:

- *nativa*: si ha il controllo di tutte le sue componenti così da poter sperimentare diverse soluzioni al fine di ottenere la massima efficienza;

- *modulare*: progettata in modo da agevolare eventuali cambiamenti e consentire future espansioni. L'interfaccia della libreria è costituita da API che implementano le operazioni di accesso ai documenti e interagiscono con le applicazioni clienti;

- *efficiente*: minimizza l'occupazione in spazio attraverso l'uso di tecniche di compressione e consente accessi veloci ai documenti attraverso l'uso di strutture dati per l'indicizzazione efficiente degli stessi:

- orientata ad operare su *documenti testuali*: prevede una serie di strutture dati e di indici (come le *liste di proximity*) che lo rendono particolarmente indicato a lavorare con documenti con larghe parti linguistiche. Dalle prove effettuate comunque il sistema si è rivelato efficiente anche con documenti provenienti da basi di dati;

- *indipendente dalla DTD*: il sistema non richiede la presenza della DTD per gestire i documenti;

- utilizza algoritmi e strutture dati che rappresentano lo *stato dell'arte*. Essendo il sistema nativo, è stato possibile individuare e realizzare durante la sua costruzione le migliori soluzioni implementative, avvantaggiandosi del controllo e della conoscenza della globalità delle componenti del sistema;

- *flessibile*: l'utente può avere un elevato controllo sulle strutture dati da adoperare nella indicizzazione dei documenti XML. Egli infatti può decidere di inserire o meno determinate strutture (eliminando così la memorizzazione di dati non necessari) e configurarne il comportamento (ad esempio determinando i caratteri separatori) per ogni singolo documento;

- orientato alla gestione di *singoli documenti*: la granularità di XCDEv1.0 è il singolo documento, sul quale possono essere effettuate le interrogazioni. I documenti possono essere a loro volta organizzati in collezioni in maniera indipendente dal sistema;

- di *pubblico dominio* e *portabile*: XCDEv1.0 si appoggia ad alcune librerie con licenza pubblica GNU. Il codice è interamente scritto in C ed è stato implementato sotto Linux, ma senza l'utilizzo di chiamate vincolate al sistema operativo.

Una delle scelte fondamentali della libreria è di operare alla granularità del *singolo documento.* Questa scelta ha un significativo svantaggio e diversi vantaggi. E' chiaro che una granularità fine comporta che ricerche su collezioni di piccoli documenti impiegano più tempo. Tuttavia, indicizzare singoli file permette di implementare efficientemente indici distribuiti, di aggiornare banalmente l'indice, di fare attenzione alla eterogeneità dei documenti XML, e di personalizzare gli indici in accordo agli aspetti di ogni documento.

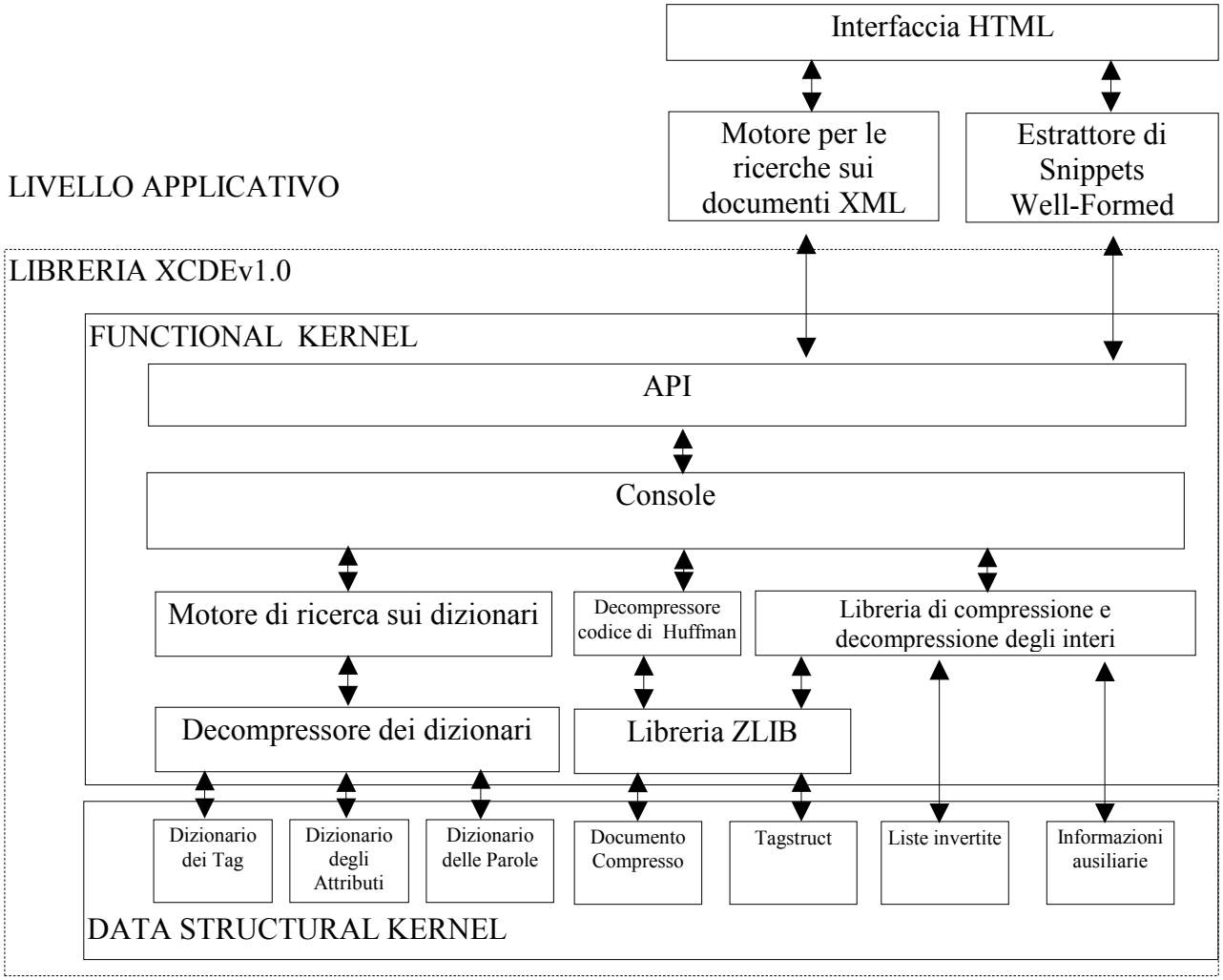


Figura 1.12 – Schema a livelli della libreria XCDEv1.0

La libreria XCDEv1.0 è realizzata a *livelli di astrazione*, di cui possiamo vedere una schematizzazione nella figura 1.12. Il livello più alto è quello *applicativo*, dove si trovano le applicazioni client che utilizzano la libreria. Lo schema contiene nel livello applicativo la struttura del motore *XCDEv1.0 Search Engine*.
L'interfaccia fra la libreria e il livello applicativo è costituita dalle *API*. Si tratta di funzioni C che effettuano varie operazioni sui livelli sottostanti e che possono essere raggruppate in maniera omogenea in base alle strutture dati a cui accedono. Tutte le API

utilizzano la *console*, che costituisce la *struttura dati di raccordo* fra le varie componenti della libreria. I moduli che si trovano fra la *console* e le strutture dati del *kernel* si occupano di effettuare principalmente delle trasformazioni sui dati in modo da renderli *accessibili al livello della console*. Esse riguardano soprattutto la loro decompressione, visto che, per minimizzare lo spazio occupato, tutte le strutture dati del *kernel* sono compresse. L'algoritmo di compressione varia in base alla singola struttura. Nel livello più basso troviamo il *kernel* delle strutture dati, il quale contiene le strutture adibite alla memorizzazione ed indicizzazione dei documenti XML. I *dizionari* contengono i diversi dati che costituiscono il documento (*tag*, parole ed attributi). Il meccanismo di indicizzazione utilizzato per questi elementi è quello delle liste invertite, implementate in modo da minimizzarne l'occupazione in spazio e consentirne un rapido accesso. Tali liste sono associate agli elementi dei dizionari e possono essere delle *liste di posizione* o di *proximity*. Le prime contengono la posizione dell'elemento all'interno del *documento compresso*, mentre le seconde sono associate solo alle parole e consentono di effettuare ricerche in base alla distanza fra esse. Le informazioni contenute nel dizionario dei *tag* e nelle *liste invertite* ad esso associate consentono di creare la *tagstruct,* una struttura dati concepita per indicizzare in modo compatto la struttura dei documenti XML, così da supportare efficienti interrogazioni di tipo *stabbing*: ossia quali sono i *tag* che contengono una posizione specificata nel documento? Infine la struttura delle *informazioni ausiliarie* è utilizzata per inizializzare la *console* e poter accedere a tutte le altre strutture dati. La creazione e l'accesso alle strutture dati sono orientate alla massima modularità: alcune delle strutture (come la *tagstruct* e le *liste di proximity)* possono anche non essere presenti, visto che all'utente è lasciata la possibilità di non costruirle. La libreria XCDEv1.0, rispetto ad altri sistemi, consente di ottimizzare lo spazio di memorizzazione delle informazioni garantendo al tempo stesso il supporto efficiente ad interrogazioni avanzate. Negli esperimenti condotti su varie collezioni di documenti XML è stato verificato che lo spazio occupato da *tutte le strutture dati del kernel* (*liste invertite*, *tagstruct, documento compresso, informazioni ausiliarie* e *dizionari*) è al più uguale a 1.2 volte la dimensione originale del documento indicizzato.
XCDEv1.0 fornisce anche un semplice linguaggio per interrogare i documenti XML da essa indicizzati. È opportuno osservare come questo linguaggio sia solo un prototipo realizzato al fine di effettuare ricerche sulla collezione di testi letterari fornita dal *CIBIT* **[CIBIT]** e per testare la libreria stessa. Le funzionalità di quest'ultima non sono pienamente utilizzate dal linguaggio, e inoltre esso consente di effettuare solo interrogazioni su *path*, ovvero interrogazioni che hanno una struttura con un *singolo cammino* contenente i *tag*.
Questi tipi di interrogazioni sono una semplificazione di quelle formulabili su XML mediante altri linguaggi, come ad esempio XQuery **[XQuery 1.0]**, ma rispondono alle richieste dell'utente umanista. Lo scopo del nostro progetto è stato quello di migliorare ed estendere le funzionalità offerte dalla libreria XCDEv1.0 attraverso un sofisticato linguaggio di interrogazione. Due erano le possibili direzioni di ricerca verso cui orientarci, la prima rivolta alle ricerche tipiche delle basi di dati e la seconda rivolta alle ricerche Full-Text (FTS). Influenzati dal documento **[W3C Full-Text]** redatto dalla W3C nel 2003, che estende le funzionalità di XQuery alle ricerche Full-Text, abbiamo optato per la seconda direzione. Inoltre dall'esigenza di effettuare interrogazioni su testi letterari, abbiamo sviluppato il nuovo linguaggio, XCDE Query Language, che combina un sottoinsieme delle proprietà di XQuery con alcuni aspetti propri dei linguaggi orientati all'Information Retrieval.
Le principali ricerche nell'ambito di tali linguaggi riguardano interrogazioni:

1. su una o più parole;

2. su parole per prefisso, per suffisso e per sottostringa;
3. basate su espressioni regolari;
4. su parole con errori;
5. per proximity su più parole.

Il modo più intuitivo per la risoluzione delle interrogazioni supportate dal linguaggio, sarebbe stato quello di utilizzare i meccanismi di ricerca su file testuali e le operazioni di trasformazione sulle stringhe. In realtà questo approccio ha il vantaggio di supportare tutte le tipologie di interrogazione orientate ai documenti puramente testuali (per parola, con prefisso, per errori), ma incontra diverse difficoltà computazionali nella gestione delle interrogazioni sulla struttura dei dati. Infatti essendo tale struttura non esplicitamente indicizzata, una ricerca sui tag e sugli attributi richiede delle operazioni di *postprocessing* che possono risultare computazionalmente molto costose in tempo, considerando anche che i documenti XML in ambito letterario hanno dimensioni rilevanti. Inoltre, non verrebbe sfruttata la proprietà interessante di XML di descrivere la "semantica" delle informazioni contenute in un documento in modo strutturato, documentato e facilmente leggibile da una macchina. La soluzione che ci ha permesso di superare queste limitazioni è stata l'utilizzo delle tecniche innovative per la memorizzazione e l'indicizzazione dei documenti XML offerte dal *kernel* XCDEv1.0.
Questa tesi, quindi, è una proposta di implementare quel sottoinsieme di XQuery orientato all'Information Retrieval mediante un motore di ricerca XML-nativo e basato su tecniche avanzate di indicizzazione e compressione.
Avendo deciso di sfruttare la libreria XCDEv1.0, è stato possibile implementare un ampio sottoinsieme delle funzionalità descritte nell'articolo **[W3C Full-Text]** e riportate nella tabella seguente in cui a sinistra è indicata la funzionalità dell'interrogazione e a destra una sua descrizione.

Funzionalità	Descrizione
Interrogazioni su una o più parole	Si vuole cercare una parola o un gruppo di parole nel documento. (par 2.2.1 e 2.2.2 di [**W3C Full-Text**])
Interrogazioni su una parola in un elemento	Si vuole cercare una parola all'interno di un particolare elemento. (par 3.2.1 di [**W3C Full-Text**])
Interrogazioni su più parole in un elemento	Si vogliono cercare più parole all'interno di uno stesso elemento. (par 3.2.2 di [**W3C Full-Text**])
Interrogazioni su un elemento e restiutiscono elementi differenti	Si vogliono porre delle condizioni su un particolare elemento e restituire elementi differenti. (par 3.2.5 di [**W3C Full-Text**])
Interrogazioni tra elementi adiacenti	Si vogliono cercare più parole anche se si trovano in due o più elementi diversi e adiacenti. (par 3.2.7 di [**W3C Full-Text**])
Interrogazioni su un elemento e i suoi discendenti	Si vuole cercare una parola in un elemento e nei suoi discendenti. (par 3.2.8 di [**W3C Full-Text**])
Interrogazioni sugli attributi	Si vuole cercare un elemento con più parole nell'attributo. (par 3.2.9 di [**W3C Full-Text**])

Interrogazioni su un elemento e i suo attributi	Si vuole cercare un elemento specificando le condizioni sugli attributi. (par 3.2.10 di [**W3C Full-Text**])
Interrogazioni sul formato di una parola	Si vuole cercare una parola solo con lettere minuscole, solo maiuscole o maiuscole e minuscole insieme o contenente caratteri speciali. (par 5.2.3 par 5.2.4 par 5.2.5 di [**W3C Full-Text**])
Interrogazioni per prefisso	Si vuole cercare una parola specificando il suo prefisso. (par 6.2.1 di [**W3C Full-Text**])
Interrogazioni per suffisso	Si vuole cercare una parola specificando il suo suffisso. (par 6.2.2 di [**W3C Full-Text**])
Interrogazioni per sottostringa	Si vuole cercare una parola specificando una sua sottostringa. (par 6.2.3 di [**W3C Full-Text**])
Interrogazioni Booleane	Si vogliono effettuare ricerche in OR, AND e NOT. (par 6.2.2 di [**W3C Full-Text**])
Interrogazioni per proximity	Si vogliono cercare due o più parole che si trovano ad una certa distanza. (par 10.2.1 di [**W3C Full-Text**])
Interrogazioni sulla struttura	Si vogliono effettuare ricerche su figli, genitori, antenati, discendenti e fratelli di un elemento (da par 13.2.1 a par 13.2.11 esclusi par 13.2.2 par 13.2.6 e par 13.2.10 di [**W3C Full-Text**])
Interrogazioni che creano nuovi elementi	Si vogliono effettuare interrogazioni inserendo nel risultato nuovi elementi definiti dall'utente. (par 15.2.1 di [**W3C Full-Text**])

Per incrementare la potenza espressiva di XCDE Query Language nell'ambito delle ricerche Full-Text, abbiamo previsto anche altri tipi di interrogazioni. Interrogazioni su parole con espressioni regolari, e soprattutto con errori (paragrafi [2.1.1.2] e [2.1.1.3]). Un altro tipo di interrogazione permette di porre condizioni sulla profondità di un elemento (dove con profondità si intende la distanza tra due nodi dell'albero XML associato al documento), o su tutti gli elementi che si trovano ad una certa profondità (paragrafo [2.1.2.3]). Per quanto riguarda la visualizzazione dei risultati, il linguaggio dà un'ampia scelta all'utente, che può settare parametri come la dimensione e la direzione dello snippet da visualizzare (paragrafo [2.1.7.1]), scegliere se visualizzare o meno i tag (paragrafo [2.1.7.2]), scegliere se visualizzare tutti i risultati trovati o solo un sottoinsieme di essi (paragrafo [2.1.8]).
Per rendere più efficienti le interrogazioni sulla profondità, il *kernel* XCDEv1.0 è stato ampliato con un'ulteriore struttura dati, la *depthstruct*. Tale struttura dati consente, dato un elemento del documento XML, di conoscere le profondità di tutte le occorrenze dell'elemento nell'albero strutturale del documento XML. Anche per tale struttura sono state utilizzate tecniche di compressione per ridurre lo spazio occupato.
Descriviamo i passi salienti dell'esecuzione di un'interrogazione nel linguaggio XCDE Query Language. Come prima cosa si ha bisogno di indicizzare il documento. Quesa operazione viene eseguita dal kernel XCDEv1.0 con il comando:

```
xcde_build [-v][-c CONFIGURATION_FILE_NAME] [-f] -t -d  filename
```

Il risultato è un insieme di file che contengono il documento compresso e le strutture dati per l'indicizzazione. Una novità, rispetto alla libreria XCDEv1.0 originale, è l'implementazione e creazione di un'altra struttura dati (con l'opzione –d) che memorizza la *profondità* di ogni elemento presente nell'albero XML. In questo modo, le interrogazioni che coinvolgono la *distanza* tra elementi discendenti vengono risolte efficientemente. Anche per tale struttura sono utilizzate tecniche di compressione per ridurre lo spazio occupato.
Dopo che gli indici sono stati costruiti, gli utenti possono eseguire le interrogazioni sul documento XML originale usando l'XCDE *query engine* che offre diverse funzionalità. Come abbiamo osservato precedentemente, XQuery è un potente linguaggio di interrogazione ma sembra che sia stato implementato pensando a documenti XML derivati da dati *strutturati*. Al contrario XML è text-based e i linguaggi di interrogazione dovrebbero essere orientati principalmente all'elaborazione e manipolazione di dati testuali. Di conseguenza il nostro obiettivo è stato quello di progettare un linguaggio semplice da usare. Per questo le interrogazioni sono formulate utilizzando un formalismo XML-based con la sintassi **`SELECT-FROM-RETURN-RANGE`**, che ricorda quella di SQL e di XML-QL. Ad esempio se si volesse trovare l'autore Boccaccio l'interrogazione si presenterebbe nella seguente forma.

```
SELECT  <autore xml_var = '$autore'>
                Boccaccio
        </autore>
FROM    esempio.xml
RETURN  $autore
```

L'interrogazione viene eseguita attraverso il comando:

```
xcde_search2 [-p] [-f Filename] query_expression
```

in cui le opzioni -p e -f possono cambiare il comportamento del *query engine*:

`-f Filename:` permette di leggere l'interrogazione da un file**.**

`-p:` indica al *query engine* che devono essere restituite le posizioni nel documento compresso degli elementi che soddisfano l'interrogazione e non gli snippet.

La clausola **`SELECT`** è costituita da un'espressione XML *ben formata.* In questo modo ogni utente con conoscenze elementari di XML è in grado di formulare l'interrogazione senza essere costretto ad apprendere la sintassi di XPath, o di XQuery. All'interno della clausola **`SELECT`** si possono specificare delle variabili attraverso un attributo speciale il cui nome è `xml_var` e il cui valore è composto dal simbolo speciale '$' seguito dal nome della variabile. In questo modo più elementi, nell'esempio solo l'elemento `autore`, possono essere marcati simultaneamente come interessanti senza usare combinazioni complicate di espressioni XPath. L'uso che si fa delle variabili è simile a quello di CXQuery.
L'output di un'interrogazione è costituito da uno o più snippet di cui è possibile gestire il formato attraverso la clausola **`RETURN`** che, nuovamente, include un'espressione XML *ben formata*. Lo strumento chiave per creare questo snippet sono le variabili presenti nella **`SELECT`**, e richiamate nell'espressione della clausola **`RETURN`** per indicare dove gli elementi marcati come interessanti, devono essere visualizzati. Un'altra particolarità del processo di estrazione dello snippet, che spiegheremo diffusamente nei prossimi capitoli, è la possibilità data all'utente di personalizzare il formato del risultato. In particolare si può

agire sulla dimensione dello snippet da estrarre, sulla direzione dello snippet, e sulla la presenza o no dei tag.
La sintassi dell'interrogazione prevede la presenza della clausola **`RANGE`**, opzionale, che permette di specificare l'intervallo di risultati da restituire all'utente.

Capitolo 2

XCDEv2.0: Query Language

Per presentare XCDE Query Language abbiamo deciso di percorrere la strada del "learning by examples". Per questo motivo la prima parte di questo capitolo è dedicata ad illustrare le funzionalità del linguaggio attraverso una carrellata di esempi su un documento ben-formato riportato completamente in Appendice C. Il linguaggio può essere usato per eseguire una varietà di interrogazioni, tipiche dell'Information Retrieval, tra le quali, ricerche su parole per prefisso, per suffisso, per sottostringa, ricerche per proximity tra più parole, ricerche per espressioni regolari o con errori, ricerche che restituiscono snippet per visualizzare il contesto in cui occorre l'interrogazione e molte altre.

2.1 Esempi di interrogazioni

La sintassi di un'interrogazione prevede le clausole **`SELECT`**, **`FROM`**, **`RETURN`** e **`RANGE`**. La clausola `SELECT` consiste di un'espressione XML *ben formata* per esprimere le condizioni della ricerca. Tale espressione si presta bene ad essere analizzata come se fosse un albero in quanto i documenti XML sono strutturati come alberi, con tutti gli elementi inseriti all'interno di una relazione padre/figlio e con solo un elemento radice. La clausola `FROM` contiene i nomi dei file sui quali effettuare la ricerca. Anche la clausola `RETURN`, come la `SELECT`, consiste di un'espressione XML per esprimere la formattazione dei risultati dell'interrogazione. Gli esempi che seguono illustrano interrogazioni significative che operano sul documento XML riportato in Appendice C. In ogni paragrafo seguente si specifica un sottoinsieme di interrogazioni che l'utente può effettuare sul documento in input, e il risultato aspettato per ogni interrogazione. Il formato generale del risultato è mostrato in figura 2.1. L'elemento `<xcde:results>` racchiude tutti i risultati di tutti i file presenti nella clausola `FROM` (in figura è stato specificato solo `example.xml`). L'elemento `<xcde:result>`, invece, racchiude tutti i risultati di un singolo file e a sua volta ogni singolo risultato è racchiuso tra l'elemento `<xcde:return>` al quale è associato l'attributo `result_number` per specificare il numero del risultato. Infine l'elemento `<xcde:return>` racchiude lo snippet richiesto dall'utente attraverso le variabili presenti nella clausola **`RETURN`**.

```
<xcde:results>
  <xcde:result>
    <xcde:filename>example.xml</xcde:filename>

    <xcde:return result_number = '1'>

       //snippet risultato per il primo risultato

    </xcde:return>
         .
         .
    <xcde:return result_number = '24'>

      //snippet risultato per il 24-esimo risultato

    </xcde:return>
  </xcde:result>
</xcde:results>
```

Figura 2.1 – Formato generale del risultato di un'interrogazione

Ogni interrogazione illustra una specifica e rilevante funzionalità del linguaggio, anche se queste possono essere combinate tra loro. Gli esempi vengono presentati seguendo lo schema del documento XQuery and XPath Full-Text Use Cases **[W3C Full-Text]** della W3C.
Tutte le interrogazioni presentate in questo capitolo sono eseguibili dal motore di ricerca XCDEv2.0 e possono essere trovate nel pacchetto di installazione della libreria.
Le interrogazioni possono riguardare parole, insiemi di parole, tag e attributi. Una parola consiste di uno o più caratteri consecutivi. Da notare che parole consecutive all'interno di un elemento TAG nell'interrogazione non possono essere separate da punteggiatura. Sono previsti anche operatori per la proximity.
Vengono mostrate:

- Interrogazioni che includono operatori booleani (e.s., and, or e not);
- Interrogazioni sui valori degli attributi;
- Interrogazioni per espressioni regolari su parole;
- Interrogazioni con errori su parole;

E vengono presentate nel seguente formato:

`Interrogazione`

- `Operandi: parole, parti di parole, attributi`
- `Commenti: commenti sul comportamento generale della query`

`Risultato atteso:`

```
…risultati…
```

2.1.1 Interrogazioni su parole

Iniziamo illustrando le interrogazioni più semplici, quelle su singole parole o su insiemi di parole. Il linguaggio però offre la possibilità di operare ricerche complesse che andremo ad esporre nei sottoparagrafi a seguire.

2.1.1.1 Esatta, per Prefisso, per Suffisso, per Sottostringa

L'interrogazione cerca una singola parola nel documento.

L'esempio seguente trova tutte le occorrenze delle parola "Usability" nel documento.

```
SELECT  <xml_exact xml_var = '$parola' xml_case = 'sensitive'>
                Usability
        </xml_exact>
FROM    esempio.xml
RETURN  <mytag> $parola </mytag>
```

- Operandi: "Usability"
- Commenti: La clausola SELECT genera l'albero XML in cui la variabile $parola viene legata al nodo etichettato 'Usability'. La clausola RETURN usa le occorrenze esatte di 'Usability' associate a $parola per generare la lista dei risultati. Nell'esempio la ricerca è *case-sensitive*, nel caso in cui l'attributo non sia presente la ricerca è considerata *case-insensitive*.

Risultato atteso:

```
<xcde:results>
      <xcde:result>
      <xcde:filename>esempio.xml</xcde:filename>

      <xcde:return result_number = '1'>
              <mytag> Usability </mytag>
      </xcde:return>
      .
      .
      .
      <xcde:return result_number = '15'>
              <mytag> Usability </mytag>
      </xcde:return>
      </xcde:result>
</xcde:results>
```

Il tipo di ricerca sulla stringa viene specificato da un tag speciale, nell'esempio `xml_exact` (esatta). Altri tipi di ricerche, per prefisso, suffisso e sottostringa possono essere effettuate sostituendo ad `xml_exact` rispettivamente i tag speciali `xml_prefix`, `xml_suffix`, `xml_contained`.

2.1.1.2 Per espressioni regolari

L'interrogazione cerca una singola parola nel documento.

L'esempio seguente trova tutte le parole "testing" o "testong".

```
SELECT  <xml_regexp xml_var = '$parola'>test[i|o]ng</xml_regexp>
FROM    esempio.xml
RETURN  $parola
```

- Operandi: "testing", "testong"
- Commenti: La clausola SELECT genera l'albero XML in cui la variabile $parola viene legata al nodo etichettato 'test[i|o]ng'. La sintassi per le espressioni regolari è quella usata dal comando *grep* **[Grep]**. La clausola RETURN usa i valori risultanti di $parola per generare la lista delle parole risultanti.

Risultato atteso:

```
<xcde:results>
      <xcde:result>
      <xcde:filename>esempio.xml</xcde:filename>

      <xcde:return result_number = '1'>
            Testing
      </xcde:return>
      .
      .
      .
      <xcde:return result_number = '17'>
            Testing
      </xcde:return>
      </xcde:result>
</xcde:results>
```

2.1.1.3 Con errori

L'interrogazione cerca una singola parola nel documento.

L'esempio seguente trova tutte le parole "ksabimity" con al più due errori.

```
SELECT  <xml_error xml_maxerr = '2' xml_var = '$parola'>
              ksabimity
        </xml_error>
FROM    esempio.xml
RETURN  $parola
```

- Operandi: "ksabimity"
- Commenti: La clausola SELECT genera l'albero XML in cui la variabile $parola viene legata al nodo etichettato 'ksabimity'. L'attributo xml_maxerr specifica il numero massimo di caratteri che è possibile cambiare, eliminare o aggiungere all'interno della parola. La clausola RETURN usa i valori risultanti di $parola per generare la lista delle parole risultanti.

Risultato atteso:

```
<xcde:results>
      <xcde:result>
      <xcde:filename>esempio.xml</xcde:filename>

      <xcde:return result_number = '1'>
            Usability
      </xcde:return>
      .
      .
      .
      <xcde:return result_number = '24'>
            usability
      </xcde:return>
      </xcde:result>
</xcde:results>
```

2.1.2 Interrogazioni su elementi

In questo paragrafo illustriamo interrogazioni che cercano uno o più elementi XML. Il linguaggio offre la possibilità di operare ricerche di vario genere che andremo ad esporre nei sottoparagrafi a seguire. Alcune interrogano un elemento e i suoi discendenti.

2.1.2.1 Interrogazioni sul contenuto di un elemento

L'esempio seguente trova tutti i `<book>` che hanno come discendente l'elemento `<content>`, che a sua volta contiene le parole 'User' e 'profiling'.

```
SELECT <book xml_var = '$libro'>
                  <content>User profiling</content>
            </book>
FROM esempio.xml
RETURN $libro
```

- Operandi: "User", "profiling"
- Commenti: La clausola SELECT genera l'albero XML in cui la variabile $libro viene legata al nodo etichettato 'book'. La clausola RETURN usa il sottoalbero con radice nel nodo associato a $libro per generare la lista dei risultati. Per le parole valgono le regole illustrate nel paragrafo 2.1.1

Risultato atteso:

```
<xcde:results>
      <xcde:result>
      <xcde:filename>esempio.xml</xcde:filename>

<book number="3">
   …..
   <content>
      …..
      <part number="1"> <container type="box">1-12</container>
         …..
         <component><container type="box">1</container>
           <componentTitle>Computers</componentTitle>
           <subComponent>
```

```
            <componentTitle>Software,
            <componentDate normalize="1946/1947">1946-1947
            </componentDate>
            </componentTitle>
         </subComponent>
         <subComponent>
            <componentTitle>Human Computer Interaction
            research, <componentDate normalize="1945/1952">
            1945-1952</componentDate>
            </componentTitle>
            <subsubComponent>
               <componentTitle>Flow diagram,
               <componentDate normalize="1950">1950
               </componentDate>
               </componentTitle>
            </subsubComponent>
            <subsubComponent>
               <componentTitle>General,
               <componentDate normalize="1947/1951">1947-1951
               </componentDate>
               </componentTitle>
            </subsubComponent>
            <subsubComponent><container type="box">2</container>
               <componentTitle>Eye Movement research,
               <componentDate normalize="1949/1950">1949-1950
               </componentDate>
               </componentTitle>
            </subsubComponent>
            <subsubComponent>
               <componentTitle>User profiling,
               <componentDate normalize="1950/1959">1950s
               </componentDate>
               </componentTitle>
            </subsubComponent>
          </subComponent>
      </component>
      <component>
        <componentTitle>Web User Appreciation Award,
        <componentDate normalize="1956">1956</componentDate>
        </componentTitle>
      </component>
   </part>
   …..
</content>
</book>

   </xcde:result>
</xcde:results>
```

2.1.2.2 Interrogazioni sul valore degli attributi associati ad un elemento

L'interrogazione cerca gli elementi con l'attributo specificato.

L'esempio seguente trova tutti i `<title>` che hanno un attributo 'shortTitle' il cui valore è la frase 'Usabilityguy Manuscript Guide'.

```
SELECT
     <title xml_var = '$titolo' shortTitle = 'Usabilityguy Manuscript Guide'>
     </title>
```

```
FROM esempio.xml
RETURN $titolo
```

- Operandi: "Usabilityguy Manuscript Guide"
- Commenti: La clausola SELECT genera l'albero XML in cui la variabile $titolo viene legata al nodo etichettato 'title'. La clausola RETURN usa il sottoalbero con radice nel nodo associato a $titolo per generare la lista dei risultati trovati.

Risultato atteso:

```
<xcde:results>

<xcde:result>
<xcde:filename>esempio.xml</xcde:filename>

<xcde:return result_number = '1'>
<title shortTitle="Usabilityguy Manuscript Guide">
      John Wesley Usabilityguy: A Register of His
      Papers</title>
</xcde:return>

</xcde:result>

</xcde:results>
```

Possono essere effettuati altri tipi di ricerca sul valore degli attributi associati ai tag. Il tipo di ricerca va specificato all'interno del valore stesso da ricercare come mostrato nella tabella di figura 2.2.

'exi:Usability Basics' 'exs:Usability Basics'	Ricerca esatta su 'Usability Basics' rispettivamente *case-insensitive* o *case-sensitive*.
'prefi:Usability' 'prefs:Usability'	Ricerca per prefisso su 'Usability' rispettivamente *case-insensitive* o *case-sensitive*.
'sufi:Basics' 'sufs:Basics'	Ricerca per suffisso su 'Basics' rispettivamente *case-insensitive* o *case-sensitive*.
'conti:Basics' 'conts:Basics'	Ricerca per sottostringa su 'Basics' rispettivamente *case-insensitive* o *case-sensitive*.

Figura 2.2 Tipi di ricerca sul valore degli attributi

Ad esempio se volessimo effettuare una ricerca sul valore di un attributo per prefisso e *case-sensitive* la clausola SELECT si trasformerebbe nel seguente modo:

```
SELECT
     <title xml_var = '$titolo' shortTitle = 'prefs:Usabilityguy Manuscript'>
     </title>
```

2.1.2.3 Interrogazioni sugli elementi e le loro profondità

L'esempio seguente trova tutti i <book> che hanno come nipote l'elemento <title> e restituisce il contenuto dell'elemento <title>.

```
SELECT <book>
            <title xml_var = '$titolo' xml_dist = '2'></title>
       </book>
FROM   esempio.xml
RETURN $titolo
```

- Operandi: Nessuno
- Commenti: La clausola SELECT genera l'albero XML in cui la variabile $titolo viene legata al nodo etichettato 'title'. La clausola RETURN usa il sottoalbero con radice nel nodo associato a $titolo per generare la lista dei risultati trovati.

Risultato atteso:

```
<xcde:results>

<xcde:result>
<xcde:filename>esempio.xml</xcde:filename>

<xcde:return result_number = '1'>
<title shortTitle="Improving Web Site Usability">Improving
      the Usability of a Web Site Through Expert Reviews and
      Usability Testing</title>
</xcde:return>

<xcde:return result_number = '2'>
<title shortTitle="Usability Basics">Usability
      Basics: How to Plan for and Conduct Usability Tests
      on Web Site Thereby Improving the Usability of Your
      Web Site</title>
</xcde:return>

<xcde:return result_number = '3'>
<title shortTitle="Usabilityguy Manuscript Guide">
      John Wesley Usabilityguy: A Register of His
      Papers</title>
</xcde:return>

</xcde:result>

</xcde:results>
```

2.1.3 Interrogazioni sui valori degli attributi senza specificare l'elemento

Questi esempi illustrano interrogazioni sul valore degli attributi. Anche in questo caso è possibile specificare diversi tipi di ricerca sul valore dell'attributo.

2.1.3.1 Esatta, per prefisso, per suffisso, per sottostringa

L'interrogazione cerca gli elementi con l'attributo specificato permettendo di non conoscere a priori l'elemento.

L'esempio trova tutti gli elementi con il valore 'Usability Basic' nell'attributo 'shortTitle'.

```
SELECT  <xml_anyvalue shortTitle = 'Usability Basics' xml_var = '$tag'>
        </xml_anyvalue>
FROM    esempio.xml
RETURN  $tag
```

- Operandi: "Usability Basics"

- Commenti: La clausola SELECT genera l'albero XML in cui la variabile $tag viene legata al nodo speciale etichettato 'xml_anyvalue' che contiene gli elementi che soddisfano la query. Nell'esempio la ricerca sul valore dell'attributo è esatta e *case-insensitive* per default. La clausola RETURN usa il sottoalbero con radice nel nodo associato a $tag per generare la lista degli elementi risultanti.

Risultato atteso:

```
<xcde:results>

<xcde:result>
<xcde:filename>esempio.xml</xcde:filename>

<xcde:return result_number = '1'>
<title shortTitle="Usability Basics">Usability
      Basics: How to Plan for and Conduct Usability Tests
      on Web Site Thereby Improving the Usability of Your
      Web Site</title>
</xcde:return>

</xcde:result>

</xcde:results>
```

Possono essere effettuati altri tipi di ricerca sul valore degli attributi. Il tipo di ricerca va specificato all'interno del valore stesso da ricercare come mostrato nella tabella di figura 2.2.

Ad esempio se si volesse effettuare una ricerca sul valore per suffisso e *case-insensitive* la clausola **SELECT** si trasformerebbe così:

```
SELECT  <xml_anyvalue shortTitle = 'sufi:basics' xml_var = '$tag'>
        </xml_anyvalue>
```

2.1.3.2 Con errori

L'interrogazione cerca gli elementi con l'attributo specificato permettendo di non conoscere a priori l'elemento che contiene l'attributo.

L'esempio trova tutti gli elementi con l'attributo 'type' e valore 'bax' con al più 1 errore.

```
SELECT  <xml_anyvalue type = 'err:1:bax' xml_var = '$tag'>
        </xml_anyvalue>
FROM    esempio.xml
RETURN  $tag
```

- Operandi: "bax"
- Commenti: La clausola SELECT genera l'albero XML in cui la variabile $tag viene legata al nodo speciale etichettato xml_anyvalue che contiene gli elementi che soddisfano la query. La stringa 'bax' può contenere al più un errore. La ricerca è considerata in tutti i casi *case-sensitive*. La clausola RETURN usa il sottoalbero con radice nel nodo associato a $tag per generare la lista degli elementi risultanti.

Risultato atteso:

```
<xcde:results>

<xcde:result>
<xcde:filename>esempio.xml</xcde:filename>

<xcde:return result_number = '1'>
<container type="box">1-12</container>
</xcde:return>

<xcde:return result_number = '2'>
<container type="box">1</container>
</xcde:return>

<xcde:return result_number = '3'>
<container type="box">2</container>
</xcde:return>

<xcde:return result_number = '4'>
<container type="box">3-5</container>
</xcde:return>

<xcde:return result_number = '5'>
<container type= »box »>5</container>
</xcde:return>

</xcde:result>

</xcde:results>
```

2.1.4 Interrogazioni per proximity

L'interrogazione cerca gli elementi che contengono più parole con una proximity massima specificata. Per *proximity* si intende una forma di distanza logica fra le parole che consente di escludere dalla distanza gli elementi della struttura del documento e di considerare solo le parole.

L'esempio trova tutti i libri che contengono le parole 'The', 'usability' e 'site' con proximity 5, ovvero entro una finestra testuale di al massimo 5 parole.

```
SELECT  <book xml_var = '$libro'>
            <xml_proximity xml_maxprox= '5'>
                  The usability site
            </xml_proximity>
        </book>
FROM    esempio.xml
RETURN  $libro
```

- Operandi: "The usability site"

- Commenti: La clausola SELECT genera l'albero XML in cui la variabile $libro viene legata al nodo etichettato 'book' che contiene gli elementi che soddisfano la query. L'attributo xml_maxprox specifica la distanza massima tra la prima e l'ultima parola dell'occorrenza trovata. Nell'esempio un'occorrenza è valida se tra la prima e l'ultima delle tre parole ci sono meno di 5 parole compresa l'altra parola da ricercare. Per le parole valgono le regole illustrate nel paragrafo 2.1.1. La clausola RETURN usa il sottoalbero con radice nel nodo associato a $libro per generare la lista degli elementi risultanti.

Risultato atteso:

```
<xcde:results>

<xcde:result>
<xcde:filename>esempio.xml</xcde:filename>

<xcde:return result_number = '1'>
<book number="1">
   <metadata>
      <title shortTitle="Improving Web Site Usability">Improving
      the Usability of a Web Site Through Expert Reviews and
      Usability Testing</title>
      …..
   </metadata>
   <content>
      <introduction>
         <author>Elina Rose</author>
         <p>The usability of a Web site is how well the
         site supports the user in achieving specified
         goals. A Web site should facilitate learning,
         and enable efficient and effective task
         completion, while propagating few errors.
         Satisfaction with the site is also important.
         The user must not only be well-served, but must
         feel well-served.</p>
         …..
      </introduction>
      …..
   </content>
</book>
</xcde:return>

<xcde:return result_number = '2'>
<book number="2">
   <metadata>
      <title shortTitle="Usability Basics">Usability
      Basics: How to Plan for and Conduct Usability Tests
      on Web Site Thereby Improving the Usability of Your
```

```
      Web Site</title>
      …..
   </metadata>
   …..
</book>
</xcde:return>

</xcde:result>

</xcde:results>
```

2.1.5 Interrogazioni con operatori booleani

Le interrogazioni che seguono includono operatori booleani quali `AND`, `OR` e `NOT` dando la possibilità di interrogare documenti senza dover conoscere a priori la loro struttura o i loro elementi.

2.1.5.1 AND

L'operatore AND non esiste esplicitamente come gli altri operatori in quanto non è previsto un tag speciale per esprimerlo. Non per questo è negata all'utente la possibilità di esprimere condizioni in AND. Infatti, se non è stato specificato alcun operatore tra due o più condizioni queste vengono considerate in AND.

2.1.5.2 OR

L'interrogazione cerca una o entrambe le parole all'interno di un elemento e dei suoi discendenti.

L'esempio trova tutti i `<title>` dei `<book>`, con le parole 'web' o 'software' o 'internet' nell'elemento `<content>`.

```
SELECT  <book>
            <title xml_var = '$titolo'></title>
            <content xml_var = '$content'>
                  <xml_or>web software internet</xml_or>
            </content>
        </book>
FROM    esempio.xml
RETURN  $titolo $content
```

- Operandi: "web" "software" "internet"
- Commenti: La clausola SELECT genera l'albero XML in cui la variabile $titolo viene legata al nodo etichettato 'title' e la variabile $content viene legata al nodo etichettato 'content'. L'elemento <title> è discendete diretto dell'elemento <book>. Per le parole valgono le regole illustrate nel paragrafo 2.1.1.La clausola RETURN usa il sottoalbero con radice nel nodo associato a $titolo e $content per generare la lista degli elementi risultanti.

Risultato atteso:

```
<xcde:results>

<xcde:result>
<xcde:filename>esempio.xml</xcde:filename>

<xcde:return result_number = '1'>
<title shortTitle="Improving Web Site Usability">Improving
      the Usability of a Web Site Through Expert Reviews and
      Usability Testing</title>
<content>
      <introduction>
         <author>Elina Rose</author>
         <p>The usability of a Web site is how well the
         site supports the user in achieving specified
         goals. A Web site should facilitate learning,
         and enable efficient and effective task
         completion, while propagating few errors.
         Satisfaction with the site is also important.
         The user must not only be well-served, but must
         feel well-served.</p>
        …..
         <p>This book has been approved by the Web Site
         Users Association.</p>
      </introduction>
      <part number="1">
         <title>Expert Reviews</title>
         <introduction>
            <p>Expert reviewers identify problems
            and recommend changes to web sites based
            on research in human computer interaction
            and their experience in the field.</p>
            …..
            <p>Expert review methods should be
            initiated early in the development process,
            as soon as paper <b>p</b>rototypes
            (hand-drawn pictures of Web pages) or
            <b>w</b>ireframes (electronic mockups) are
            available. They should be conducted using
            the hardware and software similar to that
            employed by users.</p>
         </introduction>
         <chapter>
            <title>Heuristic Evaluation</title>
            <p>Expert reviewers critique an interface to
            determine conformance with recognized
            usability principles. <footnote>One of the
            best known lists of heuristics is <citation>
            Ten Usability Heuristics by Jacob Nielson</citation>.
            Another is <citation> Research-Based Web
            Design and Usability Guidelines</citation>
            </footnote></p>
         </chapter>
         <chapter>
            <title>Cognitive Walk-Through</title>
            <p>Expert reviewers evaluate Web site
            understandability and ease of learning while
            performing specified tasks. They walk through
            the site answering questions such as "Would a
            user know by looking at the screen how to
            complete the first step of the task?" and "If
            the user completed the first step, would the
            user know what to do next?," with the goal of
```

```
            identifying any obstacles to completing the
            task and assessing whether the user would
            cognitively be aware that he was successful in
            completing a step in the process.</p>
         </chapter>
      </part>
      …..
   </content>
</xcde:return>

<xcde:return result_number = '2'>
<title shortTitle="Usability Basics">Usability
      Basics: How to Plan for and Conduct Usability Tests
      on Web Site Thereby Improving the Usability of Your
      Web Site</title>
      <content>
      <introduction>
         <p>This is a basic handbook for planning and
         conducting usability tests on Web sites. Usability
         testing should be used in conjunction with other
         expert review methods.</p>
         <p>This book has not been approved by the Web Site
         Users Association.</p>
      </introduction>
      …..
      <part number="2">
         <chapter>
            …..
            <p>Give the user the script, then assure them
            that you are testing the Web site, not them.
            Users are asked to verbalize their thoughts as
            they complete the tasks. The event is recorded
            or someone takes notes. It is often preferable
            to have two testers, <footnote>Usability
            testing can be done at great expense or on a
            shoe string, using <testingProcedure>in-house
            expertise</testingProcedure> or
            <testingProcedure>contracting with human
            computer interaction professionals
            </testingProcedure>.</footnote> one to ask the
            questions, another to take notes. Testers should
            offer no guidance or comments to the user. Mouse
            movements, typing, expressions, and the user's
            words should be recorded.</p>
         </chapter>
         <chapter>
            <title>Evaluating and Implementing Results</title>
            <p>Compile the results and review collectively.
            Make changes to the site to alleviate the problems
            found in Web site components which were propagating
            the largest number of or the most devastating errors.
            Begin new iterations of testing and changes, until
            users are successful in the accomplishing the
            tasks.</p>
         </chapter>
      </part>
   </content>
</xcde:return>

<xcde:return result_number = '3'>
   …..
   <content>
      <introduction>
```

```
        <p>The papers of John Wesley Usabilityguy span the
        years 1946-2001, with the bulk of the items
        concentrated in the period from 1985 to 2001. The
        papers feature his career as a developer of software
        applications and usability specialist. The collection
        consists of correspondence, memoranda, journals,
        speeches, article drafts, book drafts, notes, charts,
        graphs, family papers, clippings, printed matter,
        photographs, r&egrave;sum&egrave;s and other materials.</p>
      </introduction>
      <part number="1">
           …..
           <subComponent>
              <componentTitle>Software,
              <componentDate normalize="1946/1947">1946-1947
              </componentDate>
              </componentTitle>
           </subComponent>
           …..
           <componentTitle>Web User Appreciation Award,
           <componentDate normalize="1956">1956</componentDate>
           </componentTitle>
           …..
      </part>
      …..
   </content>
</xcde:return>

</xcde:result>

</xcde:results>
```

2.1.5.3 NOT

L'interrogazione cerca i titoli che non contengono più parole all'interno di un elemento.

L'esempio trova tutti i <book> che non contengono le due parole 'usability' e 'testing' consecutive ma contengono un elemento <title> discendente di <book>.

```
SELECT  <book>
           <title xml_var = '$titolo'></title>
           <xml_not><xml_proximity xml_maxprox= '1'>
                usability testing
           </xml_proximity></xml_not>
        </book>
FROM    esempio.xml
RETURN  $titolo
```

- Operandi: "usability testing"
- Commenti: La clausola SELECT genera l'albero XML in cui la variabile $titolo viene legata al nodo etichettato 'title' che contiene gli elementi che soddisfano la query. Per le parole valgono le regole illustrate nel paragrafo 2.1.1. La clausola RETURN usa il sottoalbero con radice nel nodo associato a $titolo per generare la lista degli elementi risultanti. A differenza degli operandi AND e OR questo richiede un solo operando.

Risultato atteso:

```
<xcde:results>

<xcde:result>
<xcde:filename>esempio.xml</xcde:filename>

        <xcde:warning> no result found for this file </xcde:warning>

</xcde:result>

</xcde:results>
```

2.1.6 Altri esempi

Gli esempi seguenti illustrano interrogazioni necessarie per mostrare altre ricerche testuali significative. Alcune interrogazioni restituiscono elementi aggiuntivi o diversi dagli elementi interrogati. Altre interrogano parole ignorando la presenza di elementi intermedi.

2.1.6.1 Interrogazioni su elementi con restituzione di elementi differenti

L'interrogazione cerca parole all'interno di un elemento e restituisce elementi differenti presenti all'interno dello stesso documento.

L'esempio trova tutti i libri che contengono le parole 'usability' e 'testing' nell'elemento `<subject>`.

```
SELECT  <book>
            <subject>Usability testing</subject>
            <xml_or>
                  <metadata>
                        <title xml_var = '$titolo' xml_dist = '1'>
                        </title>
                  </metadata>
                  <authors xml_var = '$autore'></authors>
                  <dateRevised xml_var = '$dataRev'></dateRevised>
            </xml_or>
        </book>
FROM    esempio.xml
RETURN  $titolo $autore $dataRev
```

- Operandi: "Usability testing"

- Commenti: La clausola SELECT genera l'albero XML in cui la variabile $titolo viene legata al nodo etichettato 'title', la variabile $autore viene legata al nodo etichettato 'authors', la variabile $dataRev viene legata al nodo etichettato 'dateRevised'. La ricerca è considerata *case-insensitive*. Un vincolo dell'interrogazione prevede che almeno un elemento tra <title>, <authors> e <dateRevised> deve essere presente all'interno dell'elemento <book>. L'attributo xml_dist (vedi paragrafo 2.1.2.3) specifica la distanza massima tra l'elemento che lo contiene e il suo antenato nel testo. Per le parole valgono le regole illustrate nel paragrafo 2.1.1. La clausola RETURN usa il

sottoalbero con radice nel nodo associato a $titolo, $autore e $dataRev per generare la lista degli elementi risultanti.

Risultato atteso:

```
<xcde:results>

<xcde:result>
<xcde:filename>esempio.xml</xcde:filename>

<xcde:return result_number = '1'>
      <title shortTitle="Improving Web Site Usability">Improving
      the Usability of a Web Site Through Expert Reviews and
      Usability Testing</title>
      <authors>
         <author>Millicent Marigold</author>
         <author>Montana Marigold</author>
      </authors>
      <dateRevised>2002</dateRevised>
</xcde:return>

<xcde:return result_number = '2'>
      <title shortTitle="Usability Basics">Usability
      Basics: How to Plan for and Conduct Usability Tests
      on Web Site Thereby Improving the Usability of Your
      Web Site</title>
      <dateRevised>2001</dateRevised>
</xcde:return>

<xcde:return result_number = '3'>
      <title shortTitle="Usabilityguy Manuscript Guide">
      John Wesley Usabilityguy: A Register of His
      Papers</title>
      <authors>
         <author>Millicent Marigold</author>
         <author>Morty Marigold</author>
      </authors>
      <dateRevised>2002</dateRevised>
</xcde:return>

</xcde:result>

</xcde:results>
```

2.1.6.2 Interrogazioni su parole ignorando gli elementi

L'interrogazione cerca una sequenza di parole anche se inizia all'interno di un elemento e termina in un altro elemento.

L'esempio trova tutti i <book> con la frase 'usability testing once the problems'.

```
SELECT  <book>
           <title xml_var = '$titolo' xml_dist = '2'></title>
           <content xml_var = '$content'>
                 <xml_proximity xml_maxprox = '4'>
                       usability testing once the problems
                 </xml_proximity>
           </content>
```

```
        </book>
FROM    esempio.xml
RETURN  $titolo $content
```

- Operandi: "usability testing once the problems"

- Commenti: La clausola SELECT genera l'albero XML in cui la variabile $titolo viene legata al nodo etichettato 'title', la variabile $content viene legata al nodo etichettato 'content'. La ricerca è considerata *case-insensitive*. La proximity tra le parole è 4 e deve esistere un elemento <title> al più a distanza 2 dall'elemento <book>. Per le parole valgono le regole illustrate nel paragrafo 2.1.1. La clausola RETURN usa il sottoalbero con radice nel nodo associato a $titolo, $content per generare la lista degli elementi risultanti.

Risultato atteso:

```
<xcde:results>

<xcde:result>
<xcde:filename>esempio.xml</xcde:filename>

<xcde:return result_number = '1'>
      <title shortTitle="Improving Web Site Usability">Improving
      the Usability of a Web Site Through Expert Reviews and
      Usability Testing</title>
      <content>
            …..
            <title>Usability Testing</title>
            <p>Once the problems identified by expert
            reviews have been corrected, it is time to
            conduct some tests of the site with your unique
            audience or audiences by conducting usability
            testing.</p>
            …..
   </content>
</xcde:return>

</xcde:result>

</xcde:results>
```

2.1.6.3 Interrogazioni su entità

L'interrogazione cerca una entità nel documento.

L'esempio trova tutte le parole contenenti l'entità 'è'.

```
SELECT  <xml_contained xml_var = '$parola'>
            &egrave;
        </xml_contained>
FROM    esempio.xml
RETURN  $parola
```

- Operandi: "è"

- Commenti: La clausola SELECT genera l'albero XML in cui la variabile $parola viene legata al nodo etichettato 'è'. La clausola RETURN usa le occorrenze esatte di '"è' associate a $parola per generare la lista dei risultati.

Risultato atteso:

```
<xcde:results>

<xcde:result>
<xcde:filename>esempio.xml</xcde:filename>

<xcde:return result_number = '1'>
D&egrave;veloppement
</xcde:return>
.
.
.
<xcde:return result_number = '6'>
r&egrave;sum&egrave;s
</xcde:return>

</xcde:result>

</xcde:results>
```

2.1.7 Formattazione del risultato

In questo paragrafo illustriamo le funzionalità che il nostro linguaggio mette a disposizione dell'utente per la formattazione dei risultati di un'interrogazione. Queste funzionalità vanno espresse nella clausola **RETURN** mediante l'elemento speciale `<xml_context>` che racchiude le variabili inserite nella clausola **SELECT**. Lo snippet estratto deve essere ben formato. Per comodità applichiamo la formattazione dei risultati sulla semplice interrogazione illustrata nel paragrafo **2.1.1.1**

2.1.7.1 Dimensione e direzione dello snippet visualizzato

L'interrogazione cerca una singola parola nel documento.

L'esempio trova tutte le parole 'Usability' nel documento e restituisce il contesto nel quale compare ciascuna occorrenza visualizzando 20 token a sinistra e 20 token a destra della parola. Per *token* si intendono tutti gli elementi che costituiscono un documento XML. Ad esempio in:

```
<AUTORE data_nascita='1861'>Italo Svevo</AUTORE>
```

sono *token* il tag *AUTORE*, l'attributo *data_nascita* con il suo valore, le parole *Italo* e *Svevo* ed i diversi caratteri '<', '>', '</' e lo spazio.

```
SELECT  <xml_exact xml_var = '$parola' xml_case = 'sensitive'>
                Usability
        </xml_exact>
FROM    esempio.xml
```

```
RETURN  <xml_context xml_size = '20' xml_view = 'bidir'>
            $parola
        </xml_context>
```

- Operandi: "Usability"
- Commenti: L'attributo speciale xml_size specifica il numero di token che devono essere estratti in aggiunta al risultato contenuto nella variabile $parola. L'attributo speciale xml_view specifica in quale direzione visualizzare il contesto.

Risultato atteso:

```
<xcde:results>

<xcde:result>
<xcde:filename>esempio.xml</xcde:filename>

<xcde:return result_number = '1'>
<book number="1">
   <metadata>
      <title shortTitle="Improving Web Site Usability">Improving
      the Usability of a Web Site Through Expert Reviews and
      Usability Testing
 […]
</title>
 […]
</metadata>
 […]
</book>
</xcde:return>
.
.
.
<xcde:return result_number = '15'>
<subjects>
 […]
<subject>
 […]
</subject>
         <subject>Software evaluation</subject>
         <subject>Usability testing</subject>
         <subject>Manuscript collections</subject>
      </subjects>
</xcde:return>

</xcde:result>

</xcde:results>
```

Il valore dell'attributo speciale `xml_view` nell'esempio è settato a 'bidir' (entrambe le direzioni). Altri valori sono ammessi per questo attributo speciale in accordo con la tabella in figura 2.3.

bidir	Permette di estrarre uno snippet ben formato visualizzando sia il contesto che precede sia quello che segue il risultato.
Forward	Permette di estrarre uno snippet ben formato visualizzando solo il contesto che segue il risultato.
Backward	Permette di estrarre uno snippet ben formato visualizzando solo il contesto che precede il risultato.
Full	Permette di estrarre uno snippet ben formato visualizzando il risultato completato con tutti gli elementi che lo contengono.
Stop-parent	Permette di estrarre uno snippet ben formato visualizzando sia il contesto che precede sia quello che segue il risultato. La visualizzazione del contesto termina nel momento in cui si incontra, rispettivamente, l'apertura e la chiusura dell'elemento padre.

Figura 2.3 – valori dell'attributo speciale xml_view

2.1.7.2 Formato dello snippet visualizzato

L'interrogazione cerca una singola parola nel documento.

L'esempio trova tutte le occorrenze della parola 'Usability' nel documento e restituisce il contesto nel quale queste compaiono visualizzando 20 token a sinistra e 20 token a destra della parola. Del risultato vengono visualizzate solamente le parole, omettendo la visualizzazione dei tag.

```
SELECT  <xml_exact xml_var = '$parola' xml_case = 'sensitive'>
                Usability
        </xml_exact>
FROM    esempio.xml
RETURN
    <xml_context xml_size = '20' xml_view = 'bidir' xml_format =
'notag'>
            $parola
    </xml_context>
```

- Operandi: "Usability"
- Commenti: L'attributo speciale xml_format specifica se visualizzare o no i tag del risultato. Nell'esempio il valore è 'notag' che non visualizza i tag nel risultato. L'altro valore amesso per l'attributo è 'tag' che invece permette di visualizzarli. Quest'ultimo è anche il comportamento di default nel caso in cui l'attributo sia omesso.

Risultato atteso:

```
<xcde:results>

<xcde:result>
<xcde:filename>esempio.xml</xcde:filename>
```

```
<xcde:return result_number = '1'>
  Improving the  Usability of a Web Site Through Expert Reviews and
Usability Testing
</xcde:return>
.
.
.
<xcde:return result_number = '15'>
 Software evaluation  Usability testing Manuscript collections
</xcde:return>

</xcde:result>

</xcde:results>
```

2.1.8 Restituzione di un sottoinsieme dei risultati

L'interrogazione cerca una singola parola nel documento.

L'esempio seguente trova tutte le parole "Usability" nel documento.

```
SELECT  <xml_exact xml_var = '$parola' xml_case = 'sensitive'>
                Usability
        </xml_exact>
FROM    esempio.xml
RETURN  <mytag> $parola </mytag>
RANGE 5 10
```

- Operandi: "Usability"
- Commenti: La clausola **SELECT** genera l'albero XML in cui la variabile $parola viene legata al nodo etichettato 'Usability'. La clausola **RETURN** usa le occorrenze esatte di 'Usability' associate a $parola per generare la lista delle parole risultanti visualizzandone solo 6 (dalla quinta alla decima comprese).

Risultato atteso:

```
<xcde:results>
      <xcde:result>
      <xcde:filename>esempio.xml</xcde:filename>

      <xcde:return result_number = '5'>
              <mytag> Usability </mytag>
      </xcde:return>
      .
      .
      .
      <xcde:return result_number = '10'>
              <mytag> Usability </mytag>
      </xcde:return>
      </xcde:result>
</xcde:results>
```

2.2 Struttura della Query

Diamo ora una descrizione della sintassi e della semantica di un'interrogazione nel linguaggio XCDE Query Language, iniziando con la grammatica ed esaminando in seguito ogni clausola dettagliatamente.

2.2.1 La grammatica

La sintassi seguente mostra la struttura di un'interrogazione:

Query	::=	**SELECT** (*Element* | *SpecialElement*) **FROM** *NameFile*+ **RETURN** *ExprReturn*+ **RANGE** *Range*
Element	::=	*StartTag Expression EndTag*;
SpecialElement	::=	(*SpecialElement*)+ | *Literal* | *WordSearch* | <**xml_or**> (*Element*|*SpecialElement*)(*Element*|*SpecialElement*)+ </**xml_or**> | <**xml_proximity xml_maxprox** = '(1|9)(0|9)*'> (*WordSearch*)(*WordSearch*)+ </**xml_proximity**> | <**xml_anyvalue** (*Literal* = '*TypeSearch*:*Literal*')+> *Expression* </**xml_anyvalue**> | <**xml_not** [*Pivot*]?> (*Element*|*SpecialElement*) </**xml_not**>;
Expression	::=	(*Element* | *SpecialElement*)*;
StartTag	::=	< *NameTag Attribute** >;
NameTag	::=	Literal;
Attribute	::=	Literal = 'Literal' | *Pivot* | **xml_dist** = '(1|9)(0|9)*';
Pivot	::=	**xml_var** = '$Literal';
WordSearch	::=	<**xml_exact** [*Pivot*]? [**xml_case** = '*CaseValue*']? > Literal </**xml_exact**> | <**xml_prefix** [*Pivot*]? [**xml_case** = '*CaseValue*']? > Literal </**xml_prefix**> | <**xml_suffix** [*Pivot*]? [**xml_case** = '*CaseValue*']? > Literal </**xml_suffix**> | <**xml_contained** [*Pivot*]? [**xml_case** = '*CaseValue*']? > Literal </**xml_contained**> | <**xml_regexp** [Pivot]?> RegExpr </**xml_regexp**> | <**xml_error xml_maxerr** = '(1|8)*'> Literal </**xml_error**>;

CaseValue	::=	sensitive | insensitive;
TypeSearch	::=	**exi** | **exs** | **prefi** | **prefs** | **sufi** | **sufs** | **conti** | **conts** | **err**☹1|8)*;
EndTag	::=	</*NameTag*>;
Literal	::=	String;
NameFile	::=	Literal;
ExprReturn	::=	$Literal | Literal | *StartReturnTag ExprReturn EndTag* | <**xml_context** *AttrReturn*> $Literal </**xml_context**>;
StartReturnTag	::=	< *NameTag* [Literal =’Literal’]* >;
AttrReturn	::=	**xml_size** = ‘[1-9][0-9]*’ [**xml_format** = ‘(tag | notag)’]? [**xml_view**= ‘backward | forward | bidir | full | stop_parent’]?;
Range	::=	(1|9)(0|9)* (1|9)(0|9)*;

Figura 2.4 – Grammatica dell’XCDE Query Language

Nella specifica descritta sopra i simboli non terminali sono scritti in *italico*. Gli operatori ‘?’, ‘+’, e ‘*’ denotano, rispettivamente, zero o una, una o più, e zero o più occorrenze del simbolo che li precede.
Un’interrogazione nel nostro linguaggio è composta da quattro clausole **SELECT**, **FROM**, **RETURN** e **RANGE** (opzionale). Nella clausola **SELECT** va inserita un’espressione XML *ben formata*. Tale espressione può essere concettualizzata come un albero XML, dove ogni nodo rappresenta un elemento XML. Le foglie possono contenere parole o elementi vuoti. Ogni parola è rappresentata da esattamente una foglia. Durante la visita dell’albero l’ordine con il quale compaiono gli elementi fratelli non influenza i risultati. Questa scelta scaturisce dal mancato supporto fornito dal kernel XCDEv1.0 nel gestire le informazioni relative ai vincoli di parentela tra elementi fratelli. I nodi rappresentano anche gli eventuali elementi speciali presenti nell’espressione XML. Gli elementi speciali sono quelli che descrivono il tipo di ricerca da effettuare sul sottoalbero del nodo relativo. Informazioni aggiuntive sulla ricerca possono derivare da attributi speciali. Un particolare attributo speciale è `xml_var` che, dove presente in un’interrogazione, lega la variabile indicata dal valore dell’attributo all’elemento che la contiene detto *Pivot*. Questo permette all’utente di usare quella variabile nella clausola **RETURN** per fare riferimento ai risultati associati all’elemento. Altri attributi speciali sono `xml_dist`, `xml_maxprox`, `xml_maxerr`, `xml_dist` e `xml_case` che descriveremo nel paragrafo 2.2.2.2. Nella clausola **FROM** vanno indicati uno o più documenti XML sui quali effettuare la ricerca. Anche l’espressione presente nella clausola **RETURN**, essendo *ben formata*, è vista come un albero. Ogni risultato da visualizzare comporta la visita di tale albero. Durante la visita, ogni volta

che si incontra un nodo se questo non rappresenta un *Pivot* si visualizza l'elemento associato al nodo, altrimenti si visualizza il risultato corrente associato al *Pivot*.
In presenza di più risultati, questi vengono numerati e l'utente può inserire nella clausola **RANGE** l'intervallo di risultati che desidera visualizzare.

2.2.2 Clausola SELECT

La clausola principale di un'interrogazione è la **SELECT** consistente di un'espressione XML *ben formata*. L'espressione può essere un *Element* o uno *SpecialElement* e racchiude al suo interno i parametri per la ricerca.
Un *Element* è composto da un'espressione racchiusa tra l'apertura e la chiusura di un elemento e permette di specificare gli elementi XML da ricercare. Uno *SpecialElement,* a sua volta, può essere composto da una semplice stringa o da una o più parole racchiuse tra l'apertura e la chiusura di un elemento speciale e permette all'utente di indicare il tipo di ricerca da effettuare sulle stringhe contenute. La presenza di elementi speciali permette anche ad un utente che ha poca familiarità con il linguaggio di formulare un'interrogazione in modo semplice. Il confronto non è ordinato.

2.2.2.1 Elementi speciali

Le interrogazioni vengono effettuate sostanzialmente su stringhe. Una stringa corrisponde a pattern diversi a seconda del tipo di ricerca specificata. Esso può essere ad esempio una parola, un prefisso, un suffisso, una sottostringa, un'espressione regolare, ecc.. Per esprimere il tipo di ricerca da effettuare sulla stringa abbiamo definito degli elementi speciali. Gli elementi speciali più semplici sono:

- `<xml_exact>` che denota una ricerca esatta. Questo significa che la stringa racchiusa tra l'elemento speciale è esattamente la parola da cercare.

- `<xml_prefix>` che denota una ricerca per prefisso. Questo significa che la stringa racchiusa tra l'elemento speciale deve essere un prefisso della parola da cercare.

- `<xml_suffix>` che denota una ricerca per suffisso. Questo significa che la stringa racchiusa tra l'elemento speciale deve essere un suffisso della parola da cercare.

- `<xml_contained>` che denota una ricerca per sottostringa. Questo significa che la stringa racchiusa tra l'elemento speciale deve essere una sottostringa della parola da cercare.

Quando la stringa non è inclusa all'interno di un elemento speciale si assume che la ricerca da effettuare su tale stringa sia esatta.
Analizziamo ora tipi di ricerche avanzate permesse dall'XCDE Query Language.
Si possono definire ricerche per espressioni regolari tramite l'elemento speciale `<xml_regexp>`. La stringa inclusa nell'elemento speciale deve seguire la sintassi del comando *grep* **[Grep]** di linux.

Per le ricerche con errori, invece, si utilizza l'elemento speciale `<xml_error xml_maxerr = 'x'>`. In questo caso la stringa inclusa nell'elemento speciale può avere fino ad *x* errori continuando a corrispondere al pattern di ricerca, dove per errore si intende qualsiasi carattere diverso, eliminato o aggiunto.

È possibile realizzare interrogazioni sulla *proximity* utilizzando l'elemento speciale `<xml_proximity xml_maxprox = 'x'>`. Come *proximity* si intende una forma di distanza logica fra le parole che consente di escludere dalla distanza gli elementi della struttura del documento e di considerare solo le parole. Un uso di tale elemento speciale è stato illustrato nel paragrafo [**2.1.4**].

Altri tipi di interrogazioni che rendono il linguaggio completo, pensando ad interrogazioni testuali, sono quelle effettuate a partire dal valore degli attributi. Questi tipi di ricerche sono realizzate tramite l'elemento speciale `<xml_anyvalue>` all'interno del quale è necessario inserire almeno un attributo. Anche in questo caso per ogni attributo richiesto può essere specificato un pattern di ricerca per il valore. Il pattern di ricerca deve essere specificato all'interno del valore dell'attributo stesso che può essere composto da una o più parole eccetto che in caso di ricerca con errori. Con riferimento all'esempio illustrato nel paragrafo [**2.1.2.2**], elenchiamo i pattern di ricerca possibili, ad esempio, sul valore dell'attributo `'Usabilityguy Manuscript Guide'`:

- `'exi:Usabilityguy Manuscript Guide'` denota una ricerca esatta *case-insensitive* sul valore dell'attributo (`exs`, rispettivamente, per ricerca *case-sensitive*).
- `'prefi:Usabilityguy'` denota una ricerca per prefisso *case-insensitive* sul valore dell'attributo (`prefs`, rispettivamente, per ricerca *case-sensitive*).
- `'sufi:Guide'` denota una ricerca per suffisso *case-insensitive* sul valore dell'attributo (`sufs`, rispettivamente, per ricerca *case-sensitive*).
- `'conti:Usabilityguy'` denota una ricerca per sottostringa *case-insensitive* sul valore dell'attributo (`conts`, rispettivamente, per ricerca *case-sensitive*).

Anche sui valori degli attributi il linguaggio permette ricerche con errori. Queste si esprimono nel seguente modo:

```
'err:n:Usabilityguy'
```

dove *n* denota il numero massimo di errori ($0<n<9$). Ovviamente *n* non può essere maggiore della lunghezza della parola, lunghezza intesa come numero di caratteri. Nel caso di ricerca con errori all'interno del valore dell'attributo oltre a '`err:n:`' deve essere specificata esattamente una parola. La presenza di più parole non genera un messaggio di errore ma non viene restituita nessuna occorrenza per tale ricerca. La parola specificata viene cercata in modalità *case-insensitive*.

Infine è possibile anche realizzare interrogazioni booleane utilizzando gli elementi speciali `<xml_or>` e `<xml_not>`. L'elemento `<xml_or>` deve racchiudere almeno due *Element* o *SpecialElement* a differenza di `<xml_not>` che deve avere necessariamente un solo *Element* o *SpecialElement*.

Osserviamo che pur non essendoci esplicitamente nessun elemento speciale che esprime l'operatore AND, questo viene applicato di default agli elementi specificati, purché non venga inserito uno dei due operatori sopra descritti.

2.2.2.2 Attributi speciali

Ogni elemento, speciale o non, all'interno dell'espressione della clausola **SELECT** può essere marcato come interessante ai fini del risultato associandogli l'attributo speciale ***xml_var*** = **'$**nome**'**. D'ora in poi ci riferiremo a tale elemento con il termine *Pivot*. La clausola **SELECT** lega ad un *Pivot* il nodo associato ad esso. Non potendo associare l'attributo speciale direttamente alle parole, l'utente deve inserirlo nell'elemento speciale che specifica il tipo di ricerca sulla parola.
Il valore dell'attributo speciale serve nella costruzione dei risultati per riferirsi ai valori che soddisfano la ricerca per il *Pivot*; d'ora in poi ci riferiremo a tale valore con il termine *nome del Pivot.* Con riferimento all'esempio di interrogazione illustrato nel paragrafo [2.1.1.1] mostriamo l'uso dei *Pivot*:
La **SELECT** lega il *Pivot* 'Usability', denotato con '$parola', al nodo etichettato 'Usability'. Durante la ricerca al *Pivot* vengono assegnati i valori di 'Usability' che soddisfano l'interrogazione. Nella costruzione dei risultati viene generato un risultato per ogni diverso valore assunto dal *Pivot*.

Un altro attributo speciale opzionale per indicare la case-sensitivity della ricerca è ***xml_case*** = **'sensitive|insensitive'.** Esso può essere inserito negli elementi speciali che operano su parole, quali <xml_exact>, <xml_prefix>, <xml_suffix> e <xml_contained> e si riferisce alle parole racchiuse all'interno di essi. Nel caso in cui non sia presente, la ricerca è considerata *case-insensitive*. Per le ricerche su espressioni regolari tramite <xml_regexp> e con errori tramite <xml_error> l'opzione sulla case-sensitivity non è prevista e se inserita viene ignorata. Tali ricerche sono sempre *case-sensitive*. Con riferimento all'esempio di interrogazione illustrato nel paragrafo [2.1.1.1] mostriamo l'uso dell'attributo:
Nell'esempio la **SELECT** associa l'attributo all'elemento <xml_exact> specificando che la parola 'Usability' deve essere cercata in modo *case-sensitive*.

L'ultimo attributo opzionale da analizzare è ***xml_dist*** = **'x'**, che permette di esprimere i vincoli sulla discendenza degli elementi. Un'espressione contenente un elemento, chiamiamolo A, al quale è associato l'attributo xml_dist seleziona tutte le occorrenze in cui A occorre nel documento al massimo a distanza x dall'elemento padre che è specificato nell'espressione dentro la **SELECT**. Ad esempio se x è uguale ad 1, significa che l'elemento deve essere figlio diretto dell'elemento padre specificato (i.e. nessun altro elemento deve trovarsi tra i due nel documento originale). Un uso è stato illustrato nel paragrafo [2.1.2.3]. L'attributo può essere inserito all'interno di qualsiasi elemento. L'inserimento dell'attributo all'interno di elementi speciali non è consentito, in quanto la nozione di 'distanza' esiste solamente tra gli elementi presenti all'interno del documento XML. Anche l'inserimento dell'attributo all'interno dei figli diretti degli elementi speciali non ha è consentito, in quanto la distanza si riferisce all'elemento padre che in questo caso è 'speciale'. In tal caso l'attributo sarà ignorato. Ad esempio se l'espressione contiene un elemento A, padre di un elemento speciale B, che a sua volta è padre di un elemento C (figura 2.5a) e l'utente inserisce l'attributo speciale all'interno di C, questo non vincola la sua distanza dall'elemento A. Per esprimere ciò, occorre modificare l'espressione facendo diventare A figlio diretto di B e padre di C (figura 2.5b).

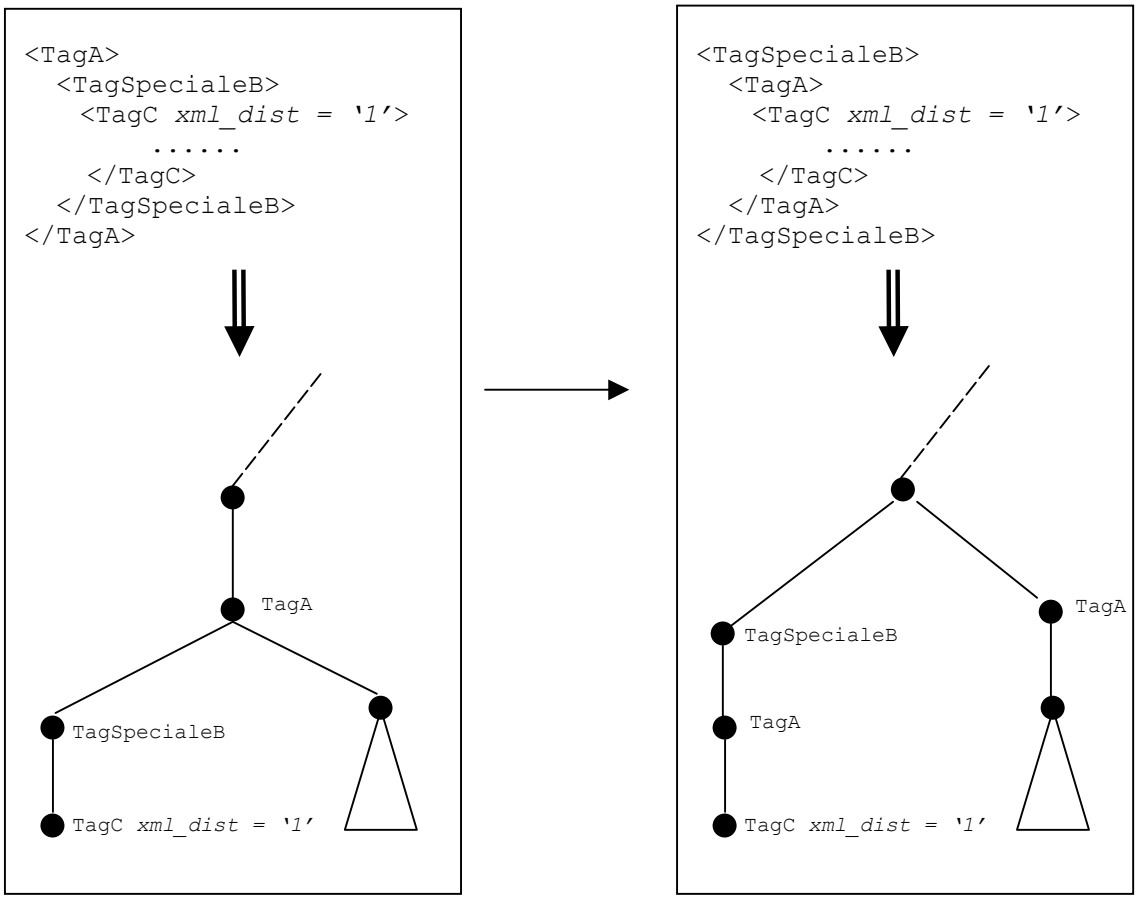


Figura 2.5a – Uso errato di 'xml_dist'

Figura 2.5b –Uso corretto di 'xml_dist'

Inoltre se posto all'interno dell'elemento radice dell'albero, l'attributo sarà ignorato.

Infine gli attributi speciali ***xml_maxerr*** = '*n*' e ***xml_maxprox*** = '*x*' sono specifici rispettivamente degli elementi `<xml_error>` e `<xml_proximity>`. Il primo attributo indica il massimo numero di errori ($0<n<9$) ammessi per la parola specificata affinché quest'ultima soddisfi ancora la ricerca. Il secondo indica la proximity massima ($x>0$) alla quale devono stare le parole racchiuse dall'elemento. Il valore 1 di proximity indica che le parole devono essere consecutive.

2.2.3 Clausola FROM

La clausola **FROM** specifica l'insieme dei documenti, separati da uno spazio, su cui effettuare la ricerca indicata nella clausola **SELECT**. Questo, quindi, dà la possibilità di effettuare la stessa ricerca su più documenti scrivendo ed eseguendo una sola interrogazione. E' comunque necessario inserire il nome di almeno un documento affinché non venga generato nessun errore sintattico.

2.2.4 Clausola RETURN

La clausola **`RETURN`** permette di esprimere quali sono i risultati da visualizzare e il formato secondo al quale visualizzarli.
L'espressione XML, contenuta al suo interno, può essere composta dal nome di uno o più *Pivot*, da un'espressione XML ben formata contenente uno o più *Pivot* tra gli elementi terminali, oppure da un'arbitraria combinazione di essi. L'espressione XML permette all'utente di specificare elementi arbitrari da restituire nel risultato.
L'uso dei *Pivot* in questa clausola permette di riferire i risultati ottenuti dall'esecuzione dell'interrogazione.
Gli elementi speciali possono racchiudere i nomi dei *Pivot* per gestire le proprietà del contesto estratto associato ad essi. Questo rende il linguaggio potente e completo non solo nella fase di ricerca ma anche in quella della gestione dei risultati. Parleremo degli elementi speciali contenuti nella clausola **`RETURN`** qui di seguito.

2.2.4.1 Elementi speciali

L'unico elemento speciale, opzionale, previsto è `<xml_context>` che racchiude solo nomi di *Pivot* (uno o più). Le proprietà definite da tale elemento speciale si applicano ad ogni nome di *Pivot* contenuto in esso.
Si possono definire proprietà diverse per *Pivot* diversi. Per esprimere ciò ogni nome di *Pivot* deve comparire nel proprio `<xml_context>`.

2.2.4.2 Attributi speciali

Le varie proprietà del contesto associato ai nomi dei *Pivot* sono definite tramite gli attributi speciali dell'elemento `<xml_context>`.
L'attributo speciale ***`xml_size =`*** *`'x'`* regola il numero di token da estrarre oltre al risultato associato al *Pivot* relativo. Il valore di `xml_size` deve essere un intero positivo (x>0). Impostare il valore a '0' oppure non inserire l'attributo speciale non genera un errore ma è equivalente all'assenza dell'elemento speciale `<xml_context>` relativo. Un uso è stato illustrato nel paragrafo [2.1.7.1].
L'attributo speciale ***`xml_view`*** specifica quali sono i token da estrarre. La presenza di `xml_view` è opzionale e può assumere i seguenti valori:

- *`'backword'`*: i token estratti sono quelli che precedono lo snippet associato al *nome del Pivot*.
- *`'forward'`*: i token estratti sono quelli che seguono lo snippet associato al *nome del Pivot*.
- *`'bidir'`*: i token estratti sono sia quelli che precedono sia quelli che seguono lo snippet associato al *nome del Pivot*. In questo caso vengono estratti gli '*x*' token che precedono e gli '*x*' token che seguono, dove '*x*' è il valore dell'attributo speciale `xml_size`.

- `'full'`: in questo caso lo snippet associato al *nome del Pivot* viene completato con tutti gli elementi del documento XML in input che lo contengono.

- `'stop_parent'`: inserendo questo valore il comportamento è simile a quello ottenuto impostando il valore a `bidir`. In questo caso, però, l'estrazione dei token che precedono si interrompe non solo nel momento in cui viene raggiunto il numero di token indicato, ma anche appena si incontra l'apertura dell'elemento padre dello snippet associato al *nome del Pivot*. Allo stesso modo l'estrazione dei token che seguono lo snippet si interrompe non solo nel momento in cui si raggiunge il numero di token indicato, ma anche appena si incontra la chiusura dell'elemento padre. In tutti e due i casi l'elemento padre viene restituito nel risultato.

In tutti i casi il risultato visualizzato è un frammento di codice XML ben-formato. Infatti, se lo snippet estratto non è ben-formato questo viene completato bilanciando gli elementi aperti e non chiusi e viceversa.
Se non è presente l'attributo speciale, il valore di default è `'bidir'`. Un uso è stato illustrato nel paragrafo [2.1.7.1].
Infine, l'attributo speciale ***xml_format = 'tag|notag'*** permette di scegliere se visualizzare o meno i tag all'interno dello snippet estratto indicando come valore rispettivamente `'tag'` e `'notag'`. Anche questo attributo è opzionale ed il suo valore di default è `'tag'`. Un uso è stato illustrato nel paragrafo [2.1.7.2].

2.2.5 Clausola RANGE

La clausola **`RANGE`** offre la possibilità di visualizzare un sottoinsieme dei risultati che soddisfano l'interrogazione. In questa clausola l'utente deve specificare due interi positivi maggiori di zero che corrispondono agli estremi, compresi, dell'intervallo dei risultati da visualizzare. Il secondo valore inserito deve essere maggiore o uguale al primo. Se il numero di risultati dell'interrogazione è maggiore del primo estremo ma minore del secondo, viene fissato automaticamente il secondo estremo al numero di risultati. Ad esempio se la ricerca restituisce 20 risultati, inserire "**`RANGE`** `10 25`" è equivalente ad inserire "**`RANGE`** `10 20`". Allo stesso modo, se il primo estremo è maggiore del numero di risultati dell'interrogazione non viene visualizzato nessun risultato, ma un messaggio in cui si avverte l'utente che non ci sono risultati nell'intervallo specificato e quanti sono i risultati. Ad esempio se la ricerca restituisce 20 risultati e l'utente inserisce "**`RANGE`** `25 30`", verrà avvisato con il messaggio:

```
<xcde:warning>
      No results in current range. Results are only 20
</xcde:warning>
```

Un uso è stato illustrato nel paragrafo **[2.1.8]**.

Capitolo 3

XCDEv2.0: Query Engine

Abbiamo descritto nel capitolo precedente la sintassi e la semantica dell'XCDE Query Language. In questo capitolo descriveremo l'implementazione del linguaggio fornendo una descrizione approfondita delle strutture dati, della loro costruzione, del loro uso, e fornendo inoltre le motivazioni che sono alla base delle nostre scelte implementative.

3.1 Strutture dati del Query Engine

L'espressione XML presente nella clausola **`SELECT`** deve essere *ben formata*, e quindi può essere strutturata come un albero con tutti gli elementi inseriti all'interno di una gerarchia padre-figlio e con un solo elemento radice.
Tale albero viene generato facendo il parsing dell'espressione XML. Il parser di tipo SAX **[SAX]** fa uso di una struttura dati ausiliaria di tipo pila chiamata `pila_parsing` e descritta nel paragrafo [3.1.1]
L'albero è formato da nodi interni che rappresentano gli elementi contenuti nell'espressione e da nodi foglia che rappresentano sia elementi vuoti sia parole. Descriveremo la struttura dei nodi nel paragrafo [3.1.2].
I *nomi dei Pivot* presenti nell'espressione XML vengono legati ai nodi marcati come *Pivot*, e tale legame viene inserito in una tabella hash così da facilitare il loro recupero al momento della valutazione della clausola **`RETURN`**. La struttura della tabella hash sarà descritta nel paragrafo [3.1.3].

3.1.1 Pila_parsing

Il parser nel momento in cui analizza un elemento non ha ancora visitato il suo sottoalbero, per questo motivo non è ancora in grado di costruire il nodo associato. La struttura dati `pila_parsing` è stata introdotta per memorizzare temporaneamente le informazioni, possedute in quel momento e utili successivamente, di tali nodi.
Tale struttura dati è un vettore di elementi di tipo `nodo_pila` la cui struttura è la seguente:

```
typedef struct {
  query_tree_node *node;
  unsigned int childs_vector_dimension;
```

```
} nodo_pila;
```

dove `node` è un vettore di puntatori ai nodi figli dell'elemento corrente e `childs_vector_dimension` indica il numero di tali figli.

3.1.2 Query_tree_node

Di ogni elemento dell'espressione XML occorre memorizzare informazioni utili ai fini dell'esecuzione dell'interrogazione e della restituzione dei risultati. Tali informazioni vengono raccolte durante il parsing ed inserite nei nodi dell'albero che hanno la seguente struttura:

```
typedef struct query_tree_node {

  char *node_name;
  unsigned int node_type;
  struct query_tree_node **childs_vector;
  unsigned int num_childs;
  char **attributes_list;
  unsigned int num_attributes;

  union {
        XCDE_IL_Couple_Type *tag_positions;
        XCDE_IL_Triple_Type *word_positions;
  };
  unsigned int num_positions;
  int *depth_vector;
  unsigned int num_depth;

  unsigned char is_potable;
  int *number_results_vector;
  unsigned char is_setted_number_results;
  } query_tree_node;
```

Possiamo distinguere tre classi di informazioni.
La prima classe raccoglie le informazioni relative all'espressione XML. Di questa classe fa parte il campo `node_name` nel quale si memorizza il nome dell'elemento o della parola associata al nodo. Il tipo dell'elemento è distinguibile dal campo `node_type` che assume valore 'TAG' o 'WORD'. Il legame padre-figlio è espresso attraverso il vettore di puntatori ai nodi figli `childs_vector` la cui dimensione, che corrisponde al numero di figli, è memorizzata in `num_childs`. Anche gli attributi degli elementi vengono mantenuti nella struttura e precisamente nel vettore `attributes_list` in cui ogni coppia nome-valore dell'attributo occupa due posizioni consecutive nel vettore. La cardinalità del vettore, pari a due volte il numero di attributi, è inserito in `num_attributes`.
Per quanto riguarda la seconda classe di informazioni, questa è relativa alla fase di esecuzione dell'interrogazione. Di questa classe fanno parte per ogni nodo le posizioni fisiche, all'interno del documento XML compresso, delle occorrenze trovate per l'elemento associato a tale nodo. Per capire cosa sono le posizioni fisiche pensiamo al fatto che la presenza di entità all'interno dei documenti XML porta alla distinzione tra il documento fisico, cioè il sorgente, e il documento logico, ossia ciò che il documento effettivamente descrive. Così gli elementi:

```
<AUTORE data_nascita="1899">&ernie;</AUTORE>

<TITOLO>G&#246;del, Escher, Bach: un'eterna ghirlanda brillante /TITOLO>
```

corrispondono in realtà dal punto di vista logico gli elementi:

```
<AUTORE data_nascita="1899">Ernest Hemingway</AUTORE>

<TITOLO>Gödel, Escher, Bach: un'eterna ghirlanda brillante </TITOLO>
```

suppenendo che all'entità `&ernie;` sia associato il valore "`Ernest Hemingway`".
Per gestire questa situazione il kernel XCDE si basa su una doppia vista del documento XML:

- La *vista logica*: che viene considerate per l'indicizzazione e la ricerca;
- La *vista fisica*: che viene considerata per la memorizzazione compressa;

L'implementazione delle due viste comporta la presenza di elementi *logici* e *fisici* all'interno delle strutture dati *dizionario* proprie del kernel XCDE. I dizionari contengono gli elementi che costituiscono il documento (tag, parole ed attributi). Il meccanismo di indicizzazione per questi elementi è quello delle liste invertite. Agli elementi dei dizionari sono associati delle *liste invertite*, che possono essere liste di posizione o di proximity. Le prime contengono le *posizioni fisiche* all'interno del documento compresso intese come le posizioni in byte nel documento compresso dei token fisici corrispondenti.
Le posizioni fisiche relative all'occorrenza di un elemento o di una parola vengono inserite all'interno del vettore `tag_positions` o `word_positions` a seconda che il `node_type` sia rispettivamente 'TAG' o 'WORD'.
`Tag_positions` è un vettore di elementi di tipo `XCDE_IL_Couple_Type` **[XCDEv1.0]**. L'occorrenza di un tag è identificata da una posizione di apertura e una posizione di chiusura che occupano posizioni consecutive nel vettore.
`Word_positions` è un vettore di elementi di tipo `XCDE_IL_Triple_Type` **[XCDEv1.0]**. L'occorrenza di una parola è identificata da un'unica posizione inserita nel vettore.
In entrambi i casi `num_positions` contiene il numero di elementi del vettore che, in caso di tag, è `tag_positions` e in caso di parole è `word_positions`. Osserviamo che `num_positions` in caso di tag è pari al doppio del numero di occorrenze perché per ogni occorrenza di un tag vengono mantenute nel vettore la sua posizione di apertura e chiusura, mentre in caso di parole è uguale al numero di occorrenze.
D'ora in poi useremo il termine 'posizione' per riferirci alla coppia (pos_apertura, pos_chiusura) in caso di un tag o alla singola posizione in caso di una parola.
Insieme alle posizioni degli elementi viene mantenuta la lista delle *profondità* associate ad un elemento, ossia il livello a cui si trova quell'elemento nell'albero del documento XML. Dal nostro punto di vista la nozione di profondità è legata solamente agli elementi di tipo tag, in quanto il linguaggio non permette di specificare la profondità di elementi di tipo parola. Il vettore che contiene tale informazione è `depth_vector`, la cui dimensione è specificata in `num_depth`.
L'ultima classe di informazioni racchiude ciò che riguarda la fase di costruzione e visualizzazione dei risultati. Si è deciso di utilizzare la tecnica della '*potatura*' per diminuire l'occupazione di memoria dovuta alla presenza di nodi utili ai fini dell'esecuzione dell'interrogazione ma non alla visualizzazione dei risultati. Ciò è dovuto al fatto che la visualizzazione dei risultati è legata alla presenza di *Pivot*, quindi tutti quei sottoalberi che non contengono nodi marcati come *Pivot* dopo l'esecuzione dell'interrogazione possono essere potati. Per identificare i nodi potabili la struttura del nodo prevede il flag `is_potable`.

Per una possibile ottimizzazione abbiamo associato ad ogni nodo un vettore, chiamato `number_results_vector`, che specifica le informazioni sul numero di risultati relativi al nodo medesimo. L'elemento i-esimo del vettore contiene il numero di risultati relativi all'i-esima occorrenza dell'elemento associato al nodo. Ogni occorrenza può far parte di più risultati. Ad esempio, con riferimento alla figura 3.1:
se il nodo B ha 3 occorrenze (b_1, b_2, b_3) relative alla posizione a_1 del nodo A e il nodo C ne ha 2 (c_1, c_2), nella posizione del vettore `number_results_vector` relativa all'occorrenza a_1 sarà presente il numero 6, frutto del prodotto dei 3 risultati di B per i 2 di C.

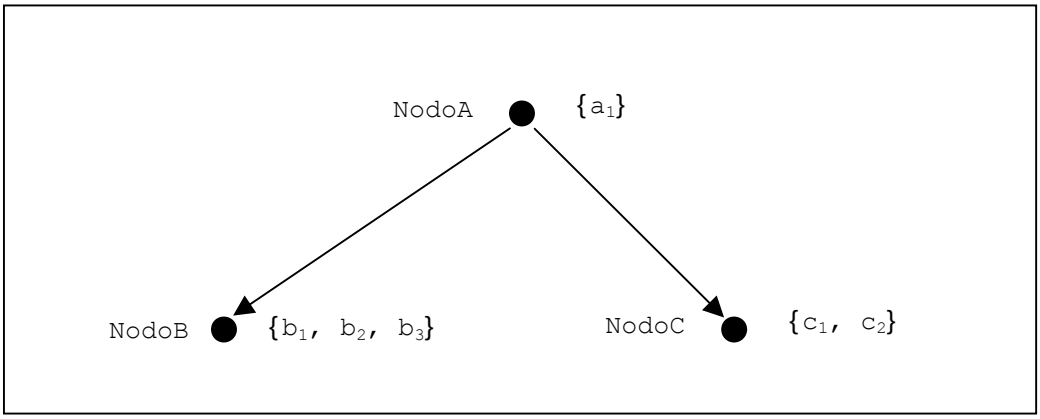


Figura 3.1 – Esempio di calcolo del vettore number_results_vector

Per calcolare la dimensione del vettore si fa riferimento a `num_positions`, in particolare in caso di elemento di tipo tag la dimensione è pari alla metà del valore di `num_positions`, in caso di elemento di tipo parola la dimensione è pari esattamente al valore di `num_positions`. Ricordiamo, infatti, che un elemento di tipo tag è caratterizzato da due posizioni, quella di apertura e quella di chiusura. Infine per conoscere lo stato di tale vettore, se è stato riempito o meno, è previsto il flag `is_setted_number_result`.

3.1.3 Tabella hash

La corrispondenza esistente tra i *nomi dei Pivot* e i nodi ad essi associati è contenuta all'interno della struttura dati `HHash_table`. La tabella hash contiene un campo `size` che indica il numero di elementi corrente e un campo `card` contenente la dimensione totale. Il corpo della tabella hash è un vettore di puntatori ad una lista di nodi. La struttura di tali nodi permette di memorizzare il *nome del Pivot* in `variable_name`, il suo indice in `index` (la cui utilità sarà illustrata nel paragrafo [**3.4**]) ed il nome del nodo associato in `node_name`.

```
typedef struct Hash_node {
  char *variable_name;
  int index;
  char *node_name;
  struct Hash_node *next;
} Hash_node;

typedef Hash_node **Hash_nodeptr_array;

typedef struct {
  int size;
  int card;
```

```
  Hash_nodeptr_array table;
} HHash_table;
```

3.1.4 Tabella dei risultati

Durante la fase di esecuzione, si utilizza la seguente struttura dati d'appoggio per memorizzare, per ogni risultato, i valori associati ai *nomi dei Pivot*. Tali valori verranno recuperati durante la fase di visualizzazione dei risultati per essere sostituiti ai *nomi dei Pivot* presenti nella clausola **RETURN**.

```
XCDE_IL_Couple_Type **Table_Result.
```

`Table_Result` è una matrice con un numero di righe pari al numero dei risultati e un numero di colonne pari al doppio del numero di *nomi dei Pivot* presenti nella clausola **RETURN**. Riferendoci al seguente esempio

```
SELECT <book>
            <author xml_var = '$author'>Elina Rose</author>
            <title xml_var = '$title'>Expert reviews</title>
      </book>
FROM esempio.xml
RETURN $author $title
```

e supponendo che questa interrogazione restituisca due risultati, la matrice avrebbe due righe ognuna della forma:

posizione di apertura	posizione di chiusura	posizione di apertura	posizione di chiusura

Le prime due colonne sono relative alle posizioni di apertura e chiusura dell'occorrenza dell'elemento associato al *nome di Pivot* `$author` e le altre due colonne si riferiscono a quella di `$title`. In totale si hanno un numero di colonne, nell'esempio quattro, pari al doppio del numero di *Pivot*. In caso di *Pivot* associati a parole non cambia il numero di colonne ma le posizioni di apertura e chiusura coincidono.

3.2 Parsing dell'interrogazione e costruzione dell'albero associato

La creazione dell'albero associato all'espressione presente nella clausola **SELECT**, che da ora in poi chiameremo *albero della select,* e dell'albero associato all'espressione presente nella clausola **RETURN**, che da ora in poi chiameremo *albero della return,* avviene durante una fase di *analisi* delle due espressioni.
Per l'analisi di tali espressioni si è utilizzata la libreria di *parsing* Expat versione 1.2, creata da J.Clark **[Clark]**. Essa è una libreria di funzioni di tipo SAX di pubblico dominio, interamente scritta in linguaggio C ed è ampiamente testata e utilizzata (su di essa si basano ad esempio le funzionalità XML di Perl e Netscape). Essendo basata sulla modalità SAX, la libreria Expat è *ad eventi*. Per ogni evento possibile (ad esempio apertura o chiusura di un elemento, apertura di un commento o di una CDATA, ecc.) sono definite dall'utente delle funzioni che vengono chiamate automaticamente dal sistema quando l'evento si verifica, e a cui sono passati dei parametri che consentono di determinare da quale evento è stata generata la chiamata alla funzione e quali siano gli elementi da

gestire. Attraverso le chiamate alle singole funzioni si individuano gli elementi, tag e parole, e gli attributi, che costituiscono l'albero, riempiendo la prima classe di informazioni della struttura `query_tree_node`, riportata nel paragrafo**[3.1.2]**, di ogni nodo.
Durante questa fase di analisi dell'interrogazione, inoltre, viene allocato e riempito un vettore contenente i nomi dei documenti sui quali effettuare la ricerca e, nel caso in cui sia presente la clausola **RANGE**, due interi contenenti i valori degli estremi dell'intervallo.
Viene, infine, inizializzata e riempita la tabella hash con tutti i valori relativi ai *Pivot* (nome, nome nodo, ecc…) **[3.1.3]**.

3.3 Esecuzione dell'interrogazione

Generato l'*albero della select* si può procedere con la ricerca delle occorrenze degli elementi che soddisfano l'interrogazione ed alla risoluzione dei legami con i *Pivot*. In particolare, alla fine della fase di esecuzione dell'interrogazione ogni nodo conterrà il vettore con le posizioni, all'interno del documento XML compresso, di tali elementi. Per ottenere questo risultato si effettua una visita bottom-up sull'*albero della select.* Prima della visita i vettori delle posizioni presenti in ciascun nodo dell'albero sono vuoti.
Illustriamo come si procede nell'esecuzione di un'interrogazione attraverso un esempio sul documento `esempio.xml` riportato in figura 3.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
  <book number="1">
      <author>Millicent Marigold</author>
      <chapter_title>Improving the Usability of a Web Site</chapter_title>
      <chapter_title>Marigold Web Site</chapter_title>
  </book>
  <book number="2">
      <author>Millicent Marigold</author>
      <author>Elina Rose</author>
  </book>
</books>
```

Figura 3.2 – Esempio di documento XML

La seguente interrogazione:

```
SELECT <book>
         <author xml_var = '$author'>Marigold</author>
         <chapter_title xml_var = '$title'></chapter_title>
       </book>
FROM esempio.xml
RETURN $author $title
```

trova tutti i libri scritti da `Marigold` restituendo l'autore ed il titolo del capitolo.
L' *albero della select* associato all'interrogazione è riportato in figura 3.3. Ad ogni nodo dell'albero è affiancata la lista di tutte le occorrenze di ciascun elemento e parola.
La visita bottom-up genera in ogni nodo un vettore contenente le posizioni che soddisfano le condizioni imposte dal suo sottoalbero. Per la generazione di tale vettore, in ogni nodo si filtra la lista delle occorrenze mantenendo quelle corrette in accordo con il sottoalbero.

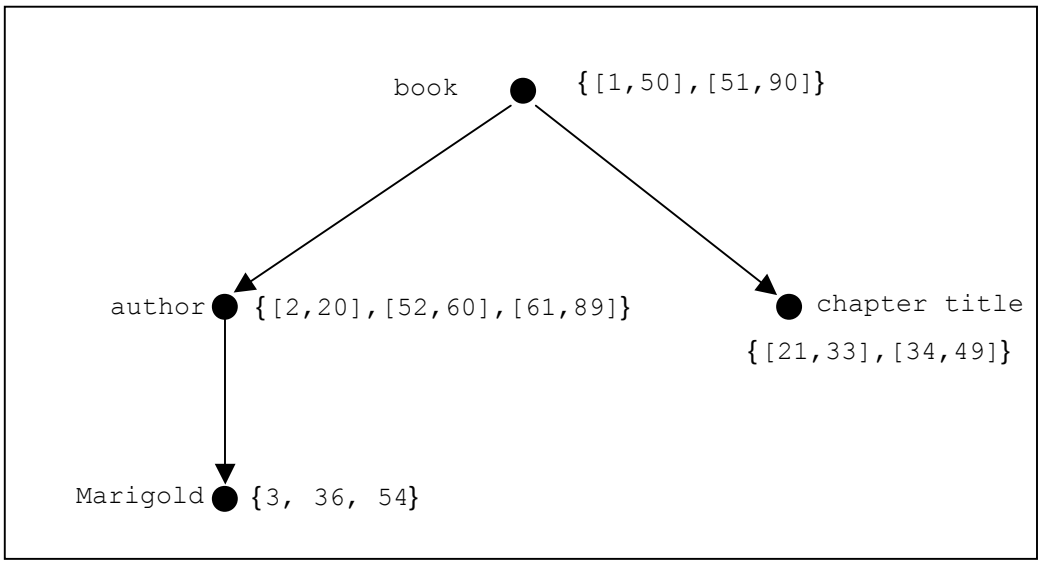


Figura 3.3 – Esempio di albero della select e liste di occorrenze

Nell'esempio, nel nodo `author`, della lista `{[2,20],[52,60],[61,89]}` la posizione `[61,89]` viene eliminata poiché si riferisce ad un `author` che non è `Marigold`. Questo lo si evince dal fatto che nessuna delle occorrenze di `Marigold` è contenuta nell'intervallo `[61,89]`.
L'albero risultante della visita bottom-up è quello mostrato in figura 3.4.

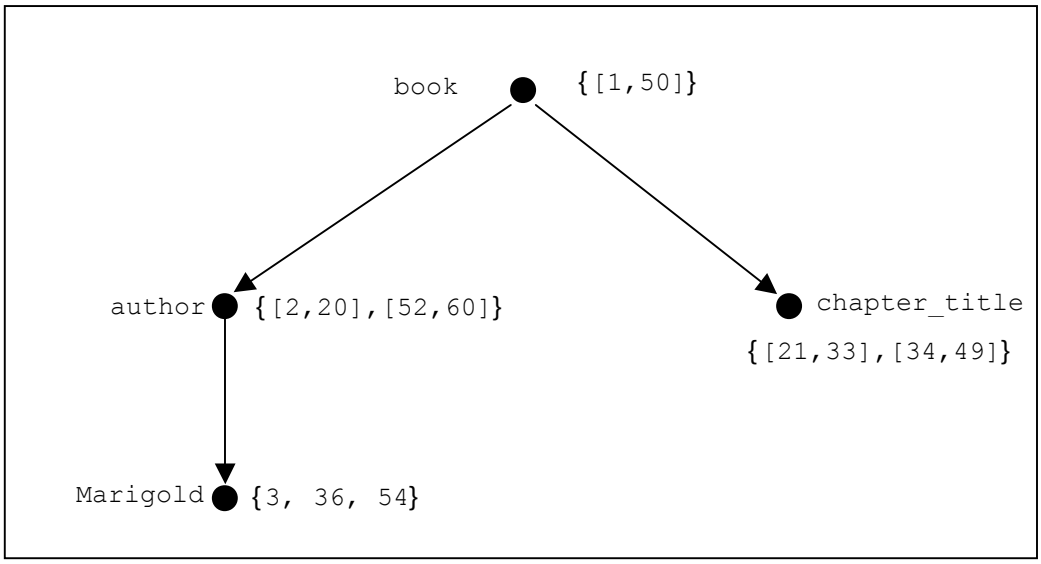


Figura 3.4 – Esempio di albero dopo la visita bottom-up

Come si può notare alla fine della visita bottom-up ci sono alcune posizioni che prima erano state mantenute, ad esempio la posizione `[52,60]` del nodo `author`, e che ora vanno eliminate. Questo è dovuto al fatto che dato un nodo, la visita bottom-up non permette di controllare i vincoli imposti dai nodi non facenti parte del suo sottoalbero. Quindi, l'unico nodo che alla fine della visita contiene le posizioni corrette è la radice. Facendo riferimento all'esempio, l'unica posizione esatta è `[1,50]` poiché è l'unica ad essere stata mantenuta dopo il filtraggio della lista. A questo punto è necessaria un'altra

visita, stavolta top-down, descritta nel paragrafo **[3.3.2]**, che a partire dalle posizioni esatte della radice filtra i vettori delle posizioni presenti nei restanti nodi. Inoltre, durante la visita bottom-up l'*albero della select* viene potato dei nodi non interessanti ai fini della costruzione del risultato ma che sono stati utili solo durante la ricerca delle posizioni esatte.
Nella visita top-down, dato un nodo, di ogni figlio vengono mantenute le posizioni il cui *intervallo* è contenuto in almeno uno di quelli del nodo. Data una posizione indichiamo come *intervallo* la finestra che ha come estremi la posizione di apertura e di chiusura. L'albero risultante da tale visita è il seguente:

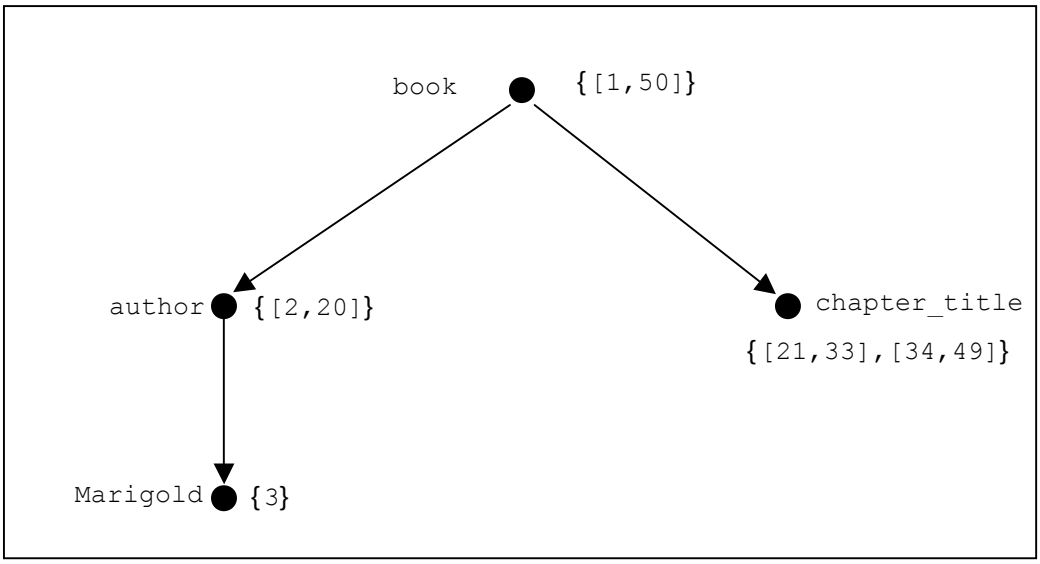


Figura 3.5 – Esempio di albero dopo la visita top-down

Notiamo che nel nodo `author` la posizione `[52,60]` è stata eliminata perché l'intervallo corrispondente non è contenuto nell'intervallo `[1,50]` del nodo padre `book`. Così come sono state eliminate le posizioni `[36]` e `[54]` del nodo `Marigold`, posizioni che nella visita bottom-up erano state mantenute.
Le posizioni fisiche che riempiono l'*albero della select* sono relative ad un singolo documento compresso. Avendo la possibilità, nella clausola **FROM**, di specificare più documenti sui quali effettuare la stessa ricerca, l'*albero della select* viene replicato un numero di volte pari al numero di tali documenti prima di essere riempito.
Nei prossimi paragrafi ci limiteremo, per semplicità descrittiva, ad illustrare il caso in cui nella clausola **FROM** sia presente un solo documento lavorando, quindi, su un singolo albero. Questo caso è facilmente generalizzabile ad un numero di documenti maggiore di uno ripetendo, per ciascuno, le operazioni descritte nei prossimi paragrafi.

3.3.1 Visita bottom-up

L'operazione principale effettuata dalla visita bottom-up dell'*albero della select* è quella di riempire i vettori delle posizioni associati ai nodi rispettando i vincoli di innestamento. Come già osservato, i nodi foglia rappresentano sia elementi di tipo "TAG" sia elementi di tipo "WORD", mentre i nodi interni rappresentano solamente elementi di tipo "TAG. A loro volta i nodi interni sono relativi sia ad elementi speciali che non. Descriviamo prima l'analisi degli elementi di tipo "WORD" e di tipo "TAG" non speciali. I primi nodi ad

essere analizzati sono le foglie i cui vettori vengono riempiti con le posizioni di tutte le occorrenze dell'elemento relativo che soddisfa il tipo di ricerca. Induttivamente, analizzando un nodo interno, in un primo momento si riempie il vettore con tutte le posizioni delle occorrenze dell'elemento "TAG" relativo e immediatamente dopo si filtra tale vettore in base alle posizioni presenti nei vettori dei figli lasciando solo le posizioni valide. Una posizione è considerata *valida* se per ogni figlio esiste almeno una posizione nel suo vettore inclusa in essa. Nell'esempio di figura 3.2, la posizione `[23,50]` del nodo A è *valida* in quanto il figlio B contiene la posizione `[24,30]` e il figlio C contiene `[31,40]`, entrambe incluse nella posizione `[23,50]` di A.

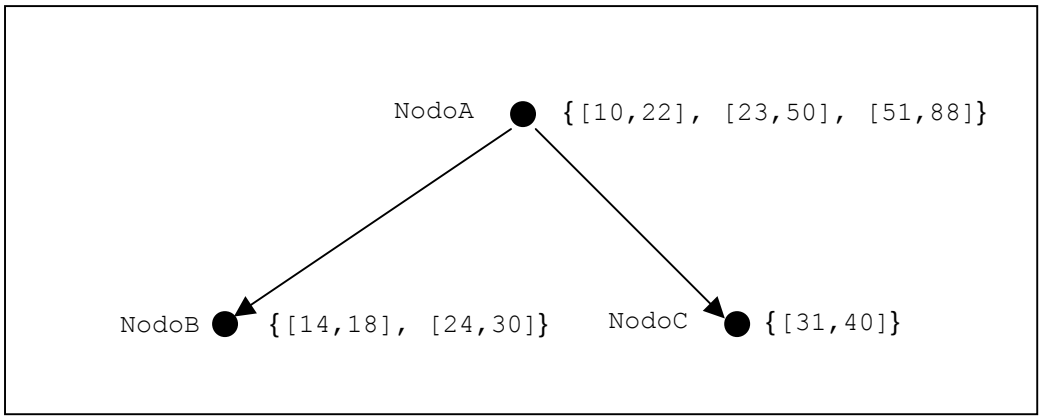


Figura 3.6 – Esempio di posizioni valide

E' possibile che l'utente abbia vincolato un elemento inserendo uno o più attributi non speciali. In tal caso si filtra ulteriormente il vettore delle posizioni eliminando quelle che non contengono gli attributi specificati.
Descriviamo, ora, il comportamento in presenza di un nodo rappresentante un elemento "TAG" speciale. Poiché agli elementi speciali che indicano il tipo ricerca da effettuare sulla parola racchiusa al loro interno (`<xml_exact>`, `<xml_prefix>`), non corrisponde nessun vettore di posizioni, verranno eliminati non appena si sono recuperate le posizioni della parola contenuta. Il vettore delle posizioni per l'elemento speciale `<xml_not>` corrisponde al vettore del suo unico figlio. Nel caso dell'elemento speciale `<xml_or>` il vettore delle posizioni si ottiene fondendo i vettori dei figli. Per quanto riguarda il trattamento dell'elemento speciale `<xml_proximity>` esso comporta l'accesso alla *lista delle proximity* di ogni parola presente all'interno dell'elemento e l'uso di un algoritmo basato sulla tecnica di *plane-sweep* **[Sadakane]**. Le liste di proximity contengono il numero sequenziale della parola nel documento, in modo da consentire interrogazioni su più parole ponendo un vincolo sulla loro distanza di occorrenza **[Navarro97]**. Tale algoritmo riempie il vettore tag_positions con le posizioni di inizio e fine della finestra in cui compaiono le parole indicate entro la *proximity* specificata. Queste posizioni verranno trattate d'ora in poi come se fossero posizioni di apertura e chiusura di un normale elemento.
Infine per l'elemento speciale `<xml_anyvalue>` si genera, inizialmente, per ogni attributo specificato una lista temporanea delle posizioni degli elementi che contengono tale l'attributo. Successivamente l'intersezione di tali liste genera il vettore delle posizioni da associare all'elemento `<xml_anyvalue>`.
Un'altra operazione effettuata durante questa fase prevede la potatura di alcuni nodi dell'*albero della select*. I nodi potati sono tutti quelli che non sono stati marcati come *Pivot*, nella fase di parsing, e che hanno tutti i figli potabili.

Durante questa fase il vettore generato nel nodo corrente contiene sì posizioni *valide* ma bisogna tener presente che nella visione globale dell'interrogazione non sono stati ancora presi in considerazione i vincoli legati agli antenati del nodo. Solo la radice *dell'albero della select* conterrà alla fine di tale fase le posizioni finali che soddisfano l'interrogazione. Per tale motivo occorre effettuare un'ulteriore visita dell'albero che a partire dalle posizioni nella radice, filtra gli altri vettori per ottenere le posizioni finali.

3.3.2 Visita top-down

Durante la visita, per ogni nodo raggiunto, si correggono i vettori delle posizioni di ciascun figlio tenendo conto di due fattori. Il primo riguarda l'inclusione degli intervalli associati alle posizioni dei figli in quelli del nodo, e il secondo il giusto grado di parentela tra gli elementi rappresentati dal nodo e dal figlio.
In ogni nodo, infatti, in corrispondenza del vettore delle posizioni è presente il vettore delle profondità. Questo significa che ad ogni posizione dell'elemento corrisponde la sua profondità nel documento XML.
Dato un figlio, se questo non contiene l'attributo `xml_dist`, le posizioni eliminate dal suo vettore sono quelle per cui non esiste nessuna posizione nel vettore del padre che le contiene. Nel caso in cui l'attributo sia presente, quest'ultimo pone un ulteriore vincolo sulle posizioni che saranno mantenute nel vettore, più precisamente rimarranno quelle per cui la differenza tra le profondità corrispondenti è minore del valore dell'attributo.
Se il nodo figlio è di tipo "WORD" si corregge il suo vettore tenendo conto solo del primo fattore.
In presenza di un nodo figlio associato all'elemento speciale `<xml_not>`, considerando che già per ogni posizione del suo vettore `tag_positions` non esiste nel padre nessuna posizione che la include, il raffinamento del vettore del padre non comporta il corrispondente raffinamento del vettore del figlio.
Termina con questa visita la fase di esecuzione dell'interrogazione. Ora, dunque, *l'albero della select* contiene le posizioni delle occorrenze degli elementi che soddisfano l'interrogazione. Descriveremo nei prossimi paragrafi la parte relativa alla costruzione e visualizzazione dei risultati.

3.4 Costruzione dei risultati

Vediamo ora il problema che giustifica la necessità di avere una fase di costruzione dei risultati ed in seguito descriviamo le operazioni che vengono effettuate in questa fase.
Ogni vettore di posizioni è partizionato logicamente in gruppi. Una posizione appartiene ad un gruppo piuttosto che ad un altro in base alla posizione del nodo padre che la include. Gruppi creati a partire dalla stessa posizione del padre sono legati tra loro.

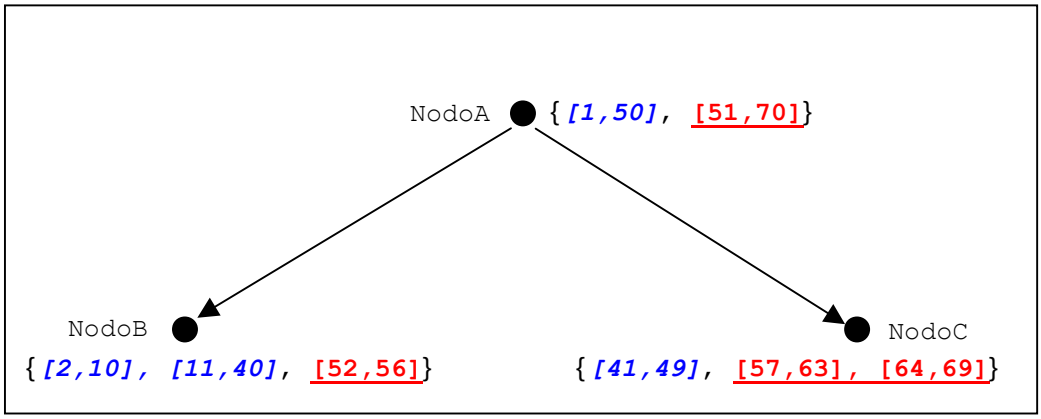


Figura 3.7 – Esempio di partizionamento in gruppi delle posizioni

Ad esempio, con riferimento alla figura 3.7, il vettore di posizioni del nodo B è suddiviso in 2 gruppi, il primo B_1 contenente le posizioni {[2,10], [11,40]} e il secondo B_2 contenente la posizione {[52,56]} poiché incluse rispettivamente nella posizione [1,50] e [51,70] del nodo padre A. Allo stesso modo il vettore di posizioni del nodo C contiene nel primo gruppo C_1 la posizione {[41,49]} e nel secondo gruppo C_2 le posizioni {[57,63],[64,69]} anch'esse incluse rispettivamente nella posizione [1,50] e [51,70] del nodo padre A. In conclusione i gruppi legati tra loro sono B_1 e C_1, e B_2 e C_2.
Questi legami impediscono di considerare in modo indipendente le posizioni di figli diversi. Quindi, lo scopo di questa fase è quello di riempire la tabella dei risultati, introdotta nel paragrafo **[3.1.4]** sfruttando le informazioni sui *Pivot* contenute nella tabella hash, introdotta nel paragrafo **[3.1.3]**, e tendendo conto di tali legami, rendendo così possibile la fase successiva di visualizzazione dei risultati.
Il riempimento della tabella richiede un'ulteriore visita bottom-up. Durante tale visita in ogni nodo viene costruita una tabella temporanea per ogni posizione del padre contenente i risultati relativi a quella posizione. La tabella costruita nei nodi foglia ha un numero di righe pari al numero di posizioni contenute nell'intervallo associato alla posizione corrente del nodo padre ed un numero di colonne fissato a priori e pari al doppio del numero di *Pivot*. Questo perché per ogni *Pivot* viene memorizzata la posizione di apertura e chiusura della finestra di testo da visualizzare. Ogni posizione viene inserita nella corrispondente riga e nella colonna *j*-esima dove *j* è l'indice del *Pivot*, recuperato dalla tabella hash **[3.1.3]**, associato al nodo. Procedendo con la visita in ogni nodo interno si fondono le tabelle dei nodi figli per generarne una nuova con un numero di righe pari al prodotto tra i numeri di righe delle tabelle temporanee dei nodi figli. Le righe della nuova tabella vengono riempite combinando tra loro le righe di ciascuna tabella dei figli (vedi figura 3.8). Inoltre, se il nodo corrente è un *Pivot*, la posizione corrente per la quale si stanno calcolando i risultati si inserisce nella colonna corrispondente all'indice associato al nodo *Pivot*. Un caso particolare si verifica in presenza del nodo speciale `<xml_or>`. In tal caso la tabella del nodo si ottiene facendo l'unione delle righe delle tabelle dei nodi figli.

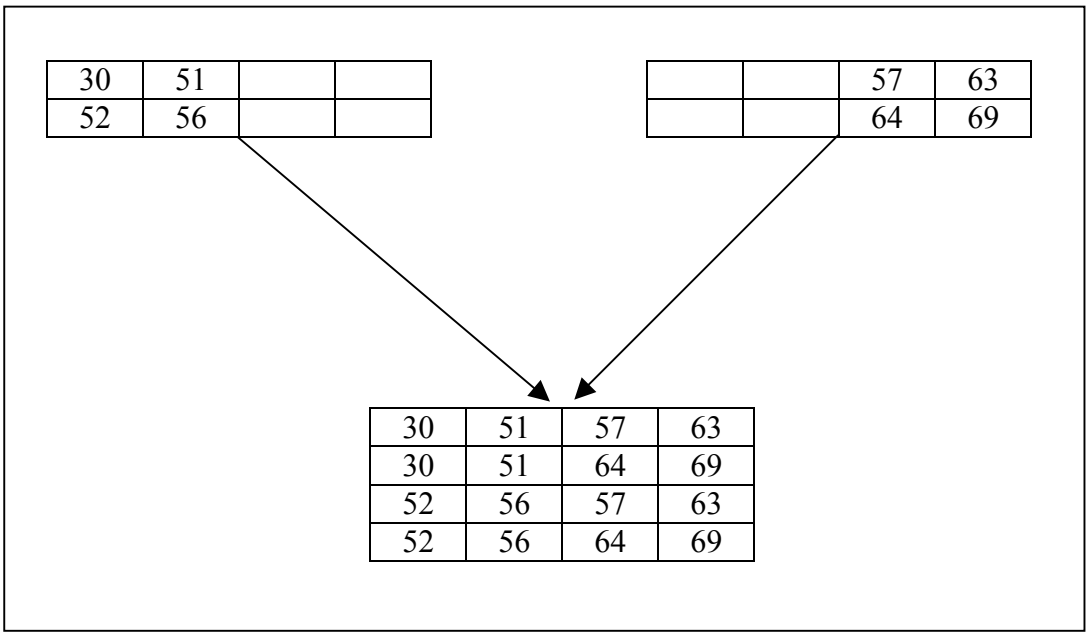


Figura 3.8 – Esempio di combinazione di righe

Alla fine della visita la tabella costruita nel nodo radice sarà la tabella dei risultati finali.
Lo spazio occupato dalla tabella è proporzionale al prodotto del numero di risultati che soddisfano l'interrogazione e delnumero di *Pivot* presenti nella clausola **RETURN**.
La presenza della clausola **RANGE** nell'interrogazione potrebbe ridurre il numero di risultati da visualizzare. Questo però non implica una riduzione proporzionale anche del tempo di risposta e dello spazio occupato dalla struttura dati poiché i risultati vengono comunque generati tutti. Il notevole aumento di complessità che si sarebbe avuto facendo costruire solamente i risultati richiesti ci ha portato a rinunciare a tale funzionalità. L'aumento di complessità è dovuto all'esistenza di una relazione tra le posizioni di nodi differenti, che a priori, senza aver effettuato la completa visita dell'albero, non permette di stabilire qual è l'i-esimo risultato. Ma come già osservato, la completa visita dell'albero ha generato l'intera tabella dei risultati.

3.4.1 Memorizzazione dei risultati nel file system

La libreria XCDEv2.0 per consentire una visualizzazione dei risultati di un'interrogazione anche in un momento successivo a quello di risoluzione della stessa, consentendo la memorizzazione di questi in un file. In tale file vengono memorizzate, secondo lo schema in figura 3.9, la tabella dei risultati e la tabella hash. Essendo la tabella dei risultati costituita da numeri interi, abbiamo scelto di memorizzare tali informazioni in maniera compressa. Il file è in formato binario e la tabella è memorizzata per righe, e al suo interno le righe della tabella sono memorizzate in maniera sequenziale. Per garantire una maggiore efficienza nel recupero di un sottoinsieme dei risultati dal file ogni dieci righe (costituenti un *blocco*) viene memorizzato nel file un indice del *blocco*, cioè un puntatore alla sua prima riga. In testa al file viene memorizzata la dimensione della tabella dei risultati e il numero di variabili presenti nella clausola **RETURN**. Seguono il vettore dei puntatori ai blocchi, e per ogni risultato la lunghezza della lista delle posizioni e la lista delle posizioni fisiche memorizzate con il *Continuation bit*, e infine le informazioni relative alla tabella hash.

Per problemi legati alla codifica dello zero i valori codificati con il *Continuation bit* vengono incrementati di uno. In sede di decompressione i valori saranno opportunamente decrementati.

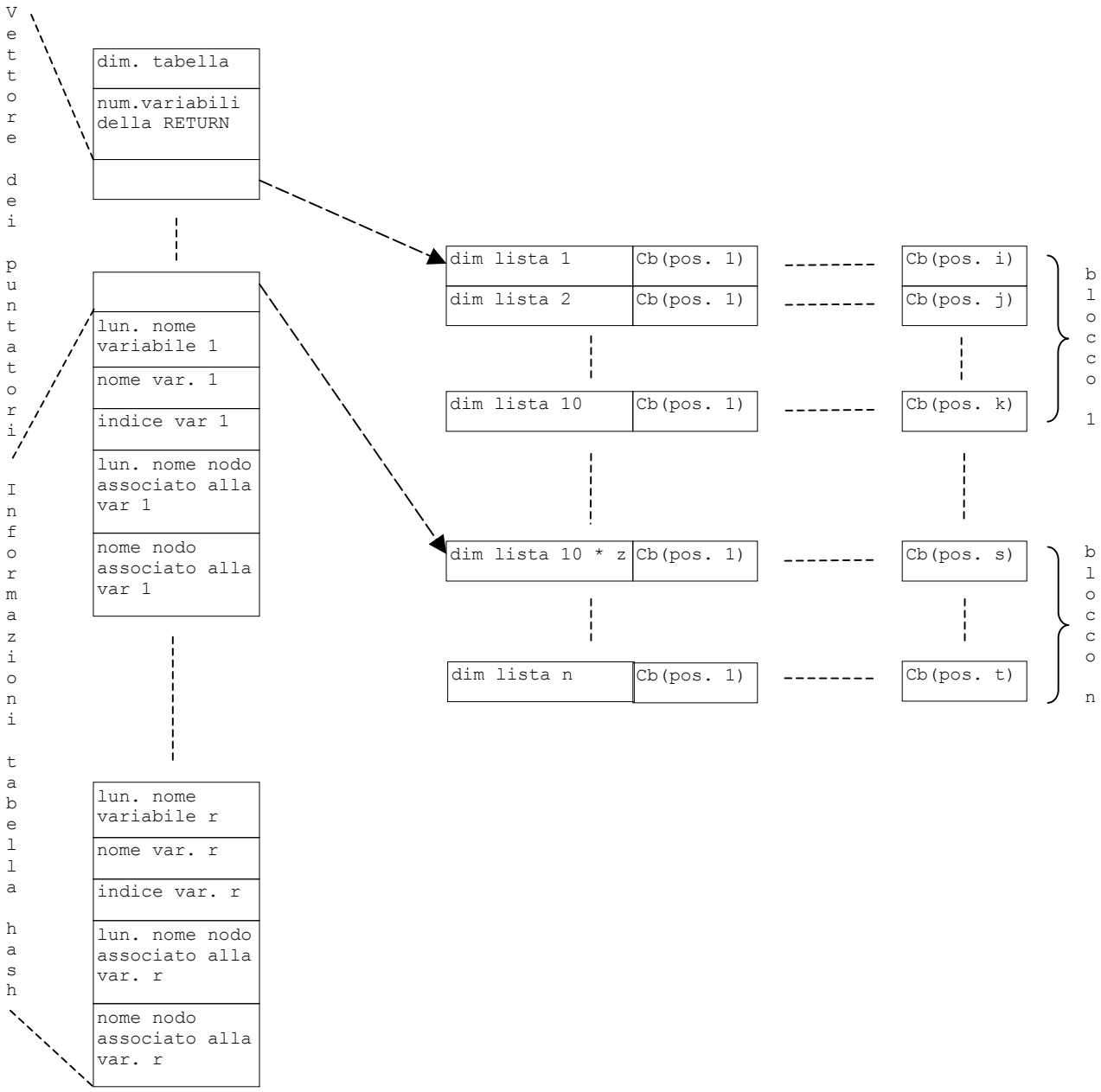


Figura 3.9 – Struttura del file che contiene i risultati

3.4.2 Recupero dei risultati dal file system

La possibilità di memorizzare i risultati di un'interrogazione in un file introduce la necessità di disporre di un meccanismo di recupero degli stessi.
La libreria offre due tipi di recupero: un recupero totale dei risultati ed uno parziale. Il recupero totale prevede la lettura dal file di tutti i risultati presenti. Quello parziale, invece, permette di recuperare un intervallo specifico di risultati. Quest'ultimo sfrutta la presenza dei puntatori ai blocchi per evitare la scansione sequenziale dei risultati.

3.5 Visualizzazione dei risultati

A partire dalla tabella dei risultati si può procedere con la fase di visualizzazione. In particolare, per ogni riga della tabella che corrisponde ad un determinato risultato si effettua la visita dell'albero associato alla clausola **RETURN**. Durante tale visita se il nome del nodo corrente non è del tipo "$nome_Pivot" o l'elemento speciale `<xml_context>`, allora il nome del nodo con la sua eventuale lista di attributi viene visualizzato nel risultato. Al contrario, se il nome del nodo è un *nome di Pivot*, occorre accedere alla riga corrente della tabella dei risultati e recuperare, dalla colonna *i*-esima dove *i* è l'indice associato alla variabile $Pivot, le posizioni fisiche di apertura e chiusura dello snippet da estrarre. Estrarre uno snippet significa ricostruire la porzione del documento XML contenente il risultato.
Nella clausola **RETURN** è prevista la possibilità di inserire l'elemento speciale `<xml_context>` che influisce sul contesto dello snippet da estrarre. Si possono regolare la dimensione, la direzione e la visualizzazione o meno dei tag nello snippet. Tali funzionalità sono state illustrate nel paragrafo **[2.2.4]**. Esistono casi in cui specificare una dimensione maggiore di zero dello snippet, comporta avere un risultato che non è ben-formato. Poiché, il risultato di un'interrogazione la maggior parte delle volte è soggetto ad ulteriori elaborazioni, abbiamo deciso di renderlo comunque ben-formato completandolo con gli elementi di tipo "TAG" aperti e non chiusi e chiusi e non aperti.
Inoltre, non è presente nessun meccanismo che eviti la presenza, nel risultato, di snippet che abbiano parti in comune a causa del fatto che l'utente può specificare una dimensione arbitraria di essi.
Per motivi di debugging, è data all'utente la possibilità di visualizzare, le posizioni di apertura e chiusura di tali snippet anzicchè il loro contenuto. Da notare che l'utilizzo di tale funzionalità è incompatibile con la presenza dell'elemento speciale `<xml_context>`.

Capitolo 4

Le funzioni di XCDEv2.0

In questo capitolo completeremo la descrizione del Query Engine descrivendo le operazioni che devono essere effettuate per eseguire un'interrogazione specificandone le caratteristiche algoritmiche e la realizzazione. In particolare, all'inizio del capitolo descriveremo la 'console query' della libreria, a cui hanno accesso le funzioni. Successivamente descriveremo l'API che rappresenta l'interfaccia della libreria verso tutte le applicazioni client e concluderemo con la specifica della struttura e dell'implementazione delle varie funzioni.

4.1 La 'console query'

La *console query* è una struttura dati che risiede in memoria centrale, a cui accedono la maggior parte delle funzioni di libreria. Essa consente di mantenere informazioni riguardanti l'interrogazione, i suoi risultati ed il suo stato di avanzamento (se è stata analizzata, eseguita, etc). La *console query* è implementata in modo da:

- *Semplificare la sintassi delle funzioni*: così da ridurre il numero di parametri da passare a queste.

- *Facilitare l'implementazione delle operazioni della libreria*: al fine di rendere il sistema di semplice utilizzo al programmatore ed estendibile.

4.1.1 Struttura della console query

La *console query* può essere suddivisa da un punto di vista logico in diverse classi di informazione, ognuna riguardante un particolare aspetto dell'interrogazione. Più precisante la prima classe contiene l'espressione della clausola **`SELECT`** in `select_string`, il vettore con i nomi dei documenti XML sui quali eseguire la ricerca in `file_vector`, il numero di tali file in `number_of_files`, l'espressione XML contenuta nella clausola **`RETURN`** in `return_string` e l'intervallo dei risultati da restituire in `first_range` e `last_range`.

Seguono le informazioni sui risultati, come l'albero associato alla clausola **SELECT** in `select_tree`, il vettore di puntatori agli alberi contenenti i risultati di ciascun file in `trees_vector`, il numero di tali alberi in `number_of_trees`, l'albero associato alla clausola **RETURN** in `return_tree`, e il puntatore alla tabella hash in `hash_table`.
Infine essa contiene le informazioni riguardo lo stato dell'interrogazione in opportuni flag come, `XCDE_query_parsed` che indica se l'interrogazione è stata analizzata e `XCDE_query_executed` che indica se l'interrogazione è stata eseguita.

```
typedef struct XCDE_Console_Query_Type_Struct {

char *select_string;
char **file_vector;
int number_of_files;
char *return_string;
int first_range;
int last_range;

query_tree_node *select_tree;
query_tree_node **trees_vector;
int number_of_trees;
query_tree_node *return_tree;
HHash_table *hash_table;

unsigned char XCDE_query_parsed;
unsigned char XCDE_query_executed;

} XCDE_Console_Query_Type;
```

4.2 L'API

L'API rappresenta l'interfaccia tra le applicazioni *client* e il nucleo della libreria XCDE2. Essa è costituita da un insieme di funzioni C. La maggior parte delle funzioni hanno fra i loro parametri la *console query* a cui si accede per riferimento. Per tutte queste funzioni è necessario che tale riferimento sia diverso dal valore 'NULL'. Le funzioni dell'API possono essere raggruppate in base alle funzionalità che realizzano nel modo seguente:

- funzioni ad alto livello;
- funzioni per l'esecuzione di un'interrogazione;
- funzioni per operare sui risultati;

L'intestazione di una generica funzione è la seguente:

***int* XCDE_Nomeoperazione (parametro1,**
.
.
parametroN);

e restituisce un codice intero che indica l'esito dell'operazione. I risultati sono restituiti attraverso delle variabili passate per riferimento. Nel caso in cui si richieda la memorizzazione del risultato all'interno di un vettore, la sua allocazione viene eseguita

dalla funzione medesima, mentre è compito dell'applicazione *client* liberare la memoria una volta che questa non sia più utilizzata.

Nelle sezioni seguenti forniremo una descrizione generale di ogni funzione e della sua implementazione, rimandando per una visione più dettagliata all'Appendice A.

4.2.1 Funzioni ad alto livello

Di questo insieme fa parte solamente la funzione **XCDE_Search**. Essa accetta come parametri la stringa contenente l'interrogazione da eseguire e il flag per la gestione della visualizzazione delle posizioni. La stringa è quella contenuta nella clausola **`SELECT`** ed il flag vale '1' se si vogliono visualizzare le posizioni nel file compresso e '0' per la visualizzazione testuale delle occorrenze stesse.
La funzione esegue l'interrogazione e restituisce, sullo standard output, i risultati.

4.2.2 Funzioni per l'esecuzione di un'interrogazione

Queste sono le funzioni che consentono all'applicazione client di gestire le fasi di esecuzione di un'interrogazione dal parsing alla costruzione della tabella dei risultati.
Il parsing viene eseguito dalla funzione **XCDE_Parse_Query_String** che analizzando la stringa che contiene l'interrogazione individua ed estrae le informazioni necessarie a riempire alcuni campi della *console query* e setta il flag `XCDE_query_parsed` a 1. A questo punto l'interrogazione può essere eseguita dalla **XCDE_Execute_Query** che accedendo alla *console query*, passata come parametro, fa un numero di copie dell*'albero della select* pari al numero di file da interrogare, e riempie ciascuna copia con le posizioni, nel corrispondente file compresso, che soddisfano la ricerca. La chiamata di questa funzione richiede che nella *console query* sia attivato il flag `XCDE_query_parsed`. Infine, nella *console query* viene settato a 1 il flag `XCDE_query_executed`. L'ultima funzione di questo gruppo di API è **XCDE_Compute_Table_Results** che permette di calcolare la tabella dei risultati relativi ad un file a partire dal suo indice nel vettore `file_vector` e dalla *console query*. La funzione restituisce la tabella dei risultati ed un intro che reppresenta il numero dei risultati ottenuti. Per l'esecuzione di questa funzione si richiede che nella *console query* sia attivato il flag `XCDE_query_executed`.

4.2.3 Funzioni per operare sui risultati di un'interrogazione

Le funzioni descritte in questo paragrafo gestiscono le operazioni relative alla visualizzazione dei risultati, alla memorizzazione delle tabelle dei risultati e al loro recupero.
La prima funzione che descriviamoè la **XCDE_Get_Number_Results** che restituisce il numero di risultati di un'interrogazione a partire dall'albero generato dalla funzione **XCDE_Execute_Query**.
La funzione che gestisce la visualizzazione è la **XCDE_View_Result.** Essa accetta come parametri l'indice del file al quale si riferiscono i risultati, la tabella dei risultati, il flag per la gestione del tipo di visualizzazione e la *console query*. La funzione richiede che nella *console query* sia attivato il flag `XCDE_query_parsed`. La memorizzazione della tabella dei risultati nel file system ed il suo recupero sono gestiti rispettivamente dalle funzioni di libreria **XCDE_Write_Result_File** e **XCDE_Read_Result_File**. La prima prende come parametri la tabella dei risultati, il nome del file nel quale memorizzare la tabella e la *console query*. I requisiti di tale funzione sono gli stessi della **XCDE_View_Result**. La

seconda permette di ricostruire in memoria la tabella dei risultati e la tabella hash a partire dal nome del file usato dalla **XCDE_Write_Result_File** per la memorizzazione. Utilizzando la funzione **XCDE_Read_Result_File** il recupero dei dati è totale, cioè vengono recuperati tutti i risultati contenuti nel file. Per un recupero parziale abbiamo previsto la funzione **XCDE_Read_Range_Result_File** che è del tutto simile alla **XCDE_Read_Result_File** con la differenza che è data all'utente la possibilità di scegliere l'intervallo dei dati da recuperare.

Capitolo 5

Conclusioni

La libreria XCDEv2.0 rappresenta uno strumento innovativo per l'interrogazione di documenti XML. Una delle sue caratteristiche principali è data dalla facilità di formulazione delle interrogazioni grazie alla sintassi XML-based. Questo permette anche utenti poco esperti di esprimere facilmente interrogazioni complesse.
Dal punto di vista delle prestazioni la libreria risulta efficiente poiché opera direttamente sul documento compresso sfruttando gli indici forniti dalla versione 1.0 della libreria e decomprimendo, se l'interrogazione trova dei risultati, solo quella parte del documento che li contiene. La libreria sfrutta inoltre altre strutture dati fornite da XCDEv1.0 che consentono di realizzare interrogazioni strutturali in maniera semplice e veloce, permettendo in più di effettuare operazioni sofisticate come l'estrazione di *snippet ben-formati*, non supportate finora da nessun altro linguaggio.
Un ulteriore vantaggio in termini di efficienza è il linguaggio con cui la libreria è stata scritta, il linguaggio C. Di seguito illustreremo alcuni spunti di ricerca e sviluppo che meritano una maggiore attenzione per il futuro di questa libreria.
XCDEv2.0 non è stata pensata per essere utilizzata direttamente dagli utenti, nonostante ciò sia possibile, ma come punto di partenza per nuove applicazioni che agevolano ulteriormente l'utente nella ricerca.
Una possibile applicazione prevede lo sviluppo di un'interfaccia *web* così da consentire all'utente di formulare *query* sui *tag* in modo guidato attraverso opportune interfacce grafiche, sfruttando eventualmente la *DTD*. In questo modo non si avrebbe la necessità di conoscere la DTD del documento da interrogare per realizzare interrogazioni complesse sui documenti. Inoltre si potrebbe prevedere la definizione di un CSS (*Cascading Style Sheet)* **[CSS]** per visualizzare in modo più elegante e chiaro gli *snippets,* sfruttando così la loro proprietà di essere *ben-formati.*
Il gruppo di lavoro della W3C di Pisa è interessato a quest'ultima applicazione ed è già partito il progetto per la sua realizzazione.
Infine indichiamo un interessante spunto per la realizzazione di un sistema di gestione dinamica di collezioni XML che sfrutta le proprietà del linguaggio e della libreria XCDEv2.0. Prevediamo, nel prossimo futuro, di *raggruppare* più documenti in un solo documento *virtuale* il quale, mediante opportuni *tag*, preserva l'individualità dei suoi documenti costituenti. A questo punto la libreria potrebbe indicizzare una collezione di documenti virtuali così da consentire la gestione semplice e veloce delle operazioni di modifica di documenti. Questo approccio in parte ricorda quello adottato nel sistema *Glimpse* **[Manber]** e descritto in letteratura con il nome di *block addressing indices*

[Baeza-Yates]. La differenza sostanziale consiste in questo caso nello sfruttare pienamente la flessibilità dell'XML per definire il concetto di *blocchi di indicizzazione* in modo indipendente dal sistema di ricerca.

Bibliografia

[Applications] http://www.oasis-open.org/cover/xml.html#applications
Elenco delle applicazioni aggiornate di XML,The XML Cover Page, By Robin Cover.

[Baeza-Yates] R. Baeza-Yates, G. Navarro: *Block-addressing indices for approximate text retrieval*. In Proceedings of the ACM CIKM'97, pp. 1-8, 1997.

[CAML] X.Leroy. *The Objective Caml System, release 2.04, Documentation and user's manual.* Institut National de Recherche en informatique et en Automatique, Nov 1999.

[CIBIT] http://cibit.humnet.unipi.it/home_index.htm
Pagina ufficiale del Centro Interuniversitario Biblioteca Italiana Telematica.

[Clark] http://www.jclark.com/xml/expat.html
Pagina ufficiale di Expat – Il parser XML di J. Clark utilizzato in XCDE.

[CSS] http://www.w3.org/TR/REC-CSS1
Specifica ufficiale del Cascading Style Sheets, livello 1, W3C Recommendation, 17 december, 1996, revised 11 january 1999
Sito della World Wide Web Organization.

http://www.w3.org/TR/REC-CSS2
Specifica ufficiale del Cascading Style Sheets,livello 2, W3C Recommendation ,17 may, 1998
Sito della World Wide Web Organization.

[CXQuery] Y. Chen, P. Revesz : *CXQuery: A Novel XML Query Language.*

[Functions and Operators] *XQuery 1.0 and XPath 2.0 Functions and Operators*, W3C Working Draft (16 August 2002), see http://www.w3.org/TR/xquery-operators.

[Galax] http://db.bell-labs.com/galax
Implementazione open-source di XQuery 1.0.

[Grep] http://www.gnu.org/directory/grep/html
Pagina ufficiale di Grep, Egrep e Fgrep ver.2.5, Free Software Foundation.

[Lorel] S. Abiteboul, D. Quass, J. McHugh, J. Widom, e J. Wiener : *The Lorel Query Language for Semistructured* Data. Journal of Digital Libraries, pp. 68-88, April, 1997.

[Manber] U. Manber, S. Wu : *GLIMPSE – A tool to search through entire file systems.* In Proceedings of the USENIX Winter 1994 Technical Conference,
pp. 23-32, 1994.

[Navarro97] G. Navarro, R. A. Baeza-Yates: *Proximal Nodes - A Model to Query Document Databases by Content and Structure*. Information Systems, vol. 15, pp. 400-435, 1997.

[OQL] R.G.Cattell. *Teìhe Object Database Standard: ADGM 2.0.* Morgan Kaufmann, 1997.

[OEM] Y.Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: *A mediation system based on declarative specifications*. In Proceedings of the Internation Conference of Data Engineering, (ICDE '96), pages 132-141, 1996.

[QEXO] http://www.gnu.org/software/qexo
Specifica dell'implementazione GNU Kawa del linguaggio XQuery.

[QUILT] D. Chamberlin, J. Robie, e D. Florescu: *Quilt : An XML query Language for hererogeneous data sources,* WebDB (Informal Proceedings), 2000, pp.53-62.

[Sadakane] K. Sadakane, H. Imai: *On k-word proximity search*. IPSJ SIG Notes 99-AL-68, 1999.

[SAX] http://www.saxproject.org
Sito ufficiale del progetto SAX contenente la definizione del SAX 2.0, David Megginson, Marzo 2002.

[SGML] ISO. Information processing – text and office systems – standard generalized markup language (SGML), 1986.

[SQL] *Information technology – database languages – SQL Multimedia and Application Packages – Part 2:Full text,* International Organization For Standardization, 2000.

[TQL] G. Conforti, G.Ghelli, A. Albano, D. Colazzzo, P. Manghi, C. Sartiani : *The Query Language TQL.* Dipartimento di Informatica, Università di Pisa, Italy 2002.

[XCDEv1.0] http://butirro.di.unipi.it/~ferrax/xcde/xcdelib.html
Specifica ufficiale della libreria XCDE versione 1.0 per comprimere e indicizzare file XML, realizzata da Paolo Ferragina e Andrea Mastroianni, 2002.

[XDuce] H. Hosoya e B.C. Pierce: *XDuce: an XML processing language*. Preliminary report, Dec 1999.

[XML] http://www.w3.org/TR/REC-xml
Specifica ufficiale dell' eXtensible Markup Language Version 1.0 (Second Edition) , W3C Recommendation, 6 october 2000.
Sito della World Wide Web Organization.

[XML-GL] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi e L. Tanca: *XML-GL: A Graphical Language for Querying and Reshaping XML Documents.* Politecnico di Milano, dipartimento di Elettronica e Informazione.

[XML-QL] http://www.w3.org/TR/NOTE-xml-ql
A. Deutsch, M. Fernandez, D. Florescu, A. Levy e D.Suciu: *XML-QL: A Query Language for XML.* Submission to the W3C Consortium 19 August 1998.

[XML-SQL] T. Pankowski. *XML-SQL: An XML Query Language Based on SQL and Path Tables.* Chair of Control, Robotics and Science, Poznan University of Technology.

[XPath] http://www.w3.org/TR/xpath
Specifica ufficiale del formalismo di indirizzamento per documenti XML
XPath versione 1.0, W3C Recommendation, 16 november, 1999
Sito della World Wide Web Organization.

[XPathLog] W. May, Integration of xml data in xpathlog, *CAiSE Workshop Data Integration over the Web (DIWeb'01)* (Interlaken, Switzerland), 4–5 2001.

[XQL] M. Fernandez, J. Simèon, P. Walder : *XML Query Language: Experiences and Exemplars.*

http://www.texcel.no/whitepapers/xql-design.html
J. Robie. *The design of XQL,* 1999.

http://metalab.unc.edu/xql/xql-proposal.html
J. Robie, editor. *XQL '99 Proposal.*

[Xquery 1.0] http://www.w3.org/TR/xquery
Specifica ufficiale del linguaggio di interrogazione per documenti XML
XQuery versione 1.0, W3C Recommendation, 20 december, 2001
Sito della World Wide Web Organization.

http://www.w3.org/TR/xmlquery-full-text-use-cases/#XQuery

[YAT_L] S. Cluet e J. Simèon : *YAT_L: a Functional and Declarative Language for XML.* Draft manuscript, Mar. 2000.

[Young] Michael J. Young
XML – Step by Step
Microsoft Press, Redmond, Washington (USA), 2000.

[W3C Req] http://www.w3.org/TR/xquery-full-text-requirements
Specifica dei requisiti del linguaggio di interrogazione XQuery.

[W3C Full Text] http://www.w3.org/TR/xmlquery-full-text-use-cases
Specifica dei casi d'uso per interrogazioni full-text di un sottoinsieme di XML Query e XPath.

[W3C XML Query] www.w3.org/XML/Query
Sito ufficiale del World Wide Web Consortium.

Indice delle figure

Figura 1.1 - Esempio di documento XML

Figura 1.2 - Regole sintattiche per i nomi di elementi XML

Figura 1.3 - Entità predefinite di XML

Figura 1.4 - Regole sintattiche per i valori degli attributi XML

Figura 1.5a - Espressione YAT_L per la creazione di un albero

Figura 1.5b - Espressione XML

Figura 1.6 - Esempio di uso dell'iteratore match

Figura 1.7 - Esempio di uso dell'operatore case

Figura 1.8 - "Coercion" per operatori di confronto

Figura 1.9a - Interrogazione in Lorel

Figura 1.9b - Interrogazione in OQL

Figura 1.10 - Struttura del sistema LORE

Figura 1.11 - DTD in CXQuery

Figura 1.12 - Schema a livelli della libreria XCDEv1.0

Figura 2.1 - Formato generale del risultato di un'interrogazione

Figura 2.2 - Tipi di ricerca sul valore degli attributi

Figura 2.3 - Valori dell'attributo speciale xml_view

Figura 2.4 - Grammatica dell'XCDE Query Language

Figura 2.5a - Uso errato di 'xml_dist'

Figura 2.5b - Uso corretto di 'xml_dist'

Figura 3.1 – Esempio di calcolo del vettore number_results_vector

Figura 3.2 – Esempio di documento XML

Figura 3.3 – Esempio di albero della select prima delle visite

Figura 3.4 – Esempio di albero dopo la visita bottom-up

Figura 3.5 – Esempio di albero dopo la visita top-down

Figura 3.6 – Esempio di posizioni valide

Figura 3.7 – Esempio di partizionamento in gruppi delle posizioni

Figura 3.8 – Esempio di combinazione di righe

Figura 3.9 – Struttura del file che contiene i risultati

APPENDICE A – L'API di XCDEv2.0

La libreria XCDEv2.0 possiede un'API formata dalle funzioni di libreria di XCDEv1.0 alle quali si vanno ad aggiungere nove nuove funzioni C. Esse richiedono l'inclusione del file d'intestazione *xcde_libsearch.h*, all'interno del quale sono definiti diversi *tipi* e diverse *costanti* utilizzate dalle funzioni della libreria. Esso deve essere inserito in tutti i programmi che la utilizzano.
Al suo interno è definita la *console query* attraverso il tipo `XCDE_Console_Query_Type` **[4.1]**. La *console query* è passata per riferimento alle funzioni della libreria e contiene tutte le informazioni necessarie per le interrogazioni. Un altro tipo definito è `XCDE_Query_Tree_Node` **[3.1.2]**.
Per quanto riguarda le costanti, esse sono quelle definite dal kernel XCDEv1.0.
Ogni programma che utilizzi la libreria XCDEv2.0 deve rispettare la seguente struttura del codice:

```
// file di intestazione
#include "xcde_libsearch.h"

int main(int argc, char *argv[])
{ XCDE_Console_Query_Type console ; // variabile di tipo console
      .
      .
      .
   int result=0 ; // variabile per gli eventuali codici di errore

      .
      .
      .

      return(1) ;
}
```

La variabile `console` deve essere passata per riferimento alle funzioni che la richiedono in *input*. I parametri che non devono essere modificati sono passati per valore. Sono passate per riferimento le variabili che conterranno i risultati della funzione.
Per quanto riguarda gli eventuali errori esistono due categorie. La prima comprende tutti gli errori di sintassi dell'interrogazione, mentre la seconda comprende tutti gli altri tipi di errori, ad esempio, problemi nell'allocazione della memoria, problemi con i parametri delle funzioni etc. Tutte le funzioni nelle quali si verifica un errore facente parte della prima categoria restituiscono il valore –2. Quelle nelle quali, invece, si verifica un errore appartenente alla seconda categoria restituiscono valore –1. Tutte le funzioni restituiscono un valore non negativo quando non si verifica alcun errore.
Elenchiamo di seguito, le nuove funzioni della libreria XCDEv2.0, seguendo la suddivisione utilizzata nella sezione **[4.2]**.

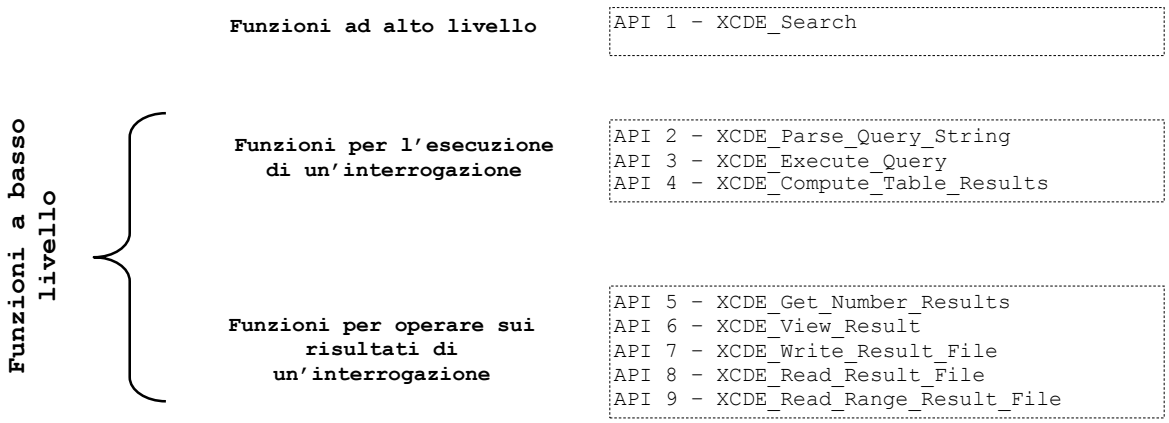


Figura A.1 – Funzioni API

Funzioni ad alto livello

Di questo gruppo fa parte una sola funzione che permette, data un'interrogazione, di visualizzare i risultati ottenuti dalla sua esecuzione.

API NR.1 - *XCDE_Search*

```
// IN Query_String: stringa contenente l'interrogazione da eseguire;

// IN Flag_Position: flag per la gestione delle posizioni;

int XCDE_Search(char *Query_String,
                unsigned char Flag_Position);
```

Esegue un'interrogazione. Prende come parametri la stringa contenente l'interrogazione da eseguire ed un flag che permette di specificare se, dell'eventuali occorrenze trovate, visualizzare la posizione all'interno del file compresso (inserendo come valore del flag 1) o gli snippet estratti (specificando come valore del flag 0).

Funzioni a basso livello

Di questo gruppo fanno parte le funzioni che consentono di gestire le varie fasi attraversate da un'interrogazione, dal parsing alla visualizzazione dei risultati. Questo gruppo di funzioni può essere diviso in due sottogruppi il primo, *funzioni per l'esecuzione di un'interrogazione*, permette di gestire l'esecuzione dell'interrogazione, mentre il secondo, *funzioni per operare sui risultati di un'interrogazione*, permette di gestire i risultati e la loro visualizzazione. Le API numero 7, 8 e 9 sono state pensate per poter memorizzare all'interno del file system i risultati di un'interrogazione in modo da poterli recuperare in un secondo momento. L'API numero 8 permette di ricostruire l'intera tabella dei risultati, mentre l'API numero 9 permette di estrarre solo un sottoinsieme di risultati.

Funzioni per l'esecuzione di un'interrogazione

API NR.2 - *XCDE_Parse_Query_String*

```
// IN Query_String: stringa contenente l'interrogazione da eseguire;
```

```
// OUT Console_Query: indirizzo della struttura 'console query';

int XCDE_Parse_Query_String (char *Query_String,
                             XCDE_Console_Query_Type *Console_Query);
```

Esegue il parsing dell'interrogazione contenuta nel primo parametro inserendo nella struttura dati `Console_Query` le informazioni riguardanti l'interrogazione e settando opportunamente i flag relativi allo stato dell'interrogazione **[4.1]**. Prende come parametri la stringa contenente l'interrogazione da eseguire ed il puntatore alla `Console_Query`. Il valore di ritorno della funzione è –2 in presenza di errori di sintassi nell'interrogazione, -1 in presenza di altri errori diversi da quelli di sintassi e 1 altrimenti.

API NR.3 - *XCDE_Execute_Query*

```
// IN Console_Query: indirizzo della struttura 'console query';

int XCDE_Execute_Query (XCDE_Console_Query_Type *Console_Query);
```

Esegue l'interrogazione le cui informazioni sono contenute nell'unico parametro `Console_Query`. Ciò significa che in precedenza bisogna aver richiamato la funzione `XCDE_Parse_Query_String`. L'effetto della chiamata di questa funzione è quello di trovare tutti i risultati dell'interrogazione inserendone le informazioni relative nella struttura dati `Console_Query` **[4.1].** Il valore di ritorno della funzione è –2 in presenza di errori di sintassi nell'interrogazione, -1 in presenza di altri errori diversi da quelli di sintassi e 1 altrimenti.

API NR.4 - *XCDE_Compute_Table_Results*

```
// IN Index_Tree: indice del documento del quale calcolare la table
                  results;

// OUT Table_Results: indirizzo della variabile che conterrà la tabella
                      dei risultati;

// OUT Size_Table_Results: indirizzo della variabile che conterrà il
                           numero di risultati trovati;

// IN Console_Query: indirizzo della struttura 'console query';

int XCDE_Compute_Table_Results (int Index_Tree,
                                XCDE_IL_Couple_Type ***Table_Results,
                                int *Size_Table_Results,
                                XCDE_Console_Query_Type *Console_Query);
```

Riempie la tabella dei risultati sfruttando le informaioni contenute nella `Console_Query`. Ciò significa che in precedenza bisogna aver richiamato la funzione `XCDE_Execute_Query`. Il valore di ritorno della funzione è –2 in presenza di errori di sintassi nell'interrogazione, -1 in presenza di altri errori diversi da quelli di sintassi e 1 altrimenti.

Funzioni per operare sui risultati di un'interrogazione

API NR.5 - *XCDE_Get_Number_Results*

```
// IN Tree: indirizzo dell'albero del quale calcolare il numero di
                  risultati associati;

// OUT Number_Results: indirizzo della variabile che conterrà il numero
                  di risultati;

// IN Console_Query: indirizzo della struttura 'console query';

int XCDE_Get_Number_Results (query_tree_node *Tree,
                             int *Number_Results,
                             XCDE_Console_Query_Type *Console_Query);
```

Calcola il numero dei risultati senza però costruire la tabella dei risultati. In precedenza bisogna aver richiamato la funzione `XCDE_Execute_Query`. Il valore di ritorno della funzione è -1 in presenza di errori, 1 altrimenti.

API NR.6 - *XCDE_View_Result*

```
// IN Index_File: inice del documento del quale visualizzare i
                  risultati;

// IN Table_Results: indirizzo della variabile che contiene la tabella
                  dei risultati;

// IN Size_Table_Results: indirizzo della variabile che contiene il
                  numero di risultati;

// IN Flag_Position: flag per la gestione delle posizioni;

// IN Console_Query: indirizzo della struttura 'console query';

int XCDE_View_Results (int Index_File,
                       XCDE_IL_Couple_Type **Table_Results,
                       int Size_Table_Results,
                       unsigned char Flag_Position,
                       XCDE_Console_Query_Type *Console_Query);
```

Visualizza i risultati contenuti nella `Table_Results` passata come prametro. Ciò significa che in precedenza bisogna aver richiamato la funzione `XCDE_Compute_Table_Results`. Il primo parametro è l'indice, all'interno del vettore dei nomi dei file presenti nella struttura dati `Console_Query`, del file al quale sono associati i risultati da visualizzare.
Il parametro `Flag_Position` se settato a 0 permette di visualizzare gli snippet associati alle occorrenze trovate, se settato a 1, invece, permette di visualizzare anzicchè lo snippet associato all'occorrenza la sua posizione all'interno del file compresso. Questa opzione è prevista per fini di debugging.
Il valore di ritorno della funzione è –2 in presenza di errori di sintassi nell'interrogazione, -1 in presenza di altri errori diversi da quelli di sintassi e 1 altrimenti.

API NR.7 - *XCDE_Write_Result_File*

```
// IN Table_Results: indirizzo della variabile che contiene la tabella
```

```
                  dei risultati;

// IN Size_Table_Results: indirizzo della variabile che contiene il
                  numero di risultati;

// IN filename: indirizzo della variabile che contiene il nome del file
                  nel quale memorizzare la tabella dei risultati. Al
                  nome del file inserito verrà aggiunto il suffiso
                  ".result";

// IN Console_Query: indirizzo della struttura 'console query';

int XCDE_View_Results (XCDE_IL_Couple_Type **Table_Results,
                       int Size_Table_Results,
                       char *filename,
                       XCDE_Console_Query_Type *Console_Query);
```

Memorizza nel file system il contenuto della `Table_Results` passata come primo parametro. Ciò significa che in precedenza bisogna aver richiamato la funzione `XCDE_Compute_Table_Results`. L'utilità di questa funzione è quella di poter memorizzare i risultati di un'interrogazione per poi utilizzarli, tutti o una parte, in un momento successivo. I primi due parametri sono, rispettivamente, la `Table_Results` da memorizzare ed il numero di risultati contenuti nella `Table_Results`. Il parametro `filename` è il prefisso del nome del file nel quale memorizzare la `Table_Results`. A questo nome verrà concatenato il suffisso ".result". La `Table_Results` viene memorizzata in maniera compressa.
Il valore di ritorno della funzione è -1 in presenza di errori, 1 altrimenti.

API NR.8 - *XCDE_Read_Result_File*

```
// IN filename: indirizzo della variabile che contiene il nome del file
                  dal quale recuperare la tabella dei risultati. Il nome
                  del file inserito non deve contenere il suffiso
                  ".result";

// OUT Table_Results: indirizzo della variabile che conterrà la tabella
                  dei risultati;

// OUT Size_Table_Results: indirizzo della variabile che conterrà il
                  numero di risultati;

// IN-OUT Console_Query: indirizzo della struttura 'console query';

int XCDE_Read_Result_File (char *filename,
                           XCDE_IL_Couple_Type ***Table_Results,
                           int *Size_Table_Results,
                           XCDE_Console_Query_Type *Console_Query);
```

Recupera dal file system il contenuto del file passato come primo parametro, inserendolo nella `Table_Results`. Nella `Console_Query`, inoltre, verranno inserite le informazioni relative al numero di variabili contenute nell'interrogazione. Nessun'altra informazione relativa all'interrogazione, ai suoi risultati o al suo stato viene modificata. Il parametro `filename` è il prefisso del nome del file dal quale recuperare la `Table_Results`. Questo nome non dovrà contenere il suffisso ".result" che sarà inserito automaticamente.
Il valore di ritorno della funzione è -1 in presenza di errori, 1 altrimenti.

API NR.9 - *XCDE_Read_Range_Result_File*

```
// IN filename: indirizzo della variabile che contiene il nome del file
                dal quale recuperare la tabella dei risultati. Il nome
                del file inserito non deve contenere il suffiso
                ".result";

// IN First_Range: estremo sinistro dell'intervallo dei risultati da
                recuperere;

// IN Last_Range: estremo destro dell'intervallo dei risultati da
                recuperere;

// OUT Table_Results: indirizzo della variabile che conterrà la tabella
                dei risultati;

// OUT Size_Table_Results: indirizzo della variabile che conterrà il
                numero di risultati;

// IN-OUT Console_Query: indirizzo della struttura 'console query';

int XCDE_Read_Range_Result_File (char *filename,
                                 int First_Range,
                                 int Last_Range,
                                 XCDE_IL_Couple_Type ***Table_Results,
                                 int *Size_Table_Results,
                                 XCDE_Console_Query_Type
                                 *Console_Query);
```

Recupera dal file system un sottoinsieme dei risultati contenuti nel file passato come primo parametro, inserendolo nella `Table_Results`. L'intervallo di risultati da recuperare è determinato dai due parametri interi `First_Range` e `Last_Range`. I due estremi sono considerati come facenti parte dell'intervallo. Nella `Console_Query`, invece, verranno inserite le informazioni relative al numero di variabili contenute nell'interrogazione. Nessun'altra informazione relativa all'interrogazione, ai suoi risultati o al suo stato viene modificata. Il parametro `filename` è il prefisso del nome del file dal quale recuperare la `Table_Results`. Questo nome non dovrà contenere il suffisso ".result" che sarà inserito automaticamente.
Il valore di ritorno della funzione è -1 in presenza di errori, 1 altrimenti.

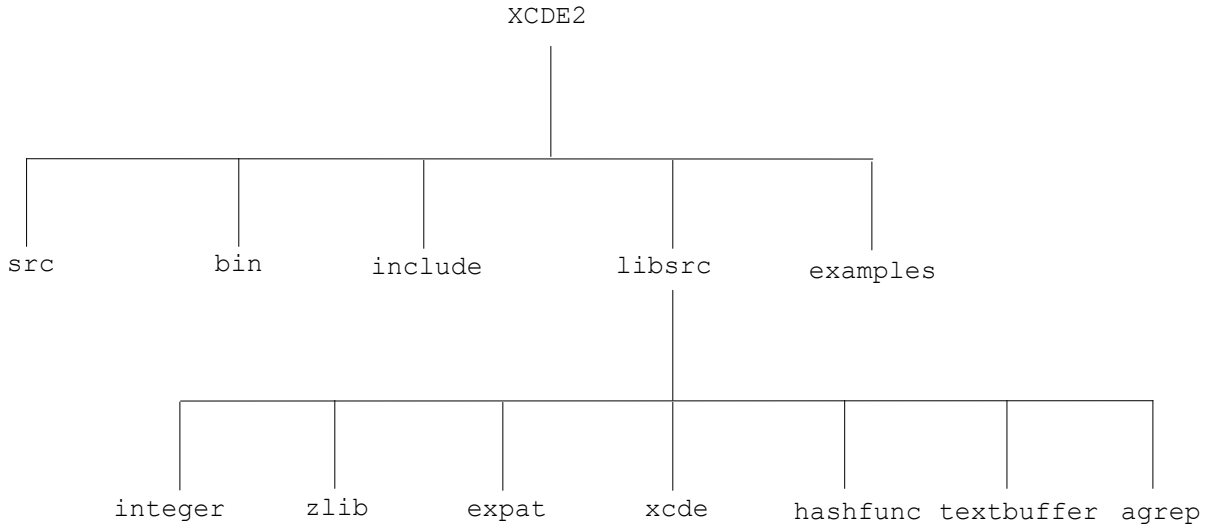
APPENDICE B – I file della libreria XCDE

La libreria XCDE2 è memorizzata per la sua distribuzione con il formato *tar* in un file chiamato *xcde2.tgz*. Dopo la sua decompressione con il comando

```
tar –zxvf xcde2.tgz
```

sarà creata la *directory* XCDE2 contenente i *file sorgenti* e gli *script* necessari per la compilazione.

L'albero della *directory* XCDE2 è il seguente:



il contenuto di ogni *directory* è illustrato di seguito:

`XCDE2/`	*script Makefile* per la compilazione dei sorgenti e altri *script* utili;
`XCDE2/src`	*file* sorgenti per la creazione dei comandi;
`XCDE2/bin/`	comandi eseguibili creati dopo la compilazione e utili per l'implementazione e l'utilizzo della libreria;
`XCDE2/include/`	*file* di intestazione necessari per la compilazione e l'uso della libreria;
`XCDE2/libsrc/`	*file* sorgenti della libreria XCDE e delle altre librerie usate per la sua implementazione;
`XCDE2/libsrc/zlib/`	*file* sorgenti della libreria Zlib;
`XCDE2/libsrc/expat/`	*file* sorgenti della libreria Expat;
`XCDE2/libsrc/integer/`	*file* sorgenti della libreria di compressione degli interi;

`XCDE2/libsrc/xcde/`	*file* sorgenti per l'implementazione delle API di XCDE2;
`XCDE2/libsrc/hashfunc/`	*file* sorgenti per l'implementazione della struttura dati *hash*;
`XCDE2/libsrc/textbuffer/`	*file* sorgenti per l'implementazione del *buffer* di testo;
`XCDE2/libsrc/agrep/`	*file* sorgenti del comando *agrep* utilizzato dalla libreria XCDE;

Per effettuare la compilazione è necessario invocare il comando seguente dalla *directory* XCDE2:

```
make –f Makefile
```

durante questa operazione si compileranno tutti i *file* sorgente presenti e si creeranno gli archivi formato *ar* della libreria XCDE e delle altre librerie utilizzate, oltre ai comandi necessari per l'utilizzo della libreria. Questi ultimi sono:

`XCDE2/bin/xcde_build`	necessario per la creazione dei *file* delle strutture dati;
`XCDE2/bin/xcde_extract`	effettua la decompressione di un *file* compresso;
`XCDE2/bin/xcde_inf`	utilizzato per ottenere informazioni su un *file* compresso e indicizzato;
`XCDE2/bin/xcde_search`	realizza le *query* sui *file* compressi e indicizzati;
`XCDE2/bin/xcde_search2`	realizza le *query* sui *file* compressi e indicizzati;
`XCDE2/bin/xcde_view`	estrae gli *snippet* dai *file* compressi e indicizzati;
`XCDE2/bin/agrep`	comando utilizzato, attraverso la chiamata *system*, per l'implementazione delle funzioni di ricerca sui dizionari della libreria XCDE;
`XCDE2/bin/_xcde_transf`	comando necessario per l'implementazione delle funzioni di ricerca della libreria XCDE, utilizzato congiuntamente al comando *agrep* precedente.

Il comando *xcde_build*

Il comando `xcde_build` crea le strutture dati necessarie per operare con la libreria XCDE. La sua sintassi è la seguente:

```
xcde_build [-v] [-c configfile] [-t] [-p] [-f] sourcefile
```

con le opzioni seguenti:

`-v` : visualizza i messaggi in maniera estesa e crea i *file* di *debug*;

`-c` : utilizza un *file* di configurazione differente rispetto a quello *standard*;

`-t` : crea la struttura *tagstruct* relativa la documento sorgente;

`-p` : non inserisce le *liste di proximity* nel *file* delle liste invertite;

Per *default* il comando non visualizza i messaggi in maniera estesa e non crea i *file* di *debug*, che contengono le informazioni sul documento indicizzato in formato testuale, in modo che possano essere consultate dall'utente. Ciò può essere effettuato attraverso il *flag* –v.
Il comando `xcde_build` deve necessariamente leggere un *file di configurazione* che è necessario al suo funzionamento. Esso è di *default* il *file* `xcde_build.conf`, che deve trovarsi nella stessa *directory* di `xcde_build`, ma un *file di configurazione diverso* può essere utilizzato facendo seguire al *flag* –c il suo nome. All'interno del *file di configurazione* si determinano, seguendo una sintassi che descriveremo in seguito, i caratteri e le entità considerati separatori, distinguendo fra quelli per le parole e quelli per il valore degli attributi.
Il comando `xcde_build` calcola di *default* le *liste di proximity* e le memorizza nel *file* delle IL. Questa operazione non viene effettuata se è presente il *flag* –p, risparmiando così spazio nel caso l'utente non sia interessato ad effettuare interrogazioni sulla *proximity* delle parole.
Come abbiamo già detto nel *file di configurazione* si inseriscono i caratteri e le entità che fungono da separatori. La struttura di un generico *file di configurazione* è mostrata nella figura B.1. Il carattere '#' indica che la riga su cui si trova un commento. Su ogni riga può essere inserito un solo separatore. Il primo carattere della riga, se è diverso da '#', rappresenta lo *scope* del separatore. Se esso è 'A' allora si tratta di un separatore per gli attributi, mentre se è 'E' si tratta di un separatore per le parole. Se è uno spazio è un separatore sia per le parole che per gli attributi.
Il carattere di specifica può essere seguito dal codice ASCII di un carattere, per indicare che esso è un separatore, o dal nome di un'entità interna o esterna, anch'essa inserita nella lista dei separatori. È importante che sia l'eventuale codice ASCII o il nome dell'entità inizino dal secondo carattere della riga. In caso contrario si verificherebbe un errore di sintassi.

```
#   Xml Compress Decompress search Engine          #
#     configuration file xcde_build.conf           #
#  c. Andrea Mastroianni - Universita' di Pisa     #
#             mastroi@cli.di.unipi.it              #
#--------------------------------------------------#
# it's possible to declare a separator by its      #
# ASCII code or by an XML entity.                  #
# In  the  first case  the  symbol # near the      #
# ASCII code (that indicates a  comment)  can      #
# be substitutes  by a  space, that means the      #
# separator  is   valid   for   all  elements      #
# (attributes too), or by the symbol E  (that      #
# means all elements except attributes) or by      #
# the  symbol A (that means attributes only).      #
# That's valid also  for  entities,  but  the      #
# name of the entity must be written.              #
#--------------------------------------------------#
 9     HT     tabulazione orizzontale
 10    LF     line feed
 32           spazio
 33    !
#34    "
 35    #
 36    $
 37    %
#38    &
#39    '
 40    (
 41    )
#42    *
#43    +
 44    ,
 45    -
 46    .
#47    /
#48    0
#49    1
#50    2
#51    3
#52    4
#53    5
#54    6
#55    7
#56    8
#57    9
 58    :
 59    ;
 60    <
 61    =
 62    >
 63    ?
 64    @
123    {
#124   |
 125   }
#126   ~
A&excl;
```

Figura B.1 – Esempio di file di configurazione

Dato il file `001.xml` facente parte della collezione del *CIBIT* il comando:

```
$ xcde_build -t -v 001.xml
```

crea i seguenti *file*:

`001.xml.dict.0.cmp`	dizionario dei *tag* compresso;
`001.xml.dict.1.cmp`	dizionario degli attributi compresso;
`001.xml.dict.2.cmp`	dizionario delle parole compresso;

`001.xml.il`	*file* contenente le *liste invertite*;
`001.xml.body`	*file* contenente il documento compresso;
`001.xml.head`	*file* contenente le informazioni ausiliarie;
`001.xml.debug.index`	*file di debug* contenente gli indici in formato testuale;
`001.xml.debug.physic`	*file di debug* contenente il documento effettivamente compresso in formato testuale. Affinché la compressione sia corretta deve essere esattamente uguale al documento sorgente;
`001.xml.debug.stats`	*file di debug* contenente alcune statistiche sul documento compresso e sugli indici;

Il comando *xcde_extract*

Il comando *xcde_extract* decomprime interamente un documento compresso. La sintassi è la seguente:

```
xcde_extract filesource
```

dove *`filesource`* è il nome del *file* sorgente (ad esempio `001.xml`). Naturalmente non è necessario che quest'ultimo sia presente nel *file system*. Il documento decompresso è visualizzato sullo *standard output* e, tramite la sua ridirezione, eventualmente inviato su un *file* determinato dall'utente

Il comando *xcde_inf*

Questo comando consente di visualizzare alcune informazioni utili su un documento compresso ed indicizzato. La sua sintassi è la seguente:

```
xcde_inf filesource
```

dove *`filesource`* è il nome del *file* sorgente (ad esempio `001.xml`). Naturalmente non è necessario che quest'ultimo sia presente nel *file system*. Ad esempio il comando:

```
$ xcde_inf 001.xml
```

visualizza le informazioni seguenti:

```
                     ------------------------------------------------------XCDE_INF :
                    Version 0.9ESP 26-02-2002
                      X.ml C.ompressed D.ocuments E.ngine
                     Paolo Ferragina - Andrea Mastroianni
                  Dipartimento di Informatica - Universita' di Pisa
                         XCDE File System Information
       ------------------------------------------------------------------


Proximity lists present
File source name : 001.xml
File source size : 866060
File header name : 001.xml.head
File header size : 2464 bytes
File body name : 001.xml.body
File body size : 248595 bytes
File IL name : 001.xml.il
File IL size : 545505 bytes
TAG compressed dictionary name : 001.xml.dict.0.cmp
TAG compressed dictionary size : 296
ATTR compressed dictionary name : 001.xml.dict.1.cmp
ATTR compressed dictionary size : 1165
WORD compressed dictionary name : 001.xml.dict.2.cmp
WORD compressed dictionary size : 64203
Maximum codeword length : 3
Number of tokens : 17053
Number of elements at level 1 : 72
Number of elements at level 2 : 7090
Number of elements at level 3 : 9891
Number of element in TAG dictionary : 56
Number of element in ATTR dictionary : 383
Number of element in WORD dictionary : 16614
Number of elements at level 1 of TAG dictionary : 3
Number of elements at level 2 of TAG dictionary : 51
Number of elements at level 3 of TAG dictionary : 2
Number of elements at level 1 of ATTR dictionary : 0
Number of elements at level 2 of ATTR dictionary : 13
Number of elements at level 3 of ATTR dictionary : 370
Number of elements at level 1 of WORD dictionary : 69
Number of elements at level 2 of WORD dictionary : 7026
Number of elements at level 3 of WORD dictionary : 9519
```

Figura B.2 – informazioni visualizzate da xcde_inf

Il comando *xcde_search*

Questo comando è utilizzato per realizzare le *query* su un documento compresso e indicizzato. La sua sintassi è:

```
xcde_search filesource query
```

dove *filesource* è il nome del documento sorgente e *query* rappresenta l'interrogazione, formulata con il linguaggio seguente:

`--tag`	il *pattern* che segue si applica a un *tag;*
`--word`	il *pattern* che segue si applica a una parola;
`--withattr`	il *pattern* che segue si applica a un nome e/o al valore di un attributo, e individua con *--withattr* i *tag* che contengono l'attributo e con *--withoutattr* quelli che non lo contengono;
`--value`	il *pattern* che segue si applica al valore di un attributo;

--xml_dist	il numero che segue specifica la distanza nell'albero del documento XML del tag corrispondente dal *tag* precedente nella interrogazione;
--xml_exact	ricerca *esatta*;
--xml_prefix	ricerca per *prefisso*;
--xml_suffix	ricerca per *suffisso*;
--xml_contained	ricerca per *sottostringa*;
--xml_regexp	ricerca per *espressione regolare*;
--xml_error	ricerca con *errore*. È seguito da un numero maggiore di 0 che indica il numero di errori sul pattern;
--xml_token_exact	ricerca *esatta* sui *token* ottenuti dal valore di un attributo;
--xml_token_prefix	ricerca per *prefisso* sui *token* ottenuti dal valore di un attributo;
--xml_token_suffix	ricerca per *suffisso* sui *token* ottenuti dal valore di un attributo;
--xml_anyvalue	non specifica un *pattern* per il nome dell'attributo;
--proximity	specifica la *proximity* su un'interrogazione con più parole. Deve essere seguito da un numero maggiore di 0 che indica la *proximity*;
--xml_nocase	specifica una ricerca *case-insensitive*.

Come risultato della ricerca il comando visualizza il *nome del documento sorgente* su cui è effettuata l'interrogazione e due numeri che indicano, rispettivamente, la posizione nel documento compresso del primo e dell'ultimo *token* della finestra minima che contiene tutte le occorrenze dei *token* che corrispondono all'interrogazione. Queste informazioni potranno poi essere utilizzate come *input* del comando `xcde_view` per l'eventuale successiva visualizzazione degli *snippet*.

Ad esempio il comando:

```
$ xcde_search 001.xml --tag --xml_exact 'TEI.2' --word --xml_exact pisa
--xml_nocase --word --xml_prefix universit --xml_nocase
```

Cerca tutte le occorrenze che corrispondono esattamente alla parola *pisa* e tutte le occorrenze di parole che hanno come prefisso la stringa *universit* in modo *case-insensitive* e all'interno di tutto il documento `001.xml`. Il risultato è il seguente:

```
001.xml 238 251
001.xml 251 259
```

Il comando *xcde_search2*

Questo comando è utilizzato per realizzare le *query* su un documento compresso e indicizzato ed è stato introdotto con la versione 2 di XCDE per supportare un linguaggio di interrogazione più potente e completo di quello presente nella versione 1. La sua sintassi è:

```
xcde_search2 [-p] [-f filename] "queryExpression"
```

Attraverso l'uso dei flag è possibile cambiare il comportamento del query engine:

-f FILENAME: questa opzione permette di eseguire la query presente nel file specificato da *filename*. Il file deve essere contenuto nella stessa directory del comando xcde_search2 (./bin o ./examples) se viene specificato un path relativo.

-p: ritorna le posizioni degli elementi nel documento compresso che soddisfano l'interrogazione anzicchè il loro contesto. Questa è un'opzione sofisticata introdotta principalmente per motivi di debugging.

Dove *queryExpression* rappresenta l'interrogazione, formulata con il linguaggio presentato nel paragrafo **[2.2]**

Ad esempio il comando seguente:

```
$ xcde_search2 "SELECT <xml_exact xml_var = '$parola' xml_case =
                'sensitive'>Usability</xml_exact> FROM esempio.xml RETURN
                <mytag> $parola </mytag>"
```

Cerca tutte le occorrenze che corrispondono esattamente alla parola *Usability* in modo *case-sensitive* e all'interno di tutto il documento `esempio.xml`. Il risultato è il seguente:

```
<xcde:results>
      <xcde:result>
      <xcde:filename>esempio.xml</xcde:filename>

      <xcde:return result_number = '1'>
             <mytag> Usability </mytag>
      </xcde:return>
      .
      .
      .
      <xcde:return result_number = '15'>
             <mytag> Usability </mytag>
      </xcde:return>
      </xcde:result>
</xcde:results>
```

Il comando *xcde_view*

Questo comando consente di estrarre degli *snippet ben-formati* da un documento compresso. La sua sintassi è:

```
xcde_view filesource <initial pos.snippet i,final pos. snippet
i+1,
                              number of words>*
```

dove *filesource* è il nome del documento sorgente e il seguente parametro è una sequenza di triple in cui i primi due elementi sono la posizione iniziale e finale della finestra da

visualizzare nel documento compresso, e il terzo indica il numero di parole da inserire nello *snippet* prima e dopo di essa.

Ad esempio il comando seguente:

```
$ xcde_view 001.xml 238 251 10 251 259 10
```

visualizza sullo *standard output* gli *snippet* relativi alle due finestre ottenute dall'interrogazione nella sezione precedente. L'output è il seguente:

```
<xml_snippet>
<xml_filename>001.xml</xml_filename>
<xml_text>
<TEI.2> ...
 <teiHeader> ...
  <fileDesc> ...
   <titleStmt> ...
    <respStmt> ...
     <resp> ... di </resp>
        <name>Pierazzo, Elena</name>
        <resp>Edizione elettronica curata presso l'unit&agrave; di ricerca di Pisa
                <address><addrLine>Universit&agrave; di Pisa</addrLine>
                <addrLine>Facolt&agrave; di Lettere e Filosofia</addrLine>
                <addrLine>Dipartimento di Studi ... </addrLine>...
               </address>...
       </resp>...
    </respStmt>...
   </titleStmt>...
  </fileDesc>...
 </teiHeader>...
</TEI.2>
</xml_text>
</xml_snippet>

<xml_snippet>
<xml_filename>001.xml</xml_filename>
<xml_text>
<TEI.2> ...
 <teiHeader> ...
  <fileDesc> ...
   <titleStmt> ...
    <respStmt> ...
     <name> ... Pierazzo, Elena</name>
        <resp>Edizione elettronica curata presso l'unit&agrave; di ricerca di Pisa
                <address><addrLine>Universit&agrave; di Pisa</addrLine>
                <addrLine>Facolt&agrave; di Lettere e Filosofia</addrLine>
                <addrLine>Dipartimento di Studi Italianistici</addrLine>
</address></resp>
</respStmt>
        </titleStmt>
        <editionStmt>
               <edition>Seconda
...    </edition>...
   </editionStmt>...
  </fileDesc>...
 </teiHeader>...
</TEI.2>
</xml_text>
</xml_snippet>
```

La possibilità di poter effettuare l'estrazione di *snippet ben-formati* rappresenta sicuramente una delle caratteristiche più innovative della libreria XCDE.

Gli *script*

Gli *script* della libreria sono:

`XCDE2/Makefile` effettua la compilazione, utilizzabile con il comando

	`make –f Makefile;`
`XCDE2/clean_exe`	effettua la rimozione di tutti i *file* oggetto, eseguibili e di libreria *ar* creati dal *makefile*;
`XCDE2/build_tar`	effettua la creazione dell'archivio *tar* dei *file* della libreria;
`XCDE2/bin/clean_xml`	effettua la rimozione dei *file* relativi ai documenti compressi ed indicizzati;
`XCDE2/bin/build_script`	effettua la creazione dei *file delle strutture dati* su tutti i *file* XML presenti nella *directory* in cui è chiamato.

I *file di libreria*

I *file di libreria* creati a seguito della compilazione sono i seguenti:

`XCDE2/libsrc/xcde/xcde_lib.a`	libreria delle API di XCDE;
`XCDE2/libsrc/xcde/xcde_libsearch.a`	libreria delle API di XCDE2;
`XCDE2/lib/libbuff.a`	libreria delle funzioni per la gestione del *buffer* di testo durante il *parsing*;
`XCDE2/lib/libexpat.a`	libreria Expat delle funzioni SAX utilizzate per il *parsing* dei documenti XML;
`XCDE2/lib/libhash.a`	libreria per la gestione della tabella Hash durante la fase di *parsing*;
`XCDE2/lib/libinteger.a`	libreria delle funzioni per la compressione degli interi con l'algoritmo *continuation bit;*
`XCDE2/lib/libz.a`	libreria con le funzioni per la compressione con l'algoritmo Lempel-Ziv;

Le librerie `libbuff.a, libexpat.a, libhash.a, libinteger.a e libz.a` sono necessarie per la costruzione del comando `xcde_build`, mentre le librerie `xcde_lib.a. libinteger.a e libz.a` sono necessarie per l'implementazione delle API e quindi per I comandi `xcde_search, xcde_search2, xcde_view, xcde_extract` e `xcde_inf` e tutte le applicazioni costruite sulla libreria XCDE2.
La struttura delle libreria è visualizzata nella figura B.3

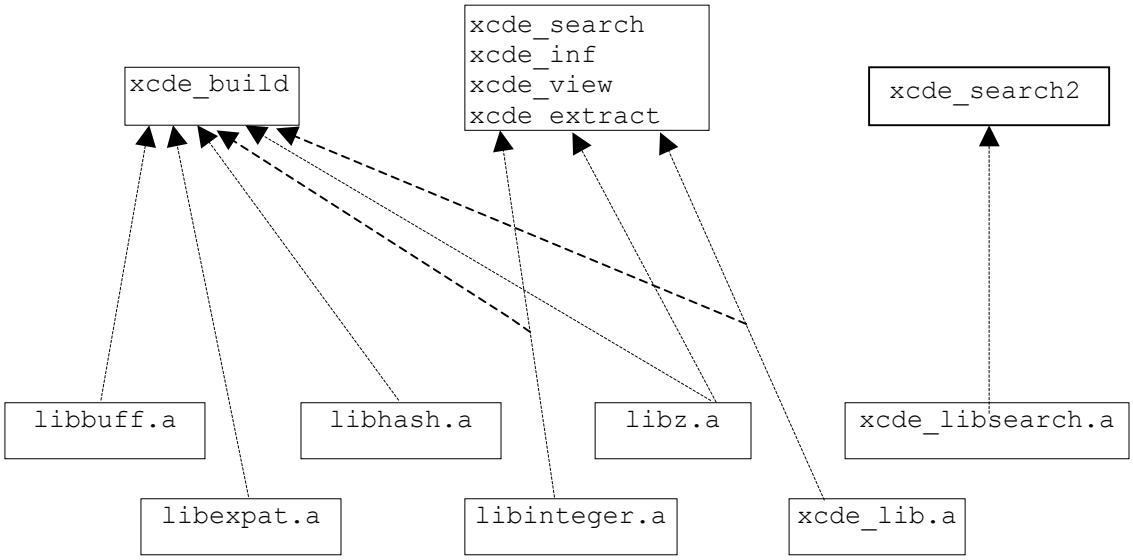


Figura B.3 – utilizzo dei file di libreria da parte dei comandi di XCDE2

I *file* sorgenti

I *file* sorgenti per la compilazione delle librerie e dei comandi sono i seguenti:

xcde_build

```
XCDE/src/xcde_build.c
XCDE/src/xml_confhand.c
XCDE/src/xcde_huffman.c
XCDE/src/xml_tagstruct.c
XCDE/src/xcde_expat_handler.c
XCDE/src/xcde_build_utility.c
```

xcde_inf

```
XCDE/src/xcde_inf.c
```

xcde_view

```
XCDE/src/xcde_view.c
```

xcde_extract

```
XCDE/src/xcde_extract.c
```

xcde_search

```
XCDE/src/xcde_search.c
```

xcde_search2

```
XCDE/src/xcde_search2.c
```

_xcde_transf

```
XCDE/src/_xcde_transf.c
```

libz.a

```
XCDE/libsrc/zlib/adler32.c
XCDE/libsrc/zlib/compress.c
XCDE/libsrc/zlib/crc32.c
XCDE/libsrc/zlib/deflate.c
XCDE/libsrc/zlib/example.c
XCDE/libsrc/zlib/gzio.c
XCDE/libsrc/zlib/infblock.c
XCDE/libsrc/zlib/infcodes.c
XCDE/libsrc/zlib/inffast.c
XCDE/libsrc/zlib/inflate.c
XCDE/libsrc/zlib/inftrees.c
XCDE/libsrc/zlib/infutil.c
XCDE/libsrc/zlib/maketree.c
XCDE/libsrc/zlib/trees.c
XCDE/libsrc/zlib/uncompr.c
XCDE/libsrc/zlib/zutil.c
XCDE/libsrc/zlib/deflate.h
XCDE/libsrc/zlib/infblock.h
XCDE/libsrc/zlib/infcodes.h
XCDE/libsrc/zlib/inffast.h
XCDE/libsrc/zlib/inffixed.h
XCDE/libsrc/zlib/inftrees.h
XCDE/libsrc/zlib/infutil.h
XCDE/libsrc/zlib/trees.h
XCDE/libsrc/zlib/zconf.h
XCDE/libsrc/zlib/zlib.h
XCDE/libsrc/zlib/zutil.h
XCDE/libsrc/zlib/Makefile
```

libexpat.a

```
XCDE/libsrc/expat/
XCDE/libsrc/expat/xmlparse.c
XCDE/libsrc/expat/xmlparse.h
XCDE/libsrc/expat/xmlrole.c
XCDE/libsrc/expat/xmlrole.h
XCDE/libsrc/expat/xmltok.c
XCDE/libsrc/expat/xmltok.h
XCDE/libsrc/expat/xmltok_impl.c
XCDE/libsrc/expat/xmltok_ns.c
XCDE/libsrc/expat/xmltok_impl.h
XCDE/libsrc/expat/xmldef.h
XCDE/libsrc/expat/ascii.h
XCDE/libsrc/expat/latin1tab.h
XCDE/libsrc/expat/utf8tab.h
XCDE/libsrc/expat/iasciitab.h
XCDE/libsrc/expat/asciitab.h
XCDE/libsrc/expat/nametab.h
```

libinteger.a

```
XCDE/libsrc/integer/int_basic.c
XCDE/libsrc/integer/continuation_bit.c
XCDE/libsrc/integer/utility1.c
XCDE/libsrc/integer/headers1.h
```

xcde_lib.a

```
XCDE/libsrc/xcde/xcde_lib.c
XCDE/libsrc/xcde/xcde_lib.h
```

xcde_libsearch.a

```
XCDE/libsrc/xcde/xcde_libearch.c
XCDE/libsrc/xcde/xcde_libsearch.h
```

libhashfunc.a

```
XCDE/libsrc/hashfunc/hashfunc.c
XCDE/libsrc/hashfunc/hashfunc.h
```

libbuff.a

```
XCDE/libsrc/textbuffer/textbuffer.c
XCDE/libsrc/textbuffer/textbuffer.h
```

agrep

```
XCDE/libsrc/agrep/agrep.1
XCDE/libsrc/agrep/agrep.algorithms
XCDE/libsrc/agrep/agrep.chronicle
XCDE/libsrc/agrep/agrep.h
XCDE/libsrc/agrep/asearch1.c
XCDE/libsrc/agrep/asearch.c
XCDE/libsrc/agrep/bitap.c
XCDE/libsrc/agrep/checkfile.c
XCDE/libsrc/agrep/checkfile.h
XCDE/libsrc/agrep/compat.c
XCDE/libsrc/agrep/contribution.list
XCDE/libsrc/agrep/COPYRIGHT
XCDE/libsrc/agrep/follow.c
XCDE/libsrc/agrep/main.c
XCDE/libsrc/agrep/Makefile
XCDE/libsrc/agrep/maskgen.c
XCDE/libsrc/agrep/mgrep.c
XCDE/libsrc/agrep/parse.c
XCDE/libsrc/agrep/preprocess.c
XCDE/libsrc/agrep/readme
XCDE/libsrc/agrep/re.h
XCDE/libsrc/agrep/sgrep.c
XCDE/libsrc/agrep/utilities.c
```

APPENDICE C – Documento XML

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE books PUBLIC "-//EXAMPLE//DTD BOOK Lite XML ver. 1//EN" "dtd/examples.dtd">
<books>
<book number="1">
  <metadata>
    <title shortTitle="Improving Web Site Usability">Improving the Usability of a Web Site Through
Expert Reviews
          and Usability Testing</title>
    <authors>
      <author>Millicent Marigold</author>
      <author>Montana Marigold</author>
    </authors>
    <publicationInfo>
      <place>New York</place>
      <publisher>Ersatz Publications</publisher>
      <dateIssued>2001</dateIssued>
      <dateRevised>2002</dateRevised>
    </publicationInfo>
    <price>25.99</price>
    <subjects xml_lang="en">
      <subject>Usability testing</subject>
      <subject>Web site development</subject>
      <subject>Heuristic evaluation</subject>
      <subject>Cognitive walk-through</subject>
      <subject>Web site usability</subject>
    </subjects>
    <subjects xml_lang="fr">
      <subject>Tests d'ergonomie</subject>
      <subject>D&egrave;veloppement de site web</subject>
      <subject>&egrave;valuation heuristique</subject>
      <subject>Parcours cognitif</subject>
      <subject>Ergonomie de site web</subject>
    </subjects>
  </metadata>
  <content>
    <introduction>
      <author>Elina Rose</author>
      <p>The usability of a Web site is how well the site supports the user in achieving specified goals. A
          Web site should facilitate learning, and enable efficient and effective task completion, while
          propagating few errors.
           Satisfaction with the site is also important. The user must not only be well-served, but must feel
          well-served.</p>
      <p>Expert reviews and usability testing are methods of identifying problems in layout, terminology,
          and navigation before they frustrate users and drive them away from your site.</p>
      <p>The most successful projects employ multiple methods in multiple iterations. As Millicent
          Marigold remarked during a recent conference, "Don't stop. Iterate, iterate, then iterate again."</p>
      <p>This book has been approved by the Web Site Users Association.</p>
    </introduction>
    <part number="1">
      <title>Expert Reviews</title>
      <introduction>
        <p>Expert reviewers identify problems and recommend changes to web sites based on research in
          human computer interaction and their experience in the field.</p>
        <p>Two expert review methods are discussed here. They are heuristic evaluation and cognitive
          walk-through.</p>
        <p>Expert review methods should be initiated early in the development process, as soon as paper
```

```
            <b>p</b>rototypes (hand-drawn pictures of Web pages) or <b>w</b>ireframes (electronic
          mockups) are available. They should be conducted using the hardware and software similar to that
          employed by users.</p>
        </introduction>
        <chapter>
          <title>Heuristic Evaluation</title>
          <p>Expert reviewers critique an interface to determine conformance with recognized usability
              principles.
            <footnote>One of the best known lists of heuristics is <citation>Ten Usability Heuristics by Jacob
              Nielson</citation>. Another is <citation> Research-Based Web Design and Usability
              Guidelines</citation></footnote></p>
        </chapter>
        <chapter>
          <title>Cognitive Walk-Through</title>
          <p>Expert reviewers evaluate Web site understandability and ease of learning while performing
            specified tasks. They walk through the site answering questions such as "Would a user know by
            looking at the screen how to complete the first step of the task?" and "If the user completed the first
            step, would the user know what to do next?," with the goal of identifying any obstacles to
            completing the task and assessing whether the user would cognitively be aware that he was
            successful in completing a step in the process.</p>
        </chapter>
      </part>
      <part number="2">
        <chapter>
          <title>Usability Testing</title>
          <p>Once the problems identified by expert reviews have been corrected, it is time to conduct some
            tests of the site with your unique audience or audiences by conducting usability testing.</p>
          <p>Users are asked to complete tasks which measure the success of the information architecture and
              navigational elements of the site.</p>
          <p>Then changes are made to improve service to users.</p>
        </chapter>
      </part>
    </content>
  </book>
  <book number="2">
    <metadata>
      <title shortTitle="Usability Basics">Usability Basics: How to Plan for and Conduct Usability Tests on
          Web Site Thereby Improving the Usability of Your Web Site</title>
      <publicationInfo>
        <place>New York</place>
        <publisher>Ersatz Publications</publisher>
        <publisher>Electronic BookWorks</publisher>
        <dateIssued>2000</dateIssued>
        <dateRevised>2001</dateRevised>
      </publicationInfo>
      <price>174.00</price>
      <subjects xml_lang="en">
        <subject>Usability testing</subject>
        <subject>Web site development</subject>
        <subject>Guides and finding aids</subject>
      </subjects>
      <subjects xml_lang="fr">
        <subject>Tests d'ergonomie</subject>
        <subject>D&egrave;veloppement de site web</subject>
        <subject>Guides et outils de recherche</subject>
      </subjects>
    </metadata>
    <content>
```

```
    <introduction>
      <p>This is a basic handbook for planning and conducting usability tests on Web sites. Usability
          testing should be used in conjunction with other expert review methods.</p>
      <p>This book has not been approved by the Web Site Users Association.</p>
    </introduction>
    <part number="1">
      <chapter>
        <title>Planning then Conducting Usability Tests</title>
        <p>Take the following steps to plan usability testing. <step number="1">Clarify and articulate the
          goal of the usability testing. </step> <step number="2">Identify tasks which are critical for users to
          be able to complete successfully. </step> <step number="3">Compile a script of questions or
          instructions which will prompt the user to attempt those tasks.</step> <step number="4">Identify
          your users and begin recruiting them.</step><step number="5">Conduct a pretest on a few users.
          </step> <step number="6">Edit the script based on insights gleaned from the pretest.</step>
           <step number="7">Resume testing.</step></p>
      </chapter>
    </part>
    <part number="2">
      <chapter>
        <title>Conducting Usability Tests</title>
        <p>Users can be tested at any computer workstation <footnote>They may be most comfortable at
          their own workstation. </footnote> or in a lab.</p>
        <p>Give the user the script, then assure them that you are testing the Web site, not them. Users are
          asked to verbalize their thoughts as they complete the tasks. The event is recorded or someone
          takes notes. It is often to have two testers, <footnote>Usability testing can be done at great expense
          or on a shoe string, using <testingProcedure>in-house expertise</testingProcedure> or
          <testingProcedure>contracting with human computer interaction professionals </testingProcedure>
          </footnote> one to ask the questions, another to take notes. Testers should offer no guidance or
          comments to the user. Mouse movements, typing, expressions, and the user's words should be
          recorded.</p>
      </chapter>
      <chapter>
        <title>Evaluating and Implementing Results</title>
        <p>Compile the results and review collectively. Make changes to the site to alleviate the problems
          found in Web site components which were propagating the largest number of or the most
          devastating errors. Begin new iterations of testing and changes, until users are successful in the
          accomplishing the tasks.</p>
      </chapter>
    </part>
  </content>
</book>
<book number="3">
  <metadata>
    <title shortTitle="Usabilityguy Manuscript Guide">John Wesley Usabilityguy: A Register of His
Papers</title>
    <authors>
      <author>Millicent Marigold</author>
      <author>Morty Marigold</author>
    </authors>
    <publicationInfo>
      <publisher>Ersatz Manuscript Library</publisher>
      <dateIssued>1998</dateIssued>
      <dateRevised>2002</dateRevised>
    </publicationInfo>
    <price>21.49</price>
    <subjects xml_lang="en">
      <subject>Computers</subject>
      <subject>Software evaluation</subject>
      <subject>Usability testing</subject>
```

```
      <subject>Manuscript collections</subject>
    </subjects>
    <subjects xml_lang="fr">
      <subject>Ordinateurs</subject>
      <subject>&egrave;valuation de logiciels</subject>
      <subject>Tests d'ergonomie</subject>
      <subject>Collections de manuscrits</subject>
    </subjects>
  </metadata>
  <content>
    <introduction>
      <p>The papers of John Wesley Usabilityguy span the years 1946-2001, with the bulk of the items
          concentrated in the period from 1985 to 2001. The papers feature his career as a developer of
          software applications and usability specialist. The collection consists of correspondence, emoranda,
          journals, peeches, article drafts, book drafts, notes, charts, graphs, family papers, clippings, printed
          matter, photographs, r&egrave;sum&egrave;s and other materials.</p>
    </introduction>
    <part number="1"> <container type="box">1-12</container>
      <title>Subject File, <date normalize="1930/1974">1930-1974</date></title>
      <introduction>
        <p>Correspondence, telegrams, memoranda, journals, logs, testimony, approved travel orders,
          invitations, charts, graphs, forms, biographical data, photographs, book drafts, clippings and other
          printed matter, r&egrave;sum&egrave;s and miscellaneous material. Organized by name of person
          or organization, topic, or type of material.</p>
      </introduction>
      <component><container type="box">1</container>
       <componentTitle>Computers</componentTitle>
       <subComponent>
         <componentTitle>Software,
                  <componentDate normalize="1946/1947">1946-1947 </componentDate>
         </componentTitle>
       </subComponent>
       <subComponent>
         <componentTitle>Human Computer Interaction research,
                  <componentDate normalize="1945/1952">1945-1952</componentDate>
         </componentTitle>
         <subsubComponent>
           <componentTitle>Flow diagram, <componentDate normalize="1950">1950</componentDate>
           </componentTitle>
         </subsubComponent>
         <subsubComponent>
           <componentTitle>General,
                  <componentDate normalize="1947/1951">1947-1951 </componentDate>
           </componentTitle>
         </subsubComponent>
         <subsubComponent><container type="box">2</container>
            <componentTitle>Eye Movement research,
                  <componentDate normalize="1949/1950">1949-1950</componentDate>
           </componentTitle>
         </subsubComponent>
         <subsubComponent>
           <componentTitle>User profiling,
                  <componentDate normalize="1950/1959">1950s </componentDate>
           </componentTitle>
         </subsubComponent>
       </subComponent>
      </component>
```

```
      <component>
        <componentTitle>Web User Appreciation Award,
                    <componentDate normalize="1956">1956</componentDate>
        </componentTitle>
      </component>
    </part>
    <part number="2"><container type="box">3-5</container>
      <title>Writings File, date normalize="1985/1999">1985-1999</date></title>
      <introduction>
        <p>Correspondence, articles, book drafts, notes, contracts, clippings, and printed matter. Arranged
           alphabetically by type (articles, books, reports, and miscellaneous) and therein alphabetically by
           type of material, subject, or title.</p>
      </introduction>
      <component>
        <componentTitle>Writings by Usabilityguy </componentTitle>
        <subComponent>
          <componentTitle><componentDate normalize="1996">1996</componentDate>
          </componentTitle>
          <subsubComponent>
            <componentTitle>"How Many Users Are Enough for User Testing?"</componentTitle>
          </subsubComponent>
          <subsubComponent>
            <componentTitle>"How to Evaluate Results from User Tests."</componentTitle>
          </subsubComponent>
          <subsubComponent>
            <container type="box">5</container>
            <componentTitle>"When Are You Done Testing?" </componentTitle>
          </subsubComponent>
          <subsubComponent>
            <componentTitle>" Do-It-Yourself User Testing " </componentTitle>
          </subsubComponent>
        </subComponent>
      </component>
      <component>
        <componentTitle>Charitable Contributions </componentTitle>
        <subComponent>
          <componentTitle>Diseases: AIDS, Hepatitis,Tuberculosis
                    <componentDate normalize="1990/1999">1990-1999</componentDate>
          </componentTitle>
        </subComponent>
        <subComponent>
          <componentTitle>Environmental Conservation: Rivers
                    <componentDate normalize="1995">1995</componentDate>
          </componentTitle>
        </subComponent>
      </component>
    </part>
  </content>
</book>
</books>
```

Figura C.1 – Esempio di documento XML