# Indice

# 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 un 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, il suo utilizzo nel mondo dell'industria e in quello accademico è aumentato 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 ad 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.

Una *query* 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. Questa visione è, attualmente, abbastanza utopistica anche se un elevato numero di persone e organizzazioni stanno lavorando affinché diventi ben presto una realtà.

Fra le applicazioni più promettenti e interessanti di XML troviamo sicuramente quella legata al campo dell' *Information Retrieval.* Diversi sistemi per la memorizzazione di documenti XML e per la successiva estrazione di informazioni da essi, sono stati

realizzati sia in ambito commerciale *(XYZ-Find, Tamino)* che accademico (*BUS, Fabric, Lore, Natix, STORED, ToX, TReSy, Xset,* ecc.). In base alle loro caratteristiche questi sistemi possono essere suddivisi in quattro grandi gruppi:

- I sistemi orientati all'*information retrieval* memorizzano i documenti XML come se fossero generici file testuali, non considerandone l'aspetto strutturale. Le successive operazioni di estrazione delle informazioni utilizzano classici meccanismi di ricerca su testi, i quali garantiscono buone prestazioni nel caso di ricerche *full-text*, ma incontrano diverse difficoltà computazionali nella gestione delle interrogazioni sulla struttura dei documenti;

- I sistemi *flat* memorizzano i documenti possibilmente in forma compressa e utilizzano dei meccanismi basati sugli *eventi* (SAX) o *su una visione ad oggetti* (DOM) per realizzare le operazioni su di essi. I sistemi di questo tipo sono lenti poichè richiedono un'analisi completa del documento ad ogni interrogazione, e nel caso DOM possono impiegare un significativo spazio di memoria;

- I sistemi orientati alle *basi di dati* utilizzano un DBMS per memorizzare i documenti. Se quest'ultimo è di tipo *relazionale* i documenti sono memorizzati come *tuple* all'interno delle tabelle, mentre nel caso *ad oggetti* essi utilizzano la visione ad albero della struttura del documento, in cui i singoli nodi sono *oggetti* della base di dati. In entrambi i casi l'occupazione in spazio dei documenti è elevata, e non sempre questi sistemi offrono efficienza nel recupero delle informazioni di carattere testuale;

- I sistemi *nativi* si basano su soluzioni *ad hoc* per la memorizzazione dei documenti. Questo consente di avere il controllo su tutto il sistema e di progettare in essi soluzioni ottimizzate per le operazioni di memorizzazione e interrogazione. I sistemi nativi sono sicuramente quelli più promettenti e flessibili, visti i buoni risultati preliminari ottenuti in letteratura sia per quanto concerne l'occupazione in spazio che per l'efficenza in tempo.

Fra i sistemi citati precedentemente: *Bus, Lore, Natix, TReSy*, *Xset,* appartengono alla classe dei sistemi nativi e, anche se non si hanno dettagli sulla loro implementazione, a questa classe fanno ragionevolmente parte anche *XYZ-Find* e *Tamino*. Di contro, il sistema *STORED* è orientato alle basi di dati, in questo caso *relazionali*, mentre *ToX* è un sistema che include componenti di vario tipo, memorizzando i documenti in modi diversi a seconda della loro struttura e natura.
Analizzando questi sistemi e la letteratura internazionale si può osservare come attualmente manchi una libreria *semplice, flessibile* e *di pubblico dominio* che consenta a un qualunque programmatore di sviluppare facilmente una applicazione che preveda l'indicizzazione, l'interrogazione, l'elaborazione o più in generale il *mining* di documenti XML. L'obiettivo di questa tesi è quello di studiare, progettare e realizzare una libreria che risponda a questi requisiti, offrendo una collezione di algoritmi e strutture dati per la memorizzazione *compressa* dei documenti XML e per la loro *efficiente* elaborazione ed interrogazione. La

libreria, chiamata XCDE (*Xml Compressed Document Engine*), 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*.

La collezione di algoritmi e strutture dati offerti dalla libreria è il risultato di una attenta selezione tra diversi *candidati*, l'approccio *nativo* ha permesso di sperimentare le varie soluzioni considerate avendo un totale controllo del sistema.

La principale struttura dati per l'indicizzazione dei documenti XML presente nella libreria è la classica lista invertita *(inverted list)*, sulla quale sono state applicate opportune tecniche di compressione in modo da ridurne l'occupazione in spazio. Le diverse componenti di un documento: *testo*, struttura dei *tag* e loro *attributi*, sono state indicizzate in maniera indipendente utilizzando per ciascuna di esse una propria lista invertita. Questa separazione consente di ottenere una grande flessibilità a livello di interrogazione in quanto è possibile effettuare efficientemente ricerche di vario tipo: *prefisso*, *suffisso, sottostringa, espressione regolare*, *proximity* e con *errori,* sia sul contenuto testuale dei documenti così come sui nomi dei tag e sui valori dei loro attributi. Per supportare la risoluzione efficiente di sofisticate *query* strutturali è stata inoltre introdotta una struttura dati innovativa, derivata dal campo della *Computational Geometry*. Questa struttura consente anche l'estrazione dai documenti compressi di *porzioni ben formate*, cioè sintatticamente corrette per il *processore* XML, come risultato delle interrogazioni effettuate dall'utente. Queste parti di documento, chiamate *snippet*, consentono alle applicazioni costruite mediante la Libreria XCDE di poter interagire facilmente con altre applicazioni scambiando dati in formato XML. Il formato compresso adottato nella libreria permette inoltre di accedere a queste *porzioni* senza che sia necessario *decomprimere interamente* un documento XML, aumentando così l'efficienza dell'intero sistema.

Il *documento compresso*, le *liste invertite* e le altre strutture dati ausiliarie comportano un'occupazione in spazio che è al più 1.2 volte la dimensione del documento originale. Questo risultato rappresenta una significativa riduzione in spazio sia rispetto ad altri sistemi *nativi*, ad esempio *Natix* (per il quale il fattore è almeno 3), sia per sistemi orientati alle *basi di dati relazionali o ad oggetti*, per i quali questo fattore moltiplicativo supera abbondantemente il 10. Come accennato precedentemente, la riduzione in spazio ha un notevole impatto sui tempi di accesso ai documenti, consentendo di aumentare virtualmente la larghezza di banda del disco e di utilizzare al meglio i meccanismi di *caching* e *bufferizzazione* offerti dal sistema operativo. Altra conseguenza della riduzione dello spazio occupato è la possibilità, garantita anche dall'elevata *modularità* della libreria, di poter aggiungere indici per implementare nuovi tipi di interrogazioni o velocizzarne alcune di quelle già offerte, senza correre il rischio di far esplodere lo spazio totale

occupato. Infine la compressione può giocare un ruolo cruciale nella implementazione di applicazioni che utilizzino la libreria XCDE su dispositivi con poca memoria disponibile, come ad esempio gli *ebooks* e i palmari.

Una delle principali scelte effettuate in sede di implementazione della libreria è stata quella di operare con *granularità* limitata al *singolo documento*. Questa soluzione comporta un aumento dei tempi delle ricerche quando queste sono effettuate su collezioni formate da *molti* documenti di *piccole* dimensioni, ma consente altresì di ottenere diversi vantaggi:

- costruzione semplificata di *indici distribuiti*;
- risoluzione immediata del problema dell'*aggiornamento* in caso di aggiunta o cancellazione di singoli documenti dalla collezione;
- possibilità di *specializzare* su *ogni documento* sia gli indici, in base al tipo di ricerca da effettuarsi su di esso, che il meccanismo di compressione, in base alla sua natura e struttura;
- possibilità di sfruttare le caratteristiche del linguggio XML per *fondere* più documenti in un solo *documento virtuale*, mantenendo da un lato intatta la loro individualità e consentendo dall'altro di non incorrere in rallentamenti significativi nell'interrogazione di collezioni molto popolose.

La libreria fornisce un API con un insieme di funzioni che operano sulla collezione di strutture dati ed algoritmi. Essa è in grado di implementare tutte le principali funzionalità di XQuery, e può supportare ricerche con funzionalità avanzate di *information retrieval*. Naturalmente la libreria XCDE è solo un *kernel* a partire dal quale si possono costruire sistemi più complessi, come motori di ricerca o sistemi per il *mining* di documenti XML.

Attualmente la libreria XCDE è stata utilizzata per implementare un motore di ricerca, chiamato *XCDE Search Engine* (http://sbrinz.di.unipi.it/~xcde), operante su una collezione di documenti in formato TEI-XML, fornita dal *CIBIT (Consorzio Interuniversitario Biblioteca Italiana Telematica)* e contenente testi della letteratura italiana. Questi documenti hanno la caratteristica di contenere ampie parti testuali, avere una struttura con numerosi *tag* innestati su più livelli, e valori dei loro attributi formati da stringhe alfanumeriche arbitrariamente complesse. Il motore può essere utilizzato per effettuare ricerche *full-text* così come ricerche attraverso *espressioni regolari*, ricerche sul contenuto testuale dei tag o sul valore dei loro attributi. La granularità del singolo documento consente inoltre al motore di ricerca di operare su porzioni dell'intera collezione, in base a quanto stabilito dall'utente al momento della formulazione dell'interrogazione. La *query* viene formulata attraverso un'interfaccia HTML semplice e intuitiva, anche per l'utente meno esperto, che consente anche di poter effettuare interrogazioni complesse, ad esempio basate su *espressioni regolari*.

La tesi risulta organizzata dei seguenti capitoli:

Nel **capitolo 1** viene fornita una panoramica sul linguaggio XML, le sue relazioni con i

linguaggi HTML e SGML, e una descrizione dei diversi approcci e tecniche messe a punto per la memorizzazione e l'interrogazione dei documenti XML. Viene fornita inoltre una descrizione generale della libreria XCDE, che sarà oggetto di studio e sperimentazione più approfondita nei capitoli seguenti.

Nel **capitolo 2** sono descritte le strutture dati utilizzate dalla libreria XCDE e le operazioni necessarie alla loro costruzione. Nel corso del capitolo sarà data giustificazione di alcune scelte realizzate in sede di progettazione e implementazione, sulla base di numerosi esperimenti eseguiti su varie collezioni di documenti XML.

Nel **capitolo 3** sono descritte in maniera approfondita le funzioni offerte dalla libreria XCDE, la loro implementazione e le API in linguaggio C per il loro utilizzo.

Nel **capitolo 4** è descritto l'*XCDE Search Engine*, il motore di ricerca per i documenti XML forniti dal *CIBIT*. Viene inoltre descritto un semplice linguaggio progettato per formulare le interrogazioni, con la sua interfaccia a linea di comando, e viene illustrata l'interfaccia web realizzata per consentire la formulazione in remoto di una interazione su quella collezione di testi.

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 XCDE con la loro dichiarazione in linguaggio C.

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

# Capitolo 1

# XML

*L'XML è stato definito dall'XML Working Group del World Wide Web Consortium (W3C) con un progetto iniziato nel 1996. La versione 1.0 della prima specifica ufficiale è stata completata nel febbraio 1998. In questa specifica il gruppo descrive il linguaggio nella maniera seguente:* **"L'XML è una estensione di SGML [...] il suo scopo è di permettere a dei file generici SGML di essere ricevuti ed elaborati in rete in una modalità non possibile finora con HTML. L'XML è stato progettato per rendere più facile l'implementazione e l'interoperabilità sia con SGML che con HTML" [XML].** *L'XML è quindi un linguaggio di markup appositamente progettato per la memorizzazione e l'interscambio di informazioni sulla rete, a differenza di HTML che è un linguaggio per la visualizzazione delle stesse. Nella prima parte di questo capitolo descriveremo brevemente la storia e la struttura del linguaggio XML e le sue relazioni con l'HTML e l'SGML. Successivamente analizzeremo alcuni approcci già consolidati per la gestione dei documenti XML e li confronteremo con l'approccio seguito nello sviluppo della libreria XCDE (XML Compressed Document Engine).*

## 1.1 Una breve storia di XML

L'XML, così come l'HTML, deriva dall'SGML *(Structured Generalized Markup Language)*, che è il padre di tutti i linguaggi di markup. L'SGML definisce una sintassi di base, ma consente di creare i propri documenti in maniera personalizzata (da qui il termine *generalizzato*). Per utilizzare l'SGML nella descrizione di particolari tipi di documenti si devono definire un insieme di elementi e una struttura del documento. Per esempio, per descrivere un libro si possono usare gli elementi BOOK, PART, CHAPTER, INTRODUCTION, SECTION e così via.

L’ insieme di elementi utilizzati per descrivere un particolare tipo di documento prende il nome di *applicazione SGML* (in realtà un'applicazione SGML può anche includere delle regole che specificano come gestire eventualmente gli elementi). Un esempio famoso di applicazione SGML è l'HTML, che è stato sviluppato nel 1991 per la descrizione di pagine Web.

L'HTML fornisce un insieme di elementi *predefiniti* che possono essere usati per marcare le componenti di una tipica pagina *web*. Esempi di questi elementi sono i titoli, i paragrafi, le liste, le tabelle, le immagini e i collegamenti. Un documento HTML consiste di un insieme di elementi annidati, ogni elemento si apre con un’etichetta iniziale: un blocco di testo preceduto da una parentesi angolata aperta (<) e seguito da una parentesi angolata chiusa (>) che contiene i nomi di elementi ed altre informazioni. Molti elementi terminano con un’etichetta finale: essa è sintatticamente uguale a quella iniziale ma contiene solo una barra inversa (/) seguita dal nome dell'elemento. Il contenuto di ogni elemento è rappresentato dal testo, di qualunque natura esso sia, fra le etichette iniziali e finali. Molti elementi possono a loro volta contenere degli elementi innestati. Abbiamo di seguito un esempio di elemento HTML:

**<H2>La mia home page</H2>**

Il *browser* che visualizza la pagina HTML riconoscerà ognuno di questi elementi standard e applicherà il relativo formato di visualizzazione collegato ad ognuno di essi. Per esempio: il tipico *browser* visualizzerà un livello più esterno con un’ampiezza maggiore rispetto a quelli usati per i titoli di livello più innestato. Un elemento di una lista non numerata sarà visualizzato indicizzandolo e affiancandolo sulla destra con un carattere enfatizzante, ad esempio un asterisco o una pallina.
L'insieme delle etichette HTML si è molto ampliato con il passare del tempo, ma ancora oggi non è sufficiente per definire molti tipi di documenti. Di seguito sono elencati degli esempi di documenti non adatti ad essere descritti attraverso l'HTML:

- un documento che non sia costituito da componenti tipici (titoli, paragrafi, liste, tabelle e così via) ;

- una base di dati : ad esempio un inventario di libri. Una pagina HTML può essere usata facilmente per visualizzare informazioni provenienti da basi di dati statiche (come una lista di descrizioni di libri), ma se si vogliono filtrare, ordinare, estrarre informazioni o lavorare in maniera flessibile con esse, ogni componente individuale dell'informazione deve essere etichettata (come in un programma per la gestione delle basi di dati, ad esempio Microsoft Access). In HTML mancano gli elementi necessari per farlo;

- un documento che si voglia organizzare in una struttura gerarchica : per esempio, un libro con le sue parti, capitoli, sezioni, ecc. Un programma potrebbe utilizzare la struttura del documento per generare delle tabelle di contenuti, per produrre degli schemi a diverso livello di dettaglio, per estrarre delle sezioni specifiche o per

lavorare con le informazioni in qualsiasi altro modo. Un elemento di titolazione HTML individua solo il testo contenuto in esso: per esempio *<H2>La mia home page</H2>*. Visto che non si collegano gli elementi al titolo, quest'ultimo non può essere utilizzato per indicare chiaramente la struttura gerarchica del documento.

L'SGML sembrerebbe essere il linguaggio estendibile perfetto per la descrizione di documenti ma in realtà è troppo complicato per permettere il trasferimento e la gestione di informazioni in maniera efficiente. La sua flessibilità eccessiva e la presenza di una serie di caratteristiche opzionali lo rendono troppo difficile da gestire da parte di un qualsiasi programma. Quello di cui si aveva bisogno era un sottoinsieme del linguaggio, più semplice e facilmente gestibile in modo automatico: per soddisfare queste esigenze è nato l'XML, che rappresenta una versione semplificata e ottimizzata per la rete di SGML. La sintassi XML offre meno opzioni di quella dell'SGML, rendendo i documenti XML più facili da leggere, ma anche più facili da gestire per le applicazioni, non ultimi gli stessi browser. Come per l'SGML, anche in XML ciascun individuo o organizzazione può definire una propria *applicazione XML* che è costituita da un insieme di elementi e da una struttura che possono essere utilizzati per descrivere dei documenti di un particolare tipo (per esempio documenti che contengono formule matematiche).

# 1.2 Caratteristiche e applicazioni di XML

Quando si crea un documento XML, invece di usare un insieme limitato e predefinito di elementi, si creano i propri elementi personalizzati e si assegnano loro opportuni nomi (da qui derivano la X e il termine *estendibile* per riferirsi al linguaggio). Si possono così descrivere tutti i tipi di documenti, da uno spartito musicale ad una base di dati. L'XML non contempla elementi predefiniti, e per questo siamo portati a pensare che si tratti di uno standard relativamente "debole". In realtà lo standard è supportato da una sintassi definita in maniera rigorosa. Per esempio, al contrario di HTML, ogni elemento XML deve avere sia un'etichetta iniziale sia una finale (o uno speciale simbolo nell'etichetta che indica un elemento vuoto) e ogni elemento deve essere completamente contenuto in un altro o essere l'elemento radice del documento (struttura innestata). La possibilità di fornire un linguaggio tanto flessibile da permettere di creare dei nuovi elementi richiede, dall'altro lato, il rispetto rigoroso della sintassi. Questo consente di scrivere più facilmente programmi che lavorano su documenti XML. Ricordiamo la citazione all'inizio del capitolo, in cui si parlava di "facilità di implementazione" come uno degli scopi del linguaggio.
Un documento XML può essere, ad esempio, il risultato di un'interrogazione di una base di dati. In questo caso, l'XML ha il vantaggio di essere un formato proprietario (come il formato Access .mdb e il formato dBase .dbf); di essere leggibile, e di basarsi su uno standard aperto e pubblico.
E' importante osservare che i nomi degli elementi di un documento XML non fanno parte del linguaggio XML. In realtà questi nomi sono scelti dal creatore del documento, con il vantaggio di poter selezionare nomi altamente descrittivi.
Un documento XML può essere conforme a due livelli di controllo sintattico: *valido* o *ben formato,* a seconda di quali criteri soddisfi. Esso è strutturato come un albero, con tutti gli elementi inseriti all'interno di una gerarchia padre-figlio e con solo un *elemento radice*.

Questa proprietà del linguaggio permette di strutturare l'organizzazione in modo gerarchico; nel caso di un libro possiamo avere le varie parti: i capitoli, le sezioni e i paragrafi.
L'XML non solo può essere usato per descrivere un singolo documento, ma chiunque può definire un insieme di elementi e una struttura per essere usati con una particolare classe di documenti. L'insieme degli elementi XML che possono apparire in un documento e la descrizione della sua struttura sono chiamati *applicazione XML.*
Un'applicazione XML è normalmente definita attraverso una *descrizione del tipo di documento* o *DTD (Document Type Definition)*. Una DTD è in un certo senso simile a uno schema di una base di dati: essa definisce il nome degli elementi che possono essere usati nel documento, l'ordine con cui gli elementi possono apparire, gli attributi che possono essere usati per i singoli elementi e altre caratteristiche. Per usare una particolare applicazione XML, usualmente si allega la DTD ai documenti; questa operazione riduce in un certo senso la libertà che si ha nella scrittura, perché i documenti devono essere conformi alla DTD. Il vantaggio di usare un'applicazione standard XML sta però nel fatto che i documenti prodotti possono essere condivisi con altri utenti ed essere elaborati e visualizzati con strumenti specifici dell'applicazione.
Alcuni esempi di applicazione sono:

- XSL (eXtensible Stylesheet Language): si utilizza la sintassi XML per creare dei potenti fogli di stile per XML **[XSL]**;

- XML Schema: rappresenta un'alternativa alla creazione delle DTD. Attraverso di esso di descrive la struttura di un documento XML usando lo stesso XML, con l'aggiunta ti poter specificare i tipi di dati contenibili da un elemento **[XSchema]**;

- XLink (XML Linking Language) o Xpointer (XML Pointer Language): consentono di collegare fra loro i documenti XML gestendo possibilmente più link alla volta. Sono notevolmente più potenti del meccanismo di collegamento dell'HTML **[Xlink]**;

Esistono applicazioni per la memorizzazione di basi di dati, per la definizione di protocolli di comunicazione (CDF - Channel Definition Format e SOAP - Simple Object Access Protocol), per la descrizione di formule matematiche (MathML), di strutture molecolari (CML - Chemical Markup Language, BSML - Bioinformatic Sequence Markup Language), di dati genealogici (GedML - Genealogical Data Markup Language), di dati astronomici (AML - Astronomical Markup Language), di spartiti musicali (MusicML - Music Markup Language), di informazioni di natura legale (XCI - XML Court Interface), di informazioni di carattere liturgico e teologico (ThML - Theological Markup Language, LitML - Liturgical Markup Language), e molte altre sono in corso di definizione **[Applications]**.

## 1.3 Visualizzazione dei documenti XML

In una pagina HTML un broswser che incontri un elemento <H1> sa che si tratta di un titolo di primo livello e conosce il tipo di carattere e l'ampiezza da utilizzare nella sua visualizzazione. In XML gli elementi non sono predefiniti e dunque esistono principalmente tre metodi per comunicare ad un browser come gestirli e visualizzarli:

- collegamenti a fogli di stile: un foglio di stile è un documento che contiene delle istruzioni per il browser su come gestire e visualizzare i singoli elementi XML. Si possono usare sia fogli di stile "a cascata" (C*ascading Style Sheet o CSS)*, che sono usati anche per l'HTML **[CSS1] [CSS2]**, o fogli di stile XSL *(Extensible Stylesheet Language o XSL),* un linguaggio considerevolmente più potente di CSS **[XSL]**.

- collegamento dei dati a una pagina HTML *(Data binding)*: questa opzione richiede che si crei un documento HTML, si colleghi il documento XML ad esso e si colleghino dei singoli elementi standard HTML (come SPAN o TABLE) agli elementi del documento XML. Gli elementi HTML in questo modo possono automaticamente visualizzare le informazioni contenute negli elementi XML collegati;

- scripting: con questa tecnica si creano dei documenti HTML, si collega il documento XML ad essa, si accede e si visualizzano gli elementi del documento XML attraverso del codice scritto in un linguaggio *script* (ad esempio JavaScript o VBScript). Il navigatore esporta il documento attraverso una visione ad oggetti (*Document Object Model o DOM)* **[DOM]**, i quali possono essere utilizzati dal linguaggio di *script* per accedere, manipolare e visualizzare gli elementi XML.

## 1.4 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 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
<?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:

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

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

```xml
<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; | & | &#38; |
| &lt; | < | &#60; |
| &gt; | > | &#62; |
| &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;
- Il 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.

## 1.5 Memorizzazione e interrogazione di documenti XML

La caratteristica sicuramente più interessante del linguaggio XML è la capacità di descrivere la "semantica" delle informazioni contenute in un documento in modo strutturato, documentato e facilmente leggibile da una macchina. Queste caratteristiche hanno condotto alcuni ricercatori a parlare di "*semantic search engine"* per sottolineare, in modo ad oggi visionario, le possibilità immense offerte da XML per il progetto di motori di ricerca "intelligenti", ossia in grado di struttura la struttura dei documenti XML per condurre ricerche più efficienti ed efficaci; da qui nasce l'esigenza di avere delle tecniche innovative per la memorizzazione e l'indicizzazione dei documenti XML. Nel corso degli ultimi anni sono state introdotte diverse tecniche, e diversi sistemi di memorizzazione basati su di esse, ognuno con particolari obiettivi e caratteristiche. Alcuni di questi sistemi nascono in ambito accademico (*Fabric, Natix, ToX, STORED, Lore, TReSy*, *XSet*, *Bus*, ecc*.)*, altri sono sistemi commerciali *(Tamino, XYZ-Find)* e per essi non si conoscono tutti i dettagli implementativi. In generale tutti possono essere ricondotti a quattro gruppi principali:

- sistemi orientati all'*information retrieval*;
- sistemi *flat*;
- sistemi orientati alle basi di dati;
- sistemi nativi.

I sistemi orientati all'*information retrieval* memorizzano i documenti XML come se fossero dei generici file testuali; la struttura interna fornita dai *tag* non viene analizzata o memorizzata, ma tutte le interrogazioni vengono effettuate utilizzando i meccanismi di ricerca sui file testuali e delle operazioni di trasformazione sulle stringhe. Ad esempio, tutti gli autori nel documento di figura 1.1 possono essere ritrovati cercando la stringa '<AUTORE'. In realtà questo approccio presenta 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 questa non esplicitamente indicizzata una ricerca sui tag e sugli attributi richiede delle operazioni di *postprocessing* che possono risultare computazionalmente molto costose. Il sistema *Tox* (sezione 1.5.5) contiene, fra gli altri, una componente orientata all'*information retrieval* per effettuare ricerche su documenti testuali.

I sistemi *flat* memorizzano il documento possibilmente in forma compressa e ogni operazione su di esso avviene accedendovi e utilizzando procedure basate sul modello ad eventi SAX (*Simple Api for XML)* **[SAX]** o su quello ad oggetti DOM (*Document Object*

*Model)* **[DOM]**. Il problema di questo approccio è dato dall'eccessiva lentezza, dovuta al fatto che ogni volta il documento deve essere esaminato interamente per scoprire, eventualmente, che non contiene una risposta all'interrogazione effettuata dall'utente. Nel caso del modello DOM inoltre è necessaria una notevole quantità di memoria, in quanto ogni volta deve essere costruito l'albero del documento. Il sistema *Tox* (sezione 1.5.5) contiene una componente che indicizza i documento costruendo l'albero DOM.

I sistemi *orientati alle basi di dati* rappresentano l'albero associato al documento XML attraverso delle *tuple* o degli insiemi di oggetti memorizzati all'interno di un DBMS. Per accedere al documento si considera la sua struttura ad albero. Nel caso *DBMS ad oggetti* tutti i nodi sono degli oggetti, e per navigare nell'albero si accede ai singoli identificatori. Nel caso di *DBMS relazionali*, per navigare all'interno dell'albero si utilizzano delle clausole *select-from-where*, mentre le interrogazioni vengono realizzate basandosi sulle tipiche operazioni di *join* e di *scan*. I vantaggi di questo approccio sono dati dal fatto che è abbastanza semplice costruire e memorizzare una struttura ad albero, soprattutto nel caso di basi di dati *object oriented*, ed inoltre le tecniche e linguaggi di interrogazione sono ben noti: si tratta essenzialmente di SQL (*Structured Query Language)* e OQL *(Object Query Language)*. La maggior parte dei dati oggi in formato XML ha origine da basi di dati e la memorizzazione dei dati può inoltre beneficiare delle ottimizzazioni previste dalla maggior parte dei DBMS. Gli svantaggi legati a questo approccio sono relativi alla memorizzazione dei dati e della loro struttura, la quale ha dei costi notevoli dal punto di vista dell'occupazione in spazio, rendendone proibitivo l'utilizzo anche per collezioni di moderate dimensioni. Inoltre alcuni tipi di interrogazione possono dar vita ad una visita completa dell'albero che può risultare molto lenta. Infine alcuni sistemi per migliorare alcuni tipi di interrogazioni si servono di indici aggiuntivi che aumentano ulteriormente l'occupazione in spazio. Un esempio di sistema orientato alle basi di dati (in questo caso relazionali) è *STORED* (sezione 1.5.4).

I sistemi nativi si occupano non solo dell'interrogazione, ma anche della *memorizzazione* dei documenti. In pratica essi implementano tutte le funzionalità. Questo consente ai progettisti di avere il pieno controllo su ogni dettaglio del sistema, implementando così algoritmi più specifici e tecniche di memorizzazione più efficaci, ma al tempo stesso facendosi carico di tutti gli aspetti computazionali che i sistemi orientati alle basi di dati delegano ai DBMS. Esempi di sistemi nativi sono *Natix* (sezione 1.5.1), *Lore* (sezione 1.5.3) *, TReSy* (sezione 1.5.6) e il nostro *XCDE* (sezione 1.5.9).

Un discorso a parte lo meritano i sistemi commerciali, come *XYZ-Find* di XYZ-Find Corp. **[XYZ-Find]** e *Tamino* di Software AG **[Tamino]**. Si tratta di sistemi sicuramente *nativi*, come dichiarato dai rispettivi produttori ma dei quali non abbiamo alcuna descrizione della loro implementazione.

### 1.5.1 Natix

Il sistema Natix è stato realizzato presso l'Università di Mannheim **[Kanne]** ed è un sistema di memorizzazione di documenti XML con dei moduli aggiuntivi (non ancora interamente implementati) di interrogazione, accesso e modifica della collezione dei documenti. L'elemento fondamentale del sistema è un gestore del disco responsabile della

sua paginazione. Esso fornisce uno spazio di memorizzazione organizzato in segmenti, cioè in un insieme lineare di pagine di disco di uguali dimensioni. Le pagine possono avere dimensioni di vari K (32K, ad esempio), e sono divise in diversi *slot* a cui possono essere assegnati uno o più *record*. Sopra questo *file system* esiste un sistema di *mapping* del documento che traduce la struttura dell'albero logico degli elementi XML in una struttura fisica che viene partizionata tra le pagine del disco.
L'elemento base è il *record*, che può contenere diversi tipi di informazioni, come interi sottoalberi, ma anche oggetti *proxy* che consentono di collegare diverse pagine fisiche nel caso che le dimensioni dell'albero logico abbiano costretto il sistema a partizionarlo. Il partizionamento dell'albero logico all'interno delle pagine fisiche può essere controllato dall'utente, entro un certo limite, attraverso dei parametri di configurazione. Questa funzionalità è utile perché consente di lasciare un minimo di controllo sulla memorizzazione all'utente, che può per esempio decidere di inserire gli stessi elementi sempre nella stessa pagina per renderne più efficiente l'accesso.
Il modello Natix essenzialmente si rivolge alla memorizzazione di documenti provenienti da basi di dati perché, grazie alla presenza della struttura ad albero e degli *slot*, l'iterazione con dati strutturati e di piccole dimensioni (quali i record) in un DB può risultare più semplice, mentre la scansione ed estrazione di parti più ampie di documenti, tipiche di un orientamento all'information retrieval, può risultare più lenta.
La critica che si può muovere al modello Natix riguarda innanzitutto la sua notevole occupazione in spazio. Ad esempio, per un file di 8Mb si produce un file indice di almeno 16-18Mb. Inoltre esistono i problemi tipici della gestione degli alberi, come ad esempio effettuare la divisione di un sottoalbero in più pagine di memoria fisica, divisione la cui gestione da parte dell'utente risulta piuttosto complessa. Anche la funzionalità di raggruppamento dei sottoalberi su singole pagine di memoria potrebbe rivelarsi inattuabile, per le limitazioni dovute alle dimensioni della pagina fisica di memoria, con conseguente peggioramento delle prestazioni del sistema.

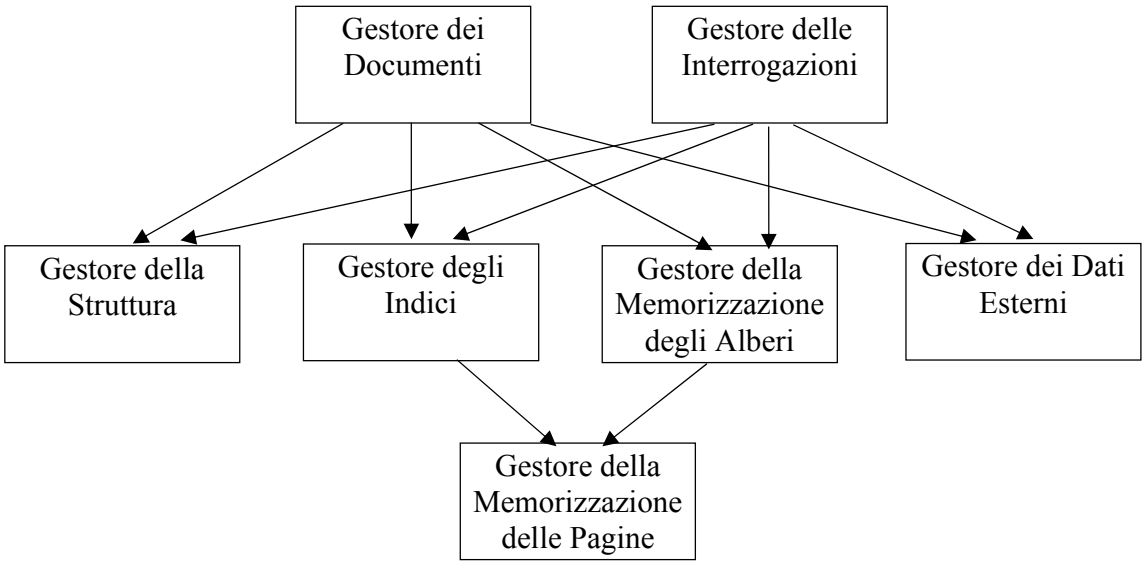



*Figura 1.5 - vista architetturale del sistema Natix*

### 1.5.2 Fabric

Il sistema Fabric **[Cooper]** è stato ideato da un gruppo di ricercatori delle Università di Stanford e Berkeley, ed è essenzialmente un'interfaccia a un generico *storage manager* che velocizza gli accessi ai documenti introducendo il concetto di indicizzazione dei cammini. In pratica vengono create delle stringhe di lunghezza arbitraria denotanti cammini nell'albero del documento XML. Queste stringhe sono poi memorizzate in un indice, ottimizzato per ricerche *full-text*. Analizzando la figura 1.5 che contiene la struttura dell'intero sistema Natix, si osserva che Fabric potrebbe essere affiancato al suo gestore degli indici.

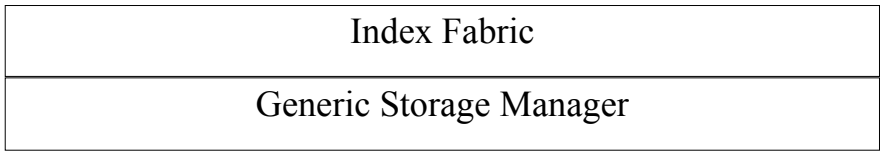



*Figura 1.6 - posizione di Fabric in un generico sistema di memorizzazione di documenti*

La struttura dati utilizzata per la memorizzazione dei cammini è un *Patricia Trie*. Essa consente di inserire un elevato numero di stringhe in modo compatto ed efficiente. L'indice inoltre risulterà abbastanza bilanciato, così che gli accessi ad una qualsiasi stringa avranno lo stesso costo, e saranno molto efficienti. I tipi di cammino creato possono essere:

- *raw paths*: si tratta di cammini che effettivamente esistono all'interno del documento originale indicizzato;

- *refined paths*: si tratta di cammini virtuali, introdotti per ottimizzare interrogazioni critiche, o per realizzare particolari interrogazioni come quelle che prevedono l'espansione di caratteri speciali.

La creazione dei cammini avviene attraverso i cosiddetti *indicatori*: ad ogni *tag* viene assegnata una stringa unica di uno o più caratteri che servirà ad identificarlo. Attraverso una sequenza di questi identificatori si può rappresentare un percorso nell'albero del documento XML. Tutti gli indicatori sono inseriti all'interno di un *dizionario degli indicatori* che sarà utilizzato per la traduzione dei cammini e per realizzare le interrogazioni. In caso di aggiornamento della base dei dati sarà aggiornato anche il *dizionario degli indicatori*, rispettando sempre la condizione di unicità.
Ad esempio, per:

```
<COLLEZIONE_LIBRI>
 <LIBRO>
   <TITOLO>Per chi suona la campana</TITOLO>
…
</LIBRO>
…
</COLLEZIONE_LIBRI>
```

gli indicatori potrebbero essere **C** per il *tag* COLLEZIONE_LIBRI, **L** per LIBRO e **T** per

TITOLO. La stringa creata sarebbe:

**C L T Per chi suona la campana**

Fabric, essendo un indice, non si lega ad una particolare sistema per la memorizzazione di documenti. Esso potrebbe essere ad esempio un *DBMS*, sia relazionale che ad oggetti. I *path* possono essere associati ad indici che si riferiscono ad un semplice file testuale, a delle *tuple* di un sistema relazionale, o ad oggetti di una base di dati XML.
Per quanto riguarda le interrogazioni, una volta formulate esse sono soggette prima ad una trasformazione, poi ad una ricerca all'interno del *Patricia Trie* e infine ad un ulteriore trasformazione per accedere agli elementi che costituiscono il risultato della ricerca. Può accadere, per alcuni tipi di interrogazioni (come ad esempio quella in cui non tutta la struttura del *cammino* viene specificata) che debba essere esplorato tutto il *Patricia Trie*, incorrendo così in un alto costo computazionale. Inoltre, siccome viene creato un cammino per ogni elemento testuale, l'occupazione in spazio rende difficili eventuali ricerche di prossimità. Si osservi che l'occupazione in spazio del sistema Fabric aumenta considerevolmente quella dello *storage manager*, considerato che si aggiunge allo spazio occupato per la memorizzazione del documento.

### 1.5.3 Lore

Lore (*Lightweight Object Repository)* è un DBMS specifico per la gestione di informazioni semistrutturate realizzato all'Università di Stanford **[McHugh]**. Il sistema consente di utilizzare tutte le caratteristiche di un DBMS tradizionale (sicurezza, accessi efficienti, recupero in caso di errori, controlli di concorrenza, interrogazioni ottimizzate, ecc.) consentendo al tempo stesso una certa flessibilità, necessaria per la memorizzazione di documenti XML: i documenti XML non corrispondono infatti sempre a degli schemi rigidi; la loro stessa struttura può essere eterogenea. La scelta del gruppo di progettisti di Lore è stata quella di realizzare interamente il sistema di memorizzazione dei documenti, in modo da avere il controllo su tutte le componenti del sistema. Il modello seguito per la rappresentazione del documento XML è quello OEM *(Object Exchange Model)*. Il documento viene rappresentato come un *grafo orientato ed etichettato*. I vertici del grafo sono oggetti; ad ogni oggetto è associato un identificatore unico. Gli oggetti che non hanno archi uscenti sono detti *atomici*, e ad essi sono associati dei valori di tipo primitivo, come interi, reali, caratteri, ecc.; gli oggetti non atomici sono detti *complessi.* Il modello è orientato a descrivere dati semistrutturati ed eterogenei, visto che tutte le informazioni sono ottenute attraverso gli identificatori degli oggetti. Nella figura 1.7 è possibile vedere un modello OEM parziale (cioè non contenente tutti i dati) del documento di figura 1.1. Il linguaggio di interrogazione di Lore, chiamato *Lorel* (*Lore Language)* **[Abiteboul]**, prevede delle forme di navigazione dichiarativa che consentono di selezionare dei cammini e degli oggetti all'interno dell'OEM. Il modulo per la gestione delle interrogazioni consiste di un compilatore per il linguaggio di interrogazione utilizzato, il *Lorel* (*Lore Language)* appunto, e si basa sul concetto di *cammino.* Esso è costituito da un nome di un oggetto, seguito da una sequenza di nomi di archi, separati da punti. Per esempio:

**COLLEZIONE_LIBRI.LIBRO.COPERTINA**

è un cammino che individua tutti gli oggetti COPERTINA all'interno del documento. Un principio fondamentale di *Lorel* è che, al momento della scrittura del cammino, l'utente non debba preoccuparsi delle possibili irregolarità del grafo. In questo caso non bisogna preoccuparsi del fatto che non tutti gli elementi LIBRO contengano l'elemento COPERTINA. Le irregolarità sono risolte al momento dell'esecuzione di un'interrogazione. Esiste un meccanismo di *general path expression* (che si affianca ai cammini visti finora, chiamati *simple)*, attraverso il quale si può navigare in forma dichiarativa nel grafo basandosi su un sistema di *pattern*. I *pattern* possono essere applicati al cammino per individuare un insieme di oggetti complessi o atomici. Per esempio con COLLEZIONE_LIBRI.L% si individuano tutti i cammini che partono dal nodo COLLEZIONE_LIBRI e seguono su un arco che inizia con L. Il simbolo ? indica che il cammino è opzionale, mentre | indica una scelta fra i cammini. L'intera sintassi si basa sulle espressioni regolari.

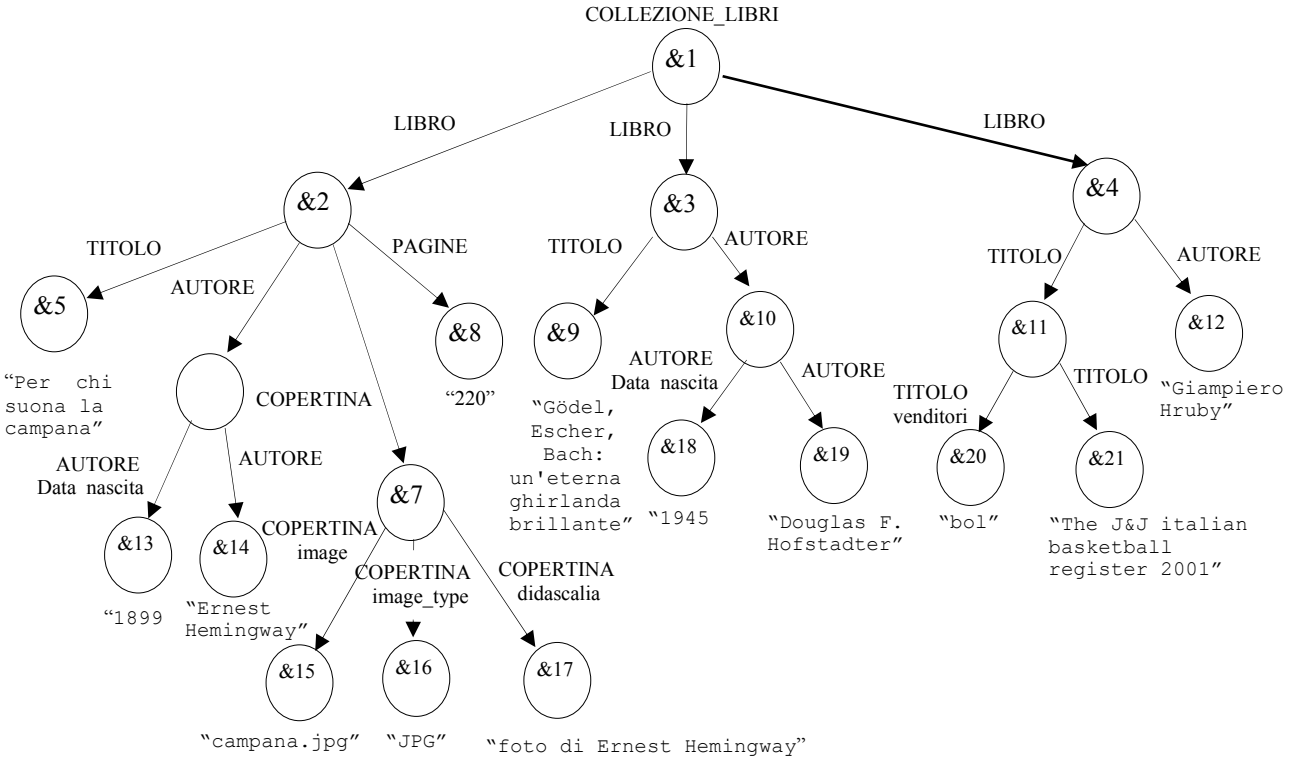



*Figura 1.7 - modello OEM parziale del documento di figura 1.1*

Un esempio di interrogazione è il seguente:

**select** COLLEZIONE_LIBRI.TITOLO.AUTORE
**where** COLLEZIONE_LIBRI.(PAGINE)?
**like** "220"

in sui si chiede di selezionare i nomi degli autori dei libri il cui tag PAGINE contenga

il valore 220. La risposta a questa interrogazione è

Ernest Hemingway

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 di *update*.

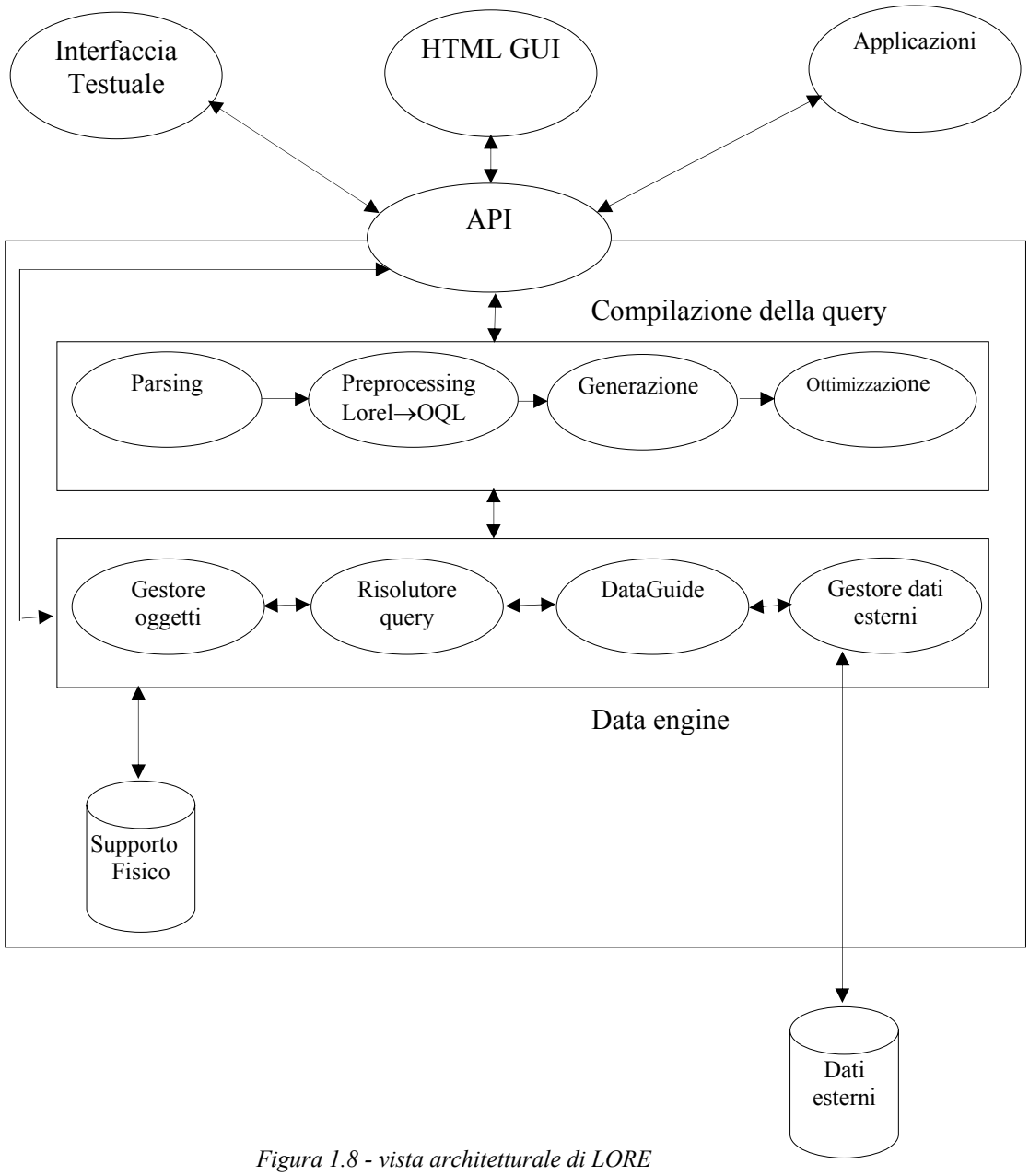



*Figura 1.8 - vista architetturale di LORE*

Possiamo vedere nella figura 1.8 una schematizzazione della struttura del sistema LORE. Le componenti fondamentali sono:

- le **API**: un insieme di classi C++ utilizzate per interfacciarsi con il sistema e costruire applicazioni *client*;

- il **modulo per la gestione delle interrogazioni**: che effettua delle operazioni di compilazione e ottimizzazione delle *query* prima di risolverle. La *query* è sottoposta ad un *parsing*, viene costruito un albero della query che in seguito è tradotto in formato OQL (*Object Query Language)*. A partire da questo formato si eseguono delle operazioni di *scan*, *select* e *join* sui dati o degli accessi a degli indici che sono creati per ottimizzare l'esecuzione dell'interrogazione. Gli eventuali risultati sono sempre degli oggetti, figli di un nuovo oggetto risultato, a cui è possibile accedere attraverso le API;

- il **motore**: che gestisce i dati veri e propri, effettuando le trasformazioni nei diversi formati fisici e logici necessari.

Ognuna di queste componenti è divisa a sua volta in una serie di moduli. All'interno del motore il modulo **gestore degli oggetti** si occupa di tradurre i riferimenti agli oggetti dello schema OEM nelle corrispondenti posizioni fisiche. Le pagine fisiche del disco sono divise in *slot* e ad ognuno di essi è assegnato un singolo oggetto. Gli oggetti possono avere dimensioni differenti e quindi sono più *slot* possono essere allocati ad un oggetto. Esiste inoltre una funzionalità di *garbage collector* attraverso la quale gli oggetti che non possono essere raggiunti vengono rimossi. Il **DataGuide** fornisce una descrizione dinamica della base di dati in modo da sopperire alla mancanza di una struttura fissa, ed è fondamentale per esplorare la sua struttura, così come per formulare ed ottimizzare le interrogazioni. Senza di esso il motore sarebbe costretto ad esplorare tutta la base di dati per reperire informazioni sulla sua struttura, operazione costosissima nel caso di collezioni di documenti molto ampie. Il **gestore dei dati esterni** consente di integrare dinamicamente dati provenienti da fonti esterne, ad esempio dalla rete, mascherando agli altri moduli questa operazione. Il modulo per la gestione delle interrogazioni rappresenta un compilatore per il Lorel. Le interrogazioni sono soggette ad un preprocessing: se contengono solo *simple paths* (cioè cammini senza espressioni regolari) questi sono sostituiti con nomi di variabili. Non è possibile fare la stessa cosa per i *general paths*, che richiedono un meccanismo di sostituzione a tempo di esecuzione. Il motore memorizza due indici, chiamati **L-index** e **V-index**. L'**L-index** *(Link index)* consente di ricondursi dalla coppia formata dall'identificatore di un oggetto a dall'etichetta di un arco a tutti gli oggetti che sono padri dell'oggetto discendente attraverso un arco con l'etichetta specificata; si tratta quindi di un indice di puntatori agli elementi padre. Il **V-index** *(Value Index)* consente di trovare, data una tripla etichetta-operatore-valore, tutti gli oggetti atomici aventi un arco entrante con l'etichetta specificata ed aventi un valore che soddisfa l'espressione descritta dall'operatore e dal valore indicato nella tripla. Per risolvere il problema del confronto fra tipi differenti vengono in realtà memorizzati tre indici diversi: per le stringhe, per i numeri interi e i reali, e per le stringhe che possono essere tradotte in numeri. Una delle caratteristiche più interessanti di Lore è di poter gestire i dati irregolari. Inoltre non ha bisogno della DTD, implementando il meccanismo della *DataGuide* e permettendo, tramite il linguaggio *Lorel*, non solo l'interrogazione ma

anche l'aggiornamento dei dati.
Esistono però in *Lore* dei problemi di ottimizzazione dello spazio e, soprattutto, delle interrogazioni: la presenza dei soli indici V e L non consente di limitare sufficientemente il numero delle operazioni di visita dell'albero, che sono costose, e aumenta al tempo stesso l'occupazione in spazio dei singoli documenti, che può diventare proibitiva anche per collezioni di moderate dimensioni.

### 1.5.4 STORED

STORED (*Semistructured TO Relational Data)* **[Deutsch]** è un sistema che si basa su un DBMS relazionale per la memorizzazione di documenti XML. Il sistema non necessita della DTD dei documenti. Essi sono tradotti dal modello dei dati semistrutturato al modello relazionale, e le interrogazioni avvengono utilizzando un particolare linguaggio di interrogazione. Gli elementi che non sono contenuti nelle tabelle relazionali, a causa della loro irregolarità o perché inseriti successivamente, sono memorizzati in formato nativo. L'obiettivo del sistema è di generare delle tabelle relazionali con occupazione spaziale minima e riduzione della frammentazione dei dati. Siccome questo problema è NP-Completo, gli autori propongono un algoritmo euristico di tipo *data mining*. Il formato dei dati utilizzato per rappresentare il documento XML è quello OEM già descritto nella sez. 1.5.3 parlando del sistema *Lore*. I dati vengono tradotti dal formato OEM ad un formato misto, che contiene uno *schema relazionale* con le relazioni e le loro cardinalità. Un "traduzione" di STORED consiste essenzialmente in una clausola *from where store*, attraverso la quale si selezionano gli elementi da memorizzare. Per esempio, per il documento di figura 1.1 possiamo introdurre le seguenti tabelle, in cui l'elemento *object identifier* rappresenta la chiave:

Tabella COLLEZIONE_LIBRI.LIBRO

| Oid | TITOLO | AUTORE | PAGINE | COPERTINA |
|---|---|---|---|---|
| &1 | Per chi suona la campana | &5 | 220 | &9 |
| &2 | Senilità | &6 | NULL | NULL |
| &3 | Gödel, Escher, Bach:un'eterna ghirlanda brillante | &7 | NULL | NULL |
| &4 | The J&J italian basketball register 2001 | &8 | NULL | NULL |

Tabella COLLEZIONE_LIBRI.LIBRO.COPERTINA

| Oid | Image | image_type | didascalia |
|---|---|---|---|
| &9 | campana.jpg | JPG | Foto di Ernest Hemingway |

Tabella COLLEZIONE_LIBRI.LIBRO.AUTORE

| Oid | Autore | Data Nascita |
|---|---|---|
| &5 | Ernest Hemingway | 1899 |
| &6 | Italo Svevo | 1861 |
| &7 | Douglas F. Hofstadter | 1945 |
| &8 | Giampiero Hruby | NULL |

*Figura 1.9 - tabelle relazionali costruite da STORED per il documento di figura 1.1*

Nelle tabelle gli oggetti sono stati divisi in base al loro tipo e alla loro composizione: nonostante ciò esistono diversi elementi vuoti, che sono presenti nelle righe corrispondenti ad oggetti non uniformi. La scelta del sistema STORED è quella di non memorizzare la struttura dei dati insieme ai dati stessi. Nel caso siano aggiunti degli elementi non contemplati nelle tabelle precedenti (questo vale per i documenti senza DTD) essi sono inseriti in formato nativo. L'accesso a questi dati potrebbe quindi risultare inefficiente, ma la probabilità che si abbia necessità di accedervi è piuttosto bassa.
Un esempio di "traduzione" è il seguente:

```
M1=FROM COLLEZIONE_LIBRI.LIBRO : $X
    { TITOLO: $T,
      AUTORE: $A
      OPT { PAGINE: $P }
      OPT { COLLEZIONE_LIBRI.LIBRO.COPERTINA: $XC }
     }
    STORE($X, SA, $P, $XC)
```

Gli elementi selezionati dalla OEM vengono memorizzati nella tabella COLLEZIONE_LIBRI.LIBRO. La clausola OPT indica che gli elementi racchiusi al suo interno sono opzionali. Ad ogni regola di traduzione corrispondono delle regole di inversione che esprimono le interrogazioni sul modello relazionale.
Il problema fondamentale di STORED riguarda i documenti altamente irregolari, per i quali le tabelle costruite possono risultare inefficienti, prevedendo un alto numero di elementi memorizzati in formato nativo e quindi accessibili con elevati costi computazionali. Inoltre il linguaggio di interrogazione non è abbastanza potente per esprimere interrogazioni complesse, ad esempio contenenti espressioni regolari. Il sistema dovrebbe essere quindi affiancato da un sistema di indici, ad esempio *Fabric* (sezione 1.5.2), ma questo comporterebbe un aumento notevole dello spazio occupato ed un rallentamento degli accessi. Questa soluzione è stata comunque applicata in *ToX* descritto nella sezione che segue.

### 1.5.5 ToX

Nel sistema *ToX – Toronto XML Engine* **[Barbosa]** vengono applicate tutte le tecnologie descritte finora. Esso è ancora in via di realizzazione ed è stato progettato da

alcuni ricercatori dell'Università di Toronto in collaborazione con l'IBM. Alla base del sistema c'è il concetto di *caratterizzazione automatica* dei documenti. I documenti vengono cioè catalogati in base ad un insieme di caratteristiche detto *strutturazione*. I documenti testuali hanno una *bassa strutturazione*, mentre i documenti provenienti da basi di dati hanno un'*alta strutturazione*. Ci si aspetta che documenti simili abbiano lo stesso grado di strutturazione. Si tratta di un approccio che è già stato utilizzato nella compressione semantica dei documenti XML introdotta in *Xmill* **[Liefke]**. Nel motore vengono *memorizzati* ed *indicizzati* sia documenti reali che documenti virtuali, cioè documenti presenti in rete. I documenti possono essere memorizzati come file testuali, all'interno di basi di dati relazionali o basi di dati ad oggetti. La scelta dipende dalla proprietà di *strutturazione* del documento. Anche gli indici sono calcolati e memorizzati in base al tipo di documento. Per ogni documento si memorizzano i *dati* e la *struttura*, per eseguire ed ottimizzare le interrogazioni. Gli elementi base del sistema sono i singoli *documenti* che possono essere memorizzati, indicizzati e, eventualmente, rimossi dal sistema.
L'architettura del sistema ToX è illustrata di seguito:

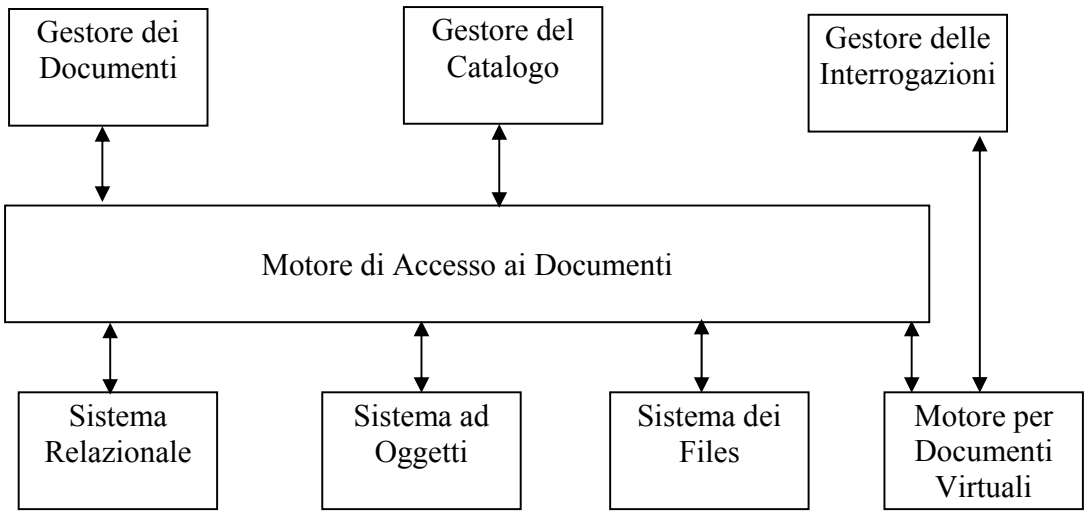



*Figura 1.10 - vista architetturale di ToX*

I moduli che interfacciano il sistema con gli utenti sono il *gestore dei documenti* e il *gestore delle interrogazioni*. Il primo permette all'utente di inserire, modificare e rimuovere i documenti. Inoltre esso si occupa di determinare il tipo di memorizzazione ed indicizzazione più adatti. Il secondo si occupa di fornire un'interfaccia per formulare ed eseguire le interrogazioni. È prevista la possibilità di utilizzare nel sistema diversi linguaggi di interrogazione, fra cui *XPath* **[XPath].** Il cuore del sistema è il *motore di accesso ai documenti*, che funziona come un'interprete fra le richieste dei moduli superiori e i moduli inferiori che memorizzano effettivamente la collezione. Il *gestore del catalogo* si occupa di memorizzare la struttura dei documenti. Esso supporta DTD e XML Schema, e vi si accede per rispondere alle interrogazioni. La base di dati ad oggetti viene utilizzata per la

memorizzazione della struttura del documento sotto forma di albero DOM, mentre la base di dati relazionali è utilizzata per memorizzare informazioni sugli elementi del documento. Per il problema dell'indicizzazione dei documenti il meccanismo utilizzato è il *ToX Indexing Mechanism* (*ToXin)*. Esso è stato progettato per consentire un rapido accesso agli elementi dell'albero XML e una rapida navigazione al suo interno. È formato da due indici distinti:

- *indice dei percorsi*: che consente una valutazione efficiente delle espressioni regolari sui percorsi;

- *indice dei valori*: per localizzare i nodi dell'albero in base al loro valore.

Il problema fondamentale di *ToXin* è che gli indici sono costruiti a partire dall'albero DOM del documento, il quale deve essere tenuto in memoria. La consultazione può risultare lenta, perché ogni volta deve essere ricostruito l'albero suddetto.
Il sistema ToX è ancora in fase di realizzazione, non vengono quindi forniti dettagli implementativi, anche se già sono stati proposti diversi miglioramenti, soprattutto relativi al meccanismo di indicizzazione. Uno dei suoi punti deboli è il sistema di valutazione della strutturazione del documento, di cui non vengono forniti dettagli, e che può dare origine a inefficienze nel caso in cui la catalogazione dei documenti non sia effettuata in maniera corretta. Il fatto inoltre che le *query* debbano accedere agli indici del documento dopo averne costruito il DOM Tree rallenta la loro esecuzione e richiede una notevole quantità di memoria.

### 1.5.6 TReSy

Il sistema *TReSy (Text Retrieval System)* **[Corti]** è frutto della collaborazione fra ricercatori del Dipartimento di Informatica dell'Università di Pisa e la Scuola Normale Superiore. Esso è orientato al suporto di ricerche su documenti XML/SGML a forte contenuto testuale. La caratteristica fondamentale di TReSy sono i suoi due indici: un *indice sui testi* e un *indice sui marcatori*. L'indice sui testi viene implementato attraverso lo *String B-Tree* **[Ferragina99]**, una struttura dati progettata specificatamente per archiviare e recuperare efficientemente informazioni da memorie a più livelli. Essa permette in particolare di effettuare ricerche su *pattern* quali parole singole, parole adiacenti, sillabe o sequenze arbitrarie di caratteri. TReSy permette inoltre di personalizzare il tipo di indicizzazione, in base all'uso che se ne intende fare, senza alterare i documenti archiviati. Ad esempio si può decidere di realizzare l'indicizzazione in modo *case insensitive*. Per l'indicizzazione degli elementi marcatori dei documenti si è invece applicata una soluzione diversa, e cioè un *Interval Tree* modificato in modo da contenere non solo le informazioni sui *tag*, ma anche la lista dei loro attributi con il relativo valore. Con l'*Interval Tree* le ricerche per nomi di *tag* possono essere realizzate in maniera molto efficiente, mentre per altri tipi di ricerche, quali quelle sui valori degli attributi, è necessaria la visita dell'intero albero.
Un altro problema di TReSy è dato dall'eccessiva occupazione in spazio dei suoi due i che è di almeno un fattore moltiplicativo 10 rispetto alle dimensione della collezione indicizzata.

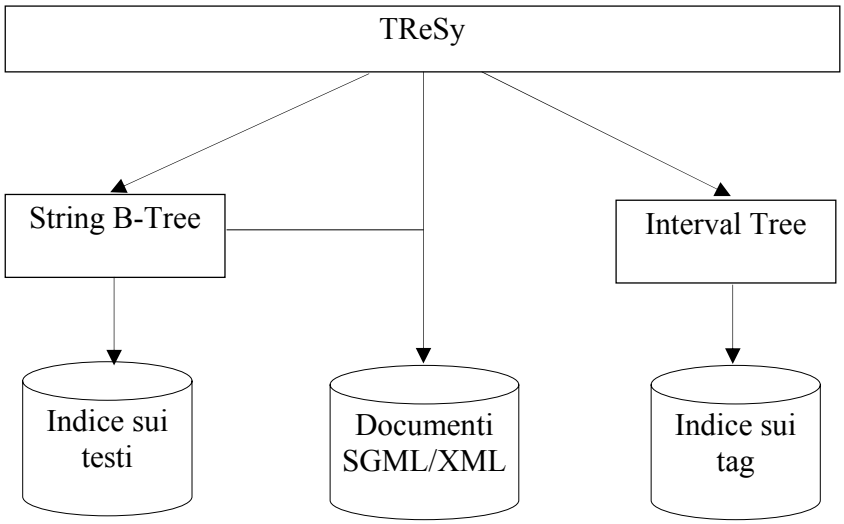



*Figura 1.11 - schema logico del sistema TReSy*

### 1.5.7 BUS

Il sistema BUS *(Bottom Up Scheme)* **[Shin]** è stato progettato da alcuni ricercatori dell'Università di Chungnam (Corea del Sud). Esso è un sistema che ha come obiettivo quello di minimizzare lo spazio occupato dagli indici consentendo delle risposte ragionevolmente efficienti alle interrogazioni. Il sistema indicizza solo la parte testuale del documento, cioè le foglie dell'albero XML, mentre la struttura viene ricreata al momento dell'interrogazione. Il meccanismo per la memorizzazione degli indici è quello delle liste invertite. Ad ogni elemento è associato un *identificatore (GID – General Element Identifier)* che è in realtà formato da una quadrupla (numero del documento, identificatore dell'elemento nel documento, livello dell'elemento e tipo), che permette di ricondursi dall'elemento alla sua lista invertita. Il sistema BUS ha dato buoni risultati per quanto riguarda il contenimento delle dimensioni degli indici (ad esempio per una collezione di 256Mb gli indici occupano 119.66Mb), ma bisogna considerare che essi contengono solo una parte delle informazioni e per tutte le altre è necessario ricostruire a tempo di interrogazione tutta la struttura del documento, determinando così un notevole rallentamento nell'esecuzione delle *query* strutturali.

### 1.5.8 XSet

Il sistema XSet **[Zhao]** è stato ideato da un gruppo di ricercatori dell'Università di Berkeley. Rispetto ai sistemi descritti precedentemente esso ha una caratteristica

fondamentale: si tratta di un sistema interamente *in-memory*. Viene costruito un albero (utilizzando i puntatori) che descrive la struttura dei documenti. Questo albero è chiamato *super-tree*. Ogni nodo dell'albero è associato ad un *tag*, e punta ad una struttura dati *Treap* (essenzialmente un albero binario di ricerca bilanciato) i cui nodi sono puntatori ad una struttura chiamata *ContextEntry Object*, che contiene informazioni riguardanti il valore del *tag* e il contesto in cui si trova. XSet, essendo *in-memory*, richiede una notevole quantità di memoria per il server (anche 2 o 3 Gb), in quanto esso indicizza un documento in uno spazio che è almeno 3 volte la sua dimensione. Inoltre il sisema XSet assume implicitamente che i *tag* abbiano valori atomici (ad esempio numeri o singole parole di testo), a causa della presenza del *Treap*, e quindi è impossibilitato a gestire documenti XML generici.

### 1.5.9 La nostra proposta: XCDE Library

XCDE è una libreria per la *compressione*, *l'indicizzazione* e *l'interrogazione* 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:

- orientato 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 XCDE 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*: XCDE 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.

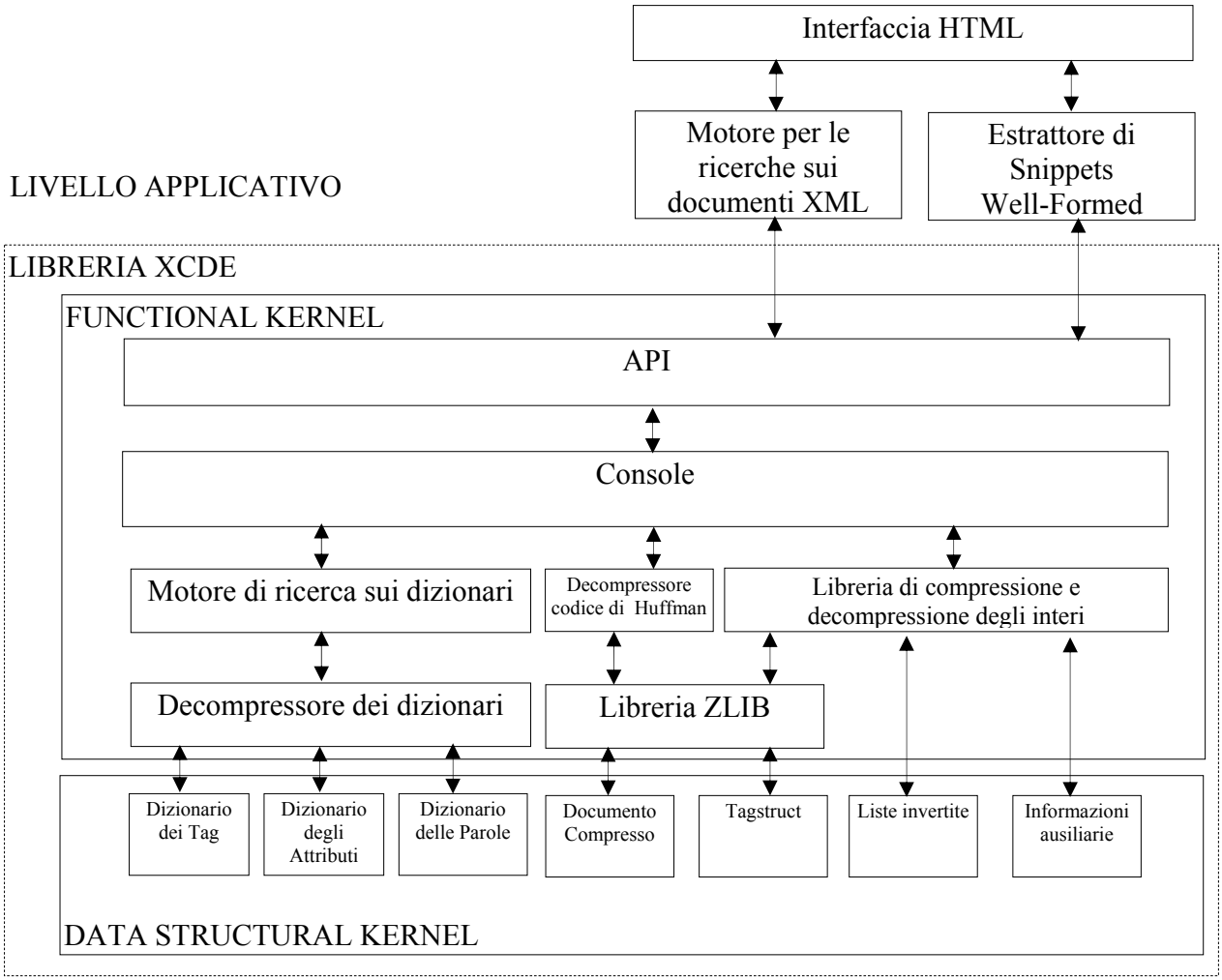



*Figura 1.12 - schema a livelli della libreria XCDE*

La libreria XCDE 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 *XCDE Search Engine*. Questo motore (descritto diffusamente nel

capitolo 4) consente di effettuare l'estrazione di *snippets ben formati* e di realizzare interrogazioni avanzate, implementando ricerche per *parola*, *prefissi*, *suffissi*, *sottostringhe* e *prossimità*.
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 in in un certo senso 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 gli elementi 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 e consentirne un rapido accesso. Agli elementi dei dizionari sono associati delle *liste invertite*, che 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 strutture dati concepita per indicizzare in modo compatto la struttura dei dati 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 orientati 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. Descriveremo nel dettaglio tutte le caratteristiche della libreria nei capitoli successivi. La libreria XCDE, rispetto ai sistemi descritti nelle sezioni precedenti, consente di ottimizzare lo spazio di memorizzazione delle informazioni garantendo al tempo stesso il supporto efficiente a 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 le dimensioni originali dei documenti.

## NOTA BIBLIOGRAFICA

In questo capitolo è stata fornita una descrizione a grandi linee del linguaggio XML. Esso è un argomento in continua evoluzione e quindi rimandiamo per ulteriori approfondimenti al sito della W3C **[XML]**. I sistemi di memorizzazione e indicizzazione sono tutti descritti a partire dagli articoli dei progettisti citati nel testo e tutti indicati in bibliografia. La maggior parte di questi sistemi è ancora in piena fase di realizzazione, per cui il più delle volte mancano *benchmark* significativi sulle loro prestazioni. Per una descrizione generale del meccanismo di indicizzazione delle liste invertite, citato a proposito di BUS e XCDE, rimandiamo a **[Witten], pagg. 109-114 e capitolo 3** del medesimo testo.

# Capitolo 2

# Le strutture dati di XCDE

*Abbiamo visto nel capitolo precedente come la libreria XCDE basi il proprio funzionamento su un nucleo di strutture dati necessarie per realizzare le operazioni sui documenti. Utilizzando come guida lo schema a livelli della figura 1.12, in questo capitolo forniremo una descrizione più approfondita delle strutture dati e della loro costruzione, fornendo inoltre le motivazioni che sono alla base delle nostre scelte implementative.*

## 2.1 Le due viste di XCDE

Abbiamo già evidenziato in sezione 1.4 di come la presenza delle *entità* all'interno dei documenti XML porti alla distinzione tra il documento fisico, cioè il sorgente, e il documento logico, ossia ciò che il documento effettivamente descrive. Così per il documento di figura 1.1 agli 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>
```

La libreria XCDE si occupa non solo della *memorizzazione* dei documenti *fisici*, ma anche

dell'*indicizzazione* della loro versione *logica*. La memorizzazione avviene servendosi di un algoritmo di compressione *lossless*, quindi senza perdita di dati (sezione 2.4). Il documento, una volta decompresso, sarà dunque perfettamente identico al documento originale, inclusi gli eventuali riferimenti alle entità. D'altro canto vogliamo che l'espansione dei riferimenti venga considerata nella fase di *indicizzazione*, per evitarne le ricerche successive. Infatti, con riferimento all'esempio precedente, nonostante la presenza dell'entità interna *&ernie;* nel documento fisico originale, l'espansione dell'entità "*Ernest Hemingway*" è logicamente presente all'interno dell'elemento AUTORE e un'eventuale ricerca dovrebbe essere in grado di individuarla. Un ragionamento analogo può essere fatto per i *commenti*. I commenti contenuti nel documenti non devono andare persi durante la compressione, ma su di essi certamente non sarà possibile effettuate interrogazioni, in quanto non fanno logicamente parte del documento XML.

Per gestire questa situazione la libreria XCDE si basa su una *doppia vista del documento XML*:

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

Per alcuni elementi le due viste possono corrispondere, ad esempio per gli elementi che non hanno al proprio interno alcun riferimento ad entità, per altri sono diverse. L'implementazione delle due viste comporta la presenza di elementi *logici* e *fisici* all'interno delle strutture dati *dizionario*, dettagliate qui di seguito.

## 2.2 I dizionari

Riferendoci alla figura 1.12 possiamo notare come tra le strutture dati siano presenti tre dizionari: il dizionario dei *tag*, il dizionario degli *attributi* ed il dizionario delle *parole*. Il vantaggio di questa divisione è di avere dei dizionari più piccoli, su cui le operazioni di ricerca saranno più efficienti. Inoltre si riducono i casi di ambiguità, visto che *tutti gli elementi in un dizionario sono dello stesso tipo*. Ad esempio, nel caso esistano nel documento un *tag* e una parola uguali, la possibile ambiguità che si presenta durante la loro ricerca può essere risolta semplicemente interrogando il dizionario corrispondente all'elemento cercato. I dizionari contengono tutti gli elementi che costituiscono un documento XML (d'ora in poi li chiameremo *token*). Ad esempio in:

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

abbiamo il tag *AUTORE*, l'attributo *data_nascita* con il suo valore, le parole *Italo* e *Svevo* ed i diversi caratteri '<', '>', '</' e lo spazio. Ognuno di questi elementi rappresenta un *token* all'interno di un dizionario. L'utente può gestire la divisione in *token* di un documento attraverso un *file* di configurazione in cui può inserire i caratteri e le entità che sono considerati separatori in sede di *parsing*, distinguendo fra i separatori per parole e *tag* e separatori per attributi. Ogni dizionario contiene dei *token* di tipo omogeneo (*tag*, attributi, parole). In realtà, come spiegato nella sezione precedente, gli elementi del

dizionario possono essere sia *token logici* che *token fisici*, a seconda che siano stati generati dal *parsing* della vista logica o fisica del documento. I due insiemi, è bene sottolinearlo, non sono disgiunti, possono quindi esistere *token* solo logici, solo fisici o sia logici che fisici. I *token fisici* sono quelli creati per esigenze di memorizzazione dei documenti: nell'esempio precedente *Italo* rappresenta un *token fisico*, perché è effettivamente presente nel documento sorgente. I *token logici* sono quelli introdotti per esigenze di *interrogazione* dei documenti. Ad essi sono associate delle *liste invertite*, che consentono di *localizzarli* all'interno della vista logica del documento. Il *token &ernie*; è un *token fisico*: esso è effettivamente presente all'interno del documento sorgente. Ad esso non è associata una *lista invertita*, perché dal punto di vista logico *&ernie;* non è un autore, ma un riferimento al nome di un autore. L'occupazione aggiuntiva in spazio causata dalla doppia presenza di *token logici* e *fisici* può essere giudicata trascurabile a seguito di alcune considerazioni:

- Il dizionario di maggior dimensione è quello delle parole, che per i documenti testuali può arrivare al 95% delle dimensioni totali di tutti e tre i dizionari. Inoltre i dizionari saranno compressi.

- Il numero dei *token fisici* presenti all'interno del dizionario delle parole è all'incirca uguale alla radice quadrata della dimensione in *byte* del documento. Questa considerazione è una conseguenza della *legge di Heaps*, che fissa la dimensione del dizionario a $O(n^{\beta})$, con $0<\beta<1$ **[Heaps]**, e di alcune prove effettuate su varie collezioni di documenti testuali (ad esempio la collezione *TREC*, utilizzata anche in **[Witten]**) che hanno fissato il $\beta$ fra 0.4 e 0.6 **[Harman]**;

- Il numero dei *token* solo *logici* è inferiore al numero dei *token fisici*.

La gestione delle entità nella creazione dei dizionari avviene in maniera differente a seconda che si tratti di entità interne o esterne. Per le *entità interne* vengono considerate, all'interno della vista logica del documento, le loro espansioni. Le *entità esterne* non vengono espanse perché in genere si tratta di entità che non hanno un significato intrinsecamente logico, ma consentono di visualizzare i documenti senza tener conto delle differenze dovute ai diversi set di caratteri (come le entità presenti nella DTD "iso_latin.dtd" del documento di figura 1.1). Questa scelta inoltre rende la memorizzazione del documento indipendente dalla DTD, e non legata alla sua presenza. Le entità esterne non espanse sono sempre considerate dei *token fisici a se stanti*, in modo da aumentare la compressione, e se non sono indicati come separatori nel *file* di configurazione, queste vanno a formare un *token logico* nel momento in cui sono concatenate ad un altro *token fisico*.

Ad esempio, l'elemento:

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

contiene i *token fisici Senilit* e *à* e il *token logico Senilità*.
Siccome la lista invertita sarà costruita a partire da un unico insieme di *token*, formato dalle parole, *tag* e attributi (detto *dizionario globale*), abbiamo bisogno di un meccanismo per relazionare un *token* in uno dei 3 dizionari fisicamente esistenti con lo stesso *token* nel

dizionario globale (virtuale). A tale scopo associeremo ad ogni token all'interno di un qualsiasi dizionario un *identificatore unico* (che chiameremo *rango del token*). Esso rappresenta la posizione del *token* all'interno del dizionario *globale.* A sua volta il *token* è identificato da una coppia (*dizionario, rango del token nel dizionario*). La relazione tra i due identificatori è biunivoca e consente di comprimere il documento in maniera trasparente rispetto alla suddivisione dei dizionari.

### 2.2.1 Il dizionario dei tag e degli attributi

Il dizionario dei *tag* contiene tutti i *tag* presenti all'interno del documento. Per semplificare le interrogazioni gli elementi di apertura delle sezioni CDATA sono considerate dei *tag*, così come gli elementi di apertura dei commenti. All'interno del dizionario sono memorizzati esclusivamente i *nomi* dei *tag*, escludendo quindi i caratteri separatori e gli attributi. Ad esempio per l'elemento:

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

sarà memorizzato nel dizionario dei *tag* il solo *token TITOLO*.

Il dizionario degli attributi contiene i *token* formati dalle coppie (nome attributo, valore). Per ottimizzare le funzioni di ricerca il valore degli attributi vengono suddivisi in una serie di *token logici* in base ad un insieme di *caratteri separatori* stabiliti dall'utente attraverso un *file* di configurazione, tenendo eventualmente conto dell'espansione di riferimenti ad entità in essi presenti. Ad esempio, all'attributo *venditori* contenuto in

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

sono associati i seguenti *token*:

- `venditori="Barnes&Noble, Bol" : token fisico;`
- `venditori="Barnes&Noble, Bol" : token logico;`
- `venditori=Barnes&Noble : token logico;`
- `venditori=Bol : token logico;`

Questa soluzione ci consente di avere una notevole flessibilità in fase di interrogazione. Ad esempio sarà possibile effettuare ricerche sia per attributi *che sono uguali esattamente* ad un valore (ad esempio "Barnes&Noble, Bol"), sia per attributi il cui valore *contiene* un determinato *pattern* (ad esempio "Bol").

### 2.2.2 Il dizionario delle parole

Il dizionario delle parole contiene tutti i *token logici* e *fisici* che non rientrano negli altri due dizionari. Esso è in genere il dizionario di dimensioni maggiori e contiene non solo i *token*

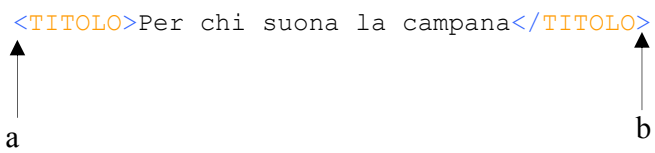
determinati a partire dal contenuto di elementi XML, ma anche tutti i *token fisici* necessari per la compressione del documento e non derivati da *tag* o attributi, come ad esempio i commenti o le direttive al *processore* o i *token* derivati da gruppi di caratteri separatori (ad esempio sequenze di spazio, caratteri di tabulazione, ecc.). I caratteri separatori sono raggruppati in sequenze che rappresentano un *token fisico*, in modo da aumentare la compressione del documento. Se un *token logico* contiene dei caratteri maiuscoli, per facilitare la ricerche *case-insensitive*, viene inserito all'interno del dizionario delle parole anche il corrispondente *token logico* derivante dalla sua trasformazione in *caratteri minuscoli*. Ad esempio, oltre al *token logico Ernest* derivante dall'espansione dell'entità interna *&ernie;*, viene inserito nel dizionario delle parole anche il *token logico ernest*. A quest'ultimo *token* sarà associata la *lista invertita* che conterrà le informazioni del *token Ernest*.

# 2.3 Le liste invertite (IL)

Le *liste invertite* (brevemente IL, per *Inverted List*) costituiscono un efficiente meccanismo di indicizzazione di documenti. Si tratta di un insieme di liste di numeri (possono essere posizioni fisiche o *ranghi* per la *proximity*) che vengono associate agli elementi di un dizionario **[Witten]**. Ad ogni *token* di un qualunque dizionario della libreria XCDE è associata una *lista invertita*, che sarà vuota per i *token* solo fisici. La presenza virtuale del dizionario *globale* ci consente di parlare da questo momento in poi di *token* senza specificare il dizionario di provenienza, interpretando così le *liste invertite* di tutti i *token* come un'unica struttura dati.

### 2.3.1 Utilizzo e topologia delle IL

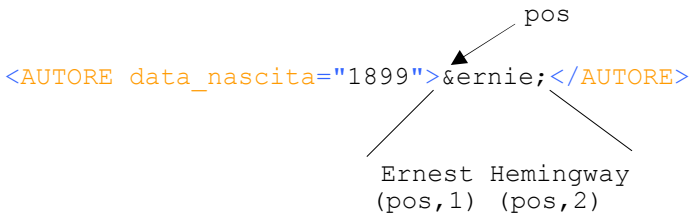
Il contenuto della lista cambia in base al tipo di elemento considerato: così per i *tag* le liste conterranno le coppie formate dalle posizioni di apertura e chiusura, per gli attributi conterranno le posizioni di apertura dei *tag* a cui gli essi si riferiscono, per le parole conterranno la posizione fisica ed eventualmente (se richiesto dall'utente) la *lista di proximity*. 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]**. Quando parliamo di *posizioni fisiche* all'interno del documento intendiamo la *posizione in byte nel documento compresso del token fisico corrispondente*.
Riferendoci ad alcuni esempi già visti all'inizio del capitolo, al *token logico ernest*, derivante dall'espansione dell'entità interna *&ernie;*, sarà associata la posizione fisica di quest'ultima nel documento compresso. Lo stesso vale per il *token logico hemingway*. Al *tag* TITOLO sarà associata la lista delle posizioni *a b c d ...*, dove le coppie *a,b* e *c,d* sono posizioni di apertura e chiusura di *tag* nel documento compresso, cioè la posizione dei *token fisici* '<' e '>'. Ad esempio, considerando il *documento compresso* (che, lo ricordiamo, riflette esattamente il documento sorgente), le posizioni saranno le seguenti:



Le *liste invertite* ci consentono di implementare in modo molto semplice le interrogazioni strutturali analizzando il loro contenuto per ogni *token*. Ad esempio si può dedurre che il *token logico suona* sia contenuto nel *tag TITOLO* se la *lista invertita* associata a *suona* contiene una posizione *x* compresa tra *a* e *b*.

### 2.3.2 Le IL e la doppia vista

Anche le *liste invertite* sono influenzate dalla doppia vista introdotta dalla libreria XCDE. Per i *token logici* che derivano dall'espansione di entità interne la sola posizione fisica di quest'ultime potrebbe non essere sufficiente per rispondere correttamente alle interrogazioni strutturali. Ad esempio utilizzando nell'indicizzazione dell'espansione dell'entità *&ernie*; la sua sola posizione fisica non saremmo in grado di distinguere la posizione di *ernest* rispetto a quella di *hemingway*. Infatti le liste di entrambe queste parole conterrebbero la posizione fisica del *token &ernie;*. È necessario pertanto aggiungere delle informazioni alla *lista delle posizioni fisiche* di un *token*. Queste rappresentano il *displacement* del *token* all'interno di una espansione di entità (se necessario). Nel nostro esempio il *token ernest* ha *displacement* 1, il *token hemingway* ha *displacement* 2. Questa soluzione, con un incremento minimo dell'occupazione in spazio, ci consente di determinare esattamente la posizione nella vista logica dei *token* che derivano dall'espansioni di entità interne e, anche in caso di definizione di entità complesse, ci permette di rispondere con esattezza a tutti i tipi di interrogazione strutturale. La posizione dei *token logici* derivanti dall'espansione di un'entità è determinata quindi dalla coppia (*posizione fisica dell'entità, displacement all'interno dell'espansione dell'entità*).




Per omogeneità un elemento non derivante dall'espansione di un'entità è individuato dalla coppia (*pos, 0*). Esso cioè ha *displacement* uguale a 0. Per completezza osserviamo che nel caso in cui l'espansione di un'entità contenga l'apertura di un *tag* con la definizione di alcuni attributi, allora il *displacement* di questi attributi sarà definito come quello dell'apertura del *tag*, in modo da poter ricondurre ogni dichiarazione di attributo al corrispondente *tag*.

### 2.3.3 La compressione delle IL

Le *liste invertite* sono ordinate in base al numero del *rango del token* a cui si riferiscono, e quindi l'accesso a ciascuna di esse può essere guidato da questo identificatore. L'implementazione delle *liste invertite* nella libreria XCDE ha come obiettivo quello di *minimizzare* lo spazio occupato consentendo al tempo stesso un accesso efficiente ai loro contenuti. Le IL rappresentano infatti la struttura dati più importante della libreria, perché occupano la maggior parte dello spazio e perché ad esse si accede più spesso nelle

operazioni di ricerca. Siccome le *liste invertite* hanno una lunghezza variabile, la conoscenza del *rango* di un token richiederebbe una scansione sequenziale delle stesse per l'individuazione della IL corrispondente. D'altro canto se memorizziamo per ogni *token* la posizione su disco della *lista invertita*, otterremmo un tempo di accesso costante ma una maggiore occupazione in spazio. La libreria XCDE per consentire una accesso veloce alle liste e un'occupazione di spazio ridotto si serve di un *indice* delle posizioni di alcune *liste invertite*, precisamente la posizione iniziale di *blocchi sequenziali di 10 liste.* Questa dimensione è il risultato di una serie di prove effettuate su una collezione di documenti XML fornita dal *CIBIT* **[CIBIT]**. I risultati e le modalità degli esperimenti sono illustrati nella sezione 2.3.4. All'interno dei singoli blocchi l'accesso è sequenziale. In testa ad ogni singola lista è memorizzata l'occupazione in *byte* della lista stessa. Questo valore può essere utilizzato per incrementare il puntatore all'interno del blocco, in modo da accedere in tempo costante alla lista successiva. Il massimo numero di accessi che possiamo effettuare all'interno di un blocco per accedere alla lista desiderata è limitato dal numero di liste contenute nel blocco stesso (10 appunto). I singoli elementi delle liste invertite sono compressi utilizzando il codice *Continuation bit* (d'ora in poi *Cb*). Si tratta di un codice a lunghezza variabile allineato al *byte*: il primo *bit* di ogni *byte* è utilizzato come *flag* per indicare se si tratta dell'ultimo *byte* della rappresentazione.

Ad esempio il numero intero codificato su quattro *byte*:

| 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 1 1 1 | 1 0 0 0 0 1 1 1 |
|---|---|---|---|

può essere rappresentato su tre *byte* come:

| 0 | 0 0 0 0 0 1 1 | 0 | 1 1 0 0 0 0 1 | 1 | 1 1 0 0 0 0 0 |
|---|---|---|---|---|---|

Le operazioni di codifica e decodifica di questa rappresentazione sono molto veloci, perché in pratica comportano semplici operazioni di *shift*. La ridondanza massima del codice è di un *byte* (derivante dai *bit* utilizzati per i *flag* e considerando che gli elementi sono allineati al *byte*, quindi un intero rappresentato in lunghezza fissa su 4 *byte* può arrivare ad occupare nel caso pessimo 5 *byte*), ma esso garantisce una buona compressione per numeri 'piccoli'. Il codice *Cb* si è rivelato abbastanza efficiente per quanto riguarda la compressione, e molto veloce nella decompressione. Riferendoci alla figura 1.12 il modulo che si occupa della compressione e decompressione di numeri interi opera mediante il *Cb*. La modularità del sistema ci ha permesso comunque di investigare altri tipi di compressione degli interi: i codici $\gamma$, $\delta$, *Golomb* e *$\gamma$-Golomb* **[Witten]** arrivando alla fine a preferire il *Cb*. I risultati sperimentali di tali prove sono illustrati nella sezione successiva.

### 2.3.4 Statistiche

Per determinare il formato di memorizzazione delle liste invertite sono state effettuate numerose prove sperimentali utilizzando alcuni documenti della collezione *CIBIT* **[CIBIT]**, costituita da opere letterarie italiane. Le prove sono state realizzate su un calcolatore equipaggiato di processore *Intel Pentium Celeron 700 Mhz* con 128 *Mb* di memoria *Ram* e 10 *Gbyte* di *Hard Disk*, *ATA-100*. Nei grafici delle figure 2.1 e 2.2 sono sintetizzati i

risultati delle prove sperimentali effettuate per determinare il numero delle liste invertite che formano un *blocco indicizzato*. Questo valore influenza sia la *velocità di accesso* alle liste, perché aumentando il numero delle liste in un blocco aumenta il tempo (sequenziale) di accesso ad esse, sia la spazio occupato visto che le posizioni di inizio dei blocchi devono essere memorizzate. Nel grafico di figura 2.1 sono illustrati i tempi medi di accesso ad una *lista invertita* in base alla dimensione dei blocchi di indicizzazione. I tempi indicati sono quelli ottenuti con l'algoritmo di compressione degli interi *Continuation bit*. I tempi sono stati calcolati effettuando gruppi di accessi casuali alle IL. Si può vedere dall'andamento del grafico come la dimensione dei singoli blocchi indicizzati influenzi fortemente i tempi di accesso alle liste. Il tempo di accesso minimo si ottiene con intervalli di valore 10, mentre già per valori intorno a 30-40 i tempi diventano considerevoli. Nel grafico di figura 2.2 è illustrata la relazione fra gli intervalli di indicizzazione delle *liste invertite* e le dimensioni degli indici memorizzati. Gli indici sono compressi applicando l'algoritmo di *Cb*. Le dimensioni diminuiscono aumentando l'intervallo di memorizzazione, com'è logico prevedere, ma la diminuzione è proporzionalmente inferiore all'aumento del tempo di accesso corrispondente. L'approccio applicato in molti motori di ricerca esistenti (ad esempio *Google* **[google]**) per la risoluzione di questo problema è quello di memorizzare *tutti gli indici delle liste invertite*. In realtà nel nostro caso questo non migliorebbe di molto le prestazioni visto che, come possiamo vedere dal grafico di figura 2.4, il tempo di accesso medio alle IL è di pochi microsecondi ($10^{-6}$). L'indicizzazione di tutte le liste comporterebbe al contrario un aumento del tempo di inizializzazione della *console* di un fattore 10, perché sarebbe necessario caricare tutti gli indici in memoria.

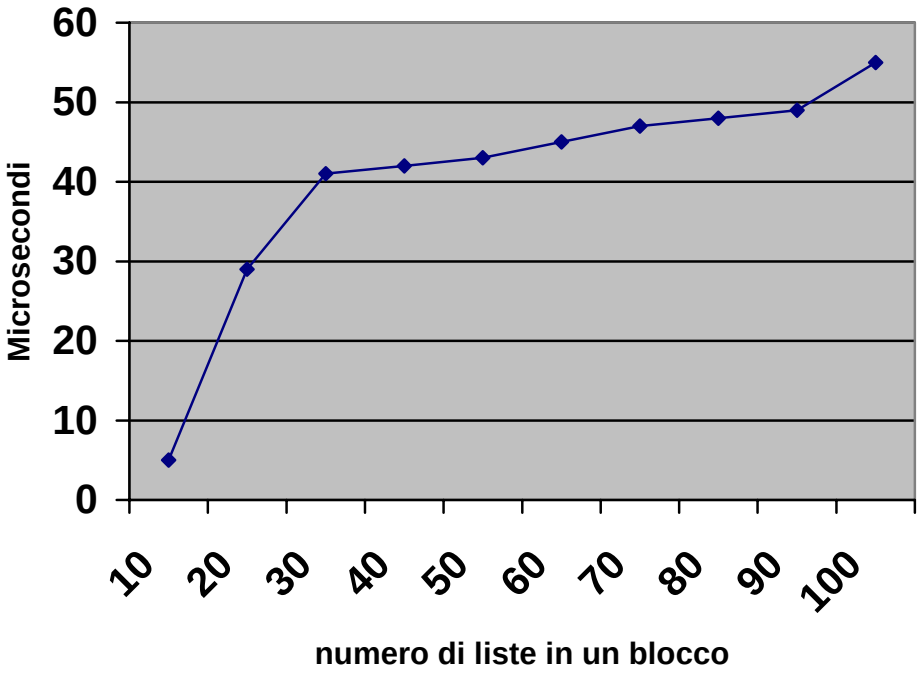



*Figura 2.1 - tempo medio di accesso alle IL*

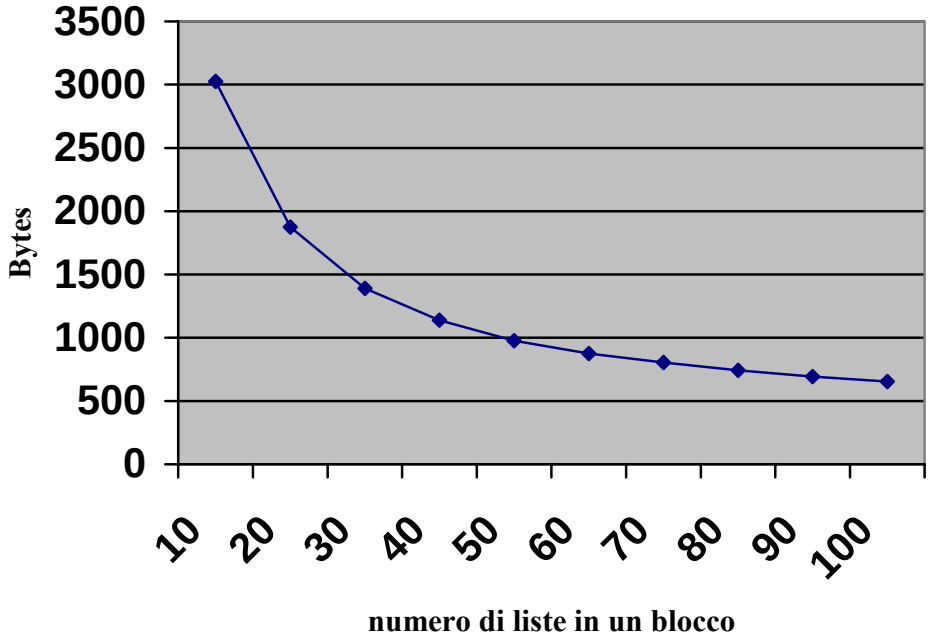



*Figura 2.2 - dimensioni medie degli indici delle IL*

Considerando che gli accessi alle liste invertite sono statisticamente le operazioni più frequenti nella risoluzione delle interrogazioni, si è deciso di minimizzare principalmente il tempo di accesso, anche perché le dimensioni degli indici memorizzati sono irrisorie rispetto alla dimensioni del documento originale XML (0.1% – 0.2%). Nei grafici delle figure 2.3 e 2.4 sono sintetizzati i risultati delle prove effettuate con diversi algoritmi di compressione degli interi. Esse sono state realizzate in modo da minimizzare l'occupazione in spazio e consentire la rapida decompressione. Ad esempio, per quanto riguarda i codici di *Golomb* e *γ-Golomb* i parametri necessari per l'esecuzione degli algoritmi sono stati calcolati per ogni lista applicando le formule suggerite in letteratura **[Witten, pagg. 119-121]**. Il grafico di figura 2.3 illustra la compressione ottenuta con l'applicazione dei diversi algoritmi, mentre il grafico di figura 2.4 illustra i tempi medi di accesso alle singole liste per le diverse codifiche. Il tempo di accesso è influenzato dal *tempo di decompressione* dei singoli algoritmi. Il *Cb* non garantisce l'ottimalità in spazio, però consente di comprimere in maniera accettabile e di accedere velocemente alle liste. Analogamente a quanto osservato per l'indicizzazione a blocchi delle IL, si è deciso di minimizzare i tempi di accesso scegliendo dunque l'algoritmo con il minor tempo di decompressione, ossia il *Cb*.

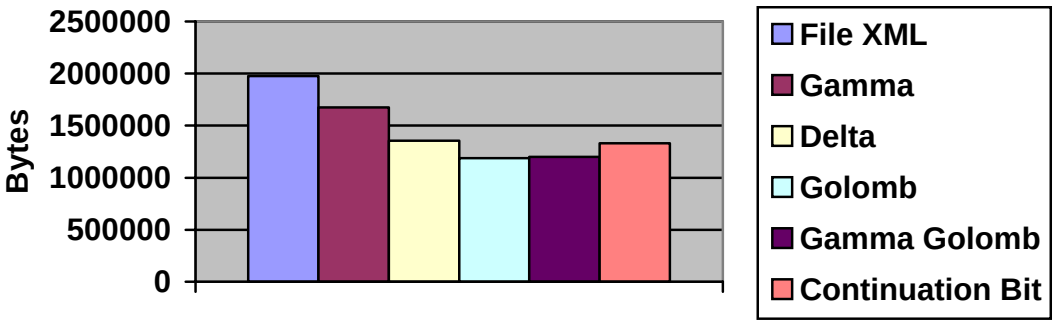



*Figura 2.3 - dimensioni medie delle IL*

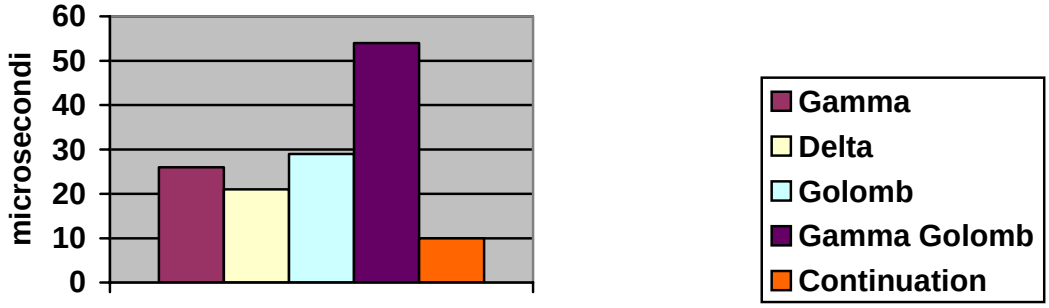



*Figura 2.4 - tempo medio di accesso alle IL per i diversi algoritmi di compressione degli interi*

## 2.4 Il documento compresso

Il documento sorgente XML viene compresso in modalità *lossless*, cioè senza alcuna perdita di informazione. Il documento, una volta decompresso, corrisponderà perfettamente al documento sorgente.

### 2.4.1 I due livelli di compressione

Il documento è soggetto a due livelli di compressione. Il primo livello di compressione consiste nell'applicazione di una versione modificata del codice di *Huffman* **[Witten, pagg. 30-51**]. Si tratta di un algoritmo di compressione basato su *parole*, nel nostro caso rappresentate dai *token fisici* già illustrati nella sezione 2.2. Più precisamente, in fase di

compressione il documento XML viene diviso in una sequenza di *token fisici* (*tag*, attributi e parole) che andranno a far parte dei diversi dizionari. I token logici sono creati allo stesso modo e durante lo stesso procedimento, al fine di consentire le interrogazioni sulla vista logica. Ogni *token* rappresenta una foglia dell'albero canonico di Huffman, e la sua probabilità è data dal numero di volte che il *token* occorre all'interno del documento. Nel caso di *token* solo logici esisteranno delle parole di codice che non saranno mai utilizzate all'interno del documento compresso. Richiamando quanto già detto nella sezione 2.2 questa ridondanza è contenuta, ed il vantaggio sta in una maggiore omogeneità di trattamento dei *token* e quindi semplicità di uso del sistema: ogni *token* possiede una parola di codice, e dalla parola di codice è possibile ricondursi ad un *rango* che lo identifica univocamente all'interno del dizionario *globale*. I dizionari dei *tag*, delle parole e degli attributi possono in realtà essere visti come un partizionamento del dizionario *globale* calcolato dall'algoritmo di Huffman. All'interno dei singoli dizionari l'ordine dei *token* è *decrescente* rispetto alla lunghezza della parola di codice, e per i *token* con la stessa lunghezza l'ordine è quello lessicografico tra le stringhe. Dalle proprietà del codice canonico di Huffman **[Witten]** segue che data una parola di codice è possibile ricondursi al dizionario e alla *rango nel dizionario* del *token* corrispondente. Poiché vogliamo poter identificare un *token* nel *file* compresso, il codice di Huffman da noi utilizzato è quello con *fan-out* 128. Qui le parole di codice sono allineate al *byte*, introducendo così una leggera perdita nel fattore di compressione, ma garantendo altresì una decompressione più efficiente e un allineamento al *byte* del *file* compresso utile per la seconda fase di compressione. Grazie a questa proprietà è possibile decomprimere efficientemente il codice di Huffman.
Il successivo livello di compressione *divide in blocchi* Huffman, e comprime ogni singolo blocco utilizzando un algoritmo a dizionario, tipo *gzip* **[gzip]**; la compressione ottenuta è elevata e i tempi di decompressione interessanti, infatti:

- si sfrutta un *effetto di località*: in genere non si decomprime un singolo *token*, ma sequenze più o meno lunghe di *token*;

- la *decompressione dei blocchi* è veloce, trattandosi di un algoritmo basato sui dizionari;

- viene implementato un meccanismo di *cache* dei blocchi: accessi successivi a parole di codice dello stesso blocco richiedono una sola decompressione del blocco medesimo;

- La diminuzione delle dimensioni del documento compresso contribuisce a diminuire i tempi di accesso, essendo il documento memorizzato sul *file system*;

- I tempi di decompressione non sono così critici in quanto questa operazione viene effettuata solo in fase di visualizzazione delle risposte alle interrogazioni o durante la decompressione di interi documenti.

La libreria utilizzata per la compressione dei blocchi è la Zlib ver. 1.1.4 **[Zlib]**, una libreria di pubblico dominio su cui si è costruito il compressore *gzip* **[Witten, pagg. 74-84]**. La

dimensione del blocco è stata fissata a 64 *Kbyte*, in seguito ad una serie di prove sperimentali illustrate nella sezione dedicata alle statistiche. Lo stesso tipo di soluzione avrebbe potuto essere applicata alle *liste invertite*, ed infatti questo approccio è stato sperimentato, dimostrando che la presenza del secondo livello di compressione comporterebbe un vantaggio irrisorio in termini di spazio e un aumento significativo dei tempi.

### 2.4.2 Statistiche

Per determinare le dimensioni del blocco di testo e per testare l'ottimalità della soluzione proposta abbiamo effettuato una serie di prove sperimentali.

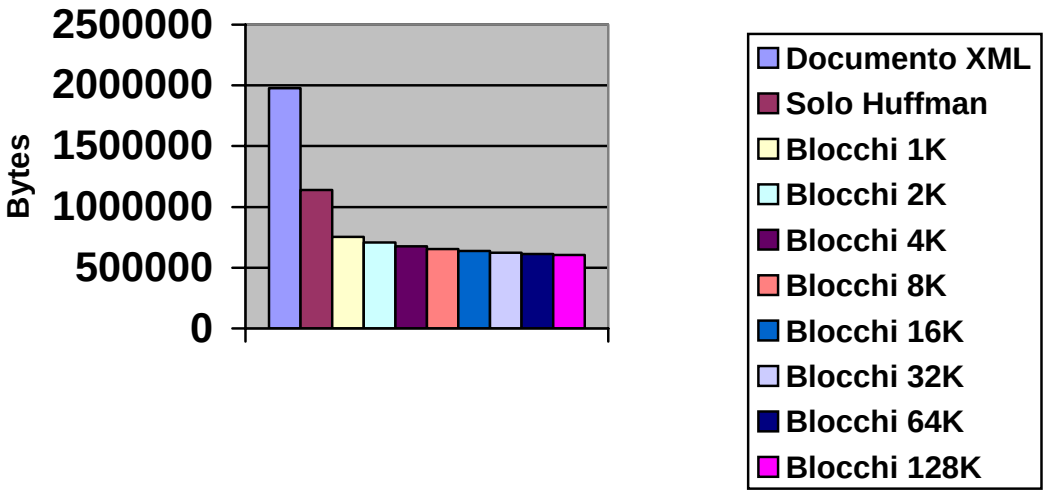



*Figura 2.5 - dimensioni medie del documento compresso*

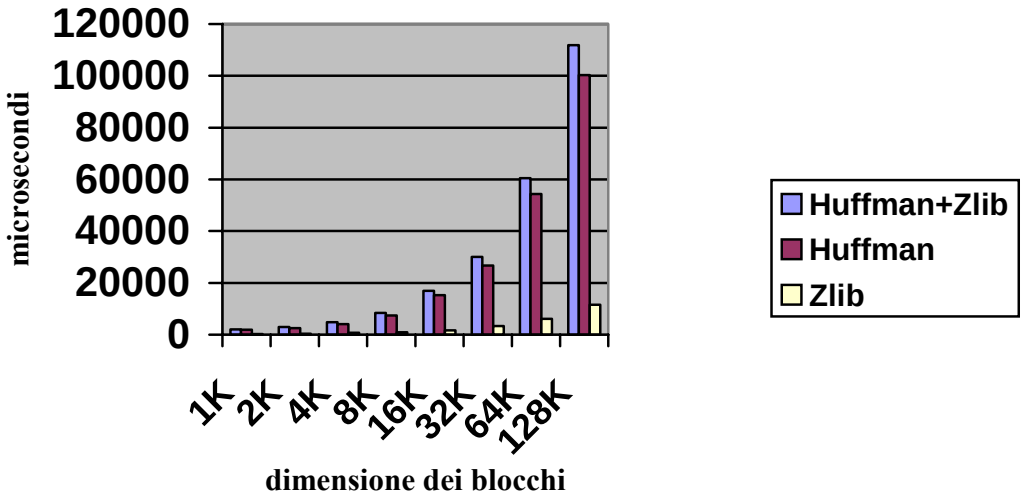



*Figura 2.6 - tempo medio di accesso ai blocchi Zlib*

Nel grafico 2.5 sono illustrati i risultati delle prove effettuate per determinare l'occupazione del *documento compresso* al variar della dimensione del blocco su cui Zlib viene eseguito. Il primo istogramma rappresenta le dimensioni medie del *documento compresso* applicando il solo algoritmo di Huffman, mentre i valori seguenti rappresentano la compressione ottenuta combinando Huffman e Zlib su dimensioni crescenti del blocco. Per dimensioni di 64-128 *Kbyte* la compressione media ottenuta è circa il doppio di quella ottenuta applicando il solo algoritmo di Huffman e la dimensione del *documento compresso* è circa il 30% delle dimensioni del documento originale. L'elevata compressione ottenuta combinando i due algoritmi è dovuta al fatto che il comprimere i singoli blocchi con il codice di Huffman consente successivamente a Zlib di operare su una dimensione di testo *implicitamente maggiore*. Inoltre l'algoritmo di Huffman è allineato al *byte* e opera su parole di codice corrispondenti a *token*. Questo consente a Zlib di operare in realtà su un dizionario di *parole* invece che di caratteri.
Possiamo vedere nel grafico 2.6 i risultati delle prove effettuate per determinare la dimensione dei blocchi Zlib. Per ogni dimensione di blocco abbiamo tre valori. Il primo valore è il tempo medio totale di decompressione di un blocco, considerando entrambe gli algoritmi di compressione. Il secondo valore è il tempo per la decompressione del solo codice di Huffman, mentre il terzo valore rappresenta i tempi per la decompressione dei blocchi utilizzando l'algoritmo Zlib. Il tempo necessario per la decompressione dei blocchi è principalmente dovuto all'algoritmo di Huffman, mentre l'algoritmo Zlib, pur garantendo una ulteriore compressione non incide in maniera significativa sui tempi di decompressione. A seguito di questo prove abbiamo quindi deciso di applicare i due livelli di compressione, utilizzando un blocco da 64 Kbyte, che garantisce un buona compressione e al tempo stesso un accettabile tempo di decompressione.

## 2.5 La "tagstruct" e le informazioni ausiliarie

La *tagstruct* è una struttura dati che consente di effettuare interrogazioni di *stabbing* su qualsiasi posizione fisica del documento. Essa ci consente cioè, data una posizione all'interno del documento compresso, di stabilire quali sono i *tag innestati* contenenti quel determinato punto, e quindi di determinare tutti i nodi predecessori di un elemento nell'albero strutturale del documento XML. Per la sua creazione ci siamo basati sull'algoritmo di *interval overlap* introdotto in un articolo di Bernard Chazelle riguardante il *filtering search* **[Chazelle]**. Nella struttura dati per le informazioni ausiliarie sono invece memorizzate tutte le informazioni necessarie per accedere alle altre strutture dati. In particolare essa contiene gli indici per l'accesso alle *liste invertite* e quelli per l'accesso ai blocchi del *documento compresso*, le informazioni sulla composizione dei dizionari e per la decompressione del codice di Huffman, la lunghezza massima delle parole di codice di Huffman e la prima parola di codice per ogni lunghezza. Tutte le informazioni sono soggette a compressione attraverso il codice *Continuation bit.*

### 2.5.1 l'algoritmo di Chazelle

L'algoritmo *interval overlap* di Chazelle elenca tutti gli intervalli appartenenti ad un insieme $S=\{[a_1, b_1], ..., [a_n, b_n]\}$ che racchiudono una determinata posizione, occupando

spazio O(n) e richiedendo tempo $O(\log_2 n)+occ$, dove *occ* è il numero di intervalli riportati.

Sia $x_{min}= \min_i a_i$ e $x_{max}=\max_i b_i$, l'intervallo $[x_{min}, x_{max}]$ include tutti gli estremi dell'insieme S. Questo intervallo viene partizionato dall'algoritmo di Chazelle in una serie di sottointervalli di dimensione variabile, detti *finestre*. Una *finestra f* ha associato l'insieme S(*f*) contenente tutti gli intervalli di S che intersecano *f* (il loro numero varia da finestra a finestra e dipende, chiaramente, dalla struttura di S). Similarmente, per ogni punto x, si usa la notazione S(x) per indicare l'insieme degli intervalli di S che contengono il punto x. Chazelle ha dimostrato che è possibile progettare le finestre in modo tale che, dato un qualunque punto x in *f,* vale $|S(f)|\leq\delta|S(x)|$, dove δ rappresenta una costante definita dall'utente. Ciò significa che per rispondere ad una interrogazione di *stabbing* su un punto x, basta trovare la finestra *f* che lo contiene, e poi esaminare tutta la lista degli intervalli C(*f*) ad essa associata alla ricerca di quelli che contengono x. Il costo dell'esame di C(*f*) è proporzionale a |C(x)|, e quindi risulta ottimo. Chazelle ha poposto un algoritmo efficiente per il calcolo delle finestre e ha dimostrato che lo spazio richiesto dalla memorizzazione delle liste di intervalli loro associate è O(n).

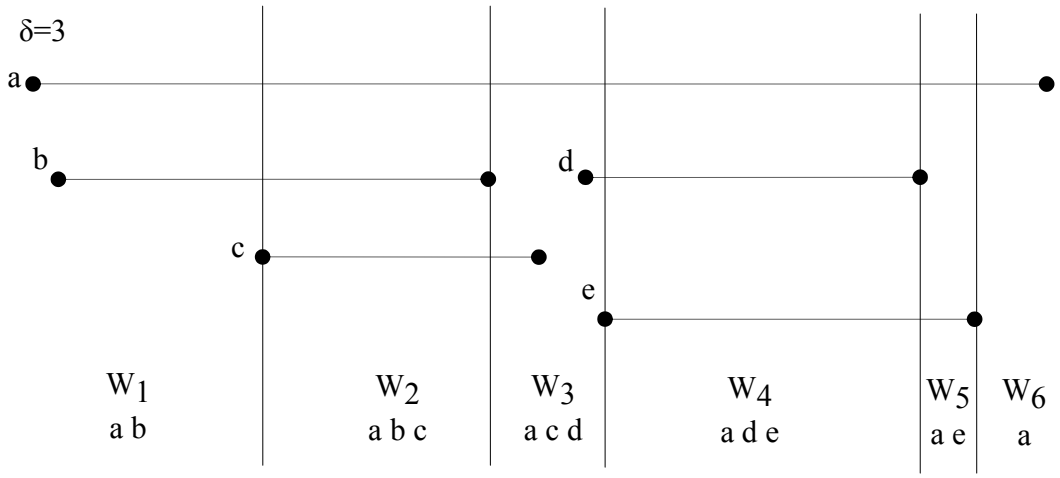



*Figura 2.7 - esempio di divisione in finestre di una serie di intervalli*

Quindi l'algoritmo di Chazelle richiede tempo $O(\log_2 n)$ per la determinazione della finestra che racchiude il punto x e tempo ottimo Θ(occ) per il recupero degli intervalli *stabbed*.
Rimandiamo per la descrizione dell'algoritmo di costruzione delle finestre all'articolo di Chazelle. Analizzeremo nel paragrafo seguente le modalità di interrogazione delle finestre e come queste possono essere inglobate nella libreria XCDE.

### 2.5.2 Applicazione dell'algoritmo di Chazelle all'interno di XCDE

L'algoritmo di Chazelle può essere facilmente utilizzato nella libreria XCDE effettuando alcune semplici modifiche. In particolare per quanto riguarda le posizioni bisogna ricordare che queste sono in verita coppie formate dalla posizione fisica e dal *displacement* di un *token*.

Naturalmente gli operatori relazionali vanno modificati, in modo che

$$(a_p, a_d) < (b_p, b_d) \leftrightarrow \{ (a_p < b_p) \text{ or } ((a_p = b_p) \text{ and } (a_d < b_d)) \}$$

Con queste modifiche l'algoritmo viene implementato considerando come intervalli le posizioni di apertura e di chiusura dei *tag*.
Una finestra è data da una sequenza di triple, ognuna contenenti il *rango del tag*, la posizione fisica del *tag* e suo *displacement*. Sfruttando la proprietà di corretto innestamento dei *tag* (ogni *tag* deve essere contenuto all'interno di un altro *tag* o essere il *tag* radice), la prima occorrenza di un *tag* indica la posizione di apertura, la seconda la posizione di chiusura. Utilizzando una pila si può facilmente determinare, durante l'analisi di una finestra, se un elemento incontrato rappresenta un'apertura o una chiusura. Per la memorizzazione della struttura, al fine di minimizzare l'occupazione in spazio, si utilizza per gli intervalli di ogni finestra l'algoritmo di compressione *Continuation bit*, e su tutta la finestra l'algoritmo di compressione Zlib in modo simile a quanto visto per il documento compresso. Il parametro $\delta$ è stato fissato a 1000, perché in seguito ad alcune prove sperimentali effettuate questo valore ci consente di avere delle dimensioni ragionevoli delle finestre non degradando significativamente il tempo di accesso alle stesse. Nel caso della *tagstruct* abbiamo deciso di far prevalere le esigenze di compressione, visto che la struttura è ben comprimibile e i tempi di accesso a essa non sono critici.

## 2.6 La creazione delle strutture dati

La creazione delle strutture dati viste finora avviene a seguito di una fase di *analisi* del documento. Il documento viene sottoposto a *parsing* e i dati così ottenuti sono soggetti a delle trasformazioni prima di essere memorizzati. L'analisi del documento può essere personalizzata dall'utente. Si possono ad esempio stabilire i caratteri e le entità che sono considerati *separatori*, e che quindi stabiliscono la divisione del documento in *token*, o si può decidere se creare le *liste di proximity* e/o la *tagstruct*. La possibilità di memorizzare solo le informazioni utili all'utente consente di risparmiare spazio nel caso non debbano essere supportati alcuni tipi di interrogazioni sui documenti. Per l'analisi del documento sorgente si è utilizzata la libreria di *parsing* Expat versione 1.2. Si tratta di un libreria di funzioni SAX creata da J.Clark **[Clark]**. Essa è di pubblico dominio, è scritta interamente 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 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 alla singole funzioni e utilizzando la lista di separatori definita nel *file* di configurazione (contenente anche le eventuali entità interne o esterne che sono considerati separatori) si individuano i *token* che costituiscono il documento. L'individuazione dei *token* è alla base dell'algoritmo utilizzato per la compressione del documento, che è una modifica dell'algoritmo di Huffman **[Huffman]** con *fan-out* 128. In pratica ad ogni *token* è associata una parola di codice allineata al *byte*. Il *bit* più significativo di una parola di codice è utilizzato come *flag* di fine parola, in maniera

simile a quanto visto per il *Continuation bit*. La parola di codice è ottenuta creando un albero canonico di Huffman (binario nell'algoritmo originale, di *a-rietà* 128 nella versione applicata per permettere di allineare le parole di codice al *byte*) le cui foglie rappresentano i *token* da comprimere (nel nostro caso come probabilità del *token* si considera la sua frequenza nel documento). Per effettuare la decompressioneè necessario avere un dizionario dei token, che permette di passare dalla parola di codice al *token*, operazione questa molto semplice nel caso di codice di Huffman canonico **[Witten]**.

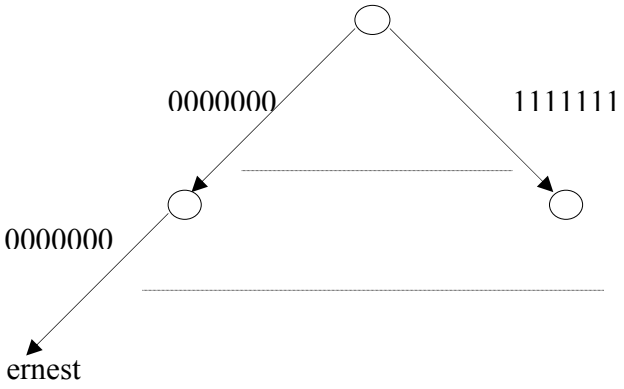
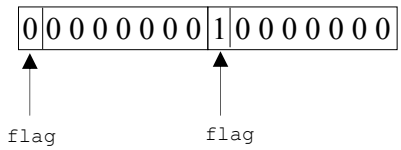



*Figura 2.8 - albero canonico di Huffman con fan-out 128*

Dato l'esempio di figura 2.8, la parola di codice di Huffman corrispondente al token *ernest* sarà:




I flag permettono l'allineamento al byte e consentono di individuare la lunghezza di una parola di codice. Durante la creazione delle strutture dati si effettuano due letture complete del documento, utilizzando Expat. Durante la prima lettura vengono creati i *token*, attraverso il loro inserimento in una tabella *Hash*, e viene calcolata la frequenza di ognuno, in modo da utilizzarla durante la creazione del codice di Huffman come *probabilità associata alla foglia del token corrispondente*. Durante la seconda lettura si crea il documento compresso e vengono create e memorizzate le *liste invertite* e la *tagstruct*.

# 2.7 Memorizzazione delle strutture dati nel file system

Tutte le strutture dati di cui abbiamo parlato precedentemente sono memorizzate all'interno del *file system*. Dato un documento XML *nomedoc.xml*, la fase di analisi crea i seguenti *file*:

*nomedoc.xml.il*: è il *file* che contiene le *liste invertite*. Esso è in formato binario e al suo interno le liste sono memorizzate in maniera sequenziale, ordinate secondo il *rango nel dizionario globale* dell'elemento a cui si riferiscono. Ogni 10 liste viene memorizzato un indice del blocco, cioè un puntatore alla posizione della sua prima lista. Naturalmente i blocchi contengono sempre 10 liste, ma hanno ampiezza variabile, visto che le dimensioni delle singole liste sono variabili. In testa ad ogni lista è memorizzata (con il *Continuation bit*) la sua dimensione in *byte*. Seguono il numero di elementi contenuti e la lista delle posizioni fisiche, memorizzate con il *Cb* applicato sulla differenza fra le posizioni (le posizioni sono tutte in ordine crescente). I *displacement* sono memorizzati solo nel caso in cui siano significativi, utilizzando un meccanismo di *escape* sulla lista (rappresentato dalla codifica del valore –1), e dopo la posizione fisica a cui si riferiscono. Se il *token* è una parola alla lista delle posizioni fisiche può seguire, se previsto dall'utente, quella delle *proximity*. Ovviamente il numero di elementi memorizzati nella lista delle *proximity* e uguale al numero di elementi memorizzati nella lista delle posizioni.
Ad esempio, sia dato un *token* la cui *k-esima* occorrenza è stata ottenuta dall'espansione di una entità; la sua *lista invertita* delle posizioni e delle *proximity* verrà memorizzata in accordo al seguente formato (Cb indica la codifica *Continuation bit* e pos_(i+1)=pos_i+Δ(i+1)*)*:

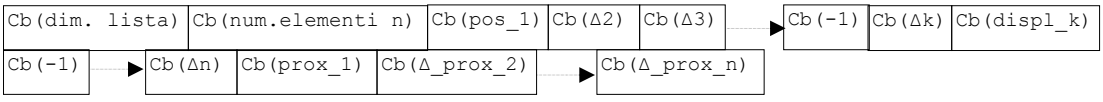



*Figura 2.9 - struttura di una lista invertita*

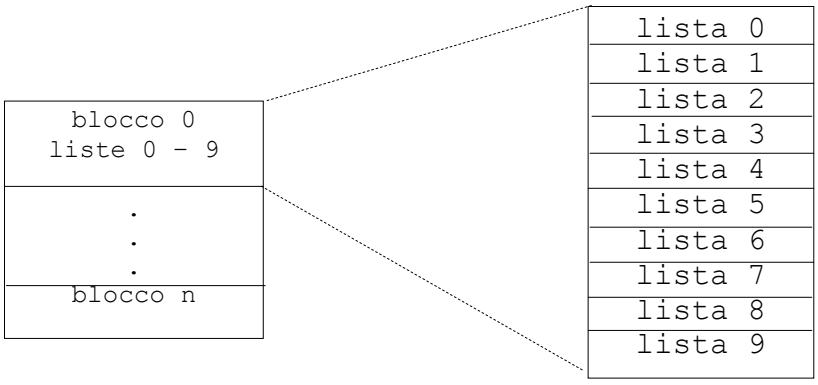



*Figura 2.10 - struttura del file delle IL*

In realtà, per problemi legati alla codifica dello zero e del valore –1 usato come *escape*, i

valori codificati con il *Cb* vengono incrementati di due o di uno a seconda che appartengano o meno alla *lista delle posizioni*. In sede di decompressione i valori saranno opportunamente decrementati;

*nomedoc.xml.body*: è il *file* che contiene il documento compresso. È un *file* binario ottenuto comprimendo prima il documento XML originale mediante la variante dell'algoritmo di Huffman "allineata al *byte*" e poi comprimendo ulteriormente i suoi blocchi di 64 *Kbyte* attraverso l'algoritmo Zlib. Ogni blocco è indicizzato, in maniera simile a quanto effettuato per le *liste invertite*. Anche in questo caso, come per le IL, l'ampiezza di ogni blocco dopo la sua compressione è variabile;

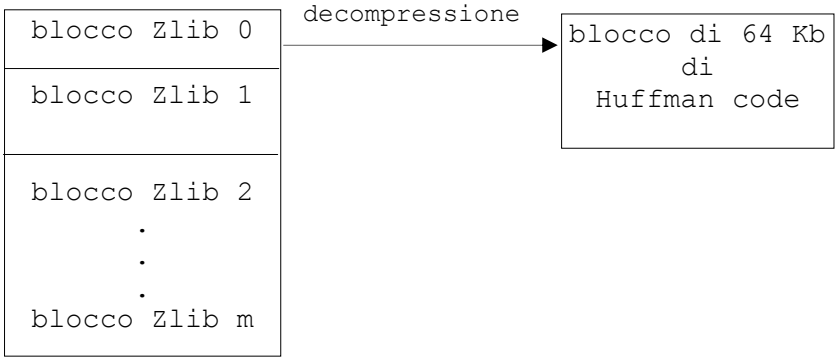



*Figura 2.11 - struttura del file che contiene il documento compresso*

*nomedoc.xml.dict.0.cmp, nomedoc.xml.dict.1.cmp, nomedoc.xml.dict.2.cmp*: si tratta dei *file* contenenti i dizionari. Ognuno di essi è compresso con Gzip **[gzip]** e contiene una sequenza di *token* in ordine descrescente in base alla lunghezza della parola di codice e all'ordine lessicografico. I *file* contengono nell'ordine, i dizionari dei *tag*, degli attributi e delle parole. All'interno di ogni dizionario i *token* sono rappresentati come una sequenza di stringhe separate dal carattere NULL. Questa particolare implementazione facilita, una volta effettuata la decompressione del dizionario il *mapping* del *file* in memoria centrale e l'accesso ai singoli *token*. Viste le dimensioni dei dizionari e la velocità di *gzip* questa fase di decompressione richiede un tempo ragionevole (ritorneremo su questo punto alla fine della tesi).

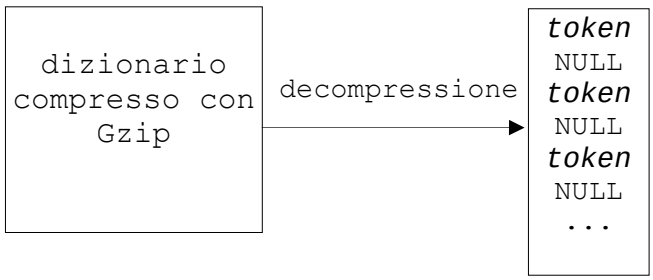



*Figura 2.12 - struttura dei file dei dizionari prima e dopo la compressione*

*nomedoc.xml.tagstruct*: è il *file* che contiene la *tagstruct*. Il *file*, così come la struttura dati originaria, è diviso in *finestre*. All'inizio è presente un *header* che contiene alcune informazioni per accedere velocemente alle singole *finestre*.

L*'header* contiene:

- il *numero delle finestre* che compongono la *tagstruct*.

e, per ogni *finestra*:

  - il numero delle triple (*rango del tag, posizione fisica, displacement*) che formano la *finestra*;
  - l'elemento (*posizione fisica, displacement*) da cui ha inizio la *finestra*;
  - il *byte* di inizio della *finestra* all'interno del *file nomedoc.xml.tagstruct*.

Tutte le componenti dell*'header* sono compresse utilizzando il *Continuation bit.* Ogni *finestra* è compressa con la libreria Zlib e al suo interno le triple (*rango del tag*, *posizione fisica, displacement*) sono compresse con il *Cb*. Per posizione fisica di ogni *tag*, essendo gli intervalli ordinati in maniera crescente, si codifica la differenza Δ rispetto all'elemento precedente, escluso il primo elemento della *finestra*.

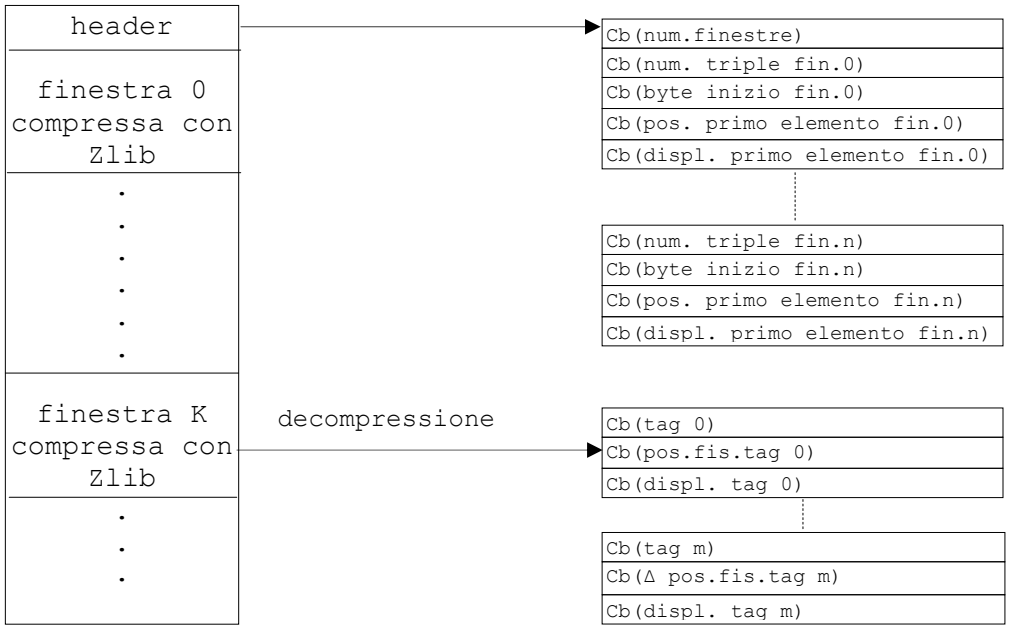



*Figura 2.13 - struttura del file che contiene la tagstruct*

*nomedocumento.xml.head:* si tratta del *file* utilizzato per contenere le informazioni ausiliarie necessarie per effettuare le operazioni sulle strutture dati. Le diverse informazioni sono compresse utilizzando il *Continuation bit*. Nel grafico seguente sono sintetizzate alcune statistiche sulle dimensioni delle diverse strutture dati costruite sulla collezione di *file* fornita dal *CIBIT* **[CIBIT]**. La dimensione media di ogni singolo documento è di circa 1Mb. Con *file* di dimensioni maggiori, la compressione migliora, come è dimostrato dal grafico di figura 2.15. Nel grafico di figura 2.14 possiamo vedere le dimensioni medie dei *file* contenenti le strutture dati. La dimensione di riferimento è quella del *file* contenente il documento XML originale. Come possiamo vedere la dimensione totale dei *file* è circa uguale a quella del file sorgente.

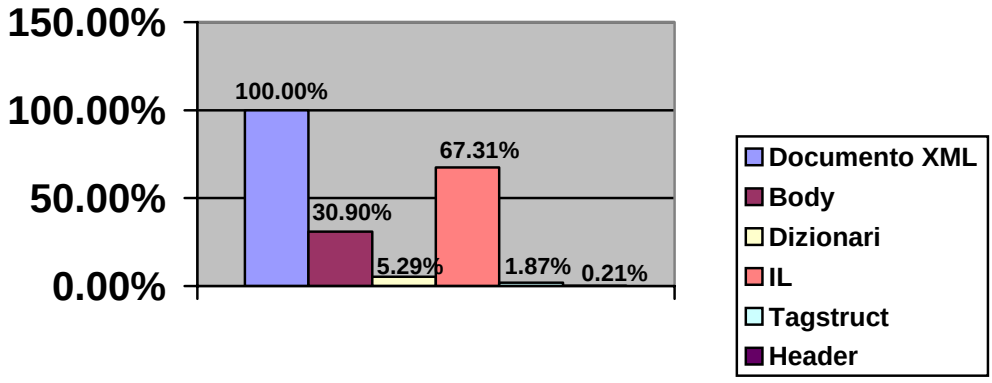



*Figura 2.14 - dimensioni dei file contenenti le strutture dati*

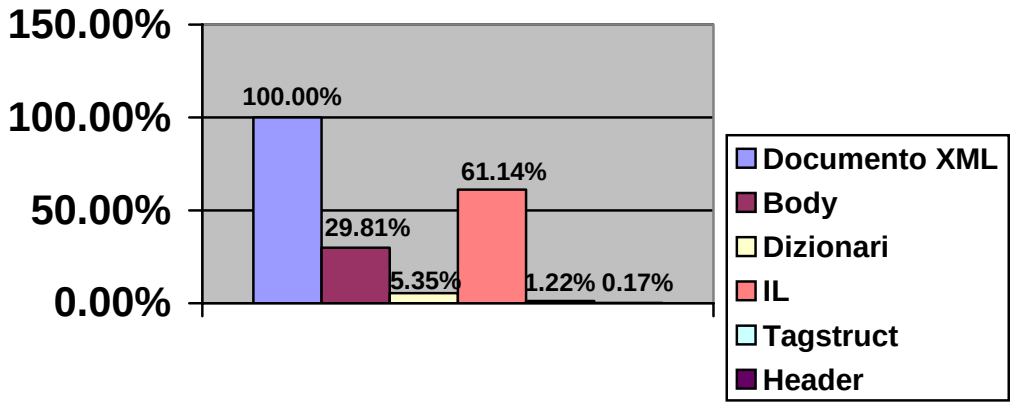



*Figura 2.15 - dimensioni delle file contenenti le strutture dati per un singolo documento contenente tutta la collezione CIBIT*

Analizzando il grafico di figura 2.13 possiamo vedere come si ottenga un miglioramento in compressione ottenuto indicizzando un singolo documento creato

unendo tutti i file della collezione *CIBIT* **[CIBIT]**. Il miglioramento ottenuto riguarda soprattutto il file contenente le *liste invertite*.
Nel grafico di figura 2.16 sono indicate le dimensioni delle strutture dati derivanti dall'indicizzazione di un documento XML contenente una descrizione di testi e articoli scientifici. Come possiamo notare la compressione ottenuta è migliore di quella ottenuta sulla collezione *CIBIT*, anche in seguito all'alta strutturazione e alla diminuzione della dimensione delle *liste di proximity* per mancanza di ampie parti testuali. L'efficacia della compressione ottenuta con XCDE non è quindi solo limitata ai documenti con prevalenza di dati testuali.

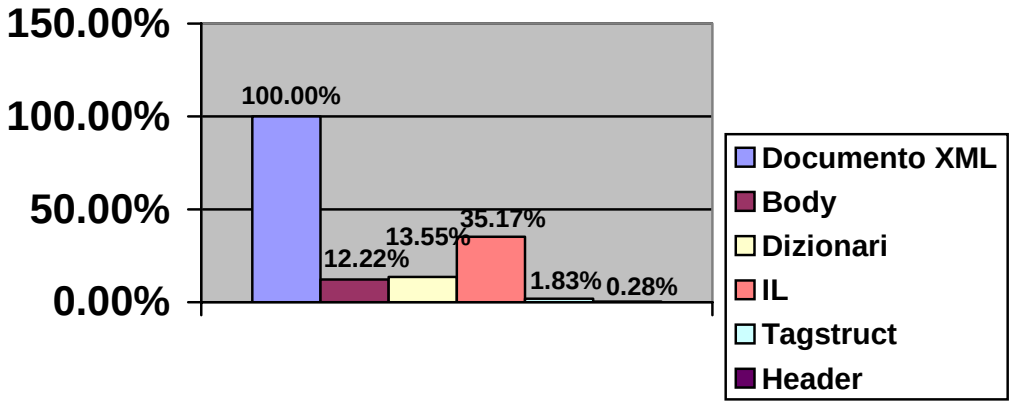



*Figura 2.16 - dimensioni dei file contenenti le strutture dati per un documento XML contenente una base di dati*

# 2.8 La creazione dei file

Una volta installata la libreria XCDE (procedura che illustreremo in maniera estesa nell'appendice B), essa contiene vari *file*: *file di supporto*, si tratta di *script* per la compilazione dei sorgenti (*Makefile),* comandi che facilitano l'utilizzo di XCDE consentendo di visualizzare informazioni sui documenti già compressi e indicizzati, comandi per decomprimere interamente un documento, o comandi per creare le strutture dati di cui abbiamo parlato nella sezione precedente.
I *file* necessari alla creazione delle strutture dati sono: il comando *xcde_build*, il suo *file* di configurazione *xcde_build.conf* e il *file di script build_script*.
Attraverso il comando *xcde_build* si costruiscono le strutture dati relative ad un singolo documento e necessarie all'utilizzo della libreria XCDE su di esso. Il comando prevede una serie di *opzioni* che consentono all'utente di modificarne il comportamento e richiede la presenza del *file* di configurazione *xcde_build.conf*. Per un corretto funzionamento il documento XML deve essere nella stessa *directory* in cui si trovano il *file di configurazione* e il comando *xcde_build*.

La sintassi di *xcde_build* è la seguente:

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

dove `sourcefile` è il nome del documento da indicizzare e comprimere. La descrizione delle opzioni è la seguente:

```
-v : verbose

-c : configuration file

-t : create tagstruct

-p : no proximity

-f : force recursive tags
```

Queste consentono all'utente di modificare le operazioni di costruzione del documento.
Il comportamento di *default* di xcde_build è di leggere il *file xcde_build.conf*, analizzare il *file sorgente* `sourcefile` e costruire i *file* delle strutture dati. Con l'opzione *–v* si indica al comando di visualizzare delle informazioni aggiuntive e di creare dei *file di debug*, i quali memorizzano le strutture dati in un formato testuale consultabile dall'utente. L'opzione *–c* consente di specificare un *file di configurazione* differente rispetto a quello standard. Le strutture dati create per *default* sono: i *dizionari*, il *documento compresso*, le *liste invertite*, con le posizioni fisiche e le *proximity*, e le *informazioni ausiliarie*. La costruzione della *tagstruct* deve essere appositamente richiesta dall'utente attraverso l'opzione *-t*. Attraverso l'opzione *–p* si indica al comando di non memorizzare le *liste di proximity*, diminuendo l'occupazione in spazio nel caso in cui l'utente non sia interessato ad effettuare interrogazioni sulla prossimità delle parole.
Una particolare attenzione deve essere rivolta all'innestamento dei *tag*: per questo è stata introdotta l'opzione *–f*. Infatti il comando *xcde_build* controlla, oltre che il documento sia *ben formato*, anche che non contenga un *tag innestato in se stesso*, ad esempio:

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

nel caso questo si verifichi la costruzione delle strutture dati non viene eseguita e l'utente è informato con un messaggio di errore. Una ricorsione nell'innestamento dei *tag* ha come conseguenza un'*ambiguità* in sede di interrogazione del medesimo e la possibilità che le interrogazioni eseguite su di esso non possano essere risolte in maniera corretta. L'utente ha comunque la possibilità di costruire le strutture dati relative al documento ignorando la presenza di uno o più *tag* innestati attraverso l'opzione *–f*. Per gestire i *tag* innestati si costruiscono tanti *tag virtuali* (distinti) quanti sono i livelli di innestamento ricorsivo. Il nome di questi *tag* è ottenuto preponendo la stringa "XML_", che è *riservata al linguaggio*, tante volte quanto è il livello di contenimento del *tag* in se stesso. Nell'esempio precedente viene costruito un solo *tag virtuale* con nome XML_TITOLO, oltre ovviamente al *tag* TITOLO, e ad esso sono associate le posizioni di apertura e chiusura del *tag innestato.*

```
$ xcde_build 001.xml

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

     Fatal Error : at raw 83.It's not possible to nest the tag list into itself
     <list><item>"conv": titoli convenzionali come "Capitolo Primo", "Parte I"</item>


$ xcde_build -f 001.xml

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

$ ls 001.*

001.xml
001.xml.body
001.xml.dict.0.cmp
001.xml.dict.1.cmp
001.xml.dict.2.cmp
001.xml.head
001.xml.il
$
```

*Figura 2.17 - esempio di utilizzo del comando xcde_build*

Possiamo osservare nella figura 2.17 un esempio di utilizzo del comando *xcde_build*. Nell'esempio si cerca di creare le strutture dati del *file* 001.xml della collezione *CIBIT* attraverso il comando *xcde_build 001.xml*, ma esso contiene il *tag list* innestato in se stesso e per questo viene visualizzato il messaggio di errore suddetto. Si forza quindi il sistema a indicizzare il *file* 001.xml con l'aggiunta dell'opzione *–f* . A questo punto, com'è possibile osservare, vengono create tutte le strutture dati (ma non la *Tagstruct,* la cui creazione si effettua con l'opzione *–t*).
Per facilitare la costruzione delle strutture dati su una collezione di file XML è stato progettato lo *script build_script*. Esso accetta gli stessi parametri di *xcde_build*, ad esclusione chiaramente del nome del documento XML, e si occupa di indicizzare tutti i documenti XML presenti nella *directory* in cui si trova. Per il corretto funzionamento di *build_script* è necessaria la presenza nella stessa *directory* del comando *xcde_build* e del *file di configurazione.*
Maggiori informazioni sulla gestione della libreria sono contenute nell'appendice B.

## NOTA BIBLIOGRAFICA

I principali algoritmi per la compressione dei numeri interi sono descritti in **[Witten]**. In particolare nelle pagg. 114-127 viene fornita una descrizione generale di tutte le tecniche citate in questo capitolo e ne vengono valutate le prestazioni. Nelle pagg. 30-51 dello stesso testo si può trovare una descrizione dell'algoritmo di Huffman canonico. Nelle pagg. 74-84 si può trovare una breve discussione degli algoritmi di compressione basati su dizionario, mentre per maggiori approfondimenti su Zlib e Gzip si può consultare il sito web **[Gzip]**. Per quanto riguarda l'algoritmo di creazione della struttura *tagstruct* rimandiamo alla sezione *Interval overlap* dell'articolo **[Chazelle]**.

# Capitolo 3

# Le funzioni di XCDE

*In questo capitolo completeremo la descrizione del nucleo della libreria XCDE definendo le caratteristiche algoritmiche delle operazioni che possono essere effettuate sui documenti XML e specificando la loro realizzazione. In particolare, all'inizio del capitolo descriveremo la console della libreria, a cui hanno accesso tutte le funzioni. Successivamente descriveremo l' API, che rappresenta l'interfaccia della libreria verso tutte le applicazioni client, e concluderemo con una specifica della struttura e implementazione delle varie funzioni.*

## 3.1 La console

La *console* rappresenta la struttura dati di raccordo fra la parte strutturale e la parte funzionale del nucleo di XCDE. Essa è una struttura dati interamente *in-memory*, a cui accedono tutte le funzioni che operano sui documenti XML compressi e indicizzati. La *console* consente di accedere ai *file* che contengono le varie strutture dati ed effettuare tutte le operazioni supportate dalla libreria su di essi. Le console è implementata in modo da:

- *minimizzare la quantità dei dati mantenuti in memoria*: qualsiasi dato viene memorizzato nella *console* solo quando effettivamente è necessario. Così ad esempio le informazioni dell'*header* del *file* che contiene la *tagstruct* vengono caricate in essa solo al momento in cui bisogna accedervi;

- *velocizzare gli accessi*: alcune tecniche di *buffering* o indicizzazione parziale sono adottate per evitare "colli di bottiglia" nell'esecuzione di operazioni sulle strutture dati. Ad esempio, i blocchi compressi mediante Zlib (blocchi del documento

compresso o della *Tagstruct*) sono mantenuti in forma decompressa all'interno di una *cache*. E sono presenti degli indici che consentono di velocizzare gli accessi alle *liste invertite* e al *documento compresso*;

- *facilitare l'implementazione delle operazioni della libreria*: in modo da rendere il sistema di semplice utilizzo ed estendibile. Ad esempio tutti i *file delle strutture dati* sono *mappati* in memoria, in modo che l'accesso ad essi possa avvenire tramite semplici assegnamenti di variabili.

La *console* deve essere *inizializzata* caricando in essa tutte le informazioni necessarie all'implementazione delle operazioni sui documenti: indici per l'accesso al documento compresso e alla IL, informazioni sulla composizione del dizionario, ecc.. Queste informazioni sono quelle contenute nel *file* delle informazioni ausiliarie (sezione 2.5).

### 3.1.1 Struttura della console

La *console* può essere suddivisa da un punto di vista logico in diverse parti, ognuna riguardante un particolare aspetto del sistema. La prima parte contiene informazioni di carattere generale sulla console medesima, come ad esempio opportuni *flag* di inizializzazione. Seguono le informazioni sulle strutture dati, come il loro stato (se sono state create, se sono state decompresse, ecc.), i riferimenti per l'accesso (nome del file, puntatore al file mappato in memoria, dimensioni, ecc.) e i dati per velocizzarne gli accessi (gli indici dei blocchi, cache dei blocchi decompressi in memoria, ecc.). Infine essa contiene le informazioni per la decodifica del codice di Huffman: divisione e dimensione dei dizionari, lunghezza massima di una parola di codice, prima parola di codice per ogni livello dell'albero canonico di Huffman, struttura che associa ad ogni rango il *token* corrispondente, ecc.. La figura 3.1 contiene uno schematizzazione della struttura della *console*.

### 3.1.2 Implementazione della console

La *console* è implementata all'interno della libreria come un tipo *struct* del linguaggio *C,* mediante un tipo chiamato *XCDE_Console_Type*. Ogni informazione contenuta nella *console* è rappresentata da un campo della struttura di tipo atomico o puntatore a un tipo atomico. *L'header* della *tagstruct*, copiato nella *console* per facilitare l'accesso alle singole finestre, è memorizzato come un vettore di elementi di tipo *struct*, ognuno contenente quattro interi che individuano univocamente una *finestra*, e corrispondenti rispettivamente alla dimensione della finestra, alla posizione nel *file* della *tagstruct*, alla posizione fisica e al *displacement* del primo elemento della finestra. Tutti i *flag* memorizzati nella *console* hanno tipo *unsigned char*, allo scopo di minimizzare l'occupazione di memoria e le informazioni riguardanti i dizionari sono organizzate in vettori dimensionati in base al numero dei dizionari e indicizzati per mezzo del tipo del dizionario. La prima posizione di ogni vettore contiene le informazioni relative al dizionario dei *tag*, la seconda quelle relative al dizionario degli *attributi* e la terza quelle relative al dizionario delle *parole*. Il vantaggio è di rendere il codice di accesso ai dizionari parametrico rispetto all'indice del vettore, e cioè rispetto al tipo di dizionario.
I dizionari, che sono memorizzati in forma compressa, possono essere decompressi in un

*file* temporaneo e caricati in memoria.

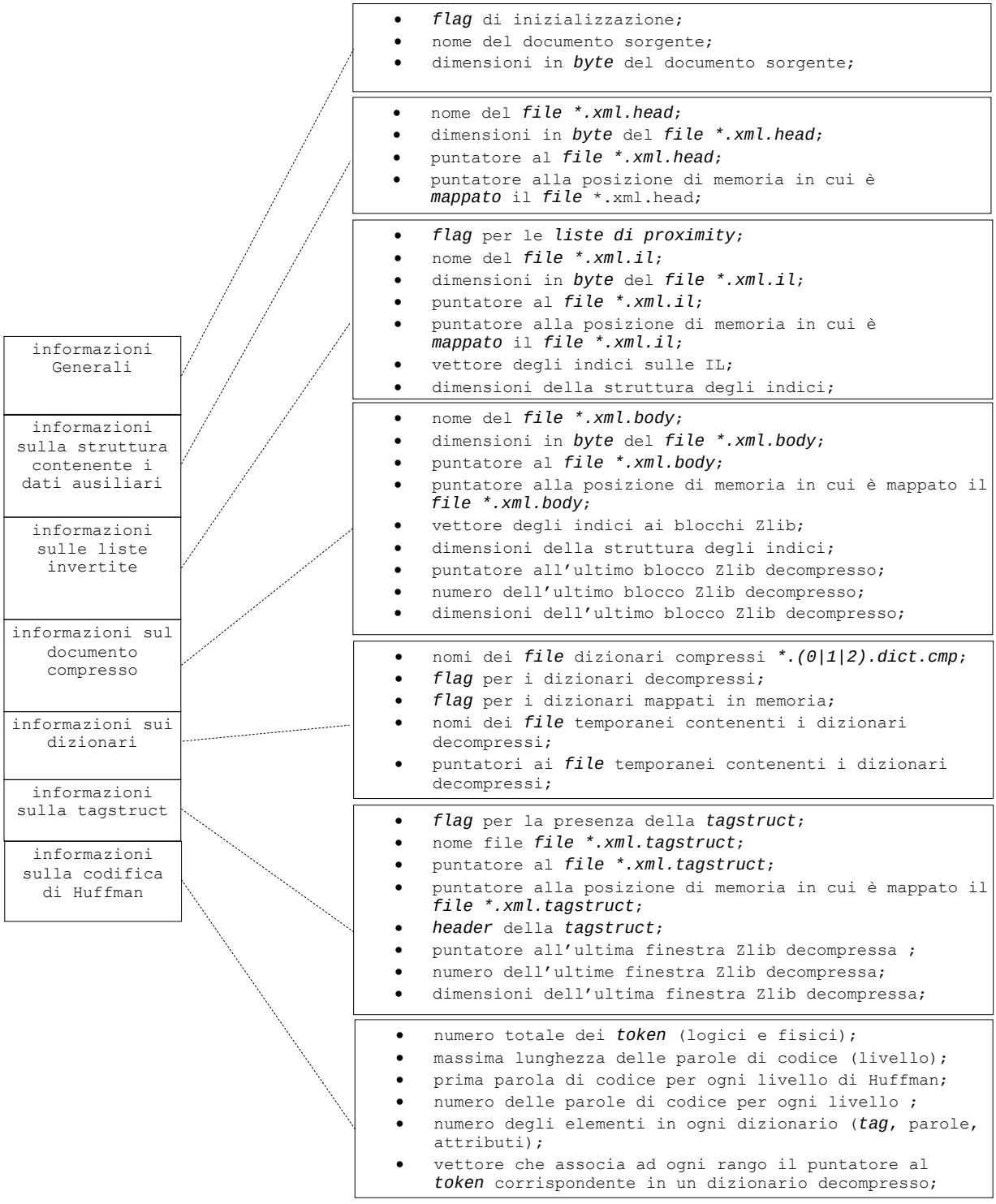



*Figura 3.1 - struttura della console*

L'operazione di caricamento ha come scopo quello di rendere i *token* del dizionario accessibili in forma immediata. All'uopo si alloca un vettore di puntatori a carattere, dimensionato con il numero dei *token* nel dizionario globale. Ognuno degli elementi del vettore punta alla posizione del *token* il cui rango corrisponde con il suo indice. Visto che i *token* sono delimitati da caratteri NULL sarà così immediato ottenere l'intero *token*. Tutti i vettori della *console* sono allocati *dinamicamente* nel momento in cui ciò è richiesto. Per quanto riguarda l'implementazione *dei meccanismi di caching* sui blocchi Zlib, ciò avviene semplicemente memorizzando il blocco decompresso in memoria e accedendo alla copia in memoria durante tutti gli accessi successivi ad esso.

## 3.2 L'API

L'API rappresenta l'interfaccia tra le applicazioni *client* e il nucleo della libreria XCDE. Essa è costituita da un insieme di funzioni C. Ogni funzione ha fra i suoi parametri la *console*, a cui si accede per riferimento. Le funzioni dell'API possono essere raggruppate in base alla struttura dati che gestiscono nel modo seguente:

- funzioni per la gestione della *console*;
- funzioni per l'accesso alle informazioni sui *file*;
- funzioni per operare sui dizionari;
- funzioni per operare sulle IL;
- funzioni per operare sul documento compresso;
- funzioni per operare sulla *tagstruct*;
- funzioni per la traduzione degli identificatori;

L'intestazione di una generica funzione è la seguente:

***int XCDE_*Nomeoperazione(parametro1,**
**.**
**.**
**parametroN,**
**indirizzo in memoria della console) ;**

e restituisce un codice che indica l'esito dell'operazione. In caso di errore l'implementazione della funzione tende a *minimizzare gli effetti collaterali prodotti dall'errore* e, quindi, l'eventuale operazione richiesta non viene eseguita e la *console* non viene modificata. 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 del codice

dell'applicazione *client* liberare la memoria una volta che questa non sia più utilizzata. Nel caso in cui la funzione esporti informazioni contenute nella *console*, di queste viene sempre fornita una copia. Si tratta di un meccanismo di *incapsulamento* che ha lo scopo di evitare che si possa accedere direttamente alla *console* e modificarne il contenuto, con possibili effetti collaterali indesiderati.

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

### 3.2.1 Funzioni per la gestione della console

Questo insieme di funzioni è fondamentale per realizzare qualsiasi operazione sui documenti, inclusa la procedura per *l'inizializzazione* e la *finalizzazione* della *console.* Tutte le applicazioni implementate utilizzando la libreria XCDE devono sempre *inizializzare* la *console* prima di iniziare a operare attraverso le funzioni della libreria, ed è buona prassi *finalizzare* la console prima di terminare.
La funzione per l'inizializzazione è la **XCDE_Init_Console**. Essa accetta come parametri il nome del *file* contenente il documento XML da processare, e una variabile di tipo *console* valida, cioè non inizializzata o che sia stata precedentemente inizializzata e poi finalizzata. Il nome del *file* passato come parametro alla funzione deve essere quello *sorgente* che è stato già compresso e indicizzato. Naturalmente non è necessario che questo *file* sia presente sul *file system*, mentre è necessario che siano presenti i *file* delle strutture dati creati a partire da esso. La funzione **XCDE_Init_Console** inizializza i campi della *console* con le informazioni necessarie per accedere alle varie strutture dati di indicizzazione, leggendole dal *file* delle informazioni ausiliarie. Inoltre la funzione controlla la presenza delle altre strutture dati richieste, restituendo un codice di errore nel caso in cui non riesca ad accedere a qualcuna di esse.

```
#include"xcde_lib.h"

int main(void)

{ // dichiarazione di una variabile di tipo console
  XCDE_Console_Type Console;

  // variabile utilizzata per memorizzare i codici di errore restituiti dalle funzioni
  int result=0 ;

  // inizializzazione della console. Notiamo come essa sia passata per riferimento, come
  // per tutte le funzioni della libreria.
  result=XCDE_Init_Console("libri.xml", &Console) ;

  if (result == -1)
       // gestione errore
       .
   // operazioni sul documento libri.xml
       .
   // finalizzazione della console.A partire da questo punto la variabile console può
   // essere utilizzata solo se riinizializzata
   result=XCDE_Finalize_Console(&Console) ;

   if (result == -1)
       // gestione errore
       .
   return(1); }
```

*Figura 3.2 - esempio di codice per l' inizializzazione e la distruzione della console*

La funzione **XCDE_Finalize_Console** distrugge la *console,* rimuovendo le informazioni contenute al suo interno. La finalizzazione della *console* non solo permette di poter riutilizzare successivamente la stessa variabile *console* su altri documenti, ma libera tutta la memoria allocata e rimuove tutti i *file* temporanei creati nel corso delle operazioni su di essa. Questa funzione accetta come parametro la sola *console.* Nella figura 3.2 possiamo osservare un esempio di codice C con l'inizializzazione e la finalizzazione della *console*. Il codice di qualsiasi applicazione *client* della libreria deve utilizzare sempre questo schema generale.

### 3.2.2 Funzioni per l'accesso alle informazioni sui file

Queste sono le funzioni che consentono all'applicazione *client* di conoscere i nomi e le dimensioni in *byte* dei diversi *file* associati alle strutture dati di un documento XML compresso e indicizzato. Per ogni *file* associato ad una struttura dati esistono due funzioni, una per ottenere il suo nome e una per derivare la sua dimensione. Ad esempio per il *file* delle IL esistono le funzioni **XCDE_Get_IL_Filename** e **XCDE_Get_IL_Filesize**. Ad entrambe si passa come parametro la *console*; inoltre la prima funzione richiede anche l'indirizzo di un variabile di tipo puntatore a carattere per contenere la copia del nome del *file* delle IL, mentre la seconda richiede l'indirizzo di una variabile di tipo intero per contenere la copia delle dimensioni delle IL.
Per i dizionari abbiamo due tipi di funzioni a seconda che si operi sui dizionari compressi o su quelli eventualmente decompressi. Si ricorda che questi ultimi sono memorizzati in *file* temporanei. Entrambe queste funzioni accettano come parametro aggiuntivo il tipo del dizionario su cui sono invocate.
Tutte le funzioni restituiscono un codice di errore nel caso in cui siano invocate su un *file* non presente. Questo può avvenire ad esempio per la *tagstruct*, nel caso in cui l'utente abbia preferito non crearla; o per i dizionari decompressi nel caso in cui l'operazione di decompressione non sia stata ancora effettuata.
Tutte queste informazioni sono ricavate dalla *console* che le memorizza al suo interno nel momento dell'inizializzazione. Esistono inoltre delle funzioni per conoscere il nome e la dimensione in *byte* del documento sorgente. Queste informazioni sono anch'esse memorizzate nella *console*, ma mentre quelle relative agli altri *file* sono memorizzate per effettuare le chiamate di sistema necessarie al *mapping*, queste ultime sono memorizzate nella struttura delle informazioni ausiliarie per consentire di creare correttamente i *nomi dei file delle strutture* e a scopo di *debugging*.
Al codice illustrato nella figura 3.2 possiamo aggiungere le seguenti dichiarazioni:

```
int DimensionIL=0 ;

char *Name_file_IL=NULL ;
```

e dopo il commento *'// operazioni sul documento libri.xml'* le chiamate:

```
result=XCDE_Get_IL_Filename(&Name_file_IL,
                            &Console) ;
```

```
result=XCDE_Get_IL_Filesize(&DimensionIL,
                            &Console) ;
```

che permettono di ottenere il nome e le dimensioni del *file* delle IL. La memoria allocata per contenere il nome del *file* dovrà essere preferibilmente liberata prima della terminazione con la chiamata di sistema *free*:

```
free(Name_file_IL) ;
```

### 3.2.3 Funzioni per operare sui dizionari

Le funzioni per la gestione dei dizionari consentono di decomprimerli e allocarne in memoria i *token* in modo da poter accedere ad essi in maniera immediata attraverso il loro *rango*, ottenere informazioni sui dizionari (come ad esempio il numero di *token* contenuti in ognuno) ed effettuare ricerche in base ad un determinato *pattern*.
I dizionari sono compressi e decompressi utilizzando il programma Gzip **[gzip]**. L'allocazione in memoria viene realizzata decomprimendo il dizionario in un *file* temporaneo (con la funzione **XCDE_Decompress_Dictionary**), *mappandolo* in memoria e allocando nella console un vettore di puntatori a *token* (con la funzione **XCDE_Load_Dictionary**). Più precisamente viene creato un vettore di puntatori a carattere, dimensionato in base al numero totale dei *token* nei tre dizionari. La memoria necessaria è allocata alla prima operazione di caricamento. Se il dizionario a cui il *token* di *rango i* appartiene è stato già caricato in memoria, allora la posizione *i* del vettore contiene un puntatore alla posizione nel *file* temporaneo dove si trova il *token*. Questo consente di ottenere in maniera semplice e veloce un *token* a partire dal suo *rango* (figura 3.3). Anche perché, ricordiamolo, i token sono separati dal carattere NULL: in questo modo è molto semplice individuare una singola stringa all'interno del *file*. Nella figura 2.12 l'operazione di "decompressione" è realizzata attraverso la chiamata **XCDE_Decompress_Dictionary**.

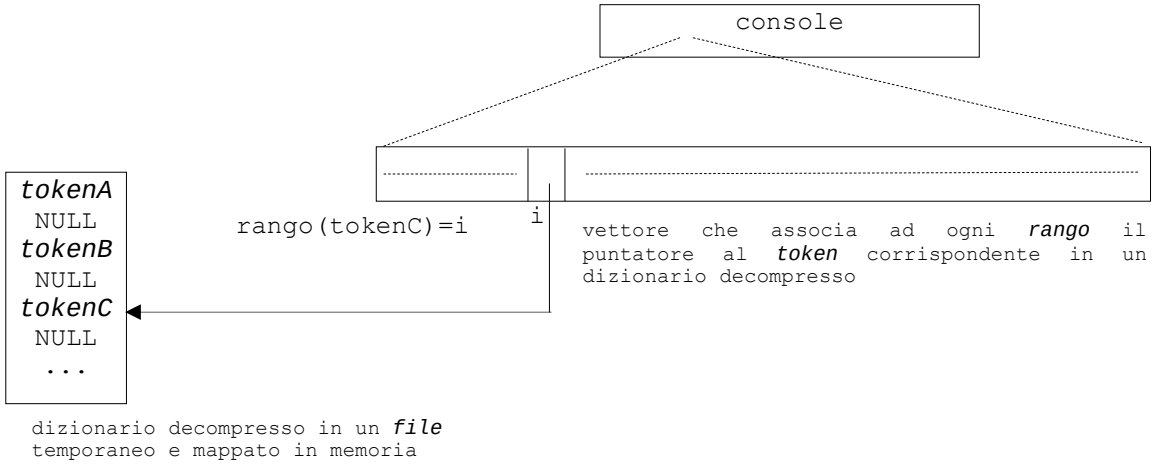



*Figura 3.3 - struttura per il collegamento fra il rango di un token e la stringa corrispondente nel dizionario*

Mentre l'allocazione richiede necessariamente la decompressione del dizionario, la decompressione può essere effettuata senza necessariamente doverlo allocare. Questo può essere utile quando si vogliono effettuare delle ricerche sui dizionari senza impiegare la memoria interna.
Le ricerche sono implementate con tre funzioni differenti, una per ogni dizionario. Questa soluzione si giustifica con l'osservazione che le ricerche sui dizionari hanno *logicamente* una natura differente. Le funzioni per la ricerca sono:

- **XCDE_Search_Tag_Dictionary** per le ricerche sul dizionario dei *tag*;
- **XCDE_Search_Attr_Dictonary** per le ricerche sul dizionario degli attributi;
- **XCDE_Search_Word_Dictionary** per le ricerche sul dizionario delle parole.

Le operazioni di ricerca sui dizionari individuano tutte le occorrenze in essi che soddisfano un *pattern* utilizzato come parametro delle ricerca e sono, insieme agli accessi alle IL, le operazioni più utilizzate per rispondere alle interrogazioni. Per effettuare una ricerca su un dizionario è necessario che quest'ultimo sia stato decompresso. Nel caso in cui questa operazione non sia avvenuta precedentemente, la funzione per la ricerca la esegue sul dizionario ad essa corrispondente.
Le ricerche sui *tag* sono realizzate utilizzando la funzione **XCDE_Search_Tag_Dictionary**, che è tra tutte la più semplice in quanto contempla solo un solo tipo di *match*: quello esatto. La funzione accetta come parametro una stringa e cerca nel dizionario dei *tag* la sua occorrenza. Nel caso in cui la stringa sia uguale ad un *token* nel dizionario, essa restituisce il suo *rango all'interno del dizionario dei tag*. Quest'ultimo, lo ricordiamo, deve essere maggiore di 0, per cui il valore 0 viene utilizzato come valore di *escape* per indicare che la ricerca si è conclusa con un insuccesso.
Le ricerche sugli attributi sono realizzate attraverso la funzione **XCDE_Search_Attr_Dictionary.** Queste sono più complesse delle predenti in quanto contemplano l'uso di due *pattern*: per il nome e per il valore di un attributo. I tipi di ricerca possibili per ognuno dei pattern sono: *esatta*, per *prefisso*, per *suffisso*, per *sottostringa*, con un numero fissato di *errori* e per *espressione regolare*. Per il valore di un attributo è possibile stabilire anche se l'eventuale *suffisso* o *prefisso* specificato attraverso il *pattern* si riferisca all'intero valore o ad un suo *token*. Ad esempio per l'attributo `venditori="Barnes&Noble, Bol",` il *pattern Bol* occorre come prefisso di un *token*, ma non come prefisso dell'intero valore dell'attributo. È anche possibile specificare un solo *pattern* di ricerca per il nome o per il valore. Infine, per ogni *pattern* si può stabilire se la ricerca debba essere *case sensitive* o meno.
A differenza delle ricerche sul dizionario dei *tag*, quelle eseguite sul dizionario degli attributi possono avere come risultato più occorrenze. La funzione **XCDE_Search_Attr_Dictionary** deve quindi restituire il numero totale di occorrenze trovate e, per ognuna di esse, il *rango del token nel dizionario*. Ciò è realizzato semplicemente restituendo il numero totale di occorrenze e un vettore contenente i ranghi nel dizionario di ognuna di esse.
Le ricerche sul dizionario delle parole sono realizzate attraverso la funzione **XCDE_Search_Word_Dictionary**. L'interfaccia della funzione è simile a quella vista per

le ricerche sul dizionario degli attributi, ma naturalmente essa contiene solo un singolo *pattern*.
Per determinare il tipo di ricerca: *case sensitive/insensitive*, per *prefissi, suffissi*, ecc., sono definite all'interno della libreria delle opportune costanti il cui valore viene passato come parametro alle funzioni per la ricerca.
Le ricerche sui dizionari sono realizzate utilizzando i comandi *grep* **[grep]** e *agrep* **[Wu],** attraverso una chiamata *system*, sui *file* temporanei contenenti i dizionari decompressi. In particolare il comando *agrep* è utilizzato per implementare le ricerche che prevedono *pattern* con un numero determinato di errori.
Questa soluzione permette di effettuare in maniera semplice ricerche anche molto complesse, ad esempio con errori o basate su espressione regolare. Chiaramente questo approccio non è scevro da inconvenienti, e infatti può essere velocizzato eliminando la chiamata *system*, di questo parleremo nel capitolo 5.
Riferendoci all'esempio della figura 3.2 possiamo effettuare una ricerca per *prefisso* sul dizionario delle parole aggiungendo le seguenti dichiarazioni:

```
// numero delle occorrenze trovate
int number_occur=0;

// puntatore al vettore dei numeri dei token corrispondenti alle singole
// occorrenze trovate
int *occur=NULL;
```

e le seguenti chiamate:

```
// ricerca per prefisso ghirl sul dizionario delle parole. La ricerca è
// case insensitive
result=XCDE_Search_Word_Dictionary((char *) "ghirl",
                                   &occur,
                                   &number_occur,
                                   XCDE_PREFIX,
                                   XCDE_NOCASE,
                                   0,
                                   &Console) ;
```

dopo l'esecuzione della funzione di ricerca, e ipotizzando che il documento sia quello di figura 1.1, la variabile *number_occur* conterrà il valore 1 (numero di occorrenze di *token* con prefisso *ghirl* nel dizionario delle parole), e *occur* avrà dimensione 1 e conterrà il *rango del token ghirlanda* nel dizionario delle parole (l'unico *token* corrispondente al pattern specificato).
Esistono delle funzioni per acquisire informazioni sulla composizione dei dizionari. Esse sono **XCDE_Get_Num_El_Dict** e **XCDE_Get_Num_El_Lev**.
La prima funzione consente di ottenere il numero di *token* contenuti in un dizionario, ad esempio il numero totale di *tag*. La seconda funzione consente di ottenere il numero dei *token* in un dizionario con una determinata lunghezza di parola di codice di Huffman (o, equivalentemente, ad una determinata profondità dell'albero canonico di Huffman).

### 3.2.4 funzioni per operare sulle IL

La funzione **XCDE_Get_IL_ListPos** accede alla *lista invertita delle posizioni e delle*

*proximity* di un *token* del dizionario delle parole il cui *rango* è passato come parametro. Essa restituisce il numero di elementi di entrambe le liste, che lo ricordiamo sono uguali, e un vettore di triple, ognuna contenente la posizione fisica, il *displacement* e la *proximity*. Il *displacement* sarà 0 nel caso in cui il *token* non derivi dall'espansione di un'entità. La funzione restituisce un codice di errore nel caso in cui le liste di *proximity* non siano state create dall'utente o nel caso in cui si richieda la *proximity* di un *token* che non fa parte del dizionario delle parole.
Per sapere se le liste di *proximity* sono state create si può utilizzare la funzione **XCDE_Setted_Proximity**, che restituisce un *flag* attivato nel caso in cui la creazione sia avvenuta.
Per implementare le coppie e le triple di cui sopra la libreria definisce i due tipi *XCDE_IL_Couple_Type* e *XCDE_IL_Triple_Type*. Per accedere alle singole *liste invertite,* le funzioni utilizzano gli indici delle liste presenti nella *console*. Una volta individuato il blocco a cui la lista appartiene, le funzioni accedono sequenzialmente ad esso possibilmente utilizzando l'ampiezza in *byte* di ciascuna lista per saltare velocemente a quella richiesta.
Riferendoci all'esempio di figura 3.2 possiamo leggere la lista delle posizioni del *token* di *rango* 10, che ipotizziamo essere una parola, aggiungendo le seguenti dichiarazioni di variabili:

```
// variabile puntatore al vettore di triple che conterrà le posizioni e
// la proximity
XCDE_IL_Couple_Type *IL_Pos=NULL;

// numero di triple nel vettore
int IL_Pos_num_el=0 ;

// contatore utilizzato all'interno di un ciclo per accedere ai singoli
// elementi delle triple
int counter=0;
```

e la seguente chiamata di funzione dopo l'inizializzazione della *console*:

```
result=XCDE_Get_IL_ListPos((int) 10,
                           &IL_Pos,
                           &IL_Pos_num_el,
                           &Console)
```

ipotizzando che il valore di *IL_Pos_num_el* sia maggiore di 0 sarà possibile accedere ai singoli elementi della lista nel modo seguente:

```
for(counter=0; counter < IL_Pos_num_el; ++counter)

 { print("\n Posizione fisica elemento %d : %d",
         counter, IL_Pos[counter].XCDE_Position) ;

   print("\n Posizione fisica elemento %d : %d",
         counter, IL_Pos[counter].XCDE_Offset) ;
 }
```

Lo spazio allocato per la IL potrà essere liberato, prima della terminazione del programma,

con la chiamata:

```
free(IL_Pos);
```

### 3.2.5 Funzioni per operare sul documento compresso

La funzione **XCDE_Get_Body_Forward_Codeword** consente, data una posizione nel documento compresso, di ottenere la parola di codice di Huffman che inizia da essa e la sua lunghezza. La parola di codice , una volta rimossi i *tagging bit*, sarà inserita in una variabile di tipo *int*, così come la sua lunghezza. Da esse sarà possibile ricondursi, attraverso le funzioni per la traduzione (sezione 3.2.7), al *token* corrispondente a quella parola di codice. Naturalmente si suppone che la correttezza della posizione nel documento compresso passata come parametro alla funzione. Nel caso questo non sia vero, ad esempio perché in quella posizione non si trova l'inizio di una parola di codice, la funzione restituisce un codice di errore.
La funzione **XCDE_Get_Body_Backward_Codeword** è simile alla precedente, ma essa accede alla parola di codice che precede la posizione nel documento compresso passata come parametro.
Nella figura 3.4 sono schematizzate le due funzioni. Il punto *x* rappresenta la posizione passata come parametro.

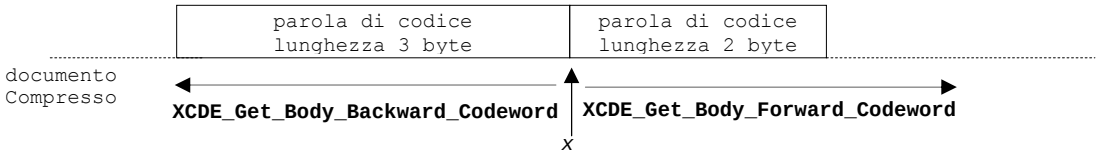



*Figura 3.4 - schematizzione del funzionamento delle funzioni XCDE_Get_Body_Backward_Codeword e XCDE_Get_Body_Forward_Codeword*

La funzione **XCDE_Get_Body_Textwindow** estende le funzionalità delle due funzioni precedenti. Essa infatti accetta come parametri una posizione all'interno del documento compresso, una funzione booleana e un contatore. Restituisce una finestra di testo del documento sorgente XML centrata nella posizione passata in *input* alla funzione. La decompressione in avanti e indietro continua fintanto che la funzione booleana non abbia restituito *true* un numero di volte pari al numero contenuto nel contatore, o fino a quando non si raggiunge l'inizio o la fine del documento. Questa funzione può essere utile ad esempio per estrarre porzioni di documento contenenti un certo numero di parole, oppure un certo numero di *tag* o altri tipi di strutture. L'implementazione utilizza le funzioni **XCDE_Get_Body_Backward_Codeword** e **XCDE_Get_Body_Forward_codeword** e le funzioni per la traduzione (sezione 3.2.7).
Infine la funzione **XCDE_Body_Eof** consente di stabilire se una posizione sia alla fine del documento compresso. Essa restituisce un *flag* con questa informazione ed è utilizzata come condizione di terminazione per la decompressione del documento.
Si noti che le funzioni di decompressione del codice di Huffman hanno bisogno innanzitutto di determinare il blocco Zlib a cui la parola di codice appariene, data la sua posizione; successivamente, esse accedono alla copia in cache di quel blocco, se presente,

altrimenti lo decomprimono utilizzando la libreria Zlib e lo caricano nella cache della console. Una volta ottenuto il blocco decompresso accedono alla parola di codice indicata dalla posizione e la copiano in una variabile di tipo *int*, dopo aver eliminato i *flag* per la terminazione presenti all'inizio di ogni *byte* e aver effettuato le opportune operazioni di *shift*. Nella figura 3.5 sono illustrate le trasformazioni effettuate su una generica parola di codice, nell'esempio di lunghezza 2:

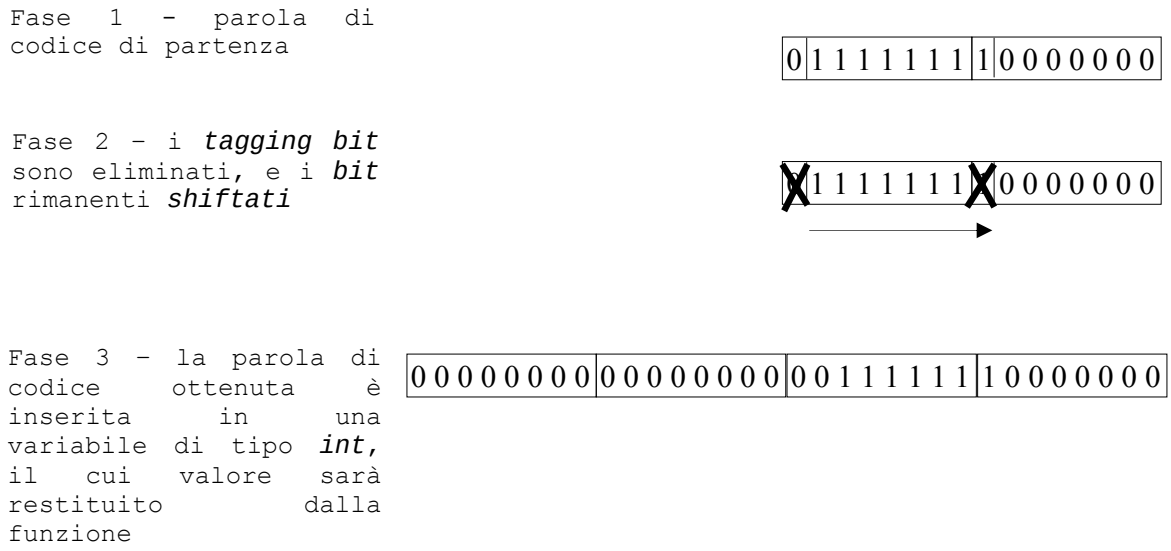



*Figura 3.5 - trasformazioni effettuate su una parola di codice di Huffman*

I blocchi Zlib hanno, una volta decompressi, dimensioni di 64*Kbyte* (ad esclusione possibilmente dell'ultimo blocco). Naturalmente la modalità di divisione in blocchi basata sul numero di *byte*, non garantisce che essi siano allineati rispetto alle parole di codice. È compito delle funzioni che effettuano la decompressione aggirare questo problema accedendo, se necessario, al blocco precedente o successivo per completare la lettura della parola di codice a cavallo tra due blocchi consecutivi.

### 3.2.6 Funzioni per operare sulla tagstruct

Ricordiamo che la *tagstruct*, come le *liste di proximity*, non è sempre presente. Per ottenere questa informazione è necessario utilizzare la funzione **XCDE_Setted_Tagstruct** che restituisce un *flag* il cui valore è 1 se la *tagstruct* è presente.
La funzione **XCDE_Tagstruct_Tags_Enclosing** riceve in *input* una coppia <*posizione fisica, displacement*> che individua una posizione del documento XML nella sua visione logica e restituisce tutti i *tag* che contengono quella posizione. Al solito, l'*output* restituito consiste del numero dei *tag* e di un vettore contenente il loro *rango nel dizionario dei tag,* ordinato secondo la relazione di contenimento (il *tag* in posizione *i* contiene il *tag* in posizione *i+1*). La funzione implementa l'algoritmo di Chazelle descritto nella sezione 2.5.1 accedendo al *file* della *tagstruct*. Ricordiamo che questo *file* contiene un *header* con le informazioni per l'accesso alle singole *finestre*. Quest'ultime sono memorizzate nella *console* la prima volta che si accede alla struttura, allo scopo di velocizzare gli accessi successivi. Anche le *finestre* della *tagstruct* sono compresse utilizzando la libreria Zlib e anche in questo caso è stato implementato un meccanismo di *caching* che evita di dover

effettuare più volte la decompressione in presenza di località temporale negli accessi alla stessa finestra.
Assumendo di avere disponibile la *lista delle posizioni* di un *token* e di aver già creato la *tagstruct*, possiamo modificare il codice presente in figura 3.2 per ottenere i *tag* che contengono la prima occorrenza di quel *token*. Si aggiungono a questo scopo le seguenti dichiarazioni:

```
// numero dei tags che contengono la posizione
int Num_tags=0;

// puntatore al vettore dei ranghi dei tags
int *Ranks_tags=NULL ;
```

e la chiamata di funzione:

```
result=XCDE_Get_Tagstruct_Tags_Enclosing(IL_Pos[0],
                                         &Ranks_tags,
                                         &Num_tags,
                                         &Console) ;
```

dove `IL_Pos` è dello stesso tipo della corrispondente dichiarazione di sezione 3.2.4. Dopo la chiamata, gli elementi del vettore *Rangs_tags* saranno i *ranghi dei tag* che contengono la posizione corrispondente a *IL_Pos[0]*.
In conclusione la funzione **XCDE_Get_Tagstruct_Tags_Enclosing** consente di realizzare interrogazioni strutturali sul documento XML, offrendo così una possibilità innovativa che è quella di poter eseguire estrazioni di *snippets ben-formati* secondo la sintassi XML. Questa proprietà verrà ampiamente utilizzata nella realizzazione del motore *XCDE_Search Engine* descritto nel capitolo 4.

### 3.2.7 Funzioni per la traduzione degli identificatori

Abbiamo visto finora che un *token* di un qualsiasi dizionario può essere identificato in diversi modi, ad esempio con il *rango nel dizionario globale* o la coppia <*dizionario*, *rango del token nel dizionario*>. Anche le parole di codice di Huffman, ottenute attaverso le funzioni illustrate nella sezione 3.2.5, rappresentano identificatori dei *token*. La presenza di più identificatori, pur complicando l'interfaccia delle funzioni, favorisce la sua modularità e garantisce l'indipendenza delle componenti della libreria. Ad esempio si può ottenere la *lista invertita* di un *token* a partire dalla sua parola di codice (ricavata dal documento compresso) senza decomprimere i dizionari; e si può ottenere la struttura del documento attraverso la *tagstruct* senza dover accedere in nessun modo al documento compresso. Inoltre la presenza di più identificatori facilita l'implementazione di alcuni moduli: il *rango nel dizionario globale* può essere usato per accedere rapidamente alle IL; e le funzioni che effettuano le ricerche sui dizionari possono determinare facilmente il *rango nel dizionario* corrispondente a un *token*.
É fondamentale che esista la possibilità di poter ottenere, in maniera semplice e veloce, un tipo di identicatore a partire da un altro equivalente. Le funzioni che seguono realizzano questa possibilità in modo efficiente.
Le funzioni **XCDE_Dict2Rank** e **XCDE_Rank2Dict** consentono di ottenere la coppia <*dizionario, rango del token nel dizionario*> a partire dal suo rango nel dizionario globale e

viceversa. Esse restituiscono un codice di errore nel caso in cui i parametri non siano corretti, ad esempio se il numero del *token* nel dizionario sia uguale o minore di 0 o se il *rango globale* è minore di 0.
Per quanto riguarda le funzioni che operano sul documento compresso, alcune di esse restituiscono la parola di codice di Huffman corrispondente ad un *token*. Per passare dalla parola di codice al *token* o al suo *rango* (nel dizionario globale) si possono utilizzare le funzioni **XCDE_Codeword2Rank** e **XCDE_Codeword2Token**.
Per effettuare le traduzioni dei diversi identificatori si utilizzano le informazioni contenute nella *console*, ad esempio quelle sulla suddivisione in livelli dei singoli dizionari, che ci consentono di passare dal *rango del token in un dizionario* al suo *rango globale*; o quelle riguardanti la prima parola di codice per ogni livello, che ci consente di passare dalla *parola di codice* al *rango* dell'elemento *nel dizionario globale*.
E' facile osservare che non è possibile effettuare *tutti i passaggi*. Ad esempio non è possibile ottenere il *rango di un token* a partire dalle sua parola di codice. Per semplificare l'API, sono state realizzate soltanto le traduzioni utili ai fini dell'implementazione della libreria.
Nella figura 3.6 è contenuta una schematizzazione delle funzioni utilizzabili per la traduzione.

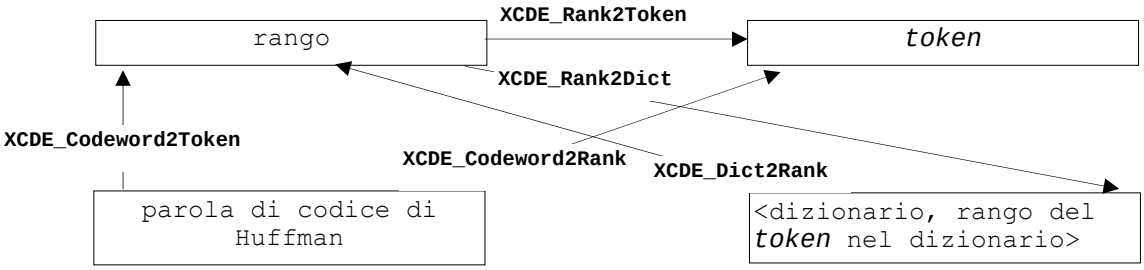



*Figura 3.6 - schematizzazione delle funzioni per la traduzione degli identificatori*

### 3.2.8 Utilizzo delle funzioni: un semplice motore di ricerca

Possiamo osservare nella figura 3.7 il codice di un semplice motore di ricerca costruito utilizzando le funzioni della libreria XCDE. Esso consente di cercare tutte le occorrenze di un *tag* in un documento e visualizzarne il contenuto. Il nome del *tag* e il nome del *file* contenente il documento su cui effettuare la ricerca sono passati al motore attraverso la linea di comando. Ad esempio se si volessero visualizzare tutte le occorrenze del *tag TITOLO* del documento *libri.xml* di figura 1.1 il comando sarebbe:

```
$ simple_engine libro.xml TITOLO
```

```
#include"xcde_lib.h"

int main(int argc, char *argv[])
{ int result=0; // variabile per la lettura dell'esito della funzione
  int cw=0, cwlen=0; // variabili che contengono la parola di codice e la sua lunghezza
  char *token=NULL; // puntatore alla stringa che contiene il token
  int numdict=0; // variabile che contiene il numero del token nel dizionario dei tag
  XCDE_Console_Type Console;  // variabile che contiene la console
  int num_IL_Pos=0; // variabile che contiene il numero di elementi della IL
  int rank; // variabile che contiene il rango del tag
  XCDE_IL_Couple_Type *IL_Pos;  // variabile puntatore alla lista delle posizioni
  /* ------------------------------------------------------------------------*/
  // inizializza la console. Il nome del documento è passato come parametro in linea
  result=XCDE_Init_Console((const char *) argv[1], &Console);

  // gestisce l'eventuale condizione di errore
  if (result== -1){ print("\ndocumento non corretto !"); exit(-1); }

  // effettua la ricerca sul dizionario dei tag
  result=XCDE_Search_Tag_Dictionary(argv[2], &numdict, &Console);

  // gestisce l'eventuale condizione di errore: finalizza la console e esce
  if (result==-1) {print("\nError!"); XCDE_Finalize_Console(&Console); exit(-1); }

  if (numdict > 0) { // esiste un'occorrenza del tag nel dizionario
   // ottiene il rango del tag a partire dal dizionario e dal numero nel dizionario
   result=XCDE_Dict2Rank(TAG, &numdict, &rank, &Console) ;

   if (result==-1) { print("\nErrore!") ; XCDE_Finalize_Console(&Console) ; exit(-1) ; }

   // accede alla liste invertita delle posizioni del tag
   result=XCDE_Get_IL_ListPos(rank, &IL_Pos, &num_IL_Pos, &Console)

   if (result==-1) { print("\nErrore!") ; XCDE_Finalize_Console(&Console) ; exit(-1) ; }

   // cicla su tutte le occorrenze del tag
   for(i=0; i<num_IL_Pos; i+=2) {
        // legge la posizione di apertura del tag
        compress_doc_position=IL_Pos[i].XCDE_Position ;

        print("\n Contenuto occorrenza numero %d", (i/2)+1) ;

        // decomprime il contenuto dell'occorrenza fino alla sua chiusura
        while(compress_doc_position<=IL_Pos[i+1].XCDE_Position) {

         // ottiene la parola di codice nel doc.compresso corrispondente alla posizione
         result=XCDE_Get_Body_Forward_Codeword(&cw,&cwlen,compress_doc_position, &Console) ;

         if (result==-1) { print("\nError!"); XCDE_Finalize_Console(&Console); exit(-1); }

         // dalla parola di codice ottiene il token
         result=XCDE_Codeword2Token(cw, cwlen,token, &Console) ;

         if (result==-1) { print("\nError!"); XCDE_Finalize_Console(&Console); exit(-1); }

         // visualizza il token e libera la memoria allocata per la stringa
         print("%s", token) ; free(token) ;

         // aggiorna la posizione nel file compresso
         compress_doc_position+=cwlen;
         } }
 // libera la memoria allocata per la lista delle posizioni
 if (Num_IL_Pos > 0) free(IL_Pos) ; }

 else print("\nNessuna occorrenza\n") ;

 // finalizza la console
 XCDE_Finalize_Console(&Console) ;

 return(1) ; }
```

*Figura 3.7 – codice di un semplice motore di ricerca costruito utilizzando l'API di XCDE*

# Capitolo 4

# Il motore di ricerca

*In questo capitolo descriveremo il motore XCDE Search Engine, realizzato a partire dalla libreria XCDE e specializzato al supporto di ricerche su documenti letterari codificati con XML-TEI e forniti dal CIBIT (Centro Interuniversitario Biblioteca Italiana Telematica). La prima parte del capitolo contiene una descrizione generale del motore; segue un semplice linguaggio utilizzato per formulare le interrogazioni, e infine sono indicati alcuni esempi di utilizzo. L'interfaccia web realizzata per formulare le interrogazioni in remoto sarà illustrata alla fine del capitolo.*

## 4.1 Struttura del motore di ricerca

Il motore di ricerca *XCDE_Search Engine* è formato da diversi moduli, ognuno realizzato per implementare una particolare funzione. Possiamo vedere una sua schematizzazione nella figura 4.1. Il *kernel* del motore è costituito da tre comandi: *xcde_search, xcde_view* e *xcde_extract*, costruiti a partire dalla libreria XCDE.
Il comando *xcde_search* utilizza un'interfaccia in linea per descrivere le interrogazioni che poi vengono eseguite sul documento passato in *input*. Il risultato della ricerca consiste di una serie di triple ciascuna formata dal *nome del documento* ove è stata effettuata la ricerca e da una coppia di posizioni fisiche nel documento compresso. Queste ultime delimitano la *finestra di dimensione minima racchiudente tutte le occorrenze dei token che rispondono all'interrogazione*.
Il comando *xcde_view* riceve in *input* una delle triple prodotte da *xcde_search* e visualizza la porzione del documento (*snippets*) ove ricade l'occorrenza specificata da quella tripla. Si tratta di porzioni *ben-formate* che possono quindi essere visualizzate o passate ad altre applicazioni per una loro successiva elaborazione.

Il comando *xcde_extract* consente di estrarre sullo *standard output* tutto il documento a partire dalla sua forma compressa.
L'interfaccia *web* del motore consente di realizzare in maniera semplice, intuitiva e in remoto le interrogazioni sulla collezione di documenti indicizzati. Queste interrogazioni vengono tradotte nel linguaggio di interrogazione di *xcde_search* attraverso degli *script* Perl e mediante il meccanismo delle *cgi-bin*. I risultati forniti da *xcde_search* vengono passati in *input* al modulo *xcde_view* per la visualizzazione degli *snippets* ed, eventualmente, al modulo *xcde_extract* per la decompressione e visualizzazione di interi documenti. L'interrogazione può essere effettuata su singoli documenti o su sottoinsiemi della collezione, specificati dall'utente a tempo di interrogazione. Questa funzione viene offerta attraverso l'utilizzo di opportuni *script* Perl che permettono di iterare l'esecuzione dei moduli *xcde_search* e *xcde_view* sui singoli documenti interrogati.

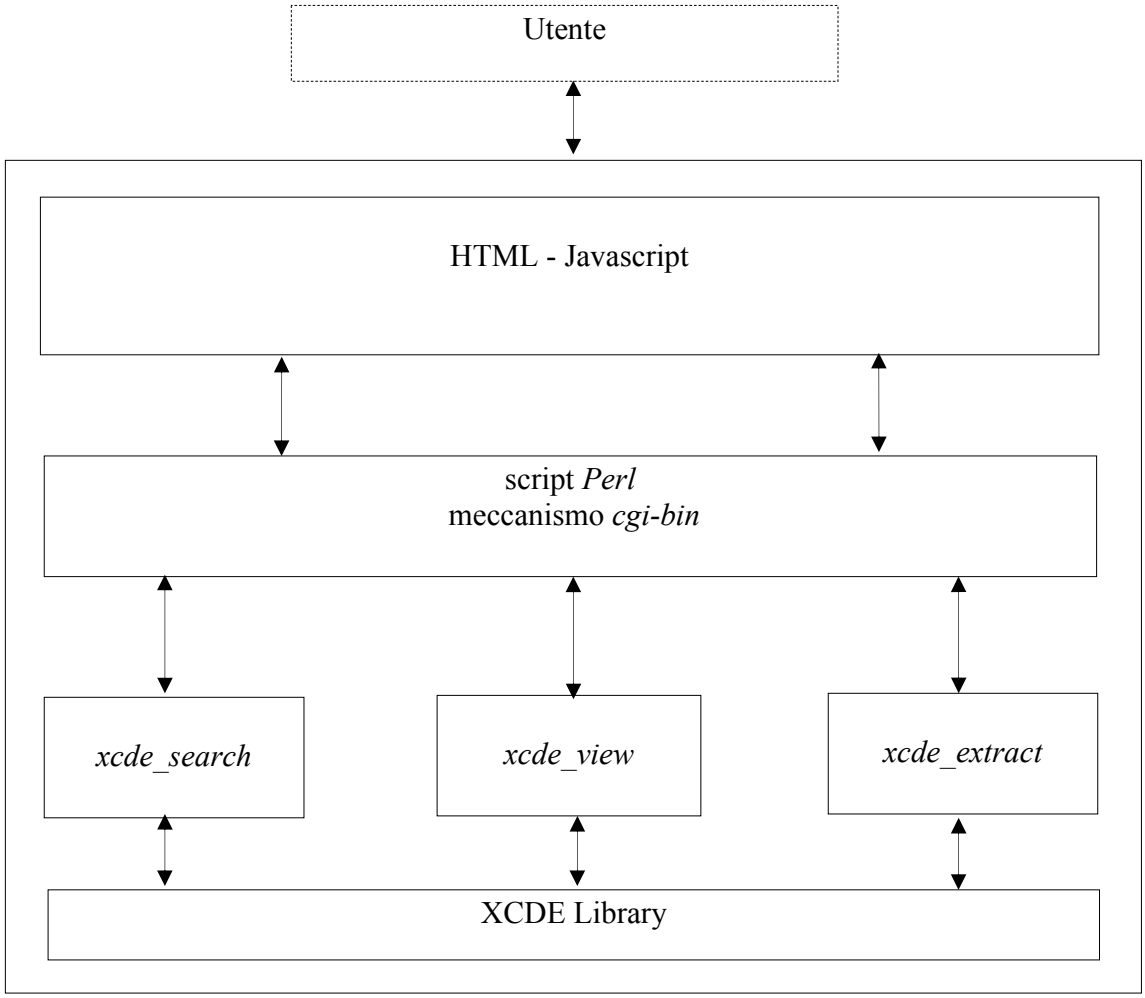



*Figura 4.1 - schematizzazione del motore di ricerca XCDE_Search Engine*

# 4.2 Il linguaggio per formulare le interrogazioni

Abbiamo già accennato nella sezione precedente all'esistenza di un linguaggio specifico per la formulazione delle interrogazioni nel comando *xcde_search*. Queste interrogazioni vengono passate in *input* ad *xcde_search* come una serie di parametri in linea, insieme al nome del documento su cui si deve effetture la ricerca. La sintassi è la seguente:

```
$ xcde_search documento interrogazione
```

dove *interrogazione* ha la struttura illustrata nella figura 4.2.

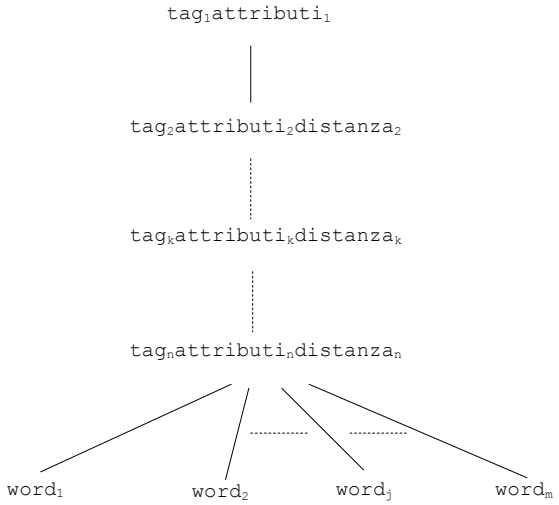



*Figura 4.2 - struttura delle interrogazioni di XCDE_Search*

si può notare come l'interrogazione abbia una struttura con un *singolo cammino* contenente i *tag*. I nodi relativi alle *parole* derivano dall'ultimo *tag* e sono tutti stesso livello. Questo tipo di interrogazione è una semplificazione di quelle formulabili su XML mediante altri linguaggi, come ad esempio XQuery **[XQuery]**, ma risponde alle richieste dell'utente umanista. Le interrogazioni vengono effettuate sostanzialmente su *stringhe*, che rappresentano i diversi *token*. Una stringa in realtà può corrispondere a *pattern* diversi a seconda del tipo di ricerca specificata. Essa può essere ad esempio una *parola,* un *prefisso*, un *suffisso*, una *sottostringa*, un'*espressione regolare*, ecc.. Per rappresentare sia le *stringhe* che i tipi di ricerca da effettuarsi su di esse abbiamo definito le *string-query*, la cui struttura è illustrata nella figura 4.3.

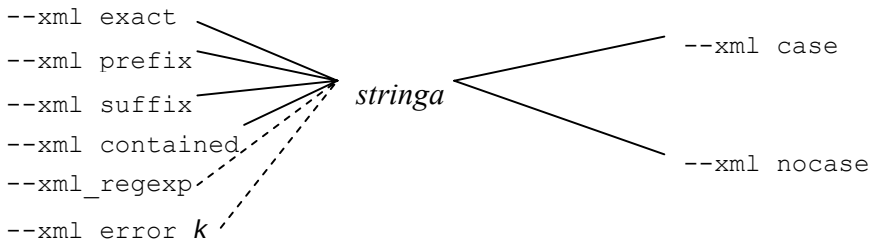



*Figura 4.3 - struttura generale di una string-query*

Attraverso la combinazione dei vari elementi è possibile definire *pattern* di ricerca sulle *query*. La prima componente indica in *tipo* di ricerca:

| | |
|---|---|
| `--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` *k* | ricerca su una stringa con un numero massimo *k* di errori (*k*>0); |

La seconda componente è la *stringa* vera e propria, che ha un significato diverso a seconda del tipo di ricerca: essa può essere una *stringa esatta*, un *prefisso*, un *suffisso*, ecc.. Nel caso di ricerca per *espressione regolare* la stringa deve essere racchiusa fra doppi apici (“) e deve corrispondere alla sintassi delle *espressioni regolari* di *grep* **[Grep]**.
La terza componente indica la *case-sensitivity* della ricerca. Essa è opzionale e nel caso in cui non sia presente la ricerca è considerata *case-sensitive*. Per le ricerche su *espressioni regolari* e con *errori* l’opzione sulla *case-sensitivity* deve essere omessa. In questi casi infatti essa non avrebbe senso, provocando un errore di sintassi. Nel caso di ricerca per *errori* dopo il *flag* `--xml_error` deve essere specificato il numero massimo degli errori, cioè dei caratteri che è possibile cambiare, eliminare o aggiungere all’interno di un *token* affinché esso corrisponda al *pattern di ricerca*.
Le *string-query* sono delle componenti delle interrogazioni che devono essere inserite in espressioni più ampie per essere significative. Un’interrogazione è formata da due parti: una relativa ai *tag* e una relativa alle *parole*. All’interno di esse le *string-query* sono utilizzate per specificare i diversi *token* richiesti. La parte delle parole può anche essere omessa, mentre è sempre necessario inserire almeno la specifica di un *tag*.

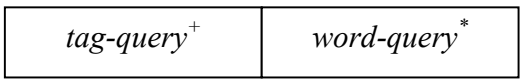



*Figura 4.4 - struttura generale di un’interrogazione*

La parte delle interrogazioni relativa ai *tag* è formata da una sequenza di una o più *tag-query*, mentre la parte relativa alle parole è formata da una sequenza eventualmente vuota di *word-query*. Le *tag-query* consentono di formulare interrogazioni sui *tag* e la loro struttura. Nella figura 4.5 è possibile osservare una generica *tag-query*. Nelle *tag-query* attraverso il *flag* `--tag` si specifica esattamente il nome del *tag*. È possibile inoltre specificare quali attributi devono essere o non essere associati ad esso affinché corrisponda al *pattern di ricerca*. Questo si effettua rispettivamente con i *f;ag* `--withattr` e `--withoutattr`.

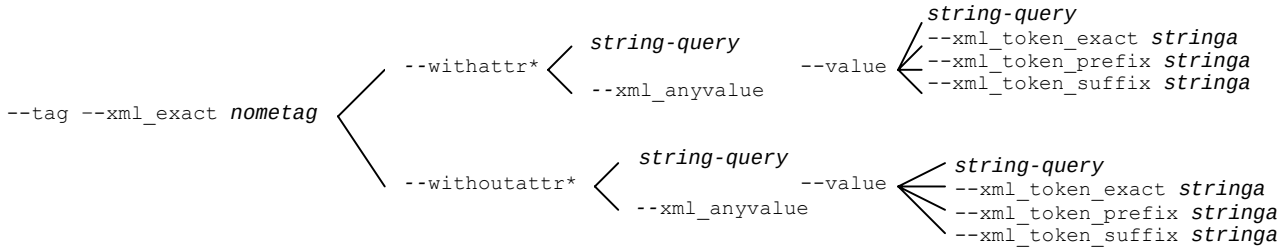



*Figura 4.5 – struttura generale di una tag-query*

Come già osservato nella sezione 3.2.3 in realtà per ogni attributo richiesto possono essere specificati un *pattern* per il nome e uno per il *valore*. Quando si specifica un attributo all'interno di una richiesta è necessario inserire almeno un *pattern* di ricerca relativo ad esso. Nel caso in cui si fornisca un *pattern* per il nome basta semplicemente far seguire al *flag* `--withattr o --withoutattr` la *string-query* corrispondente ad esso. Eventualmente la *string-query* può essere seguita dal *flag* `--value` e dalla specifica del *pattern* per il valore. Nel caso in cui si voglia specificare solo quest'ultimo si deve inserire il *flag* `--withattr o --withoutattr` seguito dal *flag* `--xml_anyvalue` e dal *pattern* per la ricerca sui valori degli attributi. Quest'ultimo è una *string-query* estesa, in cui in aggiunta possono sono presenti i *flag* `--xml_token_exact`, `--xml_token_prefix` o `--xml_token_suffix` che consentono di individuare, rispettivamente, una ricerca *esatta*, per *prefisso* e per *suffisso* non sull'interno valore dell'attributo ma su un suo *token*. È opportuno ricordare che i valori degli attributi sono divisi in *token* in base ad una lista di separatori presente in un *file* di configurazione modificabile dall'utente. Per ogni *tag* è possibile naturalmente specificare più attributi.

Esempi di *tag-query* sono i seguenti:

```
--tag --xml_exact TITOLO
```

richiede tutte le occorrenze del *tag* TITOLO (ma non titolo);

```
--tag --xml_exact TITOLO --withattr --xml_exact venditori
```

richiede tutte le occorrenze del *tag* TITOLO con un attributo *venditori*;

```
--tag --xml_exact TITOLO --withattr --xml_prefix vend
```

richiede tutte le occorrenze del *tag* TITOLO con un attributo il cui nome contiene il prefisso *vend*;

```
--tag --xml_exact TITOLO --withattr --xml_suffix ori
```

richiede tutte le occorrenze del *tag* TITOLO con un attributo il cui nome contiene il suffisso *ori*;

```
--tag --xml_exact TITOLO --withattr --xml_contained endi
```

richiede tutte le occorrenze del *tag* TITOLO con un attributo il cui nome contiene la sottostringa *endi*;

```
--tag --xml_exact TITOLO --xml_withattr --xml_error 1 venditoro
```

richiede tutte le occorrenze del *tag* TITOLO con un attributo il cui nome può essere ottenuto dalla stringa *venditoro* con al massimo 1 errore;

```
--tag --xml_exact TITOLO --xml_withattr --xml_regexp "venditor[ei]"
```

richiede tutte le occorrenze del *tag* TITOLO con un attributo il cui nome corrisponda all'*espressione regolare venditor[ei]*. Esso può essere quindi uguale a *venditore* o *venditori*;

```
--tag --xml_exact TITOLO --withattr --xml_exact venditori --value
--xml_prefix Barnes
```

richiede tutte le occorrenze del *tag* TITOLO con un attributo il cui nome è *venditori* e il cui valore contenga il prefisso *Barnes*;

```
--tag --xml_exact TITOLO --xml_withattr --xml_exact venditori --value
--xml_token_prefix bol
```

richiede tutte le occorrenze del *tag* TITOLO con un attributo il cui nome è *venditori* e il cui valore contenga un *token* con *prefisso bol*.

Se applicata al documento di figura 1.1 il penultimo esempio consente di individuare l'occorrenza:

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

mentre l'interrogazione dell'ultimo esempio individua in aggiunta:

```
<TITOLO venditori="bol">
```

Una sequenza di *tag-query* richiede tutte le occorrenze del *tag i+1* contenute in occorrenze del *tag i*. Ad esempio l'espressione:

```
--tag --xml_exact LIBRO --tag --xml_exact TITOLO
```

richiede tutte le occorrenze del *tag* TITOLO contenute nel *tag* LIBRO. È possibile raffinare ulteriormente le interrogazioni sul contenimento fornendo il *grado*, cioè il numero di *tag* che devono essere presenti nell'albero del documento fra i due *tag* richiesti affinché questi ultimi facciano parte della risposta. Questo si realizza facendo seguire ad una *tag-query* il *flag* `--xml_dist` seguito dal grado del contenimento rispetto al *tag* precedente. Ad esempio l'interrogazione:

```
--tag --xml_exact LIBRO --tag --xml_exact TITOLO --xml_dist 1
```

richiede tutte le occorrenze del *tag* TITOLO contenute direttamente nel *tag* LIBRO. Naturalmente non è possibile specificare la distanza del primo *tag* di un'interrogazione.
Le *word-query* consentono di effettuare interrogazioni sul dizionario delle parole. La struttura di una *word-query* è illustrata all'interno della figura 4.6.

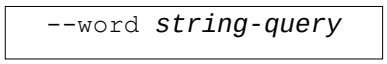



*Figura 4.6 - struttura generale di una word-query*

Una *word-query* consente di richiedere tutte le occorrenze della *parole* specificate in essa. Nel formulare un'interrogazione una sequenza di *word-query* deve essere sempre preceduta da una o più *tag-query*. In questo caso le occorrenze individuate delle *word-query* devono trovarsi all'interno della struttura descritta dalle *tag-query*. Ad esempio l'interrogazione:

```
--tag --xml_exact TITOLO --word --xml_exact campana
```

richiede tutte le occorrenze della parola *campana* all'interno del *tag* TITOLO. Applicata al documento di figura 1.1 essa ha come risposta:

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

Naturalmente le *word-query* specificate possono essere più di una. In questo caso la sequenza degli elementi non è importante, al contrario delle *tag-query*. Le due interrogazioni seguenti individuano quindi le stesse occorrenze:

```
--tag --xml_exact TITOLO --word --xml_exact campana -word --xml_exact
suona
```

```
--tag --xml_exact TITOLO --word --xml_exact suona -word --xml_exact
campana
```

È possibile realizzare interrogazioni sulla *proximity* inserendo il *flag* `--proximity` seguito da un numero maggiore di 0, che indica la *proximity massima* alla quale devono trovarsi le occorrenze delle parole individuate. Come *proximity* s'intende una forma di *distanza logica fra le parole* che consente di escludere dalla distanza gli elementi della struttura del documento e di considerare sole le parole, cioè le foglie dell'albero del documento XML.
L'interrogazione seguente:

```
--tag --xml_exact TITOLO -proximity 1 --word --xml_exact suona -word --
xml_exact campana
```

richiede tutte le occorrenze dei *token suona* e *campana* che si trovano ad una distanza logica 1 e sono contenute nel *tag* TITOLO. Applicata al documento di figura 1.1 questa interrogazione non fornirebbe risultati, mentre la stessa interrogazione con *proximity* 2 sullo stesso documento fornirebbe lo stesso risultato dell'esempio precedente.
L'implementazione delle interrogazioni avviene traducendo il linguaggio in opportune operazioni di ricerca sui dizionari e di accesso alle *liste invertite*. I valori delle liste

vengono confrontati in modo da determinare la presenza di determinate condizioni, come ad esempio l'innestamento dei *tag* ed il contenimento delle parole al loro intero. Per implementare l'interrogazione sulla distanza di innestamento dei *tag* (*--xml_dist*) si utilizza la struttura dati *tagstruct.*. Per quanto riguarda l'implementazione della *proximity* essa comporta l'accesso alla *lista delle proximity* di ogni parola presente nell'interrogazione e l'uso di un algoritmo basato sulla tecnica di *plane-sweep* **[Sadakane]**. In particolare l'algoritmo mantiene una finestra che induttivamente contiene almeno una occorrenza per ogni parola presente nell'interrogazione. Se la finestra corrente è minimale, ossia non contenuta in altre finestre già calcolate, allora essa fa parte del risultato della interrogazione a patto che la dimensione della finestra sia inferiore al valore specificato per la *proximity*. Altrimenti la finestra viene aggiornata prendendo la parola interrogata che occorre più a sinistra nella finestra corrente, e a sostituendola con la sua successiva occorrenza fornitaci dalla corrispondente *lista di proximity.* Se la nuova occorrenza ricade nella finestra corrente, allora questa non è minimale e si procede nuovamente alla cancellazione della parola più a sinistra; altrimenti siamo nel caso minimale già descritto precedentemente. L'algoritmo si ripete fino a quando non si giunge al termine di una delle liste di *proximity*. La figura 4.7 illustra un esempio di esecuzione dell'algoritmo su tre liste di *proximity* con distanza 2.


Fase 1 – la finestra individuata è [2, 4]. Si incrementa il puntatore alla la lista dell'elemento 2, che è quello più a sinistra. Il nuovo elemento puntato ha valore 9 e ricade fuori della finestra precedente. Quindi la finestra era minimale, e siccome la sua dimensione (4-2) era minore della ***proximity***, tale finestra fa parte della risposta all'interrogazione. La nuova finestra si ottiene calcolando il minimo (3) e il massimo (9) fra gli elementi puntati.

Fase 2 – la finestra è [3, 9]. Il 3 è l'elemento più a sinistra nella finestra e non dispone di occorrenze successive nella sua lista di ***proximity***, quindi l'algoritmo termina. La dimensione della finestra è 6=(9-3) che non è minore della ***proximity***. Per cui questa finestra non fa parte della risposta.

*Figura 4.7 - esempio di applicazione dell'algoritmo di plane-sweep*

L'algoritmo di *plane-sweep* viene applicato anche nel caso in cui la *proximity* non sia specificata esplicitamente dall'utente, settandola *a* $\infty$ in modo da individuare le finestre di *dimensione minima* che contengono tutte le occorrenze. Questo facilita la loro visualizzazione.

Ricapitolando, gli elementi del linguaggio per l'interrogazione sono i seguenti:

| | |
|---|---|
| `--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 per 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 il numero di ***proximity***; |
| `--xml_nocase` | specifica una ricerca ***case-insensitive***; |

È opportuno osservare come questo linguaggio sia solo un *prototipo* di un semplice linguaggio di interrogazione realizzato al fine di effettuare ricerche sulla collezione di testi letterari fornita dal *CIBIT* e per *testare* la libreria XCDE. Le funzionalità di quest'ultima non sono certamente utilizzate in maniera completa dal linguaggio, ma esso consente comunque di offrire un buon saggio della potenza della libreria. Nella figura 4.8 è visualizzata una sessione di ricerca con il motore *xcde_search*: viene realizzata un'interrogazione e la finestra risultante è passata al visualizzatore *xcde_view*.

```
$ xcde_search 001.xml --tag --xml_exact body --proximity 5 --word --xml_exact bosco --word
--xml_exact anime
001.xml 82952 82962

$ xcde_view 001.xml 82952 82962 10
<xml_snippet>
<xml_filename>001.xml</xml_filename>
<xml_text>
<TEI.2> ...
 <text> ...
  <body> ...
   <div1> ...
    <div2> ...
     <div3> ...
      <p> ...
         de gran longa megliore invescatore de
         bosco che pastore de anime umane
             ...
      </p>...
     </div3>...
    </div2>...
   </div1>...
  </body>...
 </text>...
</TEI.2>
</xml_text>
</xml_snippet>
```

*Figura 4.8 - esempio di utilizzo dei comandi xcde_search e xcde_view*

La ricerca viene effettuata sul *tag body* del documento, individuando tutte le finestre contenenti le parole *bosco* e *anime* con una *proximity* uguale o inferiore a 5. Le posizioni fisiche di inizio e fine di ogni finestra nel documento compresso sono restituite in *output* insieme al nome del *file*. Esse sono poi passate in *input* al modulo *xcde_view* che visualizza lo *snippet* contenente le parole richieste. Come è possibile osservare lo *snippet* è contenuto nel *tag* virtuale *xml_text*, a sua volta contenuto in *xml_snippet*. Esso può essere visualizzato come testo puro, come avviene nell'*XCDE Search Engine*; o può essere passato in *input* ad un'altra applicazione per un *rendering* più accattivante mediante, per esempio, l'uso di fogli di stile. Si noti che l'interrogazione in figura 4.8 è stata effettuata su un singolo documento, ma attraverso l'uso di opportuni *script* è facile ripetere una interrogazione su vari documenti.

## 4.2 L'interfaccia web

Il motore *XCDE_Search Engine* si interfaccia con l'utente attraverso una pagina *web* che consente di inserire in una *form* le interrogazioni da realizzare su parte o tutta la collezione fornita dal *CIBIT*.
L'interfaccia può essere divisa logicamente in tre parti. La prima parte, illustrata nella figura 4.9, consente di selezionare su quali documenti della collezione *CIBIT* effettuare l'interrogazione.

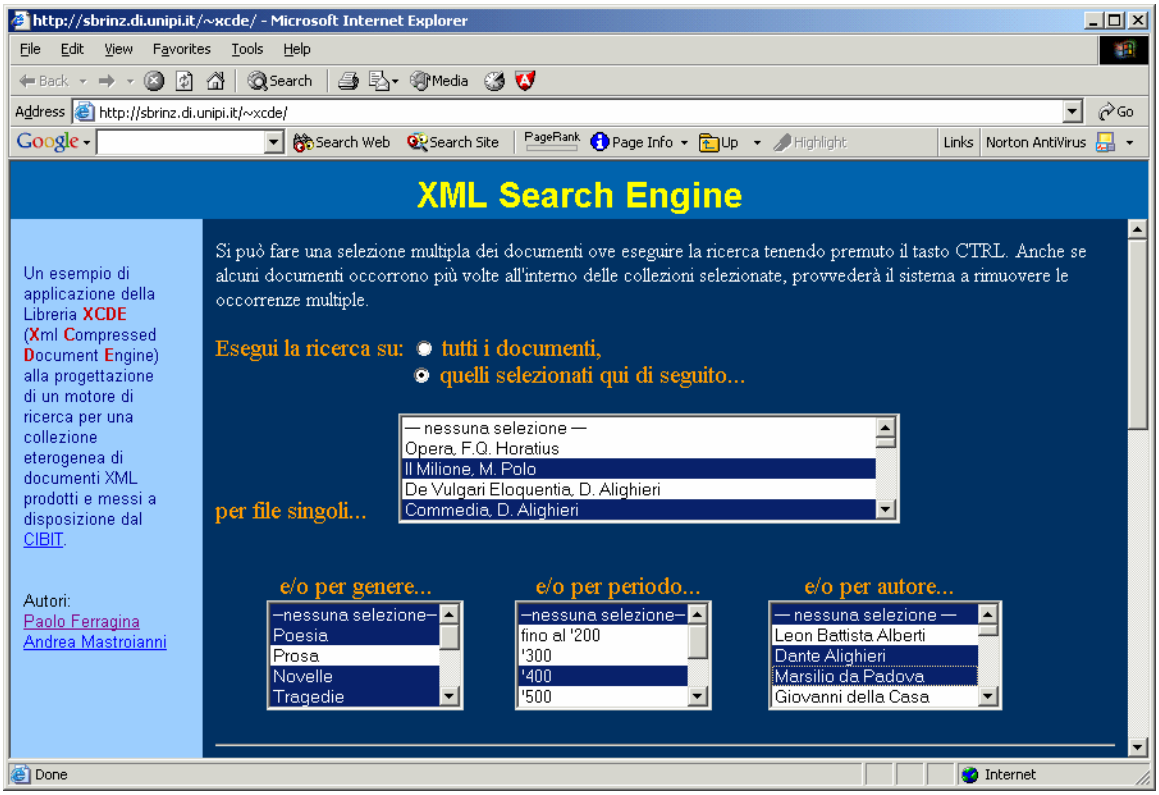



*Figura 4.9 - form per selezionare su quali documenti della collezione CIBIT effettuare l'interrogazione*

Come possiamo osservare è possibile selezionare tutti i documenti della collezione, oppure si possono scegliere singoli documenti o sottocollezioni suddivise per: *genere*, *periodo* e *autore*. Questa suddivisione è stata suggerita dagli utenti del *CIBIT*.
Scorrendo la pagina si incontra la seconda parte dell'interfaccia, visualizzata nella figura 4.10. Attraverso questa possiamo inserire una ricerca strutturale su un massimo di tre *tag*. I primi due possono essere scelti da una lista che rispecchia la DTD di XML-TEI. Fissato ciascuno di questi *tag* l'interfaccia procede a creare dinamicamente i loro possibili attributi e valori, sempre in accordo a quanto specificato nella DTD di XML-TEI. Questo aggiornamento dinamico avviene attraverso dei semplici *script javascript*. Il terzo *tag* può essere invece liberamente specificato dall'utente, insieme al nome e/o al valore di un attributo e al tipo di ricerca da effettuarsi su ognuno di essi. Nel momento in cui esso è specificato esso deve intendersi come innestato negli altri due secondo l'ordine con cui

figurano nella pagina.


*Figura 4.10 - form per la specifica delle interrogazioni strutturali*


*Figura 4.11 - form per la specifica delle interrogazioni sulle parole*

La terza, e ultima, parte dell'interfaccia consente di formulare le interrogazioni sulle parole. Come è possibile osservare in figura 4.11 si possono inserire un massimo di tre parole sulle quali può essere specificata la *proximity*. Tutte le parole si intendono incluse negli eventuali *tag* specificati nella seconda parte della pagina *web*. Per ogni parola è necessario specificare, attraverso la tendina associata, il tipo di ricerca desiderato: parola, sottostringa, prefisso, suffisso, 1 errore, 2 errori, espressione regolare. Esiste infine un campo con il quale si può ulteriormente restringere l'ambito della ricerca: corpo del documento, *header* TEI, o tutto il *file* XML.
Una volta che l'interrogazione è stata specificata in tutte le sue parti si può dar corso alla sua esecuzione attraverso il pulsante *submit query*.

Vediamo ora un esempio di interrogazione e di visualizzazione dei risultati. L'interrogazione è effettuata su *tutta la collezione di documenti CIBIT*. Specifichiamo la struttura dei *tag* come all'interno della figura 4.10, cioè un *tag body* (che indica il *corpo* del documento nella DTD TEI) che contiene un *tag foreign*, che indica una citazione in una lingua straniera, con un attributo *lang*, che indica la lingua della citazione, il cui valore è *latino*.

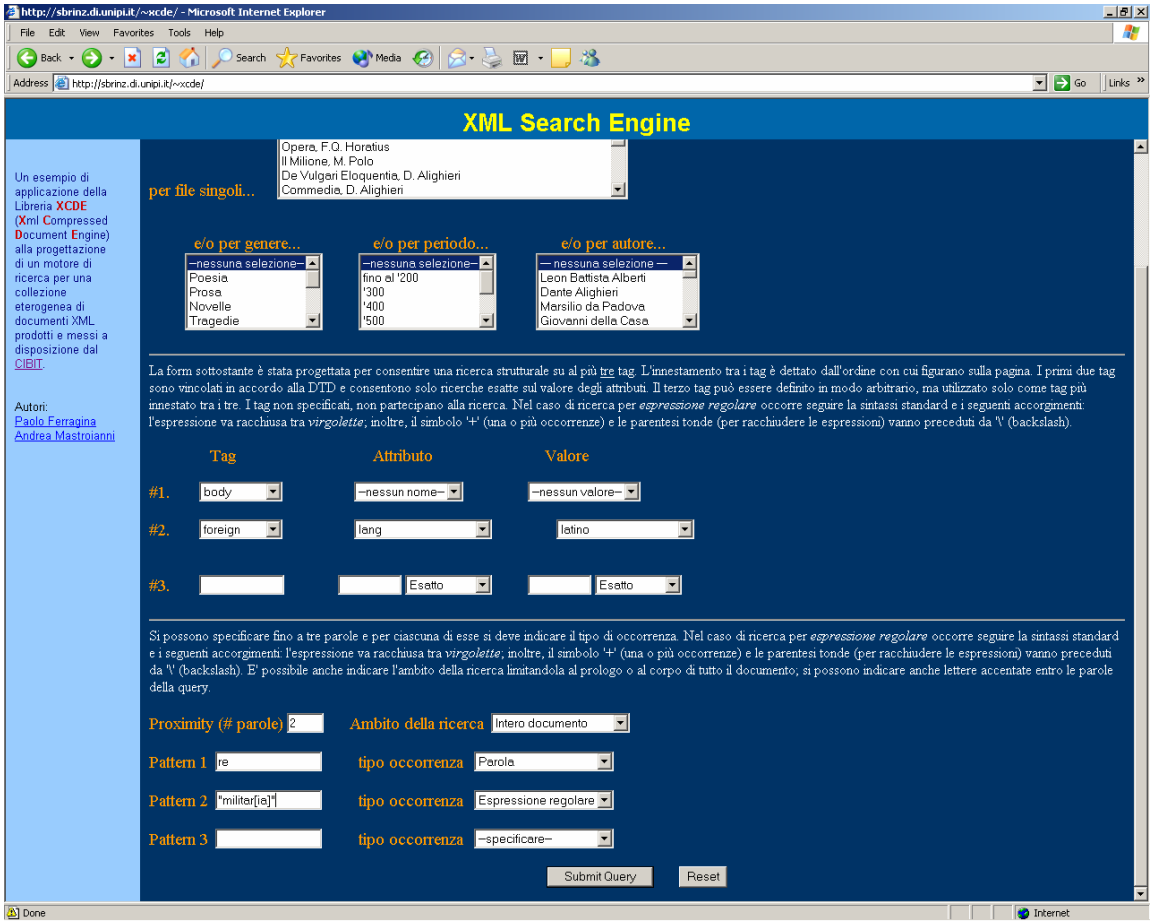



*Figura 4.12 - esempio di un'interrogazione*

La *proximity* applicata sulle parole specificate è *2*. Esse sono *res*, che è cercata in modo *esatto*, e l'*espressione regolare militar[ie]* indica che si cercano i *token militari* oppure *militare*.

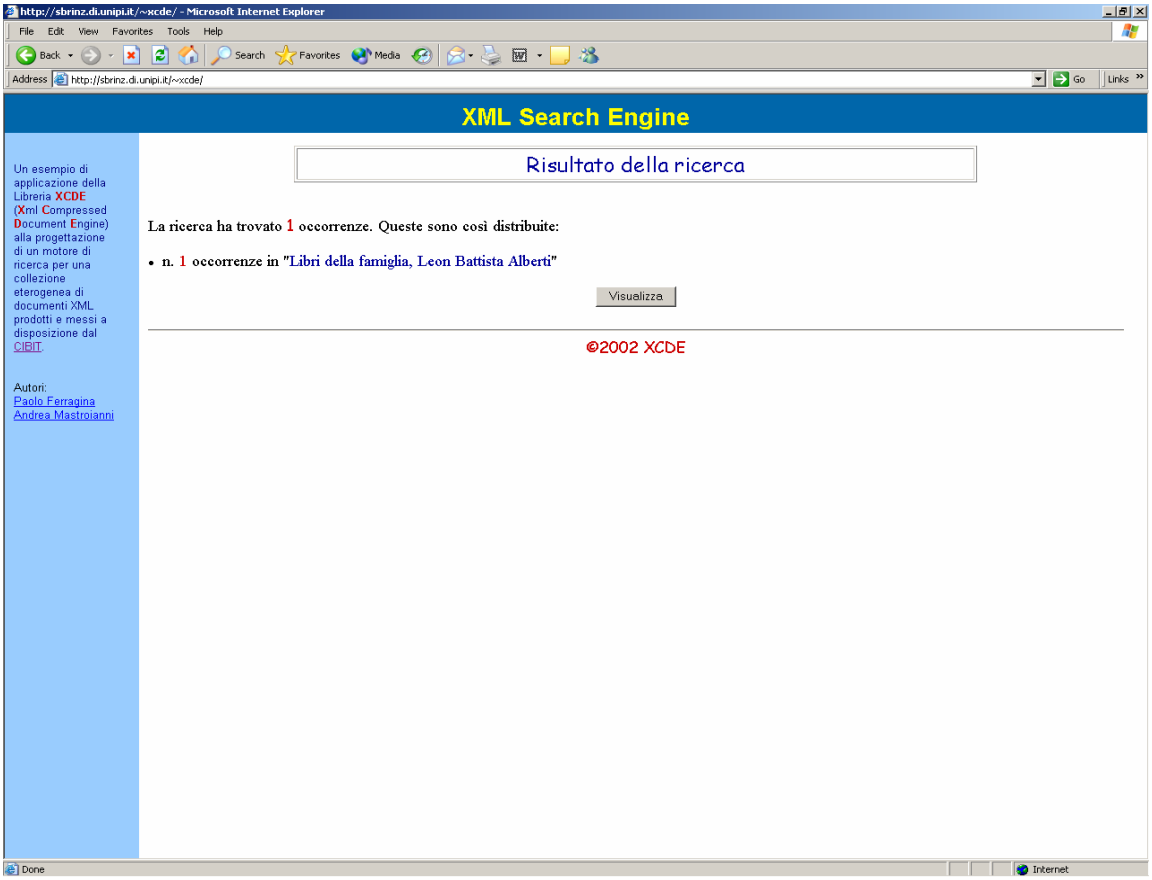



*Figura 4.13 - esempio di visualizzazione dei risultati dell'interrogazione*

La figura 4.13 mostra l'esistenza di una sola occorrenza in un solo documento della collezione, il *Libro della Famiglia* di *Leon Battista Alberti*. Visualizzando l'occorrenza tramite l'apposito pulsante si ottiene la schermata contenuta nella figura 4.14. Si tratta di una delle funzionalità più innovative del motore: l'estrazione di *snippets ben formati*. È possibile da parte dell'utente anche aumentare o diminuire la dimensione della finestra di visualizzazione dell'occorrenza attraverso gli appositi pulsanti presenti in fondo alla pagina. Questa operazione comporta l'esecuzione del comando *xcde*_view.

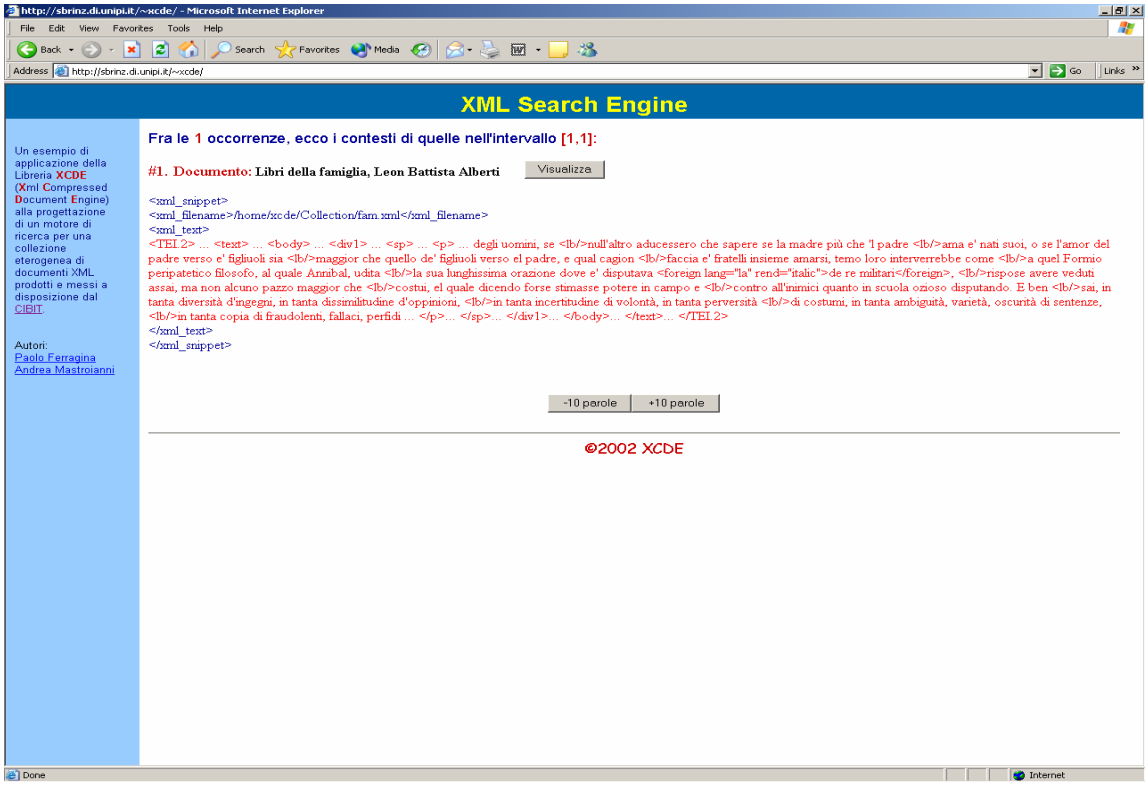



*Figura 4.14 - esempio di visualizzazione di uno snippet*

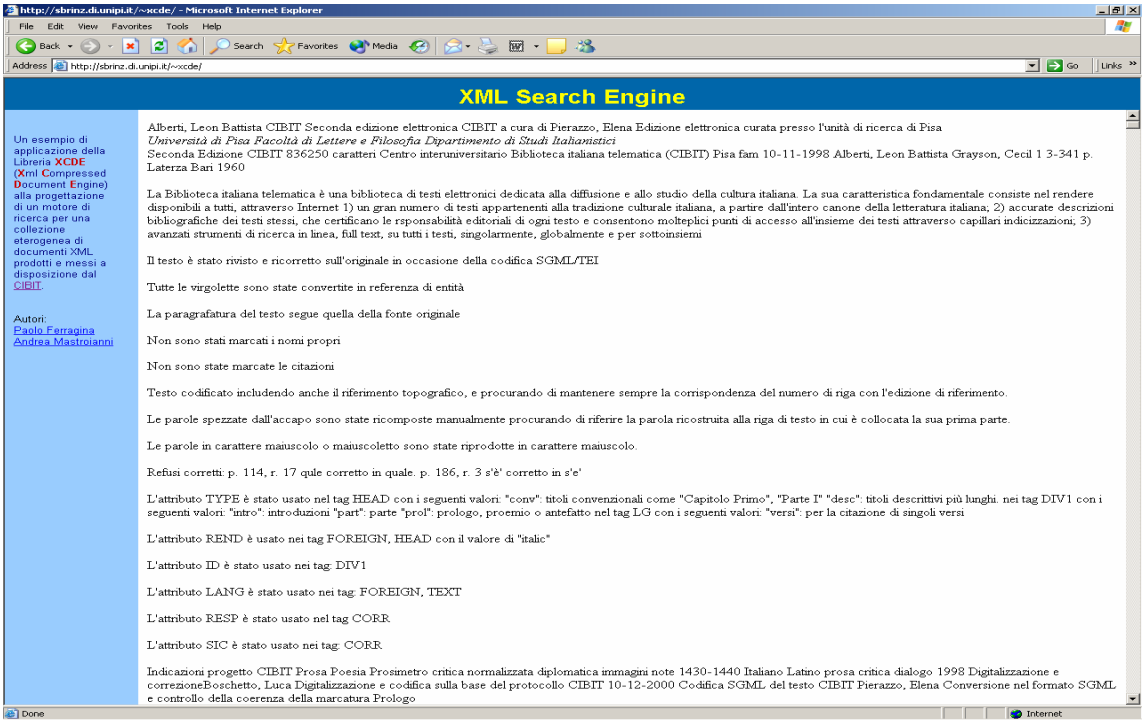



*Figura 4.15 - esempio di visualizzazione di un documento decompresso*

Nel grafico di figura 4.12 si indicano i tempi medi richiesti dal motore per rispondere ad alcune interrogazioni simili a quelle di figura 4.11, ossia consistenti di due *tag* e due *parole*. Per effettuare la ricerca sull'intera collezione *CIBIT,* formata da 30 documenti, si impiega circa 0.8 *sec*. Di questi circa 0.1 *sec* sono necessari per l'apertura e la chiusura dei *file;* un tempo simile è richiesto per accedere alle *liste invertite* e per eseguire le operazioni su di esse; il tempo restante è utilizzato dalle operazioni di decompressione e ricerca sui dizionari.

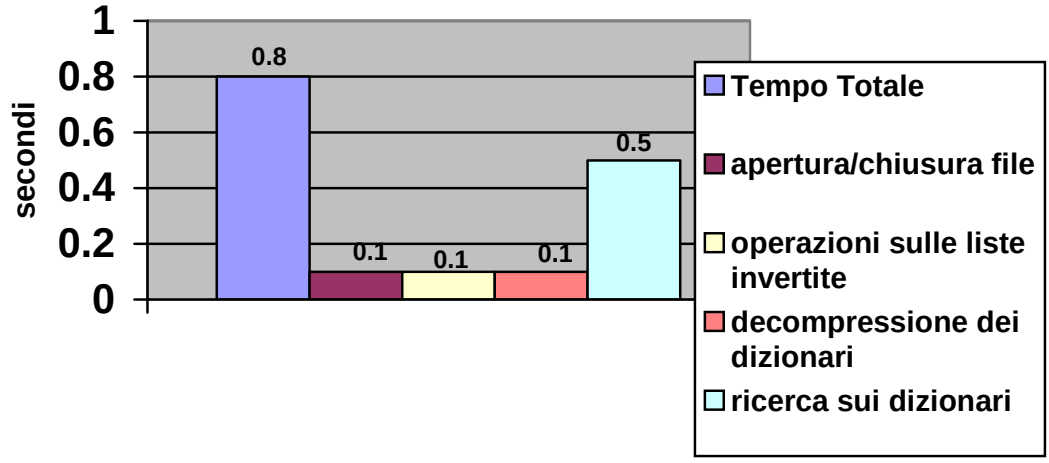



*Figura 4.16 - tempo medio di esecuzione delle interrogazioni*

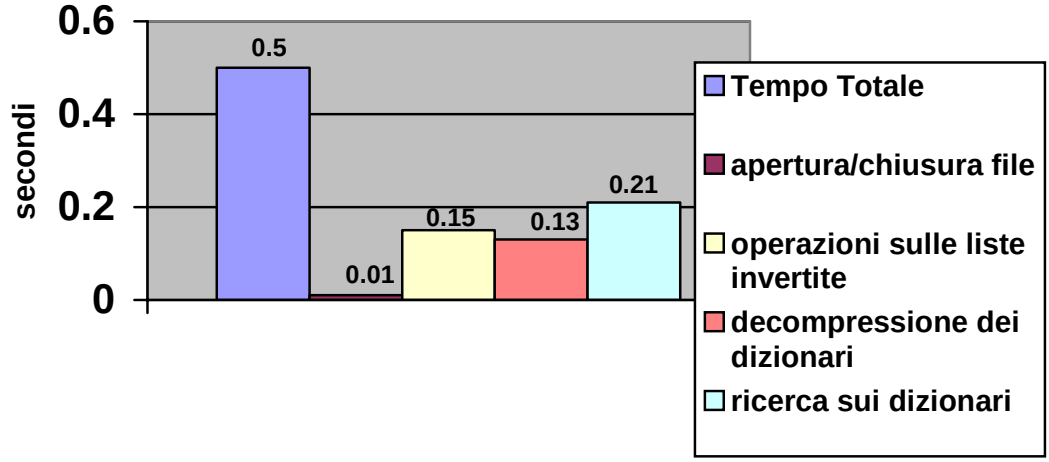



*Figura 4.17 - tempo medio di esecuzione delle interrogazioni su un documento virtuale contenente tutti i documenti della collezione CIBIT*

L'istogramma della figura 4.17 rappresenta il tempo medio di esecuzione di

un’interrogazione per una collezione formata unendo tutti i documenti della collezione *CIBIT* in un unico documento virtuale il quale, mediante opportuni *tag*, preserva l’individualità dei singoli documenti costituenti. In questo modo è possibile rendere più efficiente i tempi di accesso, dovendo accedere ad un unico *file*. Gli accessi ai dizionari sono più efficienti, considerato che si diminuisce il numero delle chiamate *system* effettuate per le ricerche, anche se aumentano le dimensioni in spazio dei singoli dizionari e quindi i tempi di decompressione. L’aumento, considerata la velocità dell’algoritmo *Gzip* **[gzip]**, è comunque di lieve entità. Diversi miglioramenti sono possibili e facilmente ottenibili in una fase di ingegnerizzazione del codice presente. Alcuni di questi sono discussi diffusamente nella sezione che segue.

# Capitolo 5

# Conclusioni

La libreria XCDE rappresenta uno strumento innovativo per l'indicizzazione e l'interrogazione dei documenti XML. Una delle sue caratteristiche principali è data dalla ridotta occupazione in spazio: la dimensione di tutte le strutture dati utilizzate è al più 1.2 volte quella del documento sorgente. Risultato questo particolarmente interessante considerando che nei sistemi concorrenti le sole dimensioni degli indici, come dichiarato dagli stessi progettisti, risultano da 3 a 10 volte più grandi del documento sorgente. La libreria XCDE utilizza inoltre alcune strutture dati (come la *tagstruct*) 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 sistema. La *modularità* e la *flessibilità* della libreria consentono di delineare un progetto *aperto* a ulteriori aggiornamenti, sviluppi, modifiche e integrazioni senza che questo, grazie all'interfaccia API realizzata, abbia eccessive conseguenze sulle eventuali applicazioni che utilizzano la libreria. Nel prosieguo illustreremo alcuni spunti di ricerca e sviluppo che meritano una maggiore attenzione per il futuro di questa libreria.
Una delle modifiche auspicabili consiste nella rimozione delle chiamate *system* all'interno del codice. I comandi *grep* e *agrep* utilizzati per le ricerche sui dizionari potrebbero essere sostituiti da funzioni di libreria aventi le stesse funzionalità ma operanti direttamente sui file compressi senza passare dalla loro completa decompressione **[Navarro00][Karkkainen]**. Quest'operazione consentirebbe di eliminare l'*overhead* dovuto alle chiamate di sistema, che è stato stimato in circa 20000μsec per ognuna di esse, velocizzando così il tempo complessivo di risposta alle interrogazioni. Un'ulteriore miglioramento potrebbe essere ottenuto sostituendo l'algoritmo di compressione *gzip* applicato sui dizionari, con indici compressi quali l'*FM-index* **[Ferragina01].** Questo indice occuperebbe uno spazio prossimo a quello dei migliori compressori esistenti, ma consentirebbe inoltre di effettuare ricerche esatte con complessità in tempo $O(p+occ \log^{\varepsilon}$

n), dove p è la lunghezza del pattern cercato, n è la lunghezza del testo e ε è una costante positiva fissata arbitrariamente dall'utente. In questo ambito sarebbe interessante, anche dal punto di vista teorico, estendere le funzionalità dell'*FM-index* all'esecuzione di ricerche complesse, quali quelle con errori o per espressioni regolari.
Per quanto riguarda le interrogazioni si potrebbe prevedere la costruzione di un linguaggio tipo XQuery **[XQuery]**, che si andrebbe a sostituire alla semplice interfaccia illustrata nel capitolo 4 e utilizzata nell'*XCDE Search Engine*. A questo proposito si potrebbero realizzare anche delle *ottimizzazioni* nell'implementazione delle *query* utilizzando opportuni *valutatori della selettività* delle *query* XML **[Aboulnaga]** che consentirebbero di formulare le *query* più opportune per l'implementazione di determinati costrutti del linguagigo.
Anche l'*XCDE Search Engine* potrebbe essere migliorato ulteriormente. Si potrebbe prevedere di arricchire l'interfaccia *web* così da consentire all'utente di formulare *query* sui *tag* in modo guidato attraverso opportune interfaccie grafiche relative alla *DTD*. In questo modo non si avrebbe la necessità di conoscere la DTD di XML-TEI 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.*
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 XCDE. 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 ridurre l'impatto che la granularità ha sulle prestazione dell'intero sistema. Rimandiamo a questo proposito alle prove effettuate e sintetizzate nel grafico di figura 4.17 nel precedente capitolo. Esso dimostra come con questo approccio l'efficienza nelle ricerche migliori. Inoltre l'uso di un unico documento virtuale consentirebbe di gestire in modo semplice e veloce le 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 appieno la flessibilità dell'XML per definire il concetto di *blocchi di indicizzazione* in modo indipendente dal sistema di ricerca.

# Bibliografia


[Aboulnaga] A. Aboulnaga, A. R. Alameldeen, J. F. Naughton : *Estimating the selectivity of XML path expression for Internet scale application*. In Proceedings the 27th VLDB Conference, VLDB'01, pp. 591-600, Morgan Kaufmann Publishers, 2001.

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

[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.

[Barbosa] D. Barbosa, A. Barta, A. Mendelzon, G. Mihaila, F. Rizzolo, P. Rodriguez Gianolli: *ToX – The Toronto XML Engine*. In Proceedings of the Workshop on Information Integration on the Web, pp. 66-73, 2000.

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

[Chazelle] B. Chazelle: *Filtering search: a new approach to query-answering.* SIAM Journal of Computing, vol. 15, n.3, 1986.

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

[Cooper] B. F. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, M.Shadmon: *A Fast Index for Semistructured Data.*In Proceedings of the 27th VLDB Conference, VLDB'01, pp. 341-350, Morgan Kaufmann Publishers, 2001.

[Corti] F. Corti, P. Ferragina, M.Paoli: *TReSy: un motore di ricerca testuale per documenti XML/SGML*. Convegno XML Italia, 2000.

[CSS1] 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.

[CSS2] 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.

[DOM] http://www.w3.org/TR/REC-DOM-Level-1
Definizione del Document Object Model, livello1, W3C Recommendation, 1 october, 1998
Sito della World Wide Web Organization.

[Deutsch] A. Deutsch, M. Fernandez, D. Suciu: S*toring Semistructured Data With STORED*. In Proceedings of the ACM SIGMOD International Conference of Management of Data: SIGMOD'99. SIGMOD Record, vol. 28, pp.431-442, ACM Press, 1999.

[Ferragina01] P. Ferragina, G. Manzini: *An experimental study of an opportunistic index*. In Proceedings of the ACM-SIAM Symposium on Discrete Algorithms, pp. 269-278, 2001.

[Ferragina99] P. Ferragina, R. Grossi: *The String B-Tree: A new data structure for string search in external memory and its applications*. Journal of the ACM, vol. 46, pp. 236-280, 1999.

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

[Gzip] http://www.gzip.org
Pagina ufficiale di Gzip.

[Harman] D. Harman: *Overview of the Third Text REtrieval Conference (TREC-3)*. In Proceedings of the 3rd Text REtrieval Conference (TREC-3), pp. 305-310, NIST 1994.

[Heaps] J. Heaps: *Information Retrieval – Computational and Theoretical Aspects*. Academic Press, New York, 1978.

[Huffman] D. A. Huffman: *A method for the construction of minimum-redundancy codes*. In Proceedings Inst. Radio Engineers, vol. 40, pp. 1098-1101, 1952.

[Kanne] C. Kanne, G. Moerkotte: *Efficient storage of XML data* In Proceedings of the 16th IEEE Conference on Data Engineering, ICDE'00, pp. 198-200, IEEE, 2000.

[Karkkainen] J. Karkkainen, G. Navarro, E. Ukkonen: *Approximate String Matching over Ziv-Lempel Compressed Text*. In Proceedings the 11th Annual Symposium on Combinatorial Pattern Matching, pp. 195-209, 2000.

[Liefke] H. Liefke, D. Suciu: *Xmill: An Efficient Compressor for XML Data*. In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data: SIGMOD'00. SIGMOD Record, vol 29, pp. 153-164, ACM Press, 2000.

[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.

[McHugh] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, J.Widom: *Lore – A database management system for semistructured data.* SIGMOD Record, vol. 26, pp. 54-70, 1997.

[Navarro00] E.Silva de Moura, G. Navarro, N. Ziviani, R.Baeza-Yates: *Fast and flexible word searching on compressed text.* ACM Transactions on Information Systems, vol. 18, pp. 113-139, 2000.

[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.

[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.

[Shin] D. Shin, H. Jang, H. Jin: *BUS - An effective Indexing and Retrieval Scheme in Structured Documents.* In Proceedings of the 3rd ACM Conference on Digital Libraries: DL'98, pp. 235-243,1998.

[Tamino] http://www.tamino.com
Sito ufficiale di Tamino, Software AG.

[Witten] I. H. Witten, A. Moffat, T. C. Bell
*Managing Gigabytes – Compressing and Indexing Documents and Images.*
Morgan Kaufmann Publishers, Los Altos, CA 94022, USA,
second edition 1999.

[Wu] S. Wu, U. Manber: *AGREP - A Fast Approximate Pattern-matching Tool.* In Proceedings of the Winter 1992 USENIX Conference, pp. 153-162, 1992.

[Xlink] http://www.w3.org/XML/Linking
Pagina ufficiale di XLink e Xpointer
Sito della World Wide Web Organization.

[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.

[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.

[XQuery] 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.

[XSchema] http://www.w3.org/XML/Schema
Pagina ufficiale delle risorse su XSchema
Sito della World Wide Web Organization.

[XSL] http://www.w3.org/TR/WD-xsl
Specifica ufficiale dell'eXtensible Stylesheet Language Version 1.0, W3C Recommendation, 15 october, 2001
Sito della World Wide Web Organization.

[XYZ-Find] http://www.xyzfind.com
Sito ufficiale di XYZ-Find Corporation.

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

[Zhao] Ben Y. Zhao: *The Xset XML search engine and XBench XML query benchmark*. Technical Report UCB/CSD-00-1112, University of California, Berkeley, 2000.

[Zlib] http://www.gzip.org/zlib
Home page della libreria Zlib ver. 1.1.4, Greg Roelofs, Jean-Loup Gailly.

# Indice delle figure

# APPENDICE A – L'API di XCDE

La libreria XCDE possiede un'API formata da 42 funzioni C. Essa richiedono l'inclusione del file d'intestazione *xcde_lib.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* attraverso il tipo *XCDE_Console_Type.* La *console* è passata per *riferimento* a qualsiasi funzione della libreria e contiene tutte le informazioni necessarie ad operare sui dati. Altri tipi definiti sono *XCDE_IL_Couple_Type* e *XCDE_IL_Triple_Type* per la lettura delle liste delle posizioni e di *proximity*.

Il tipo *XCDE_IL_Couple_Type* è definito come:

```
typedef
struct XCDE_IL_Couple_Struct_Type
{
  int XCDE_Position ;

  int XCDE_Offset ;
} XCDE_IL_Couple_Type ;
```

dove `XCDE_Position` rappresenta la posizione dell'elemento e `XCDE_Offset` il suo *displacement*.

Il tipo *XCDE_IL_Triple_Type* è definito come:

```
typedef
struct XCDE_IL_Triple_Struct_Type
{
  int XCDE_Position ;

  int XCDE_Offset ;

  int XCDE_Proximity ;

} XCDE_IL_Triple_Type ;
```

dove naturalmente i primi due elementi sono uguali, e `XCDE_Proximity` rappresenta la *proximity* dell'elemento.

Le liste delle posizioni hanno tipo **XCDE_IL_Couple_Type*. Il tipo **XCDE_IL_Triple_Type* contiene sia le informazioni della lista delle posizioni che quelle relative alla *proximity*. Esso può essere utilizzato solo nel caso in cui il *token* la cui lista è richiesta sia una parola.

Per quanto riguarda le costanti definite, esse sono utilizzate per indicano i dizionari, il tipo

di ricerca e se quest'ultima è *case-sensitive* o meno.
Le costanti per indicare i dizionari sono:

| | |
|---|---|
| `TAG` | dizionario dei *tag*; |
| `ATTR` | dizionario degli attributi; |
| `WORD` | dizionario delle parole; |

Quelle per la ricerca sono:

| | |
|---|---|
| `XCDE_EXACT` | ricerca *esatta*; |
| `XCDE_PREFIX` | ricerca per *prefisso*; |
| `XCDE_SUFFIX` | ricerca per *suffisso*; |
| `XCDE_ERROR` | ricerca con *errori*; |
| `XCDE_CONTAINED` | ricerca per *sottostringa*; |
| `XCDE_REGEXP` | ricerca per *espressione regolare*; |
| `XCDE_ANYVALUE` | ricerca con ogni valore (solo per il nome di un attributo); |
| `XCDE_TOKEN_EXACT` | ricerca esatta su un *token* (solo per il valore di un attributo); |
| `XCDE_TOKEN_PREFIX` | ricerca esatta per prefisso su un *token* (solo per il valore di un attributo); |
| `XCDE_TOKEN_SUFFIX` | ricerca esatta per suffisso su un *token* (solo per il valore di un attributo); |
| `XCDE_CASE` | ricerca *case-sensitive*; |
| `XCDE_NOCASE` | ricerca *case-insensitive*; |

Queste costanti sono utilizzate come parametro per determinare il comportamento di alcune funzioni.

Esistono dei tipi generali: i *ranghi* dei *token* sono sempre di tipo *int*, così come le stringhe di tipo *char* * e i *flag* di tipo *unsigned char*. Alle funzioni restituiscono gli eventuali risultati in variabili passate per riferimento.

Ogni programma che utilizzi la libreria XCDE deve rispettare la seguente struttura del codice:

```
// file di intestazione
#include ”xcde_lib.h”
```

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

   // inizializza la console. La variabile console è passata per
   // riferimento
   result=XCDE_Init_Console(argv[1], &console) ;

   // gestisce l'eventuale condizione di errore
   if (result == -1)

      { printf("\nErrore !") ;

         exit(-1) ;
      }

      .
      .
      .
   // distrugge la console, che a questo punto non può più essere
   // utilizzata se non inizializzata un'altra volta prima
   result=XCDE_Finalize_Console(&console) ;

   if (result == -1)

      { printf("\nErrore !") ;

         exit(-1) ;
      }

return(1) ;
}
```

Tutte le funzioni della libreria restituiscono sempre il valore –1 in caso di errore e 1 altrimenti. Esse prendono in *input* la variabile *console* passata per riferimento. 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 delle *condizioni generali*, cioè delle cause di errore comune a tutte le funzioni. Ad esempio se la variabile *console* passata ad una generica funzione non è stata inizializzata (a meno, naturalmente, della *XCDE_Init_Console* per cui è vero il contrario) o se al posto del riferimento di una variabile viene inserito il valore *NULL*. Ad esempio, nell'esempio precedente la chiamata:

```
result=XCDE_Init_Console(NULL) ;
```

restituirebbe il codice d'errore –1, così come in:

```
result=XCDE_Init_Console(&Console) ;
```

```
result=XCDE_Init_Console(&Console) ;
```

la seconda chiamata restituisce il codice di errore -1, visto che si tenta di inizializzare due volte la stessa funzione. In caso di errore una funzione non ha *effetti collaterali*. Nell'esempio precedente la seconda chiamata, avendo esito negativo, non modifica in nessun modo la variabile *console*, permettendo quindi eventualmente al programma di proseguire senza problemi.

Elenchiamo di seguito, insieme ad tutte le funzioni della libreria, seguendo la suddivisione utilizzata nella sezione 3.2.

| | |
|---|---|
| **Funzioni per la gestione della console** | API 1 – XCDE_Init_Console<br>API 2 – XCDE_Finalize_Console<br>API 3 – XCDE_Console_Initialized |
| **Funzioni per l'accesso alle informazioni sui *file*** | API 4 – XCDE_Get_Source_Filesize<br>API 5 – XCDE_Get_Source_Filename<br>API 6 – XCDE_Get_Header_Filesize<br>API 7 – XCDE_Get_Header_Filename<br>API 8 – XCDE_Get_Body_Filesize<br>API 9 – XCDE_Get_Body_Filename<br>API 10 – XCDE_Get_IL_Filesize<br>API 11 – XCDE_Get_IL_Filename<br>API 12 – XCDE_Get_Dict_Cmp_Filesize<br>API 13 – XCDE_Get_Dict_Cmp_Filename<br>API 14 – XCDE_Get_Dict_Dcmptmp_Filesize<br>API 15 – XCDE_Get_Dict_Dcmptmp_Filename<br>API 16 – XCDE_Get_Tagstruct_Filesize<br>API 17 – XCDE_Get_Tagstruct_Filename |
| **Funzioni per operare sui dizionari** | API 18 – XCDE_Decompress_Dictionary<br>API 19 – XCDE_Load_Dictionary<br>API 20 – XCDE_Get_Num_El_Dict<br>API 21 – XCDE_Get_Num_El_Dictlev<br>API 22 – XCDE_DictEl2Lev<br>API 23 – XCDE_Search_Tag_Dictionary<br>API 24 – XCDE_Search_Attr_Dictionary<br>API 25 – XCDE_Search_Word_Dictionary |
| **Funzioni per operare sulle liste invertite** | API 26 – XCDE_Get_IL_ListPos<br>API 27 – XCDE_Get_IL_ListPosProx<br>API 28 – XCDE_Setted_Proximity |
| **Funzioni per operare sul documento compresso** | API 29 – XCDE_Get_Body_Forward_Codeword<br>API 30 – XCDE_Get_Body_Backward_Codeword<br>API 31 – XCDE_Get_Body_TextWindow<br>API 32 – XCDE_Get_Body_Eof |
| **Funzioni per operare sulla tagstruct** | API 33 – XCDE_Get_Tagstruct_Tags_Enclosing<br>API 34 – XCDE_Get_Setted_Tagstruct |
| **Funzioni per la traduzione degli identificatori** | API 35 – XCDE_Dict2Rank<br>API 36 – XCDE_Rank2Dict<br>API 37 – XCDE_Codeword2Rank<br>API 38 – XCDE_Codeword2Token<br>API 39 – XCDE_Rank2Token |
| **Funzioni varie** | API 40 – XCDE_Get_Num_Tokens<br>API 41 – XCDE_Get_Maxcwlen<br>API 42 – XCDE_Get_Num_El_Lev |

## Funzioni per la gestione della console

Di questo gruppo fanno parte le funzioni che permettono di inizializzare e finalizzare la *console* e di determinare se essa sia stata inizializzata o meno.

### API NR.1 - *XCDE_Init_Console*

```
// IN name_source: nome del file sorgente;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Init_Console(const char *name_source,
                      XCDE_Console_Type *XCDE_Console);
```

Inizializza la *console*. Prende come parametri il nome del *file* sorgente (non necessariamente presente nel *file system*) e il puntatore alla *console*. Associa il documento alla *console*, legge il file *nomedocumento.xml.head* con le informazioni ausiliarie e le memorizza all'interno della *console* (ad esempio carica gli indici per l'accesso al *documento compresso* e alle *liste invertite*). Il *flag* di inizializzazione del documento è posto al valore 1. Restituisce il valore -1 nei casi generali già visti precedentemente o quando uno o più *file* necessari per inizializzare la *console* non sono presenti nel *file system*.

### API NR.2 - *XCDE_Finalize_Console*

```
// IN XCDE_Console: indirizzo della struttura console

int XCDE_Finalize_Console(XCDE_Console_Type *XCDE_Console);
```

Distrugge la *console*. Prende come parametro il solo riferimento alla variabile *console* e chiude tutti i *file* aperti, elimina gli eventuali *file* temporanei creati per la decompressione dei dizionari, libera la memoria allocata e pone il *flag* di inizializzazione a 0. Restituisce il valore -1 quando si cerca di distruggere una variabile console non inizializzata.

### API NR.3 - *XCDE_ConsoleInitialized*

```
// OUT InitializeFlag: indirizzo del flag

// IN XCDE_Console: indirizzo della struttura console;

int XCDE_ConsoleInitialized(unsigned char *InitializeFlag,
                            XCDE_Console_Type *XCDE_Console);
```

Scrive in una variabile *flag* il cui indirizzo è passato come parametro il valore 1 o 0 a seconda che la *console*, il cui indirizzo è passato anch'esso come parametro, sia inizializzata oppure no.

## Funzioni per l'accesso alle informazioni sui *file*

Di questo gruppo fanno parte le funzioni che consentono di accedere alle informazioni sui *file* in cui sono memorizzate le strutture dati. In generale queste informazioni sono il nome del *file* e le sue dimensioni. L'interfaccia di tutte le funzioni del gruppo è simile: oltre alla *console*, a quelle che leggono il nome del *file* si passa un puntatore ad una stringa, su cui si copierà il nome del *file* richiesto. La funzione si occuperà di allocare la memoria per il nome, che dovrà poi essere liberata dall'utente. Per i dizionari abbiamo due gruppi di funzioni: il primo riguarda quelli compressi e il secondo quelli decompressi, che sono memorizzati in *file* temporanei. Esse prendono come parametro una costante di tipo dizionario (*WORD, TAG, ATTR*); nel caso in cui si utilizzi una delle funzioni per i dizionari decompressi ed il dizionario corrispondente non sia stato decompresso esse restituiscono il valore -1. Lo stesso vale per le funzioni relative alla *tagstruct*, che hanno esito positivo solo se quest'ultima è effettivamente presente (ricordiamo che non è necessario creare la *tagstruct*. La possibilità di escluderla dall'operazione di creazione è a discrezione dell'utente). Esistono infine delle funzioni riguardanti il *file* contenente il documento sorgente *.xml. Esse naturalmente non richiedono la presenza di questo *file*, ma acquisiscono le informazioni necessarie dalla struttura per le informazioni ausiliarie.

### API NR.4 – *XCDE_Get_Source_Filesize*

```
// OUT Source_Filesize: indirizzo della variabile che conterrà le
// dimensioni del file sorgente;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Get_Source_Filesize(int *Source_Filesize,
                             XCDE_Console_Type *XCDE_Console);
```

Scrive in una variabile, il cui indirizzo è passato come parametro, le dimensioni in *byte* del *file* che contiene il documento sorgente associato alla *console*.

### API NR.5 – *XCDE_Get_Source_Filename*

```
// OUT Source_Filename: indirizzo della variabile che conterrà il nome
// del file sorgente;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Get_Source_Filename(char **Source_Filename,
                             XCDE_Console_Type *XCDE_Console);
```

Scrive in una variabile, il cui indirizzo è passato come parametro, il nome del *file* sorgente.

### API NR. 6 – *XCDE_Get_Header_Filesize*

```
// OUT Header_Filesize: indirizzo della variabile che conterrà le
// dimensioni del file header (informazioni ausiliarie);

// IN XCDE_Console: indirizzo della struttura console
```

```
int XCDE_Get_Header_Filesize(int *Header_Filesize,
                             XCDE_Console_Type *XCDE_Console);
```

Scrive in una variabile, il cui indirizzo è passato come parametro, le dimensioni in *byte* del *file* che contiene le informazioni ausiliarie.

**API NR.7 – *XCDE_Get_Header_Filename***

```
// OUT Header_Filename: indirizzo della variabile che conterrà il nome
// del file header (informazioni ausiliarie);

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Get_Header_Filename(char **Header_Filename,
                             XCDE_Console_Type *XCDE_Console);
```

Scrive in una variabile, il cui indirizzo è passato come parametro, il nome del *file* che contiene le informazioni ausiliarie.

**API NR.8 – *XCDE_Get_Body_Filesize***

```
// OUT Body_Filesize: indirizzo della variabile che conterrà le
// dimensioni del file body (documento compresso);

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Get_Body_Filesize(int *Body_Filesize,
                           XCDE_Console_Type *XCDE_Console);
```

Scrive in una variabile, il cui indirizzo è passato come parametro, le dimensioni in *byte* del *file* che contiene il documento compresso.

**API NR.9 – *XCDE_Get_Body_Filename***

```
// OUT Body_Filename: indirizzo della variabile che conterrà il nome
// del file Body (documento compresso);

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Get_Body_Filename(char **Body_Filename,
                           XCDE_Console_Type *XCDE_Console);
```

Scrive in una variabile, il cui indirizzo è passato come parametro, il nome del *file* che contiene il documento compresso.

**API NR.10 – *XCDE_Get_IL_Filesize***

```
// OUT IL_Filesize: indirizzo della variabile che conterrà le dimensioni
// del file IL (liste invertite);

// IN XCDE_Console: indirizzo della struttura console
```

```
int XCDE_Get_IL_Filesize(int *IL_Filesize,
                         XCDE_Console_Type *XCDE_Console);
```

Scrive in una variabile, il cui indirizzo è passato come parametro, le dimensioni in *byte* del *file* che contiene le *liste invertite*.

**API NR.11 – *XCDE_Get_IL_Filename***

```
// OUT IL_Filename: indirizzo della variabile che conterrà il nome del
// file IL (liste invertite);

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Get_IL_Filename(char **IL_Filename,
                         XCDE_Console_Type *XCDE_Console);
```

Scrive in una variabile, il cui indirizzo è passato come parametro, il nome del *file* che contiene le liste invertite.

**API NR.12 – *XCDE_Get_Dict_Cmp_Filesize***

```
// IN Dict: costante che indica il dizionario (TAG, WORD, ATTR);

// OUT Dict_Cmp_Filesize: indirizzo della variabile che conterrà le
// dimensioni del file dizionario compresso;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Get_Dict_Cmp_Filesize(unsigned char Dict,
                               int *Dict_Cmp_Filesize,
                               XCDE_Console_Type *XCDE_Console);
```

Scrive in una variabile, il cui indirizzo è passato come parametro, le dimensioni in *byte* del *file* che contiene il dizionario compresso specificato. Per individuare il dizionario richiesto viene passato come parametro una delle costanti (TAG, WORD, ATTR). Restituisce –1 nel caso essa non sia corretta (cioè sia diversa da TAG, WORD, ATTR).

**API NR.13 – *XCDE_Get_Dict_Cmp_Filename***

```
// IN Dict: costante che indica il dizionario (TAG, WORD, ATTR);

// OUT Dict_Cmp_Filename: indirizzo della variabile che conterrà il nome
// del dizionario compresso;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Get_Dict_Cmp_Filename(unsigned char Dict,
                               char **Dict_Cmp_Filename,
                               XCDE_Console_Type *XCDE_Console);
```

Scrive in una variabile, il cui indirizzo è passato come parametro, il nome del *file* che contiene ildizionario compresso specificato. Per individuare il dizionario richiesto viene

passato come parametro una delle costanti (TAG, WORD, ATTR). Restituisce –1 nel caso essa non sia corretta (cioè sia diversa da TAG, WORD, ATTR).

**API NR.14 – *XCDE_Get_Dict_Dcmptmp_Filesize***

```
// IN Dict: costante che indica il dizionario (TAG, WORD, ATTR);

// OUT Dict_Dcmptmp_Filesize: indirizzo della variabile che conterrà
// le dimensioni del dizionario decompresso;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Get_Dict_Dcmptmp_Filesize(unsigned char Dict,
                                   int *Dict_Dcmptmp_Filesize,
                                   XCDE_Console_Type *XCDE_Console);
```

Scrive in una variabile, il cui indirizzo è passato come parametro, le dimensioni in *byte* del *file* temporaneo in cui è stato decompresso il dizionario specificato. Per individuarlo si utilizza come parametro una delle costanti (TAG, WORD, ATTR). La funzione restituisce –1 nel caso in cui il dizionario non sia stato decompresso.

**API NR.15 – *XCDE_Get_Dict_Dcmptmp_Filename***

```
// IN Dict: costante che indica il dizionario (TAG, WORD, ATTR);

// OUT Dict_Dcmptmp_Filename: indirizzo della variabile che conterrà
// il nome del file temporaneo che contiene il dizionario decompresso;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Get_Dict_Dcmptmp_Filename(unsigned char Dict,
                                   char **Dict_Dcmptmp_Filename,
                                   XCDE_Console_Type *XCDE_Console);
```

Scrive in una variabile, il cui indirizzo è passato come parametro, il nome del *file* temporaneo in cui è stato decompresso il dizionario specificato. Per individuarlo si utilizza come parametro una delle costanti (TAG, WORD, ATTR). Nel caso in cui il dizionario non sia stato decompresso, la funzione restituisce il valore -1.

**API NR.16 – *XCDE_Get_Tagstruct_Filesize***

```
// OUT Tagstruct_Filesize: indirizzo di memoria della variabile che
// conterrà le dimensioni del file *.xml.tagstruct (tagstruct);

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Get_Tagstruct_Filesize(int *Tagstruct_Filesize,
                                XCDE_Console_Type *XCDE_Console);
```

Scrive in una variabile, il cui indirizzo è passato come parametro, la dimensione in *byte* del *file* che contiene la *tagstruct*. Nel caso in cui non sia stata creata, restituisce il valore –1.

**API NR.17 – *XCDE_Get_Tagstruct_Filename***

```
// OUT Tagstruct_Filename: indirizzo della variabile che conterrà il nome
// del file *.xml.tagstruct (tagstruct);

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Get_Tagstruct_Filename(char **Tagstruct_Filename,
                                XCDE_Console_Type *XCDE_Console);
```

Scrive in una variabile, il cui indirizzo è passato come parametro, il nome del *file* che contiene la *tagstruct*. Nel caso in cui essa non sia stata creata restituisce il valore -1.

## Funzioni per operare sui dizionari

Le funzioni per la gestione dei dizionari consentono di decomprimerli, allocarne in memoria gli elementi in modo da poter ottenere il *token* corrispondente ad un determinato *rango*, ottenere informazioni su di essi, come ad esempio il numero dei *token*, ed effettuare ricerche.
I dizionari sono compressi utilizzando il programma *Gzip*. L'allocazione in memoria viene realizzata decomprimendo il dizionario in un *file* temporaneo, mappandolo in memoria e allocando un vettore di puntatori, ognuno dei quali punta al *token* che ha come *rango nel dizionario globale* quella posizione del vettore. Le stringhe che rappresentano i *token* all'intero dei dizionari sono separate dal carattere NULL. In questo modo risulta immediato individuare una singola stringa all'interno del *file* a partire dal suo primo carattere. Mentre l'allocazione richiede necessariamente la decompressione del dizionario, che avviene utilizzando un *file* temporaneo, quest'ultima può essere effettuata senza necessariamente dover effettuare l'allocazione, ad esempio per effettuare delle ricerche. Esse sono realizzate con tre funzioni differenti, una per ogni dizionario, visto che *logicamente* si tratta di ricerche di natura diversa. Nella loro implementazione sono utilizzati i comandi *grep* e *agrep*. In particolare quest'ultimo viene utilizzato soltanto per effettuare ricerche con errori sul dizionario delle parole.

**API NR.18 – *XCDE_Decompress_Dictionary***

```
// IN Dict: costante (TAG, WORD, ATTR) che indica il dizionario  da
// decomprimere;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Decompress_Dictionary(unsigned char Dict,
                               XCDE_Console_Type *XCDE_Console);
```

Decomprime il dizionario specificato in un *file* temporaneo. Prende come parametro una costante che indica il dizionario da decomprimere (TAG, WORD, ATTR). Nei casi in cui il dizionario non sia specificato correttamente (ad esempio il parametro passato sia diverso da

TAG, WORD e ATTR) o il dizionario specificato sia già stato decompresso, la funzione restituisce il valore –1.

**API NR.19 – *XCDE_Load_Dictionary***

```
// IN Dict: costante (TAG, WORD, ATTR) che indica il dizionario  da
// allocare in memoria;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Load_Dictionary(unsigned char Dict,
                         XCDE_Console_Type *XCDE_Console);
```

Alloca il dizionario in memoria, cioè effettua il *mapping* del *file* decompresso e crea le strutture dati necessarie per effettuare la traduzione dal *rango di un token* alla *stringa* che lo rappresenta. Nel caso in cui il dizionario non sia stato decompresso o esso non sia stato specificato correttamente (ad esempio il parametro passato sia diverso da TAG, WORD, ATTR) la funzione restituisce il valore –1.

**API – NR. 20 – XCDE*_Get_Num_El_Dict***

```
// IN Dict: costante che indica il dizionario (TAG, WORD, ATTR);

// OUT Num_El: indirizzo della variabile che conterrà il numero di
// elementi che formano il dizionario;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Get_Num_El_Dict(unsigned char Dict,
                         int *Num_El,
                         XCDE_Console_Type *XCDE_Console);
```

Scrive in una variabile, il cui indirizzo in memoria è passato come parametro, il numero di *token* che formano il dizionario specificato attraverso una costante (TAG, WORD, ATTR). Non è necessario aver decompresso o allocato in memoria il dizionario per utilizzare questa funzione. Essa restituisce il valore –1 nel caso in cui il dizionario non sia stato specificato correttamente (ad esempio il parametro passato sia diverso da TAG, WORD, ATTR).

**API – NR. 21 – XCDE*_Get_Num_El_Dictlev***

```
// IN lev: livello dell'albero di Huffman (o, è lo stesso, lunghezza
// della parola di codice) di cui si vuole conoscere il numero degli
// elementi corrispondenti nel dizionario;

// IN Dict: costante che indica il dizionario (TAG, WORD, ATTR);

// OUT Num_El: indirizzo della variabile che conterrà il numero di
// elementi del dizionario e con la lunghezza di parola di codice
// indicata;

// IN XCDE_Console: indirizzo della struttura console
```

```
int XCDE_Get_Num_El_Dictlev(int lev,
                            unsigned char Dict,
                            int *Num_El,
                            XCDE_Console_Type *XCDE_Console);
```

Scrive in una variabile, il cui indirizzo in memoria è passato come parametro, il numero di *token* di un dizionario aventi una specificata lunghezza *lev* di parola di codice. Il dizionario è individuato da una costante (TAG, WORD, ATTR). Non è necessario aver decompresso o allocato in memoria il dizionario per utilizare questa funzione. Essa restituisce il valore –1 nei casi in cui il dizionario non sia specificato correttamente (ad esempio il parametro passato sia diverso da TAG, WORD, ATTR) o quando il livello non è corretto (ad esempio è minore o uguale di 0 o maggiore della massima lunghezza della parola di codice).

**API – NR. 22 – XCDE_*DictEl2Lev***

```
// IN Dict: costante che indica il dizionario (TAG, WORD, ATTR);

// IN NDictElem: variabile che contiene il rango (>0) di un token
// nel dizionario indicato;

// OUT Lev: lunghezza di una parola di codice corrispondente
// al un token individuato dalla due variabili precedenti;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_DictEl2Lev(unsigned char Dict,
                    int NDictElem,
                    int *Lev,
                    XCDE_Console_Type *XCDE_Console);
```

Determina la lunghezza in *byte* della parola di codice corrispondente ad un *token* del dizionario specificato attraverso una costante (TAG, WORD, ATTR) e il *rango del token nel dizionario*. La funzione restituisce il valore –1 nel caso in cui il dizionario o il *rango del token* non siano specificati correttamente (ad esempio la costante dizionario sia diversa da TAG, WORD o ATTR o il *rango del token* sia minore di 0 o maggiore o uguale del numero dei *token* all'interno del dizionario).

**API – NR. 23 – XCDE_*Search_Tag_Dictionary***

```
// IN String_To_Search: nome del tag da cercare;

// OUT dictrank: indirizzo della variabile che contiene il rango nel
// dizionario dei tag dell'eventuale occorrenza trovata;

// IN XCDE_Console: indirizzo della struttura console


int XCDE_Search_Tag_Dictionary(char *String_To_Search,
                               int *dictrank,
                               XCDE_Console_Type *XCDE_Console);
```

Questa funzione effettua le ricerche sul dizionario dei *tag*. L'unico tipo di ricerca possibile

è quella *esatta*: la stringa passata in *input* alla funzione deve *corrispondere esattamente* al nome del *tag* cercato, senza considerare i caratteri separatori (i caratteri < e > sono esclusi dal nome dei *tag*). La ricerca è *case-sensitive*. La variabile *dictrank* conterrà, se la funzione è terminata con esito positivo, il *rango nel dizionario globale* dell'occorrenza trovata. In caso non sia stata trovata nessuna occorrenza essa conterrà il valore 0.

**API – NR. 24 – XCDE_*Search_Attr_Dictionary***

```
// IN attributename: stringa in base alla quale effettuare la ricerca sui
// nomi dell'attributo;

// IN attributevalue: stringa in base alla quale effettuare la ricerca
// sui valori dell'attributo;

// OUT TokenDict: indirizzo della variabile che conterrà il vettore con i
// ranghi nel dizionario degli attributi delle occorrenze trovate;

// OUT NumToken: indirizzo della variabile che conterrà il numero di
// occorrenze trovate;

// IN typeattributesearch: variabile che contiene il tipo di ricerca da
// effettuare sui nomi degli attributi (XCDE_EXACT, XCDE_PREFIX,
// XCDE_SUFFIX, XCDE_CONTAINED, XCDE_ERROR, XCDE_REGEXP, XCDE_ANYVALUE);

// IN typevaluesearch: variabile che contiene il tipo di ricerca da
// effettuare sui valori degli attributi (XCDE_EXACT, XCDE_PREFIX,
// XCDE_SUFFIX, XCDE_CONTAINED, XCDE_ERROR, XCDE_REGEXP,
// XCDE_TOKEN_EXACT, XCDE_TOKEN_SUFFIX,  XCDE_TOKEN_PREFIX);

// IN lettercaseattribute: variabile che specifica se la ricerca sui nomi
// degli attributi debba essere case-sensitive o meno (XCDE_CASE,
// XCDE_NOCASE);

// IN lettercasevalue: variabile che specifica se la ricerca sui valori
// degli attributi debba essere case-sensitive o meno (XCDE_CASE,
// XCDE_NOCASE);

// IN nerrorattribute: variabile che specifica, in caso di ricerca con
// errore sui nomi degli attributi, il numero massimo di errori;

// IN nerrorvalue: variabile che specifica, in caso di ricerca con
// errore sul valore degli attributi, il numero massimo di errori;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Search_Attr_Dictionary(char *attributename,
                                char *attributevalue,
                                int **TokenDict,
                                int *NumToken,
                                unsigned char typeattributesearch,
                                unsigned char typevaluesearch,
                                unsigned char lettercaseattribute,
                                unsigned char lettercasevalue,
                                unsigned int nerrorattribute,
                                unsigned int nerrorvalue,
```

```
                                XCDE_Console_Type *XCDE_Console);
```

Questa funzione effettua le ricerche sul dizionario degli attributi. Per facilitare le ricerche la funzione consente di specificare in maniera distinta il tipo di ricerca per i nomi degli attributi e per i loro valori. Le occorrenze che saranno individuate dalla funzione sono quelle che soddisfano entrambe le ricerche. Per quanto riguarda il nome, la ricerca può essere *esatta* (XCDE_EXACT), per *prefisso* (XCDE_ PREFIX), per *suffisso* (XCDE_SUFFIX), per *sottostringa* (XCDE_CONTAINED), con *errori* (XCDE_ERROR, e in questo caso deve essere specificato il numero degli errori tramite il parametro *nerrorattribute*, che deve essere maggiore di 0), per *espressione regolare* (XCDE_REGEXP), o può essere specificato il valore XCDE_ANYVALUE, che non effettua la ricerca sui nomi, considerando quindi solo i risultati della ricerca sui valori degli attributi (lo stesso tipo di ricerca è possibile per i valori, ma non è possibile che entrambe siano di tipo XCDE_ANYVALUE). Per la ricerca sui nomi degli attributi può essere specificata la *case-sensitivity* o *case-insensivity* attravererso il parametro *lettercaseattribute* ed i valori XCDE_CASE e XCDE_NOCASE.
Per quanto riguarda le ricerche sui valori, esse possono essere dello stesso tipo dei nomi, ma sono previste alcune opzioni aggiuntive: XCDE_TOKEN_EXACT, XCDE_TOKEN_SUFFIX e XCDE_TOKEN_PREFIX. In pratica queste opzioni ci consentono di isolare i valori, i prefissi e i suffissi non *dell'intero valore* di un attributo, ma di un suo *token*. Ricordiamo infatti che il valore di ogni attributo è diviso in più *token* in base ad un elenco di separatori che sono forniti dall'utente nel file di configurazione *xcde_build.conf*. Gli altri tipi di ricerca sono uguali, utilizzando i parametri *lettercasevalue* e *nerrorattribute* per determinare se la ricerca sia o meno *case-sensitive* e il numero di errori nella ricerca di tipo XCDE_ERROR
La funzione restituisce il valore –1 nei casi in cui i parametri non siano corretti. In questo caso la correttezza riguarda non solo le ricerche sul nome e l'attributo ma anche la combinazione delle due. A questo proposito se i due parametri *typeattributesearch* e *typevaluesearch*, attraverso cui si determinano i tipi di ricerca rispettivamente per i nomi e i valori degli attributi, hanno entrambi valore *XCDE_ANYVALUE* la funzione restituisce il valore –1. Per tutti gli altri casi vale la regola generale che i parametri non significativi devono avere valore 0 (ad esempio i campi *nerrorattribute* e *nerrorvalue* nel caso di ricerche diverse da XCDE_ERROR);

### API – NR. 25 – XCDE_*Search_Word_Dictionary*

```
// IN String_To_Search: stringa da ricercare all'interno del dizionario
// delle parole. Eventualmente può contenere un prefisso, un suffisso,
// una sottostringa o un'espressione regolare a seconda del tipo di
// ricerca;

// OUT TokenDict: indirizzo di una variabile che conterrà un vettore
// contenente i ranghi nel dizionario delle parole delle occorrenze
// trovate;

// OUT NumToken: indirizzo della variabile che conterrà il numero delle
// occorrenze trovate;

// IN type: variabile che contiene la costante per il tipo di ricerca da
```

```
// effettuare (XCDE_EXACT, XCDE_PREFIX, XCDE_XCDE_SUFFIX, XCDE_CONTAINED,
// XCDE_ERROR, XCDE_REGEXP);

// IN lettercase: variabile che contiene la costante per considerare
// o meno le differenze fra lettere maiuscole o minuscole nella ricerca
// (XCDE_CASE, XCDE_NOCASE);

// IN nerror: in caso si effettui una ricerca per errore contiene il
// numero massimo di errori ammessi nella ricerca. In tutti gli altri
// casi deve contenere il valore 0;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Search_Word_Dictionary(char *String_To_Search,
                                int **TokenDict,
                                int *NumToken,
                                unsigned char type,
                                unsigned char lettercase,
                                unsigned int nerror,
                                XCDE_Console_Type *XCDE_Console);
```

Questa funzione effettua le ricerche sul dizionario delle parole. Le ricerche possono essere dei tipi seguenti: *esatta* (XCDE_EXACT), per *prefisso* (XCDE_PREFIX), per *suffisso* (XCDE_SUFFIX), per *sottostringa* (XCDE_CONTAINED), con *errori* (XCDE_ERROR) e per *espressione regolare* (XCDE_REGEXP). Per tutte le ricerche, ad eccezione di quella con errori, è possibile stabilire se si debba considerare (XCDE_CASE) o meno (XCDE_NOCASE) la differenza fra i caratteri maiuscoli o minuscoli. In caso di ricerca con errori questo parametro dovrà sempre essere impostato a XCDE_NOCASE. In questo caso sarà anche necessario passare in *input* il numero degli errori da considerare, che dovrà naturalmente essere maggiore di 0. In tutti gli altri casi questo parametro dovrà avere valore 0. La stringa passata in *input* viene utilizzata in un *pattern di ricerca* differente nei diversi casi. Nel caso di ricerca per *espressione regolare* essa sarà ad esempio considerata un'espressione regolare e dovrà essere conforme alle regole sintattiche per questo tipo di espressioni definite dal comando *grep*. La funzione restituisce il numero di occorrenze trovate, e nel caso in cui esso sia maggiore di 0, sarà restituito anche un vettore contenente per ogni posizione il *rango nel dizionario delle parole* dell'occorrenza corrispondente. La funzione restituisce il valore –1 nel caso in cui i parametri non siano corretti, ad esempio se il tipo di ricerca è *XCDE_ERROR* e il numero di errori è 0.

## Funzioni per operare sulle IL

Queste funzioni consentono di accedere alle *liste invertite* e di ottenere informazioni su di esse. È possibile accedere solo alla lista invertita delle posizioni o anche alla *lista delle proximity*. Abbiamo già parlato dei tipi previsti dalla libreria per l'accesso alle liste. Si tratta di *XCDE_IL_Couple_Type* per la *lista delle posizioni* e di *XCDE_IL_Triple_Type* per la *lista delle posizioni e della proximity*. *Le liste invertite delle posizioni* e *delle proximity* sono implementate come vettori di questi due tipi. L'accesso alla IL richiesta si effettua accedendo all'indice del blocco di cui la lista fa parte e effettuando degli accessi sequenziali, attraverso l'incremento di un puntatore, all'interno del blocco fino a quando non si raggiunge la *lista* richiesta.

**API – NR. 26 – XCDE_*Get_IL_ListPos***

```
// IN Rank: rango nel dizionario globale del token di cui si richiede la
// lista invertita della posizioni ;

// OUT IL: indirizzo di memoria della variabile che conterrà la lista
// invertita delle posizioni, implementato come un vettore di coppie di
// tipo XCDE_IL_Couple_Type;

// OUT IL_Num_El: indirizzo di memoria della variabile che conterrà il
// numero di elementi della lista invertita. La variabile conterrà valore
// 0 nel caso la lista sia vuota. In questo caso il token è fisico e non
// logico

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Get_IL_ListPos(int Rank,
                        XCDE_IL_Couple_Type **IL,
                        int *IL_Num_El,
                        XCDE_Console_Type *XCDE_Console);
```

Questa funzione accede alla *lista invertita delle posizioni* di un *token* di cui viene fornito in *input* il *rango nel dizionario globale*. La funzione restituirà il valore –1 nel caso in cui quest'ultimo valore non sia corretto e sia ad esempio minore di 0 o maggiore o uguale del numero massimo di *token*.

**API – NR. 27 – XCDE_*Get_IL_ListPosProx***

```
// IN Rank: rango nel dizionario globale del token facente parte
// del dizionario delle parole di cui si richiede la lista
// invertita della posizioni;

// OUT IL: indirizzo di memoria della variabile che conterrà la lista
// invertita delle posizioni e della proximity, implementato come
// un vettore di triple di tipo XCDE_IL_Couple_Type;

// OUT IL_Num_El: indirizzo di memoria della variabile che conterrà il
// numero di elementi della lista invertita. La variabile conterrà valore
// 0 nel caso la lista sia vuota;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Get_IL_ListPosProx(int Rank,
                            XCDE_IL_Triple_Type **IL,
                            int *IL_Num_El,
                            XCDE_Console_Type *XCDE_Console);
```

Questa funzione accede alla *lista invertita delle posizioni* e *della proximity* di un *token* facente parte del dizionario delle parole e di cui viene fornito in *input* il *rango nel dizionario globale*. La funzione restituisce il valore –1 nel caso in cui il *rango* non sia corretto, ad esempio perché minore di 0 o maggiore o uguale del numero massimo di *token*, nel caso in cui ad esso non corrisponda un *token* nel dizionario delle parole e nel caso in cui

non sia stata creata dall'utente la *lista delle proximity*.

**API – NR. 28 – XCDE_*Setted_Proximity***

```
// OUT ProxFlag: indirizzo di memoria della variabile che conterrà il
// flag di proximity. Esso ha valore 1 se le liste di proximity sono
// state create, e 0 altrimenti;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_SettedProximity(unsigned char *ProxFlag,
                         XCDE_Console_Type *XCDE_Console);
```

Questa funzione consente di leggere il *flag di proximity* dal quale possiamo determinare se le *liste di proximity* sono state create o meno.

## Funzioni per operare sul documento compresso

Questo gruppo di funzioni consentono di ottenere delle *parole di codice di Huffman* o delle finestre di testo a partire dal documento compresso. Dalle *parole di codice* è possibile, attraverso le funzioni di traduzione, ricondursi ad altre informazioni come ad esempio il *token* corrispondente. Esiste una funzione che, data una posizione nel documento compresso, ci consente di determinare se la posizione si trova o meno alla fine del documento stesso. L'operazione di lettura di una generica parola di codice prevede l'accesso ad un blocco Zlib e la sua decompressione.

**API – NR. 29 – XCDE_*Get_Body_Forward_Codeword***

```
// IN Position : posizione nel documento compresso da cui iniziare la
// lettura della parola di codice di Huffman;

// OUT Codeword : indirizzo della variabile che conterrà la parola di
// codice che inizia dalla posizione passata in input;

// OUT Cwlen: indirizzo della variabile che conterrà la lunghezza in byte
// della parola di codice;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Get_Body_Forward_Codeword(int Position,
                                   int *Codeword,
                                   int *Cwlen,
                                    XCDE_Console_Type *XCDE_Console);
```

Questa funzione legge la parola di codice di Huffman che inizia dalla posizione passata in *input*, restituendola insieme con la sua lunghezza in *byte* attraverso due variabili intere i cui indirizzi sono passati in *input*. In particolare la lunghezza può essere utilizzata per aggiornare la posizione, così da leggere le parole del documento in maniera sequenziale. Essa restituisce il valore –1 nel caso in cui la posizione fornita non sia corretta, cioè sia minore di 0 o maggiore dell'ultima posizione di una parola di codice nel compresso o non

sia l'inizio di una parola di codice.

### API – NR. 30 – XCDE_*Get_Body_Backward_Codeword*

```
// IN Position: posizione nel documento compresso da cui inizia la
// parola di codice successiva a quella a cui accedere;

// OUT Codeword: indirizzo in memoria della variabile che conterrà la
// parola di codice che termina nella posizione passata in input;

// OUT Cwlen: indirizzo della variabile che conterrà la lunghezza in byte
// della parola di codice;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Get_Body_Backward_Codeword(int Position,
                                    int *Codeword,
                                    int *Cwlen,
                                    XCDE_Console_Type *XCDE_Console);
```

Questa funzione legge la parola di codice di Huffman che termina nella posizione passata in *input*, restituendola insieme con la sua lunghezza in *byte* attraverso due variabili intere i cui indirizzi sono passati in *input*. In particolare la lunghezza può essere utilizzata per aggiornare la posizione, così da aggiornare la posizione all'interno del documento compresso. Esso restituisce il valore –1 nel caso in cui la posizione fornita non sia corretta, cioè sia uguale a 0 o non corrisponda con la fine di una parola di codice nel documento.

### API – NR. 31 – XCDE_*Get_Body_TextWindow*

```
// IN Position: posizione nel documento compresso in cui è centrata la
// finestra di testo estratta;

// OUT Window: indirizzo in memoria della variabile che conterrà la
// finestra di testo estratta dal documento;

// IN Count: variabile che contiene il numero di elementi a cui si
// inizializza il contatore per determinare l'ampiezza della finestra;

// IN unsigned char direction: flag che contiene la direzione
// di lettura dei caratteri a partire dalla posizione nel documento
// compresso; Si tratta di una costante definita nel file intestazione
// della libreria. La costante può essere XCDE_BACKWARD, per
// effettuare la lettura all'indietro a partire dalla posizione
// passata in input, XCDE_FORWARD per leggere in avanti a partire
// dalla posizione passata in input e XCDE_BACKWARD_FORWARD per
// leggere sia in avanti che all'indietro;

// OUT backward_new_position: indirizzo della variabile intera
// che conterrà l'eventuale posizione raggiunta nel documento
// compresso dopo aver terminato la fase di estrazione
// all'indietro;

// OUT forward_new_position: indirizzo della variabile intera che
// conterrà l'eventuale posizione raggiunta nel documento compresso
```

```
// dopo aver terminato la fase di estrazione in avanti;

// IN Counter_function: è l'indirizzo della funzione contatore
// utilizzata per determinare la condizione di terminazione della
// lettura, sia in avanti che all'indietro. Essa prende
// come parametri la parola di codice letta dal documento compresso,
// la sua lunghezza e la console e deve restituire un valore 0 o 1
// intero che sarà sottratto al contatore. Quando il contatore
// raggiungerà il valore 0 in avanti si termina la lettura in questa
// direzione, e allo stesso modo si calcola la funzione di terminazione
// all'indietro. Condizione di terminazione ulteriore in entrambe le
// direzioni è che si raggiunga l'inizio o la fine del documento
// compresso;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Get_Body_TextWindow(int Position,
                             char **Window,
                             int Count,
                             unsigned char direction,
                             int *backward_new_position,
                             int *forward_new_position,
                             int (*Counter_function)(int cw, int cwlen,
                                      XCDE_Console_Type *XCDE_Console),
                             XCDE_Console_Type *XCDE_Console);
```

Questa funzione estrae dal documento compresso una finestra di testo. Quest'ultima è calcolata a partire dalla posizione nel documento compresso passata in *input*. Da questa posizione si estrae dei caratteri in avanti, all'indietro, o in entrambe le direzioni fino a quando la funzione contatore, passata anch'essa in *input*, assume il valore 1 un numero di volte pari a *Count* in tutte le direzioni di estrazione, il che vuol dire che in realtà abbiamo due contatori inizializzati allo stesso valore per l'estrazione in avanti e per quella all'indietro. La funzione restituisce –1 nel caso in cui i parametri passati non siano corretti, ad esempio se la posizione passata in *input* è minore di 0 o maggiore della dimensione del documento compresso.

### API – NR. 32 – XCDE_*Get_Body_Eof*

```
// IN body_position: contiene la posizione passata in input di cui si
// vuol sapere se si trova o meno alla fine del documento;

// OUT eofflag: indirizzo in memoria della variabile che conterrà il flag
// per la terminazione del documento compresso. Essa avrà valore 1 se la
// posizione passata in input si trova alla fine del documento compresso
// e 0 altrimenti;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Body_Eof(int Body_Position,
                  unsigned char *Eof_Flag,
                  XCDE_Console_Type *XCDE_Console) ;
```

Questa funzione consente di determinare se una posizione, il cui valore viene passato ad

essa in *input*, si trova o meno alla fine del documento compresso.

## Funzioni per operare sulla tagstruct

Questo gruppo di funzioni consentono di accedere alle informazioni contenute nella *tagstruct*, cioè di effettuare interrogazioni di *stabbing* e di determinare se la struttura sia stata creata o meno. L'accesso alla *tagstruct* viene effettuato attraverso un *header* che contiene la posizione nel *file* delle diverse finestre create dall'algoritmo di Chazelle. Queste informazioni sono memorizzate nella *console* al primo accesso alla struttura *tagstruct*, in modo da velocizzare tutti gli accessi successivi. Anche per l'acceesso alle finestre della *tagstruct* è previsto un meccanismo di *caching*. L'ultima finestra decompressa (ricordiamo che le finestre sono compresse con la libreria Zlib) viene memorizzata nella console.

### API – NR. 33 – XCDE_*Get_Tagstruct_Tags_Enclosing*

```
// IN Stab_Point: puntatore ad una variabile di tipo XCDE_IL_Couple_Type
// che contiene la posizione nel documento su cui effettuare
// l'interrogazione di stabbing, cioè di cui determinare l'insieme
// dei tag che lo contengono;

// OUT Enclosing_tags_ranks: indirizzo della variabile vettore che
// conterrà i ranghi nel dizionario globale dei tag che contengono
// la posizione di stabbing. La posizione del rango di un tag nel
// vettore indicato indica il suo grado di innestamento. Così il
// tag nella posizione i contiene il tag della posizione i+1.

// OUT Num_enclosing_tags: indirizzo della variabile intero che
// conterrà il numero di tag che contengono la posizione di stabbing;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Get_Tagstruct_Tags_Enclosing(XCDE_IL_Couple_Type *Stab_point,
                                      int **Enclosing_tags_ranks,
                                      int *Num_enclosing_tags,
                                      XCDE_Console_Type *XCDE_Console) ;
```

Questa funzione consente di determinare la struttura dei *tag* che racchiudono nel documento originale un determinato elemento che si trova alla posizione passata come input. La funzione restituisce un vettore con i ranghi nel dizionario globale dei *tag*, ordinati seconda la relazione di contenimento, che racchiudono la posizione insieme con il loro numero. Essa restituisce il valore –1 nel caso la *tagstruct* non sia stata creata o la posizione passata in *input* non sia corretta.

### API – NR. 34 – XCDE_*SettedTagstruct*

```
// OUT TagstructFlag: indirizzo della variabile che conterrà il flag
// per determinare se la tagstruct è stata creata. Il valore della
// variabile sarà 1 in caso affermativo e 0 altrimenti.
```

```
// IN XCDE_Console: indirizzo della struttura console

int XCDE_SettedTagstruct(unsigned char *TagstructFlag,
                         XCDE_Console_Type *XCDE_Console);
```

Questa funzione consente di determinare se la struttura dati *tagstruct* è stata creata.

## Funzioni per la traduzione degli identificatori

Nella libreria XCDE ogni *token* è identificato in diversi modi: dal *rango all'interno del dizionario globale*, dal *rango all'interno del dizionario* di cui esso fa parte (TAG, WORD, ATTR) e dalla *parola di codice di Huffman*. Le funzioni che fanno parte di questo gruppo consentono di ottenere un identificatore a partire da un altro. Queste operazioni sono particolarmente utili nel codice di qualsiasi applicazione che utilizzi la libreria XCDE, in modo da consentire di operare le diverse funzioni senza dover essere vincolati ad acquisire un particolare identificatori. Ad esempio è possibile accedere alla *lista invertita* di un *token* a partire dal suo *rango nel dizionario globale* o conoscendo il dizionario di cui esso fa parte e il *rango* al suo interno.

### API – NR. 35 – XCDE_*Dict2Rank*

```
// IN Dict: costante (WORD, TAG, ATTR) che indica il dizionario di
// cui fa parte il token di cui ottenere il rango nel dizionario
// globale;

// IN NdictElem: rango nel dizionario del token da
// tradurre ;

// OUT Rank: indirizzo della variabile di tipo intero che conterrà
// il rango nel dizionario globale corrispondente al rango nel
// dizionario passato in input;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Dict2Rank(unsigned char Dict,
                   int NDictElem,
                   int *Rank,
                   XCDE_Console_Type *XCDE_Console);
```

Questa funzione consente di ottenere il *rango nel dizionario globale* del *token* identificato dal *rango nel dizionario* e dal dizionario passati in *input*. Restituisce il valore –1 nel caso quest'ultima informazione non sia corretta, ad esempio se la costante dizionario è diversa da TAG, WORD, ATTR o il *rango nel dizionario* è minore o uguale di 0 o maggiore del numero di *token* contenuti nel dizionario indicato.

### API – NR. 36 – XCDE_*Rank2Dict*

```
// IN Rank: rango nel dizionario globale di un token;

// OUT Dict: indirizzo della variabile che conterrà la costante
```

```
// dizionario (TAG, ATTR, WORD) corrispondente al token di cui sopra;

// OUT NdictElem: rango nel dizionario corrispondente al token;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Rank2Dict(int Rank,
                   unsigned char *Dict,
                   int *NDictElem,
                   XCDE_Console_Type *XCDE_Console);
```

Questa funzione consente di ricondursi dal *rango nel dizionario globale* di un *token* al suo dizionario e al *rango all'interno del dizionario*. Restituisce il valore –1 nel caso in cui il *rango nel dizionario globale* passato in *input* sia minore di 0 o maggiore o uguale del numero di totale di *token*.

### API – NR. 37 – XCDE_*Codeword2Rank*

```
// IN Codeword: parola di codice di Huffman del token di cui si vuole
// ottenere il rango nel dizionario globale;

// IN Cwlen: lunghezza in byte della parola di codice di cui sopra;

// OUT Rank: indirizzo in memoria della variabile intera che conterrà il
// rango nel dizionario globale del token;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Codeword2Rank(int Codeword,
                   int Cwlen,
                   int *Rank,
                   XCDE_Console_Type *XCDE_Console);
```

Questa funzione consente di ottenere il *rango nel dizionario globale* di un *token* a partire dalla sua parola di codice di Huffman, ottenuta ad esempio con una delle funzioni per l'accesso al documento compresso. Restituisce –1 nel caso in cui la parola di codice passata in *input* non sia corretta, cioè non corrisponda a nessun *token* del dizionario globale.

### API – NR. 38 – XCDE_*Codeword2Token*

```
// IN Codeword: parola di codice di Huffman del token di cui si vuole
// ottenere la stringa corrispondente nel documento sorgente;

// IN Cwlen: lunghezza in byte della parola di codice di cui sopra;

// OUT Token: indirizzo in memoria della variabile che punterà alla
// stringa ottenuta;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Codeword2Token(int Codeword,
                        int Cwlen,
```

```
                        char **Token,
                        XCDE_Console_Type *XCDE_Console);
```

Questa funzione consente di ottenere la stringa nel documento sorgente corrispondente ad un *token* a partire dalla sua parola di codice di Huffman, ottenuta ad esempio con una delle funzioni per l'accesso al documento compresso. Restituisce –1 nel caso in cui la parola di codice passata in *input* non sia corretta, cioè non corrisponda a nessun *token* del dizionario globale.

**API – NR. 39 – XCDE_*Rank2Token***

```
// IN Rank: rango nel dizionario globale del token di cui si vuole
// ottenere la stringa corrispondente;

// OUT Token: indirizzo in memoria della variabile che punterà alla
// stringa ottenuta;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Rank2Token(int Rank,
                char **Token,
                XCDE_Console_Type *XCDE_Console);
```

Questa funzione consente di ottenere la stringa nel documento sorgente corrispondente ad un *token* a partire dal suo *rango nel dizionario globale*. Restituisce –1 nel caso in cui esso non sia corretto, perché minore di 0 o maggiore o uguale del numero dei *token* nel dizionario globale.

## Funzioni varie

Di questo gruppo di funzioni fanno parte tutte quelle che non rientrano nelle categorie precedenti. Si tratta di funzioni attraverso le quali si possono ottenere informazioni di carattere generale: il numero dei *token* che fanno parte del dizionario globale, la lunghezza massima di una parola di codice di Huffman o il numero di *token* nel dizionario globale ad un determinato livello dell'albero canonico di Huffman associato al documento.

**API – NR. 40 – XCDE_*Get_Num_Tokens***

```
// OUT Num_Tokens: indirizzo in memoria della variabile intera
// che conterrà il numero dei token che formano il dizionario
// globale;

// IN XCDE_Console: indirizzo della struttura console


int XCDE_Get_Num_Tokens(int *Num_Tokens,
                        XCDE_Console_Type *XCDE_Console)
```

### API – NR. 41 – XCDE_*Get_Maxcwlen*

```
// OUT Maxcwlen: lunghezza massima di una parola di codice di Huffman
// in byte;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Get_Maxcwlen(int *Maxcwlen,
                      XCDE_Console_Type *XCDE_Console)
```

### API – NR. 42 – XCDE_*Get_Num_El_Lev*

```
// IN Lev: livello nell'albero canonico di Huffman associato costruito
// sul dizionario globale di cui si vuol conoscere il numero di elementi
// che ne fanno parte;

// OUT Num_El: variabile che conterrà il numero dei token nel
// dizionario globale che fanno parte del livello di cui sopra o, è
// lo stesso, hanno una lunghezza della parola di codice di Huffman
// uguale a Lev;

// IN XCDE_Console: indirizzo della struttura console

int XCDE_Get_Num_El_Lev(int Lev,
                        int *Num_El,
                        XCDE_Console_Type *XCDE_Console)
```

Questa funzione consente di conoscere, dato un livello nell'albero canonico di Huffman associato al dizionario globale, il numero dei *token* nel dizionario globale che fanno parte di quel livello dell'albero o, è equivalente, il numero dei *token* la cui lunghezza della parola di codice di Huffman è uguale a *Lev*. Restituisce il valore –1 nel caso in cui il livello non sia stato specificato correttamente, ad esempio perché minore o uguale di 0 o maggiore della profondità massima dell'albero canonico.

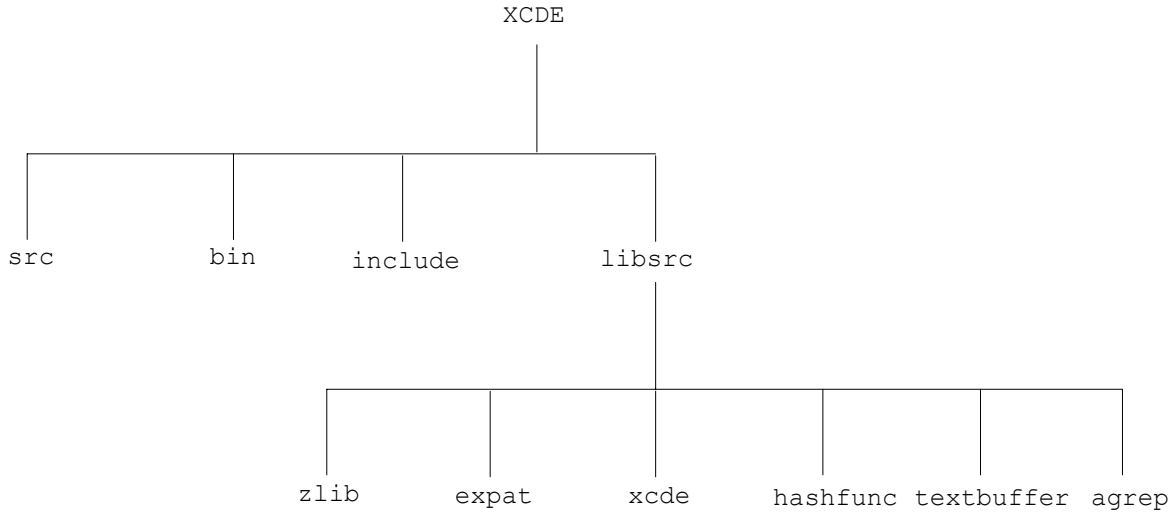
# APPENDICE B – I FILE DELLA LIBRERIA XCDE

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

```
tar –zxvf xcde.tgz
```

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

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




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

| | |
|---|---|
| `XCDE/` | *script Makefile* per la compilazione `dei` sorgenti e altri *script* utili; |
| `XCDE/src` | *file* sorgenti per la creazione dei comandi; |
| `XCDE/bin/` | comandi eseguibili creati dopo la compilazione e utili per l'implementazione e l'utilizzo della libreria; |
| `XCDE/include/` | *file* di intestazione necessari per la compilazione e l'uso della libreria; |
| `XCDE/libsrc/` | *file* sorgenti della libreria XCDE e delle altre librerie usate per la sua implementazione; |

| | |
|---|---|
| `XCDE/libsrc/zlib/` | *file* sorgenti della libreria Zlib; |
| `XCDE/libsrc/expat/` | *file* sorgenti della libreria Expat; |
| `XCDE/libsrc/integer/` | *file* sorgenti della libreria di compressione degli interi; |
| `XCDE/libsrc/xcde/` | *file* sorgenti per l'implementazione delle API di XCDE; |
| `XCDE/libsrc/hashfunc/` | *file* sorgenti per l'implementazione della struttura dati *hash*; |
| `XCDE/libsrc/textbuffer/` | *file* sorgenti per l'implementazione del *buffer* di testo; |
| `XCDE/libsrc/agrep/` | *file* sorgenti del comando *agrep* utilizzato dalla libreria XCDE; |

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

```
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:

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

| | |
|---|---|
| `XCDE/Makefile` | effettua la compilazione, utilizzabile con il comando `make –f Makefile;` |
| `XCDE/clean_exe` | effettua la rimozione di tutti i *file* oggetto, eseguibili e di libreria *ar* creati dal *makefile*; |
| `XCDE/build_tar` | effettua la creazione dell'archivio *tar* dei *file* della libreria; |
| `XCDE/bin/clean_xml` | effettua la rimozione dei *file* relativi ai documenti compressi ed indicizzati; |
| `XCDE/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:

| | |
|---|---|
| `XCDE/libsrc/xcde/xcde_lib.a` | libreria delle API di XCDE; |
| `XCDE/lib/libbuff.a` | libreria delle funzioni per la gestione del *buffer* di testo durante il *parsing*; |
| `XCDE/lib/libexpat.a` | libreria Expat delle funzioni SAX utilizzate per il *parsing* dei documenti XML; |

| | |
|---|---|
| `XCDE/lib/libhash.a` | libreria per la gestione della tabella Hash durante la fase di *parsing*; |
| `XCDE/lib/libinteger.a` | libreria delle funzioni per la compressione degli interi con l'algoritmo *continuation bit;* |
| `XCDE/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_view, xcde_extract` e `xcde_inf` e tutte le applicazioni costruite sulla libreria XCDE.
La struttura delle libreria è visualizzata nella figura B.3

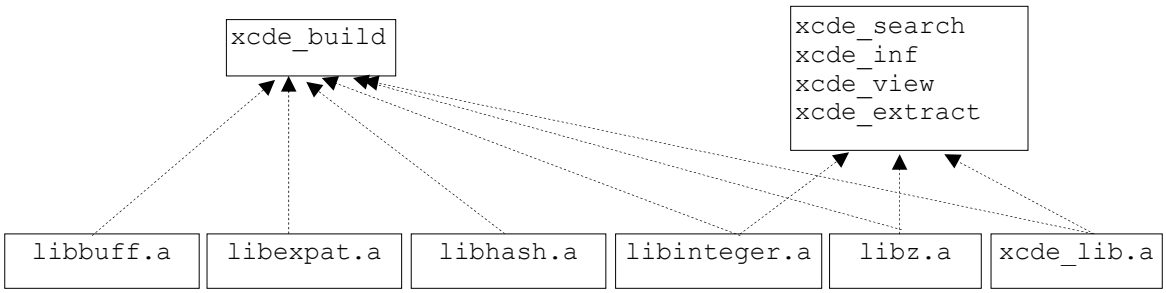



*Figura B.3 – utilizzo dei file di libreria da parte dei comandi di XCDE*

## 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_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_lib.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
```