UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

FACOLTA' DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA

TESI DI LAUREA
IN
SISTEMI INFORMATIVI

XML E SISTEMI DI GESTIONE DI BASI DI DATI:

IL RUOLO DEI DB XML NATIVI

RELATORI

Chiar.mo Prof. Antonio D'Acierno

Chiar.mo Prof. Antonio Picariello

CANDIDATO

Giacinto Montereale

Matr: 015/18797

ANNO ACCADEMICO
2002-2003

Ringraziamenti

Ringrazio tutti coloro che mi hanno fornito un aiuto alla realizzazione della presente tesi, in particolare la mia fidanzata Fortuna; senza di Lei non avrei raggiunto questo traguardo.

Un Ringraziamento speciale va alla mia famiglia per l'appoggio fornito in questi anni.

Indice

INTRODUZIONE

Il vasto panorama di tecnologie che ruotano attorno ad internet è condizionato sicuramente dal linguaggio XML (*eXtensible Markup Language)*.

XML rappresenta l'elemento base di una famiglia di tecnologie di cui tutti ormai sentiamo parlare in diversi contesti ma non sono in molti ad utilizzarlo direttamente in progetti reali.

Comprendere come XML sia nato e si sia affermato aiuta anche a capire quale sia il suo ruolo , quali i suoi obiettivi e quali le ragioni del suo successo.

La storia di XML[1] affonda le radici in un progetto più ambizioso: SGML, il cui acronimo sta per *Standard Generalized Markup Language.*
SGML è stato ideato e sviluppato da Charles Goldfarb, il rilascio ufficiale dello standard risale al 1986 ma solo da pochi anni ha cominciato a guadagnare consensi e ad essere utilizzato in un vasto spettro di applicazioni concrete.

L'idea alla base di SGML è quella dei linguaggi di markup. Tali linguaggi derivano concettualmente dai sistemi usati nelle tipografie per "marcare"le porzioni di testo indicandone le caratteristiche.

I linguaggi di markup[2] possono essere distinti in due gruppi:

- Linguaggi di markup di tipo procedurale
- Linguaggi di markup di tipo descrittivo

La differenza tra i due sta nel meccanismo usato per definire la formattazione del testo ovvero la rappresentazione.

I linguaggi di tipo procedurale (i cui testimoni più illustri sono lo Script, il TROFF, il TEX) indicano le procedure di trattamento del testo aggiungendo le istruzioni che devono essere eseguite per visualizzare la porzione di testo referenziata. I linguaggi di tipo

descrittivo lasciano la scelta del tipo di rappresentazione da applicare al testo, al software che di volta in volta lo riprodurrà.

I linguaggi del secondo tipo risultano più vantaggiosi perché,oltre a lasciare il focus della concentrazione sui problemi strutturali di leggibilità, prescindono in fase di lettura dal software con cui sono stati generati. Essi sono, in altre parole, quelli che permettono di garantire una corretta separazione tra struttura e rappresentazione.

SGML rientra in questa seconda classe.

SGML:

SGML, più che un linguaggio, è un *metalinguaggio*. Esso prescrive precise regole sintattiche per definire un insieme di marcatori e di relazioni tra marcatori in una tabella, denominata *Document Type Definition* (DTD). SGML, però, non dice nulla per quanto riguarda la tipologia, la quantità e il nome dei marcatori. Quest'astrazione, che permette di definire infiniti linguaggi di marcatura adatti per le varie esigenze particolari, costituisce il nucleo e la potenza di SGML.

HTML:

Un linguaggio di marcatura che rispetti le specifiche SGML, e gli eventuali sistemi informativi ad esso collegati, viene definito "applicazione SGML" (*SGML application*). Senza dubbio la più diffusa in assoluto di queste applicazioni è proprio HTML, sebbene il legame con SGML sia sconosciuto alla maggioranza dei suoi stessi utilizzatori. Presentato per la prima volta nel 1990 da Tim Berners-Lee del CERN, HTML (HyperText Markup Language) è un linguaggio di formattazione dei documenti che sfrutta il concetto di markup o marcatura, e consiste in un insieme di keywords, dette **tag**, racchiuse tra i caratteri '<' e '>' ed inserite in determinati punti del documento. Tali tags specificano le proprietà del testo da mostrare, così da permettere ad un programma detto ***browser*** di interpretare tale testo nel modo voluto. HTML è divenuto ormai lo standard per la

creazione di documenti ipertestuali in Internet grazie alla facilità di sviluppo, alla versatilità ed alla resa intuitiva.

XML:

Oggi però l'HTML è in crisi, difatti era stato disegnato per lo scambio di documenti scientifici e non certo per le moderne esigenze della realizzazione di vere e proprie riviste elettroniche.
Negli anni ha rilevato i suoi limiti e le numerose versioni che si sono succedute hanno cercato di ampliarne le capacità estendendo il linguaggio di marcatura ma, ben presto è tramontata la possibilità di realizzare un "linguaggio di marcatura generale". Si è quindi affacciata l'idea di poter definire i propri tag per creare il proprio linguaggio di marcatura adatto alle proprie esigenze. Nasce in questo modo *Extensible Markup Language* (XML), probabilmente il più importante progetto dell'organizzazione dalla nascita di World Wide Web.
Il progetto XML ha avuto inizio alla fine del 1996, nell'ambito della *SGML Activity* del W3C. Ma l'interesse che ha attirato sin dall'inizio (testimoniato da centinaia d'articoli sulle maggiori riviste del settore) ha portato il W3C a creare un apposito gruppo di lavoro (*XML Working Group*), composto da oltre ottanta esperti mondiali delle tecnologie SGML, ed una commissione (*XML Editorial Review Board*) deputata alla redazione delle specifiche. Nel febbraio del 1998, dopo oltre un anno di lavoro, le specifiche sono divenute una raccomandazione ufficiale, con il titolo *Extensible Markup Language (XML) 1.0*. Come di consueto tutti i materiali relativi al progetto, documenti ufficiali e informazioni ed aggiornamenti, sono pubblicati sul sito del consorzio all'indirizzo http://www.w3.org/XML.
XML, come accennato sopra, è un sottoinsieme di SGML semplificato ed ottimizzato specificamente per applicazioni in ambiente World Wide Web.

XML e HTML:

Per dare subito un input sulle potenzialità di XML facciamo un confronto con HTML.
I due "linguaggi" sono profondamente differenti tra di loro. Una prima, ma fondamentale, differenza sta nel fatto che HTML è solo una serie di regole per la formattazione del testo mentre XML è un vero e proprio linguaggio.
Come abbiamo appena detto, alla nascita HTML era un insieme di semplici regole per la formattazione del testo e la sua forza stava proprio nella semplicità. Questo pregio è stato perso a causa della continua guerra tra le case produttrici dei principali browser, che hanno cercato di personalizzarlo, gettando nella confusione gli sviluppatori che al momento della creazione non sanno mai come il loro lavoro verrà visualizzato sui vari browser.
Inoltre HTML è rivolto alla visualizzazione dei dati e non al loro trattamento e questo ne rappresenta un ulteriore grosso limite, forse il peggiore. Se, infatti, osserviamo un documento HTML non otteniamo nessun'indicazione sul significato dei dati in esso contenuti, cosa che invece non succede con XML.
Vediamo l'esempio[3] qui sotto riportato ed il risultato che con esso otteniamo così da chiarire meglio quanto appena detto:

```
<html>
 <head></head>
  <body>
   <TABLE border="1">
    <TR ALIGN="CENTER" BGCOLOR="#00ccff">
     <TD><B> Tipo di documento </B></TD>
     <TD><B> Autore</B></TD>
     <TD><B> Anno</B></TD>
     <TD><B> Titolo</B></TD>
    </TR>
    <TR ALIGN="CENTER">
     <TD><B> TESI</B></TD>
     <TD><B> Giacinto Montereale</B&gtltTD>
```

```
    <TD><B> 2003</B></TD>
    <TD><B> DB XML</B></TD>
   <TR>
  </TABLE>
 </body>
</html>
```

Tipo di documento	Autore	Anno	Titolo
TESI	Giacinto Montereale	2003	DB XML

figura 1

Tale codice si occupa solo di visualizzare la tabella di figura, ma, come notiamo, non c'è nessun tipo di associazione tra il codice e i dati stessi.
Se invece osserviamo il codice XML sotto riportato notiamo subito la profonda differenza con quello HTML:

```
<?xml version="1.0"?>
<documento>
 <tipo_di_documento>TESI</tipo_di_documento>
 <Autore>
  <Nome>Giacinto</Nome>
  <Cognome>Montereale</Cognome>
 </Autore>
 <Anno>2003</Anno>
 <Titolo>DB XML</Titolo>
</documento>
```

Si può notare la presenza di tag personalizzati che rendono immediatamente evidente la caratteristica di *estensibilità* di XML. Questa caratteristica riveste un'importanza fondamentale perché, in questo modo, riusciamo a capire il significato dei dati semplicemente guardando il codice del documento e risulta semplice accedere ai dati per poterli manipolare.
Come possiamo inoltre vedere, il codice XML si occupa semplicemente di racchiudere i dati in modo ordinato, o meglio strutturato, senza preoccuparsi della visualizzazione dei dati stessi. XML, infatti, si preoccupa di tenere ben separati tra di loro i tre aspetti fondamentali di un documento:

- il contenuto
- la struttura
- lo stile

È molto importante tener separate queste tre componenti, sia per avere un maggior controllo sul documento, sia perché una buona strutturazione del documento stesso dà un grosso aiuto a "leggerlo" meglio.

DOCUMENTO XML:

Per la creazione di un documento XML possiamo usare un qualsiasi editor di testo ma ne esistono di specializzati che ne semplificano la creazione.
Un documento XML[4] ha una struttura gerarchica ad albero (treelike) con una radice (root) e tanti nodi figli (child)

Una descrizione breve degli oggetti che compongono tale struttura è qui di seguito riportata:

- ELEMENTI:

Sono i “mattoni” principali di cui un documento XML è costituito,possono contenere testo,altri elementi,o essere vuoti.

I nomi degli elementi non possono essere qualsiasi ma devono rispettare alcune regole.

- TAG:

I tag vengono usati per delimitare gli elementi,un tag del tipo

<element_name>

delimita l’inizio di un elemento,mentre un tag del tipo:

</element_name>

delimita la fine di un elemento.

L’elemento vuoto ha un solo tag (di inizio e fine insieme) del tipo

<element_name />

infine occorre ricordare che i tag XML sono case sensitive ad esempio

<Lettera> diverso da <lettera>

- ATTRIBUTI:

Gli attributi forniscono un informazione extra su un elemento,sono posti all’interno del tag di inizio ed ognuno è formato da una coppia attributo –valore

Ad esempio

<memo data=”24/01/2001”>

<memo data=’24/01/2001’>

- ENTITA’:

Sono delle variabili per definire dei caratteri o sequenze di caratteri, che altrimenti

non potremmo utilizzare perché sono riservati.

XML definisce cinque entità che devono essere utilizzati al posto dei corrispondenti caratteri

1. & definisce il carattere &
2. &lt ; definisce il carattere <
3. 3&gt ; definisce il carattere >
4. 4&quot ; definisce il carattere “
5. 5&apos ; definisce il carattere ‘

Ma XML prevede anche la possibilità di definire entità personalizzate, difatti possiamo immaginare un entità XML come un alias , cioè un secondo nome che identifica un file o una risorsa il cui nome sarebbe troppo complicato da gestire.

- PCDATA – CDATA:

Per evitare che il parser interpreti porzioni di testo contenente ad esempio codice XML o altri caratteri riservati è possibile includere il testo in una sezione CDATA.
Una sezione CDATA è una porzione di testo racchiusa tra <![CDATA[‘contenuto’]]>.
Tutto ciò che è contenuto all’interno di questi delimitatori verrà ignorato dal parser.

- COMMENTI:

La sintassi dei commenti è
<!--This is a comment -- >

DTD, documenti “*validi*” e “*ben formati*”

Esiste un livello di correttezza dell'XML superiore alla semplice buona formattazione. Si chiama validazione.

A che serve la validazione ?

La validazione serve in tutti quei casi in cui XML è utilizzato per lo scambio di dati provenienti da fonti diverse.

Ad esempio, immaginiamo che c'è un fornitore di libri e le ordinazioni dai dettaglianti li giungono automaticamente al suo sistema nella forma di piccoli documenti XML,contenenti tutti i dati necessari per la transazione (esempio di e-commerce business-to-business). Se si vuole evitare che il sistema arbotisca la transazione, occorre che i messaggi in XML abbiano una struttura ben definita. Ad esempio, è plausibile che il riferimento ad un libro debba contenere anche il codice ISBN di quel libro.

La validazione tratta esattamente questo: appurare che un documento X ML verifichi certe regole, quale potrebbe essere per esempio l'esistenza di un elemento o di un attributo contenente il numero ISBN. Il metodo classico per definire regole di questo tipo passa attraverso la definizione di un nuovo tipo di documento chiamato DTD (Document Type Definition) quindi il DTD detta quali elementi sono obbligatori, quali opzionali, come possono (e non possono) essere annidati l'uno dentro l'altro e a quali elementi i diversi attributi appartengano (obbligatoriamente o facoltativamente).

Il DTD può essere sia interno al documento XML sia esterno:

- Validazione esterna

  < ? xml version = "1.0" ?>

  < ! DOCTYPE *nome del root* SYSTEM "validtubes.dtd">

  < *nome del root*>

  .

  La seconda linea indica che il documento deve rispettare le regole definite da un DTD esterno chiamato "validtubes.dtd"

- Validazione interna

  < ? xml version = "1.0" ?>

  < ! DOCTYPE *nome del root* [

  < ! ELEMENT *nome del root* ....

.

.

]>

I DTD non sono l'unico metodo per definire la struttura dei documenti. Microsoft, ad esempio, propone gli schemi (XML SCHEMA) che intendono essere un modo migliore per definire le regole di validità per documenti, poiché si basano ancora su XML in pratica la sintassi è ancora XML.
La validazione comunque non sempre è necessaria , difatti XML (a differenza di SGML) ammette anche la distribuzione di documenti privi di DTD,questi documenti sono definiti *ben formati* (well- formed).
Un documento XML si dice *ben formato* se è conforme ad una serie di regole generali:

- tutti i tag di apertura e di chiusura corrispondono;
- i tag vuoti utilizzano una sintassi XML speciale;
- tutti i valori degli attributi sono racchiusi tra singoli o doppi apici;
- tutte le entità sono dichiarate.

La violazione di una qualsiasi di queste regole fa in modo che il documento risultante non venga considerato ben formato. Su questi principi si basano i *parser XML*

PARSER XML

Un parser XML è un software autonomo o incluso in un'applicazione specifica come ad esempio un browser,il cui compito consiste nell'analizzare un documento XML e verificarne la correttezza.
In effetti un tool per leggere un documento XML consta di due parti: il Parser che esegue il controllo semantico e gestisce gli errori e il Processor che, utilizzando un altro file in cui è definita la formattazione dei vari tag, visualizza il documento.

Esistono due tipologie di parser:

- parser validante
- parser non validante

Il primo tipo, come dice il nome stesso, controlla la validità del documento in base alle regole sintattiche e alle informazioni sulla struttura contenute nel DTD. Il secondo si limita, invece, a controllare se il documento rispetta le regole sintattiche del linguaggio verificando in questo modo solo se il documento è ben formato.
Una volta scelto se utilizzare un parser validante o non validante possiamo suddividere i parser ancora in due categorie:

- Parsers che supportano il Document Object Model(DOM)
- Parsers che supportano le Simple API for XML (SAX)

Document Object Model & Simple API for XML

Il DOM [5] è un set di interfacce indipendenti dal particolare linguaggio che permette ai programmi di accedere e modificare dinamicamente contenuto, struttura e stile di un documento. I parsers che supportano il modello DOM implementano tale interfaccia.

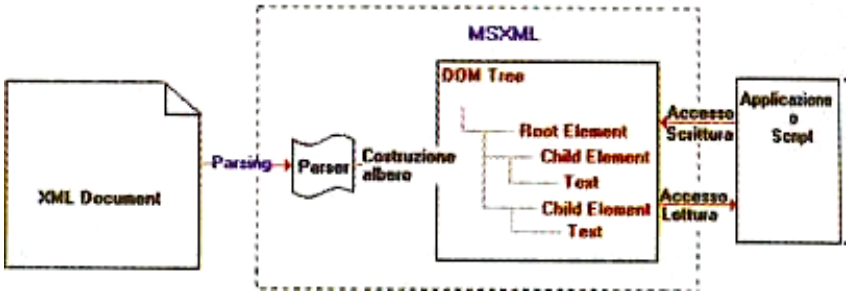


figura 3 : Mostra l'ambiente in cui opera MSXML

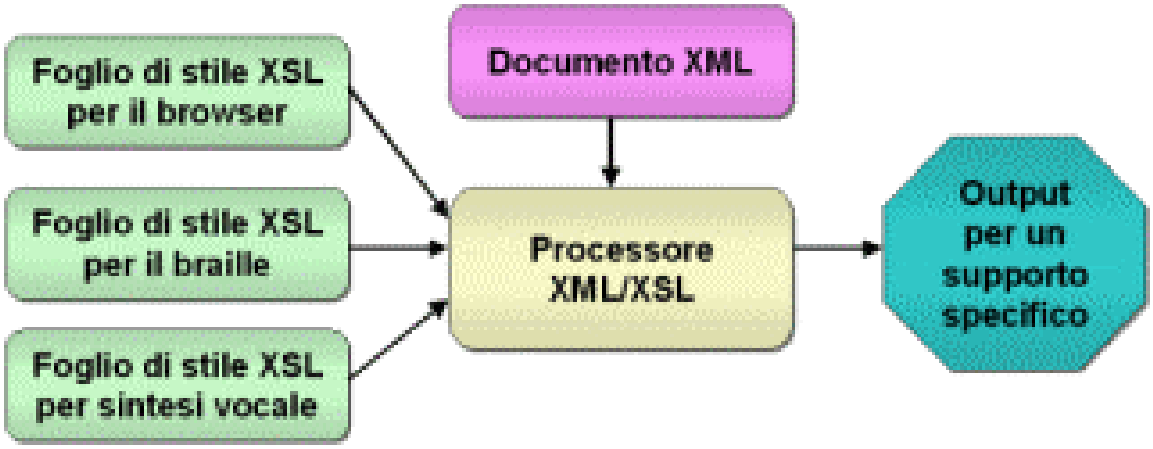
Durante l'elaborazione del documento, il parser costruisce esplicitamente in memoria una struttura ad albero che contiene tutti gli elementi del documento XML e quindi diventa possibile :

- navigare il documento spontadoci tra padre e figlio e tra fratelli
- accedere al contenuto e agli altri attributi
- svolgere operazioni fondamentali quali *selectSingleNode , selectNodes che* consentono l'esecuzione di una query espressa secondo la notazione XSL pattern e restituiscono il nodo (o i nodi) che la soddisfano.

Mentre con l'approccio DOM vengono elaborati tutti i nodi dell'albero, con l'approccio SAX il parser genera un evento ogni qual volta si verificano, durante la lettura del documento XML, determinate condizioni (ad esempio ad ogni occorrenza di un certo elemento o attributo o testo), e spetta al programmatore stabilire come reagire a tali eventi.

DOCUMENTO XSL:

La rappresentazione dei dati contenuti nel documento XML è delegata al documento XSL. L'XSL[4] si basa su un meccanismo di *fogli di stile* i quali vengono generalmente usati per applicare in modo coerente stili o formattazione ai documenti.



Dunque è evidente come XML mantenga separata la struttura e il contenuto dalla presentazione dei dati, questo ci consente innanzi tutto di veicolare un documento XML attraverso device diversi, inoltre permette a XML di proporsi come formato intermedio di scambio tra DB eterogenei, ma questo è solo uno dei nuovi scenari che si sono aperti nel mondo dei DB.

D'ora in poi il nostro obiettivo sarà quello di esplorare tutte le potenzialità offerte dall'accoppiata XML- DB.

CAPITOLO 1

XML e DB Relazionali

In questo capitolo affronteremo i vari aspetti riguardanti il rapporto tra XML e i db relazionali, in particolare faremo alcune considerazioni sulla progettazione dei db , e poi esamineremo qualche soluzione per poter tradurre documenti XML in tabelle e viceversa. Quest'ultimo aspetto è di fondamentale importanza per comprendere come ,e se, db relazionali e db xml nativi possono interagire.

1.1 Modellare i dati con XML

Il primo passo , dopo la fase di raccolta ed analisi dei requisiti della base dati ,è la progettazione concettuale ,tale fase consiste nella costruzione di uno schema in grado di descrivere la realtà di interesse.

In questo contesto, spiccano il modello Entità-Relazioni ed UML (Unifed Modeling Language),limitatamente ai diagrammi di classe.

Anche XML grazie alla possibilità di descrivere lo schema di un documento,ha dimostrato di poter essere utilizzato a questo scopo[6]. Fin dalla sua nascita, XML è stato affiancato da strumenti che permettessero di descrivere non soltanto le informazioni contenute nei documenti , ma anche la loro struttura.

Il primo di questi strumenti è stato il DTD (Document Type Definition), ma si è poi passati, negli ultimi anni, a XML-SCHEMA,un nuovo formato, proposto inizialmente da Microsft , che ha il pregio, rispetto a DTD , di essere a sua volta basato su XML e di permettere di definire il tipo di ognuno degli attributi del documento.

DTD e XML-SCHEMA servono, quindi , a descrivere la struttura dei dati, ovvero a modellarli.Il metodo utilizzato è molto simile a quello usato per il modello E-R, in pratica cambiano solo (o quasi) i vari elementi del modello:

- il primo passo è quello di identificare gli object type , ovvero i vari tipi di oggetti da prendere in considerazione durante la costruzione del modello del sistema preso in esame.Questi oggetti corrispondono all' entità del modello E-R.
- il secondo passo è quello di identificare le relazioni tra gli oggetti
- il terzo passo è quello di identificare i vari attributi di ogni oggetto e quindi di specificare per ognuno di essi il tipo di valori che può assumere.

ESEMPIO 1

La realtà che vogliamo rappresentare è quella di una Libreria, nella Libreria
troviamo Libri, Riviste, e Persone.
Il primo passo l'abbiamo quindi compiuto, avendo identificato gli object type, il passo successivo è quello di identificare il tipo di relazione che c'è tra questi oggetti, bene la relazione è di tipo gerarchico poiché i libri, le riviste nonché le persone, in particolare autori ed editori, sono subordinati all'esistenza della libreria; quanto detto può essere rappresentato nel DTD al modo seguente:

```
< ! DOCTYPE Libreria [
        < ! ELEMENT Libreria     ( Libro* , Rivista*, Persone* )>
```

l'elenco racchiuso tra ( ) viene definito *modello di contenuto* e identifica appunto gli elementi secondari che l'elemento Libreria deve contenere, e l'ordine in cui sono gli elementi.
Il modello di contenuto[4] è solo una tra le 4 possibili specifiche di contenuto di un elemento:

- Un elenco di altri elementi, denominato *modello di contenuto*
- La parola chiave *EMPTY*, per dichiarare che un elemento non può avere contenuto

- La parola chiave *ANY,*per dichiarare che quel elemento può avere qualsiasi tipo di contenuto in base alle disposizioni del DTD , disposto in un ordine qualsiasi.
- Contenuto di vario tipo, in tal caso le specifiche di contenuto possono essere costituite da un singolo insieme di alternative separate dal simbolo pipe (|)

Nel nostro caso, il Libro e la Rivista nono hanno elementi secondari quindi nel DTD ciò viene espresso al modo seguente:

```
< ! ELEMENT Libro      (EMPTY)>
 < ! ELEMENT Rivista      (EMPTY)>
```

A questo punto rimane da identificare i vari attributi di ogni oggetto, nel linguaggio XML gli attributi vengono dichiarati nel DTD utilizzando la sintassi seguente:

```
< ! ATTLIST   ElementName    AttributeName    Type    Default>
```

In questo caso < ! ATTLIST > rappresenta il tag che identifica una dichiarazione di attributo . La voce *ElementName* rappresenta il nome dell' elemento a cui vengono applicati gli attributi .
La voce *AttributeName* rappresenta il nome dell'attributo. La voce *Type* identifica il tipo di attributo dichiarato.
La voce *Default* specifica le impostazioni predefinite relative all'attributo.
Ci sono 10 tipi di attributo XML che possiamo sintetizzare al modo seguente:

- Un attributo di tipo carattere (CDATA);
- Sette tipi di attributi tokenized, chiamati così perché il loro valore rappresenta uno o più token (pensiamo ad un token come ad un'unità di informazione);
- Due tipi di attributi elencati (Elenco, NOTATION);

Per il nostro esempio è sufficiente specificare solo alcuni di questi :

ATTRIBUTO CDATA:

E' l'attributo di tipo più semplice, costituito solo da dati in formato carattere, come può essere il titolo di un libro:

< ! ATTLIST Libro titolo CDATA >

ATTRIBUTI ID & IDREF:

Ogni elemento può avere un solo ID [7] cioè il valore dell'attributo deve essere un' identificatore univoco. Se un documento contiene attributi ID con lo stesso valore, l'elaboratore produrrà un errore.

Gli attributi di tipo IDREF sono attributi il cui valore deve riferirsi ad un ID dichiarato in un altro punto del documento. È però possibile avere più di un attributo IDREF che si riferisce allo stesso ID , anzi è possibile averne quanti ne sono necessari, a propria discrezione.

Per avere tanti IDREF che "puntano" allo stesso elemento si deve dichiarare un attributo di tipo IDREFS.

Nel nostro esempio, volendo imporre che gli autori e gli editori si riferiscono all' elemento 'Persone' e cioè che i valori (nomi) degli editori e degli autori siano compresi in quelli che compaiono in Persone dobbiamo dichiarare gli attributi Libro , Rivista e Persone al modo seguente:

< ! ATTLIST Libro autori IDREFS >
< ! ATTLIST Rivista editore IDREF >
< ! ATTLIST Persone personeID ID >

La parte finale della dichiarazione di attributo è l'impostazione predefinita che può essere di 4 tipi:

- #REQURED: Ogni elemento contenente questo attributo deve specificarne un valore. Un valore mancante può causare un errore

- #IMPLIED: Questo attributo è opzionale , l'elaboratore può ignorare questo attributo se non viene rilevato alcun valore;
- #FIXED: Questo attributo deve avere il valore *fixedvalue*. Se l'attributo non è incluso nell'elemento, viene stabilito il valore *fixedvalue;*
- Default: Identifica un valore predefinito per un attributo, se l'elemento non include l'attributo, viene stabilito il valore *default*

Tenendo presente anche della parte finale della dichiarazione di attributo possiamo concludere la descrizione della nostra Libreria .
Il documento DTD finale che descrive tale realtà è il seguente:

```
< ! DOCTYPE Libreria [
      < ! ELEMENT  Libreria      ( Libro* , Rivista*, Persone* )>
       < ! ELEMENT  Libro        ( EMPTY)>
       < ! ATTLIST    Libro      titolo CDATA  #REQUIRED  autori IDREFS
                                 #REQUIRED      publicdata   CDATA
                                  #REQUIRED >
        < ! ELEMENT  Rivista     ( EMPTY)>
        < ! ATTLIST    Rivista   titolo CDATA  #REQUIRED  editore IDREF
                                  #IMPLIED      publicdata   CDATA
                                  #IMPLIED >
         < ! ELEMENT  Persone     ( Sposo?, Persone* )>
          < ! ATTLIST   Persone     personeID    ID  #REQUIRED   nome
                                    CDATA  #IMPLIED>
          < ! ELEMENT  Sposo        (#PCDATA)>
]>
```

La stessa realtà è possibile rappresentarla in modo più efficiente utilizzando XML-SCHEMA [8], infatti, esso permette di ovviare ai seguenti tre principali difetti del DTD:

- Non è un linguaggio XML ,invece la sintassi degli Schemi XML, è XML, quindi sia il documento XML che lo Schema possono essere scritti utilizzando lo stesso strumento e inoltre non occorre imparare una nuova sintassi,cosa che invece dovevamo fare se utilizzavamo i DTD ;
- Non gestisce i tipi di dato, dunque non è possibile sapere se le singole informazioni siano da interpretare come numeri o stringhe ecc…. Ad esempio un DTD può garantire l'esistenza di un elemento DATANASCITA, però non può controllare che il contenuto sia in effetti una data, e non, ad esempio, una cosa del tipo:

  <DATANASCITA> il mattino </DATANASCITA>

  Gli Schemi supportano invece la definizione di tipi di dato e quindi possiamo avere un controllo maggiore sui dati immessi;
- Non gestisce in alcun modo i namespace, che consentono al documento XML di contenere altri documenti.

  Gli Schemi hanno un *content model* aperto, nel senso che possono validare documenti che contengono tag estranei (ovvero elementi provenienti da namespace diversi)

Vediamo lo SCHEMA[9] per la nostra Libreria, essa è composta da riviste, libri, persone, quindi è un elemento complesso formato da una sequenza di altri elementi , tale tipo di dato viene etichettato come *complexType*, precisamente abbiamo:

```
<element name =’Libreria’>
  <complexType>
    <sequence>
      <element ref=’t:Libro’ minOccurs=’0’  maxOccurs=’unbounded’/>
      <element ref=’t:Rivista’ minOccurs=’0’  maxOccurs=’unbounded’/>
       <element ref=’t:Persone’ minOccurs=’0’  maxOccurs=’unbounded’/>
    </sequence>
```

```
    </complexType>
  </element>
```

Notiamo che con la sintassi

```
<element ref=’nome_elemento’/>
```

vogliamo fare riferimento ad un elemento definito globalmente in un punto qualsiasi dello SCHEMA XML , difatti abbiamo scelto di elencare innanzitutto gli elementi che compongono la Libreria e poi di andarli a definire come segue:

```
<element name='Libro'>
<complexType>
  <sequence> <element ref='t:EMPTY'/> </sequence>
  <attribute name='titolo' type='string' use='required'/>
  <attribute name='autori' type='IDREFS' use='required'/>
   <attribute name='publicationDate' type='date' use='required'/>
</complexType>
</element>
<element name='Rivista'>
  <complexType>
    <sequence> <element ref='t:EMPTY'I> </sequence>
     <attribute name='titolo' type='string' use='required'/>
     <attribute name='edjtore' type='IDREF' use='optional'/>
      <attribute name='publicationDate' type='date' use='optional'/>
</complexType>
</element>
<element name=’ Person ' >
  <complexType>
   <sequence>
   <element ref='t:Spouse' minOccurs='0' maxOccurs='1/>
```

<element ref='t:Person' mìnOccurs='0' maxOccurs='unbounded'/>
</sequence>
<attribute name='personID' type='ID' use='requìred'/>
<attrjbute name='name' type='string' use='optìonal'/>
</complexType>
</element>
<element name='Spouse'> <complexType mixed='true'/> </element>

In definitiva lo SCHEMA XML che rappresenta la nostra Libreria è il seguente:

<schema xmlns:t=’http://www.w3.org/namespace/’>
<element name =’Libreria’>
<complexType>
<sequence>
<element ref=’t:Libro’ minOccurs=’0’ maxOccurs=’unbounded’/>
<element ref=’t:Rivista’ minOccurs=’0’ maxOccurs=’unbounded’/>
<element ref=’t:Persone’ minOccurs=’0’ maxOccurs=’unbounded’/>
</sequence>
</complexType>
</element>
<element name='Libro'>
<complexType>
<sequence> <element ref='t:EMPTY'/> </sequence>
<attribute name='titolo' type='string' use='required'/>
<attrìbute name='autori' type='IDREFS' use='requìred'/>
<attribute name='publìcatìonDate' type='date' use='required'/>
</complexType>
</element>
<element name='Rivista'>
<complexType>
<sequence> <element ref='t:EMPTY'I> </sequence>
<attribute name='titolo' type='string' use='required'/>

```
      <attribute name='edjtore' type='IDREF' use='optional'/>
       <attribute name='publicationDate' type='date' use='optìonal'/>
</complexType>
</element>
<element name=’ Person ' >
  <complexType>
   <sequence>
   <element ref='t:Sposa' minOccurs='0' maxOccurs='1/>
      <element ref='t:Person' mìnOccurs='0' maxOccurs='unbounded'/>
</sequence>
<attribute nome='personID' type='ID' use='requìred'/>
<attrjbute nome='nome' type='string' use='optìonal'/>
</complexType>
</element>
<element name='Sposa'> <complexType mixed='true'/> </element>
</schema>
```

Abbiamo presentato i due modi possibili per modellare i dati utilizzando XML. In realtà l’uso più comune che si fa del DTD e di XML-SCHEMA è quello per la validazione dei documenti XML ; a tal proposito un possibile documento XML che risulta valido secondo lo SCHEMA-XML (quindi che rispetta la struttura specificata nello SCHEMA) è il seguente:

```
<libreria>

  <libro titolo="Data Strluctures and Algorithms " autori="aho hopcroft
        ullman" publicationDate="Genn, 1983"/>

 <libro titolo="Principles of Compiler Design" autori="aho ullman"
        publicationDate="1979"/>
```

```
<libro titolo="Introduction to Automata Theory" autori="hopcroft ullwan"
        publicationDate="1979"/>

<rivista titolo="Communication of ACM" editore=" aho" />

<rivista titolo="IEEE, Cowp." editore="ullman" publicationDate="Sett,2000"/>

<person personID="aho" nome="Alfred. V. Aho"/>
    <sposa>WifeOfAho</sposa>
  <person personID="sonl" name="Junior_l Aho"/>
  <person personID="son2" name="Junior_2 Aho"/>
</person>

<person personID="ullman" name="Jeffrey. D. Ullman">
    <sposa>WifeOfUllman</sposa>
 </person>

 <person personID="hopcroft" name="John. E. Hopcroft"/>
</1ibreria>
```

1.2 Le Insidie del Data Modeling con XML

Durante la progettazione occorre tenere conto del fatto che XML permette di modellare in modo naturale le relazioni di tipo gerarchico[10], mentre non offre un supporto diretto per altri tipi di relazioni, quindi comporta le stesse anomalie del modello gerarchico:

- *Anomalia di inserimento*: non è possibile inserire un'entità figlia senza un'associazione a quella madre (p.e. non è possibile inserire i dati dei nuovi assunti, prima di aver loro assegnato un reparto);

- *Anomalia di cancellazione* : non è possibile eliminare la relazione tra entità madre e figlia se non eliminando l'entità figlia (perdendone quindi le informazioni);
- *Asimmetria delle query*: per conoscere le informazioni sulle entità figlie data un'entità madre, l'accesso è veloce, mentre la conoscenza dell'entità madre data un'entità figlia, richiede la scansione di tutte le entità madri per verificare se queste hanno l'entità figlia specificata.

A differenza però del modello gerarchico possiamo definire più documenti XML che contengono le informazioni sulle diverse entità ,implementando poi l'associazione gerarchica attraverso delle chiavi esterne:

```
<Madre>
        <attributo_1>…..</ attributo_1>
        …..
        <attributo_n>…..</ attributo_n>
        <figlie>
            <attributo_chiave_figlia_1>
            <attributo_chiave_figlia_2>

            <attributo_chiave_figlia_m>
        </figlie>
</Madre>
<Figlia>

<attributo_chiave>….  </ attributo_chiave>
        <altri _attributi>
</Figlia>
```

……(altre istanze di figlia)

Questa soluzione comporta le seguenti problematiche:

- Occorre decidere , in fase di progettazione ,se inserire le entità in un unico documento (per facilitare le interrogazioni in linguaggi come XQL) oppure in documenti separati (per gestire meglio le attività di inserimento e cancellazione , nonché di condivisione delle entità). In entrambi i casi i programmi sono influenzati dalla strutturazione fisica dei dati (p.e. si ha dipendenza dei programmi dai dati) ;
- Qualunque sia la soluzione adottata , le associazioni sono *sempre* definite esplicitamente e realizzate fisicamente . Il noto modello basato sui valori, che attua la vera indipendenza dei programmi dai dati , non è *realizzabile in pratica* con XML . Esso potrebbe essere realizzato *in teoria* , attraverso la definizione di documenti XML non strutturati, cioè ogni documento XML conterebbe solo gli attributi dell'entità e delle associazioni che si desiderano rappresentare.In questo modo ,vi sarebbero tanti documenti XML quante sono le entità e le associazioni. Tuttavia, questo modello si avvicinerebbe ad una implementazione (inefficiente) di una base di dati relazionale e non si avvantaggerebbe delle caratteristiche del linguaggio XML;
- Agli svantaggi dovuti alla forte vicinanza al modello gerarchico, occorre aggiungere la povertà del linguaggio XQL, che non prevede il meccanismo di join tra documenti diversi (evidenza della mancanza del modello basato sui valori), oltre all'assenza di tutti i meccanismi forniti da un linguaggio più consolidato come SQL.

XML risulta ,quindi, un ottimo strumento di rappresentazione , ma visto che la modellizzazione dei dati che ne consegue è molto vicina a quella gerarchica , il suo uso diventa più proficuo qualora la base dati modellata, venga successivamente implementata su un database non relazionale (database object-oriented)

1.3 XML & EER

Abbiamo osservato che disponiamo di diversi strumenti per modellare i dati, crediamo quindi che sia significativo comprendere come si può passare da l'uno all'altro, in particolare da schema XML al modello Entità– Relazione e viceversa.

A tal proposito ci riferiamo ad una nuova descrizione formale XGrammar , la quale nasce dal tentativo di formalizzare in qualche modo le caratteristiche più importanti dei linguaggi per strutturare i dati (DTD , XML-SCHEMA).

XGrammar , il cui simbolo è G, è un'estensione della "Regular Tree Grammar" [9].Si compone di sette elementi:

$$G = (N_T , N_H , T , S , E, H , A)$$

le cui specifiche sono quelle di [9].

Le tre principali caratteristiche di XGrammar che aiutano a modellare i dati utilizzando XML sono le seguenti:

- *Associazioni binarie ordinate:* XGrammar può rappresentare associazioni binarie ordinate usando: 1) Associazioni elemento-sottoelemento e 2) gli attributi IDREFS.

  Tali associazioni si trovano spesso nel mondo reale ,per esempio gli autori di un libro sono in genere ordinati ,c'è un primo autore ,un secondo autore,

  un terzo e così via. L' estensione del modello E-R (EER) permette di rappresentare le relazioni binarie ordinate.

- *Unione di tipi:* DTD e XML-SCHEMA sono chiusi rispetto all' intersezione ma non sono chiusi rispetto all'operazione di differenza e unione quindi ,non consentono di definire un nuovo tipo a partire dalla differenza o dall'unione , invece XGrammar consente di fare ciò e questo risulta utile per molti problemi di integrazione di dati.

- *Tipi ricorsivi:* XGrammar può rappresentare relazioni ricorsive usando la definizione di tipi ricorsivi .

La strutturazione dei dati in XML, a differenza del modello E-R, prevede due differenti modalità di ricorsione, quella semantica e quella strutturale, XGrammar è in grado di esprimerle entrambe usando rispettivamente la nozione di *element production rule* e la nozione di *attribute production rule.*Per esempio consideriamo la relazione di subordinazione che c'è tra l'impiegato (employee) e il suo capo (manages), volendola rappresentare con il modello E-R dobbiamo tener presente che sia l'impiegato che il capo sono entrambi impiegati ciò vuol dire che una certa occorrenza dell'entità impiegato è collegata a qualche altra occorrenza dell' entità impiegato quindi la rappresentazione avviene attraverso un' associazione ricorsiva:

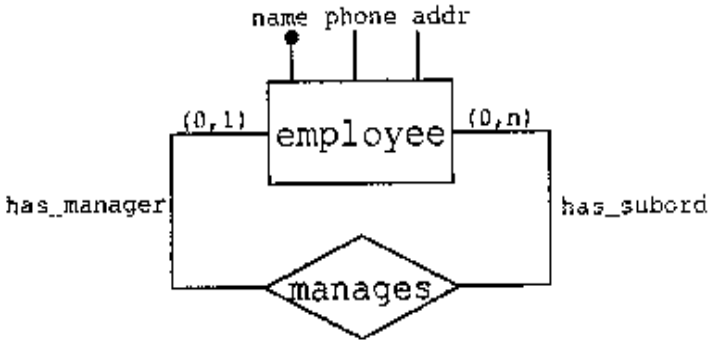


Fig. 1 Un esempio di ricorsione nel modello E-R

Questo schema E-R può essere rappresentato in due diversi modi utilizzando XML in particolare attraverso due DTD:

DTD (a)

```
< ELEMENT employee  (EMPTY)
<ATTLIST    employee
                name   ID          #REQUIRED
                phone  CDATA     #IMPLIED
                addr   CDATA     #IMPLIED
```

subord IDREFS #IMPLIED >

Tale DTD rappresenta la ricorsione dal punto di vista semantico attraverso l'attributo subord ,il cui valore (nome) è tra quelli dei nomi di tutti gli impiegati.

DTD (b)

< ELEMENT employee (subord*)

<ATTLIST employee

name ID #REQUIRED

phone CDATA #IMPLIED

addr CDATA #IMPLIED

subord IDREFS #IMPLIED >

< ELEMENT subord (employee)>

Tale DTD rappresenta la ricorsione dal punto di vista strutturale poiché l'elemento employee può contenere come elemento secondario l'elemento subord il quale a sua volta può contenere l'elemento employee .

Entrambe la DTD possono essere tradotte in XGrammar ,precisamente attraverso l'attributo production rule viene tradotta la ricorsione rappresenta dal DTD(a):

$A = \{ \text{Employee} \rightarrow \text{employee} (@\text{subord}^{?} :: \text{IDREFS} \rightsquigarrow \text{Employee}^{+}) \}$

Mentre la ricorsione rappresentata dal DTD(b) viene tradotta utlizzando l'element production rule :

E ={ Employee → employee (Subord*), Subord → subord(Employee)}

ESEMPIO 2

Date le rappresentazioni attraverso DTD e XML-SCHEMA, della realtà 'Libreria' (ESEMPIO 1) vogliamo mostrare come queste possano essere tradotte

in XGrammar.

La traduzione avviene individuando le varie componenti di XGrammar secondo le specifiche [9]:

N_T = {Libreria, Book, Magazine, Person, Spouse}

T = {libreria, book, magazine, person, spouse, title, authors, editor,

publicationDate, personID, name}

S = {Libreria}

E = {Libreria → libreria (Book*, Magazine*, Person*), Book → book (ε),

Magazine → magazine (ε), Person → person (Spouse $^{?}$, Person*),

Spouse → spouse (string) }

A = { Libreria → libreria (ε), Book → book (@title :: string,

@authors :: IDREFS⇝ Person^{+}, @publicationDate :: date),

Magazine → magazine (@title ::string,

@editor $^{?}$ :: IDREF⇝ Person,

@publicationDate $^{?}$ :: date),

Person → person (@personID :: ID, @name $^{?}$ :: string),

Spouse → spouse (ε) }

Quindi le rappresentazioni tramite DTD e XML-SCHEMA, vengono tradotte in $G_1 = (N_T , T, S, E, A)$.

Conversione da XGrammar a EER

Il passaggio da una XGrammar (e quindi da un DTD o XML-SCHEMA) ad un modello EER viene effettuato tramite la conversione dei *tree type,* degli *element production rule,* e degli attributi IDREF- IDREFS:

- *Tree Type:* Tutti i tre type (elementi di N_T) sono rappresentati come ENTITA'. Nel caso della G_1 abbiamo cinque *Tree Type* per cui abbiamo cinque entità:

  {Libreria, Book, Magazine, Person, Spouse}

- *Element production rule:* Per tutti i "figli" degli *Element production rule* (ossia dell'insieme E) è creata un' associazione ordinata di nome 'has' .

  Le cardinalità dei 'parent e child type' vengono fissati in base ai simboli "," , "+" , "*" , "?" .

  Per esempio consideriamo l' *Element production rule:*

  Libreria $\rightarrow$ libreria (Book*, Magazine*, Person*)

  In tale elemento individuiamo tre child type :

  Book*, Magazine*, Person*

allora tale elemento sarà rappresentato nel modello EER con tre associazioni ordinate di nome 'has' :

1. Library (1,1) $\Rightarrow$ Book (0,*)
2. Library (1,1) $\Rightarrow$ Magazine (0,*)
3. Library (0,1) $\Rightarrow$ Person (0,*)

Notiamo che poiché Book e Magazine possono trovarsi nel documento solo come 'child' di Library , difatti si trovano solo in Library $\rightarrow$ library (Book*, Magazine*, Person*), allora le cardinalità di Library in tali associazioni sono (1,1); invece Person può trovarsi nel documento sia come 'child' di Library che come 'child' di Person, difatti si trova in Library $\rightarrow$ library (Book*, Magazine*, Person*) e in Person $\rightarrow$ person (Spouse$^?$, Person*), quindi le cardinalità di Library in tale associazione sono (0,1).

- *Attributo IDREF:* L'attributo IDREF è rappresentato da un'associazione binaria non ordinata tra il tipo che specifica l'attributo ed il tipo a cui fa riferimento (tipo target).

  Le cardinalità del tipo che specifica l'attributo sono (0,*) mentre le cardinalità del tipo target variano a seconda dei casi:

  CASO 1: Il tipo target di un' attributo IDREF è l'unione di almeno due dei tree type . Per esempio supponiamo di avere il tipo X così specificato:

  $$X \rightarrow x (@ A :: IDREF \rightsquigarrow (B|C))$$

X ha un attributo IDREF di nome A , il quale ha come tipo target: (B|C) , allora l'attributo IDREF verrà rappresentato con due associazioni non ordinate, di nome A:

$X\ (0,*) \rightarrow B\ (0,1)$

$X\ (0,*) \rightarrow C\ (0,1)$

CASO 2: Il tipo target di un' attributo IDREF è solo uno dei tree type, inoltre l'attributo è opzionale.Questo è il caso dell' attributo editor del tree type Magazine:

@editor $^{?}$ :: IDREF$\rightsquigarrow$ Person

in tal caso l'attributo IDREF viene rappresentato da un'associazione non ordinata di nome 'edited':

Magazine (0,*) $\rightarrow$ Person (0,1)

CASO 3 : Il tipo target di un' attributo IDREF è solo uno dei tree type , inoltre l'attributo IDREF è obbligatorio. Ad esempio possiamo avere per il tipo X l'attributo di nome A così specificato:

@ A :: IDREF$\rightsquigarrow$ (B)

in tal caso l'attributo viene rappresentato da un'associazione non ordinata di nome A:

X (0, *) $\rightarrow$ B (1,1)

- *Attributo IDREFS:* L'attributo IDREFS è rappresentato da un'associazione binaria ordinata tra il tipo che specifica l'attributo ed il tipo a cui fa riferimento (tipo target).

  Come per l'attributo IDREF , le cardinalità del tipo che specifica l' attributo sono (0, *), invece le cardinalità del tipo target variano a seconda dei casi :

CASO 1: L 'attributo IDREFS è opzionale (?). Per esempio supponiamo di avere il tipo X così specificato

$$X \rightarrow x\ (\ @\ A^{?} :: \text{IDREFS} \rightsquigarrow B^{+}\ )$$

X ha un attributo IDREFS di nome A che risulta opzionale (?), allora l'attributo viene rappresentato dalla seguente associazione di nome A:

$$X\ (\ 0, *) \rightarrow B\ (0, *)$$

CASO 2: L'attributo IDREFS è obbligatorio, in tal caso viene rappresentato da un' associazione binaria ordinata .

Per esempio consideriamo l'attributo authors del tree type Book :

$$@\text{authors} :: \text{IDREFS} \rightsquigarrow \text{Person}^{+}$$

Tale attributo viene rappresentato nel modello EER da un' associazione di nome authored:

$$\text{Book}\ (0, *) \Rightarrow \text{Person}\ (1, *)$$

Data quindi la XGrammar $G_I = (N_T , T, S, E, A)$, se applichiamo le conversioni anzidette otteniamo la rappresentazione della realtà: Library, secondo il modello EER:

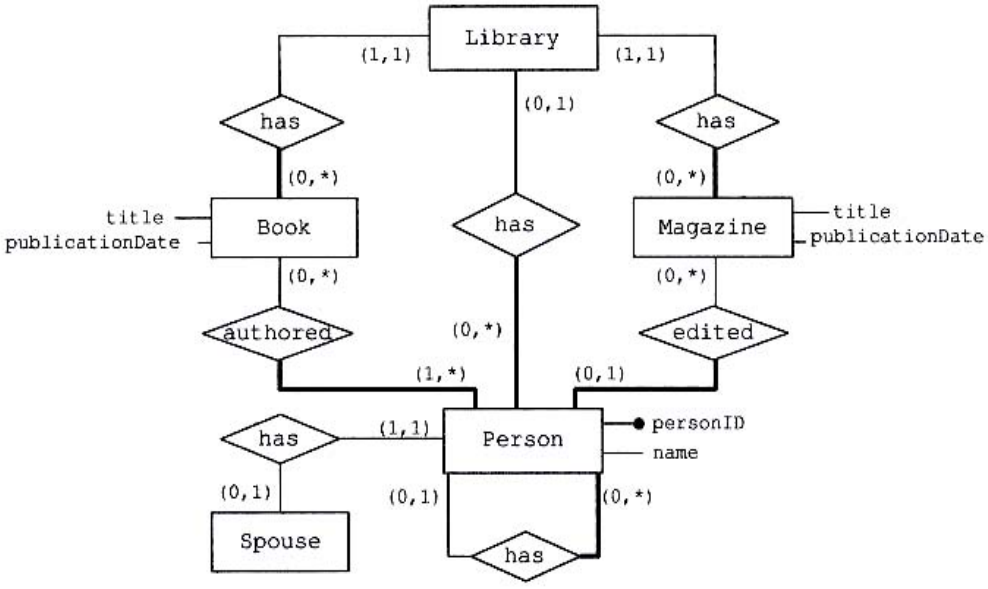


fig. 2 Esempio di modello EER

Siamo quindi passati, utilizzando XGrammar , da una rappresentazione della realtà secondo uno schema XML , ad una rappresentazione secondo il modello EER.

Conversione dal modello EER ad XGrammar

Prima di vedere come avvenga la traduzione dal modello EER ad XGrammar è bene osservare che uno schema XML supporta vincoli come quelli di *key* e di *foreign key*.Basandoci su questi definiamo i *"joinable tree types" – tree type* che hanno un vincolo di key – foreign key .

La traduzione di un dato modello EER in XGrammar si ottiene attraverso i seguenti passi:

1. Ciascuna Entità del modello EER viene tradotta in XGrammar in tree type, per esempio, riferendoci allo schema di fig. 2, avremo un tree type per Book, un tree type per Magazine e cosi via.

   Tutti gli attributi semplici possono essere tradotti in XGrammar in attributi dei tree type .

   Un attributo composto può essere tradotto in un subelement del tree type .

   Ad esempio nel caso del modello di fig.2 abbiamo un entità Book con due attributi semplici: title e publicationdate.

   L'entità Book viene tradotta nel tree type Book e i suoi attributi vengono tradotti in attributi del tree type attraverso la seguente "attribute production rule":

   $$\text{Book} \rightarrow \text{book (@title, @publicationDate)}$$

2. Consideriamo la seguente associazione ordinata($\Rightarrow$)

   $$A\ (m_1, M_1) \Rightarrow B\ (m_2, M_2)$$

   con $M_1 = M_2 = 1$.

   Se $m_1 = 0$ e $m_2 = 0$, allora tale associazione può essere tradotta in

   XGrammar in due modi differenti, un tree type con IDREF, oppure un

   tree type con un vincolo di foreign key (joinable tree types).

Se $m_1 = 1$ (o anolagamente $m_2 = 1$) allora l'associazione può essere tradotta usando la relazione element- subelement con B "figlio" di A o viceversa.Nel caso che $m_1 = 1$ e $m_2 = 1$, allora il content model per A specifica B, altrimenti se $m_2 = 0$, allora il content model specificherà $B^?$.

Nello schema EER illustrato in fig .2 abbiamo l'associazione:

$$\text{Person}(1,1) \Rightarrow \text{Spouse}(0,1)$$

quindi A = Person, B = Spouse, $m_1 = 1$, $m_2 = 0$, pertanto tale relazione in XGrammar viene tradotta come una relazione element-subelement con $\text{Spouse}^?$ subelement di Person.

3. Consideriamo l'associazione in cui $M_1 = 1$, $M_2 = n$; se l'associazione risulta ordinata possiamo tradurla in XGrammar con una relazione element-subelement oppure con l'attributo IDREFS.

   Per esempio l'associazione:

   $$\text{Library}(1,1) \Rightarrow \text{Book}(0,*)$$

   viene tradotta in XGrammar con una relazione in cui Book^* è un subelement di Library.

   Se l'associazione non è ordinata allora viene tradotta in XGrammar usando l'attributo IDREFS oppure usando i joinable tree types. Ad esempio l'associazione:

   $$\text{Magazine}(0,*) \rightarrow \text{Person}(0,1)$$

   Viene tradotta in XGrammar definendo un'atributo IDREFS, di nome editor, per il tipo Magazine:

   $$@\text{editor}^? :: \text{IDREF} \rightsquigarrow \text{Person}$$

4. Consideriamo l'associazione in cui M_1 = n, M_2 = m; se l'associazione risulta ordinata possiamo tradurla in XGrammar usando l'attributo IDREFS.

   Per esempio l'associazione:

   $$\text{Book}(0,*) \Rightarrow \text{Person}(0,*)$$

   viene tradotta in XGrammar definendo un attributo IDREFS,di nome authors, per il tipo Book:

   $$@\ \text{authors}^{?} :: \text{IDREF} \rightsquigarrow \text{Person}^{+}$$

   Se l'associazione non è ordinata, viene rappresentata in XGrammar utilizzando i joinable tree types.

5. Consideriamo un'associazione ricorsiva, è noto che in XGrammar possiamo rappresentare un'associazione ricorsiva sia dal punto di vista semantico che dal punto di vista strutturale.

   Ad esempio data l'associazione ricorsiva:

   $$\text{Person}(0,1) \Rightarrow \text{Person}(0,*)$$

   Volendola rappresentare dal punto di vista strutturale definiamo in XGrammar una relazione element-subelement in cui Person^{*} è subelement di Person.

Le fasi anzidette sono sintetizzate nella seguente tabella:

Relationship Type	Ordered/Unordered	How to represent them
1 : 1	-	{element-subelement, IDREF, joinable tree types}
1 : *n*	Ordered	{element-subelement, IDREFS}
1 : *n*	Unordered	{IDREF, joinable tree types}
n : *m*	Ordered	{IDREFS}
n : *m*	Unordered	{joinable tree types}

ESEMPIO 3

Consideriamo il seguente schema EER:

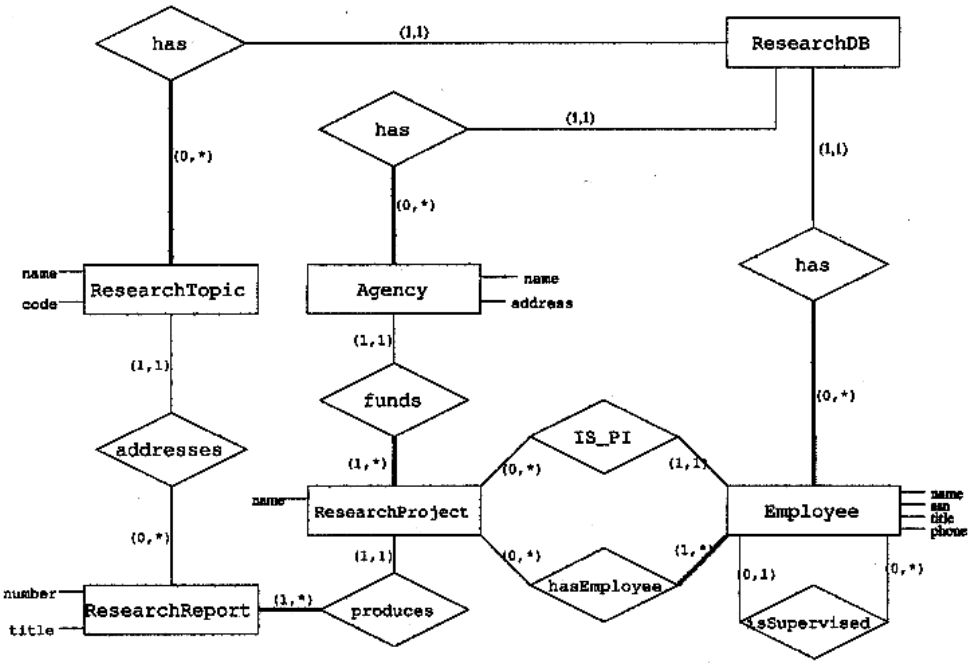


fig. 3

Applicando le fasi della conversione dal modello EER vogliamo tradurre tale schema in XGrammar.

La prima fase è quella di tradurre ogni entità dello schema EER in un tre type, quindi abbiamo:

N_T = {ResearchDB, Agency, ResearchTopic, Employee, ResearchProject,

ResearchReport}

Di questi tree type identifichiamo, osservando lo schema di fig.3, il root tree type

ResearchDB.

La fase successiva è quella di considerare le associazioni presenti nello schema EER:

Type	Order	Relationships
$1:n$	Ordered	$\{Agency(1,1) \overset{funds}{\Longrightarrow} ResearchProject(1,*),$
		$ResearchProject(1,1) \overset{produces}{\Longrightarrow} ResearchReport(1,*),$
		$ResearchDB(1,1) \overset{has}{\Longrightarrow} ResearchTopic(0,*),$
		$ResearchDB(1,1) \overset{has}{\Longrightarrow} Employee(0,*),$
		$ResearchDB(1,1) \overset{has}{\Longrightarrow} Agency(0,*)\}$
$n:1$	Unordered	$\{ResearchReport(0,*) \overset{addresses}{\longrightarrow} ResearchTopic(0,1),$
		$Employee(0,*) \overset{isSupervised}{\longrightarrow} Employee(0,1),$
		$ResearchProject(0,*) \overset{IS\_PI}{\longrightarrow} Employee(1,1)\}$
$n:m$	Ordered RHS	$\{ResearchProject(0,*) \overset{hasEmployee}{\Longrightarrow} Employee(1,*)\}$

e di tradurle in base al tipo, precisamente traduciamo tutte le associazioni 1: n in cui c'è un ordine con la relazione element-subelement; le associazioni non ordinate n : 1 vengono invece tradotte utilizzando l'attributo IDREF, mentre le associazioni n : m in cui c'è un ordine sono rappresentate utilizzando l'attributo IDREFS.

In definitiva la XGrammar risultante $G_2 = (N_T, T, S, E, A,)$ è la seguente:

N_T = {ResearchDB, Agency, ResearchTopic, Employee, ResearchProject,

ResearchReport}

T={researchDB, agency, researchTopic, employee, researchProject,

researchReport}

S = { ResearchDB }

E = {ResearchDB → researchDB(Agency*, ResearchTopic*, Employee*),

Agency → agency(ResearchProject+),

ResearchProject→ researchProject (ResearchReport+),

Employee →employee(ε), ResearchReport →researchReport(ε),

ResearchTopic→ researchTopic(ε) }

A = {ResearchDB → researchDB(ε),Agency → agency(@name :: string,
@address :: string),
ResearchTopic→ researchTopic(@name :: string, @code :: integer),
ResearchProject → researchProject(@name :: string,
@P I :: IDREF⇝ Employee,
@members :: IDREFS⇝ Employee^{+}),
ResearchReport → researchReport(@number :: integer, @title :: 8tring,
@topic :: IDREF⇝ ResearchTopic),
Employee→ employee(@name :: string, @ssn :: integer, @title :: string,
@phone :: integer,@supervisor$^{?}$ :: IDREF⇝ Employee)}

1.4 Conversione di uno schema XML in uno schema relazionale e viceversa

Dato uno schema XML (espresso tramite DTD o XML-SCHEMA), si può avere l'esigenza di tradurlo in uno schema relazionale[9], a tal proposito introduciamo l'algoritmo *Constraint-preserving Inlining Algorithm* (CPI) .

Consideriamo ad esempio il seguente schema XML

```
<!ELEMENT conf     (title,date,editor?,paper*)>
<!ATTLIST conf     id      ID            #REQUIRED>
<!ELEMENT title    (#PCDATA)>
<!ELEMENT date     EMPTY>
<!ATTLIST date     year    CDATA         #REQUIRED
                   mon     CDATA         #REQUIRED
                   day     CDATA         #IMPLIED>
<!ELEMENT editor   (person*)>
<!ATTLIST editor   eids    IDREFS        #IMPLIED>
<!ELEMENT paper    (title,contact?,author,cite?)>
<!ATTLIST paper    id      ID            #REQUIRED>
<!ELEMENT contact  EMPTY>
<!ATTLIST contact  aid     IDREF         #REQUIRED>
<!ELEMENT author   (person+)>
<!ATTLIST author   id      ID            #REQUIRED>
<!ELEMENT person   (name,(email|phone)?)>
<!ATTLIST person   id      ID            #REQUIRED>
<!ELEMENT name     EMPTY>
<!ATTLIST name     fn      CDATA         #IMPLIED
                   ln      CDATA         #REQUIRED>
<!ELEMENT email    (#PCDATA)>
<!ELEMENT phone    (#PCDATA)>
<!ELEMENT cite     (paper*)>
<!ATTLIST cite     id      ID            #REQUIRED
                   format  (ACM|IEEE)    #IMPLIED>
```

fig.1

Il primo passo compiuto dal CPI è quello di costruire un grafo per rappresentare la struttura del DTD .Il grafo si costruisce tramite un'operazione di parsine sul DTD,

i nodi sono elementi, attributi o operatori del DTD.

Ogni elemento appare esattamente una sola volta, mentre attributi ed operatori appaiono tante volte quante ne appaiono nel DTD,inoltre vengono annotate le cardinalità delle associazioni.

Il grafo relativo al DTD di fig.1 è il seguente:

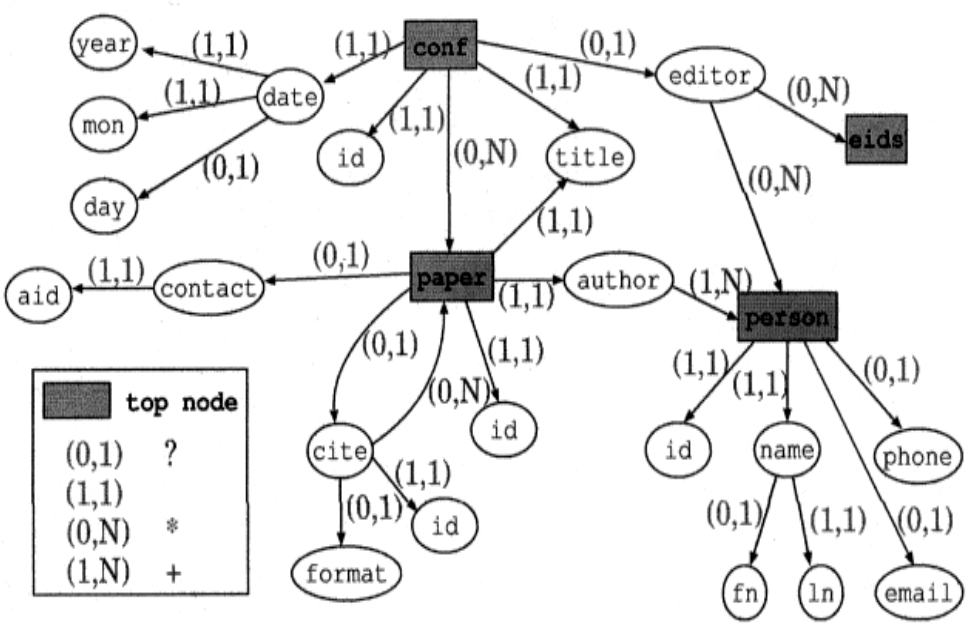


fig.2

Una volta costruito il grafo, CPI scorre a partire dai TOP NODE (secondo il metodo base di navigazione fornito dall' *hibrid algoritmo* [11]) tutti gli elementi e attributi dei suoi nodi foglia e automaticamente produce la corrispondente sintassi DDL , si costruisce così lo schema relazionale corrispondente al DTD dato.

Lo schema relazionale relativo al DTD di fig.1 definito tramite la sintassi DDL, è il seguente:

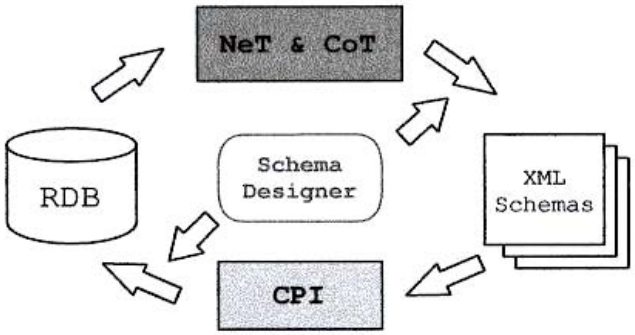
```
CREATE TABLE paper (
  id          NUMBER      NOT NULL,
  title       VARCHAR(50) NOT NULL,
  contact_aid VARCHAR(20),
  cite_id     VARCHAR(20),
  cite_format VARCHAR(50) CHECK (VALUE IN ("ACM", "IEEE")),
  root_elm    VARCHAR(20) NOT NULL,
  parent_elm  VARCHAR(20),
  fk_cite     VARCHAR(20) CHECK (fk_cite IN (SELECT cite_id FROM paper))
  fk_conf     VARCHAR(20),
  PRIMARY KEY (id),
  UNIQUE (cite_id),
  FOREIGN KEY (fk_conf) REFERENCES conf(id),
  FOREIGN KEY (contact_aid) REFERENCES person(id)
).
```

Per completezza accenniamo degli algoritmi [9] che consentono di passare da uno schema relazionale ad uno schema XML.

Un primo algoritmo semplice chiamato *Flat Translation* (FT) mappa in modo 1:1 uno schema flat relazionale in uno schema flat XML quindi utilizza solo una parte delle caratteristiche del modello XML, per ovviare a ciò si ricorre ad un secondo algoritmo chiamato *Nesting-based Translation* (NeT) il quale utilizza l'operatore di *nest* per generare uno schema XML da uno schema relazionale.

Il NeT però, può essere applicato solo ad una tabella per volta, quindi non è in grado di catturare un "big picture" di uno schema relazionale, pertanto dobbiamo utilizzare un altro algoritmo chiamato *Constraint-based Translation* (CoT) il quale prende in considerazione anche i vincoli che ci sono tra le tabelle utilizzando i *costraint semantic* .

Tutti e tre gli algoritmi sono "corretti" nel senso che tutti preservano l'informazione originale dello schema relazionale.



1.5 Relazioni e documenti XML

Oltre a tradurre lo schema relazionale in uno schema XML, è possibile utilizzando dei tools , comporre documenti XML da un DB relazionale, tra i vari tools disponibili descriviamo brevemente SilkRoute [12] ,esso viene utilizzato come middleware tra un DB relazionale ed un'applicazione che accede ai dati attraverso il WEB:

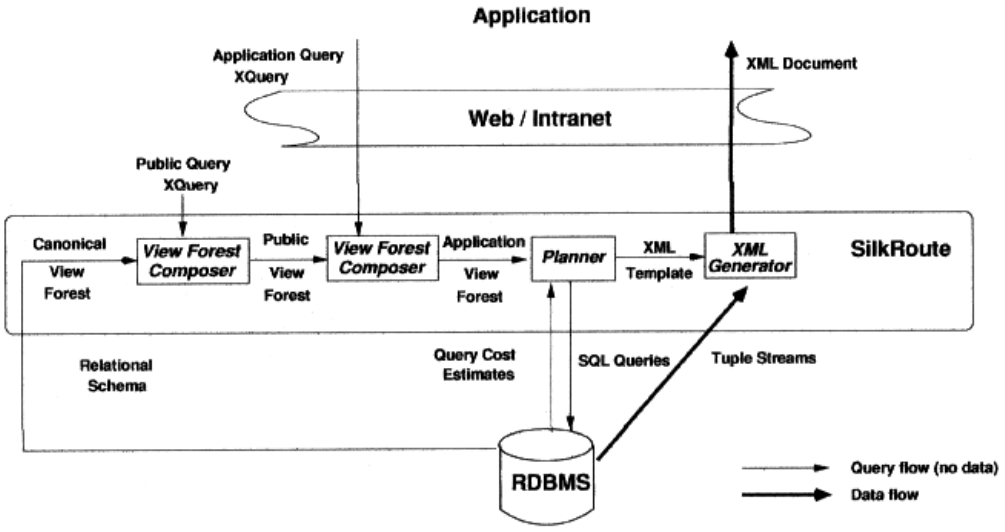


fig. 1

Le tabelle sono presentate al DBA in una virtuale "*canonical XML view"* precisamente il DBA scrivendo una "*public query"* sulla *canonical XML view* definisce la "*public XML view"* . Tale query è in genere complessa perché trasforma i dati relazionali in dati XML inseriti l'uno nell'altro. L'applicazione ha solamente accesso alla *public XML view* ma non al DB relazionale.

Per accedere ai dati bisogna formulare un "*application query"* sulla *public XML view* dopodiché tale *application query* viene composta con la *public query* ed il risultato viene poi ceduto al *"query Planner"* che lo traduce in una o più query SQL, le quali vengono poi eseguite dal motore relazionale.

Il risultato è un serie di tuple che *l'XML Generator* fonde in una relazione virtuale e costruisce un documento XML, che viene poi ritornato all'applicazione.

La *public query* e l' *application query* sono espresse in XQuery .

Lo scenario mostrato in fig. 1 è probabilmente il modo più comune per usare SilkRoute ma sono possibili altri modi di utilizzo, ad esempio il DBA può esportare un documento XML materializzandolo in una *public query* ciò può essere fatto passando la *public XML view* direttamente al planner .

Con SilkRoute abbiamo esaminato la possibilità di passare dalle tabelle a documenti XML , in realtà possiamo anche avere l'esigenza di voler archiviare, all'interno di un DB relazionale dei dati in formato XML[13].

Il problema non è però così semplice, perché un documento XML contiene delle informazioni,considerare il documento come un unico oggetto permetterebbe di memorizzarlo nel DB e reperirlo tramite una query ma, non consentirebbe di sfruttare le informazioni in esso contenute, quindi ciò che è necessario è un meccanismo di conversione, che permetta di trasferire le informazioni XML all'interno del DB, per gestirle in modo "nativo", mantenendo la capacità di recuperare, su richiesta, l'intero documento.

Questo obiettivo viene tipicamente raggiunto, dai vari DBMS, con specifiche estensioni ai loro linguaggi di manipolazione dei dati .Utilizzando Transact-SQL, ad esempio, il linguaggio disponibile in SQL SERVER, si possono utilizzare *sp_xmlpreparedocument* ,

una “ stored procedure” che analizza un testo in formato XML tramite il parser Micrsoft MSXML e un’istruzione OPENXML che restituisce un insieme di righe a partire dal documento analizzato.

L’istruzione OPENXML può ricevere, come parametro, un’espressione XPath che permette di estrarre solo un sottoinsieme di nodi del documento e una clausola WITH, che specifica il formato delle tabelle corrispondenti.E’ dunque possibile governare il processo di importazione, definendo precise regole di mapping.

CAPITOLO 2
Database XML-NATIVI (NXD)

In questo capitolo affronteremo i vari aspetti riguardanti i database XML-nativi, cercheremo innanzitutto, di comprendere il perché tali db si stanno diffondendo, presenteremo poi, le caratteristiche di NXD ed infine faremo una panoramica sui vari prodotti.

2.1 Perché usare NXD ?

L'affermarsi di NXD è legato allo sviluppo di Internet che di fatto sta cambiando il ruolo dei tradizionali DBMS : nuove funzionalità di trasferimento dati sono oggi divenute indispensabili nell'ambito di transazioni commerciali sia tra gli operatori che con i consumatori[14].

È certamente un grave errore non considerare, nella strategia di sviluppo di una attività, l'impatto che potrà avere su di essa il commercio elettronico e, in senso più vasto, tutto quello che viene raggruppato nel concetto di E.-business.

Parlare di commercio elettronico (E-commerce) significa avere un Sito Web e sviluppare su di esso le seguenti funzionalità:

1. Marketing e vendita: funzioni di promozione vendite, pubblicità e commercio diretto
2. Acquisti: gestione di acquisti diretti su Internet
3. Scambio diretto di transazioni con partner abituali (clienti e fornitori)
4. Banca elettronica: funzioni di gestione finanziaria.

Fare E-commerce quindi non è semplice, soprattutto perché occorre affrontare la sfida dell'integrazione tra sistemi eterogenei[15]. Ogni applicazione rappresenta i propri dati in formati differenti ed anche se, molte applicazioni consentono ad altre di accedere ai propri dati attraverso protocolli di rete o per mezzo di API (Application Programming Interfaces), i meccanismi per farlo cambiano di volta in volta. I dati sono tipicamente memorizzati in formato binario con una struttura non ovvia. I programmi che leggono e

scrivono tali informazioni si aspettano un formato ben preciso e se questo cambia, anche in piccola misura, le applicazioni esistenti non sono più in grado di interpretarne in maniera appropriata il contenuto informativo. Il significato dei dati è quindi cablato nella logica delle applicazioni che scrivono o leggono le informazioni ed inoltre il loro formato spesso dipende dalla piattaforma sulla quale i dati sono stati generati.

XML rappresenta la chiave di svolta per lo sviluppo delle attività di commercio elettronico. Difatti un documento XML è rappresentato da testo ASCII: questa caratteristica rende tale formato universale ed indipendente dalla piattaforma (un file di testo su Windows è lo stesso su UNIX) conferendogli di conseguenza una notevole portabilità; inoltre un documento XML non richiede l'utilizzo di formati di presentazione quindi il contenuto del documento può essere riutilizzato in diversi contesti senza adattamenti. In pratica lo stesso contenuto potrà essere inserito in brochure, pagine web, pubblicità e documenti interni all'azienda.

Risulta quindi evidente l'importanza di immagazzinare, richiamare, e modificare dati in un documento XML (un documento XML è in effetti un db nel senso stretto del termine), in altre parole gestire documenti XML.

Possiamo utilizzare un database XML-Enabled (cioè in grado di trattare file XML) oppure privilegiare quelli XML-Nativi (NXD).

Database XML-Enabled:

Sono prodotti che dichiarano il supporto all'XML come formato di Input/Output, in modo concreto, sottintendono che la traduzione, nelle due direzioni, viene applicata sui formati dei dati proprietari, comprese le API, verso lo standard XML. In realtà il supporto all'XML[16] (l'archiviazione dell'informazione che risiede nel documento XML) dipende dal tipo di documento, esaminiamo i vari casi:

1. **Document-Centric:** il documento XML come tale è il dato da inserire nel DB, cioè quello che conta è il documento nel suo complesso, e non i singoli dati contenuti. In questo caso gli interi documenti XML sono inseriti in campi del

database, o riferiti ad essi. I possibili campi possono essere di due tipi, BLOB o CLOB, la differenza tra i due è che il CLOB contiene dati in formato ASCII mentre il BLOB contiene dati in formato binario.Un esempio di Document-Centric può essere la descrizione di un prodotto:

```
<Product>

 <Intro>
 The <ProductName>Turkey Wrench</ProductName> from <Developer>Full
 Fabrication Labs, Inc.</Developer> is <Summary>like a monkey wrench,
 but not as big.</Summary>
 </Intro>

 <Description>

 <Para>The turkey wrench, which comes in <i>both right- and left-
 handed versions (skyhook optional)</i>, is made of the <b>finest
 stainless steel</b>. The Readi-grip rubberized handle quickly adapts
 to your hands, even in the greasiest situations. Adjustment is
 possible through a variety of custom dials.</Para>

 <Para>You can:</Para>

 <List>
 <Item><Link URL="Order.html">Order your own turkey wrench</Link></Item>
 <Item><Link URL="Wrenches.htm">Read more about wrenches</Link></Item>
 <Item><Link URL="Catalog.zip">Download the catalog</Link></Item>
 </List>

 <Para>The turkey wrench costs <b>just $19.99</b> and, if you
 order now, comes with a <b>hand-crafted shrimp hammer</b> as a
 bonus gift.</Para>

 </Description>

 </Product>
```

2. **Data –Centric:** i singoli dati del documento XML sono molto importanti quindi sono loro che devono essere inseriti nel DB. In questo caso la struttura gerarchica

dei documenti XML viene mappata su una serie di tabelle relazionali. Le relazioni tra tabelle permettono poi, di ricostruire la struttura originaria del documento XML. Si possono applicare query molto raffinate su questa struttura, ma di solito il numero di join implicati è molto alto.Un esempio di documento Data-Centric può essere il documento di un' ordine di vendita:

```
<SalesOrder SONumber="12345">
    <Customer CustNumber="543">
      <CustName>ABC Industries</CustName>
      <Street>123 Main St.</Street>
      <City>Chicago</City>
      <State>IL</State>
      <PostCode>60609</PostCode>
    </Customer>
    <OrderDate>981215</OrderDate>
    <Item ItemNumber="1">
      <Part PartNumber="123">
        <Description>
          <p><b>Turkey wrench:</b><br />
          Stainless steel, one-piece construction,
          lifetime guarantee.</p>
        </Description>
        <Price>9.95</Price>
      </Part>
      <Quantity>10</Quantity>
    </Item>
    <Item ItemNumber="2">
      <Part PartNumber="456">
        <Description>
          <p><b>Stuffing separator:<b><br />
          Aluminum, one-year guarantee.</p>
        </Description>
        <Price>13.27</Price>
      </Part>
      <Quantity>5</Quantity>
    </Item>
     </SalesOrder>
```

3. **Trattamento Ibrido:** il documento XML è un misto delle due tipologie viste precedentemente (document-centric, data-centric), in tal caso viene mappato in modo data-centric, ma alcuni sottoalberi vengono trattati in maniera document-centric. Si tratta del metodo più avanzato per gestire i documenti XML, ma anche il più difficile da realizzare. In generale, si trattano in maniera document-centric i sottoalberi in cui sono rari gli "accessi in profondità" ma che sono di solito recuperati come totalità.

Abbiamo quindi vari modi per trattare i documenti XML ma sono pochi efficienti soprattutto dal momento che il volume e la complessità delle transazioni di E-Business è in continua crescita, difatti il carico di lavoro necessario alla conversione tra XML e qualche altra forma di rappresentazione dati incide negativamente sulla velocità, affidabilità e funzionalità; inoltre quando si archivia un documento XML in una tabella è possibile che vengono perse informazioni quali l'ordinamento degli elementi e la distinzione tra elementi e attributi, infine normalmente sono necessari join tra non poche tabelle per recuperare anche il più semplice documento di Business.

Tra i prodotti XML-Enable citiamo DB2 e Oracle 8.i, essi permettono di gestire documenti XML in maniera document-centric ed inoltre permettono una gestione data-centric piuttosto limitata, utilizzando le loro estensioni OODBMS per rappresentare elementi XML come oggetti.

Database XML-Nativi:

Lo scenario cambia radicalmente se si sceglie di utilizzare un database NXD perché, in questo caso, si gestiscono direttamente i documenti XML, e quindi i problemi di traduzione da o verso formati XML sono già da subito superati,inoltre grazie ad un supporto esteso dell'XML, che si spinge fino a toccare l'architettura interna, presenta una maggiore scalabilità, affidabilità, e una minore manutenzione rispetto al database XML-Enable.

2.2 Che cosa è un NXD ?

Gli NXD[17,18] sono DB specializzati a memorizzare documenti XML, come ogni altro DB, supportano transazioni, accesso multiplo, API, linguaggi di interrogazione e così via. L'unica differenza con gli altri DB è che il loro modello interno è basato su XML, ciò gli consente di memorizzare i document-centric, in modo più efficiente dei DB XML-Enable, perché appunto permettono di preservare l'ordine del documento, i commenti, le sezioni CDATA.

Inoltre poiché gli NXD supportano linguaggi di interrogazione in XML, è possibile risolvere query che difficilmente possono essere risolte con SQL, come ad esempio " Dammi tutti i documenti nei quali il terzo paragrafo inizia con una certa parola".

Si osserva infine che un NXD può accettare, memorizzare, e comprendere ogni documento XML senza schema (invece il trasferimento di dati in documento XML da un DB relazionale o a oggetti, richiede la creazione di uno schema, ed il successivo mapping). La mancanza dello schema è un vantaggio per le applicazioni , quali i motori di ricerca sul WEB in cui nessun DTD o insieme di DTD può essere applicato a tutti i documenti.

NXD: Definizione

Una possibile definizione di NXD è presentata da *XML:DB mailing list* , in cui si afferma che un NXD:

- Definisce un modello logico per un documento XML (in contrasto con i dati in quanto documento); memorizza e ricerca il documento secondo quel modello. Come minimo il modello deve includere: gli elementi, gli attributi, PCDATA e

l'ordine del documento. Alcuni esempi di tali modelli sono l' XPATH data model, l' XML Infoset, ed i modelli impliciti come il DOM e gli eventi in SAX 1.0

- Ha come unità fondamentale di memorizzazione (logica) un documento XML, così come il DB relazionale ha la riga di una tabella come unità fondamentale di memorizzazione (logica).

- Non richiede alcun modello fisico particolare di memorizzazione. Per esempio può essere costruito su un DB relazionale, gerarchico, o orientato ad oggetti oppure, può usare un formato proprietario per la memorizzazione.

La prima parte di questa definizione è simile alla definizione degli altri tipi di DB, difatti si concentra sul modello usato dal DB. È da notare però, che un dato NXD potrebbe memorizzare più informazioni di quelle che sono contenute nel modello.

Per esempio l'NXD potrebbe supportare query basate sul XPATH data model ma, memorizzare i documenti come testo. In questo caso, cose come sezioni CDATA ed entità sono memorizzate nel DB ma non sono incluse nel modello.

La seconda parte della definizione, afferma che l'unità fondamentale di memorizzazione in un NXD è un documento XML (anche se sembra possibile che tale ruolo possa essere ricoperto dal frammento del documento). La sua esistenza non preclude però, il fatto che possiamo recuperare piccole unità di dati, come ad esempio un frammento di un documento, o singoli elementi; e non preclude nemmeno, la combinazione di frammenti tra documenti diversi. In termini relazionali, questo equivale a dire che l'esistenza di righe, non preclude il recupero di valori individuali da colonne, o la creazione di nuove righe da righe esistenti.

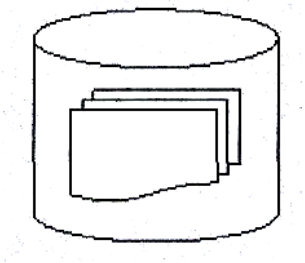
La terza parte della definizione, afferma che il formato fisico del dato memorizzato non è importante.

2.3 Architetture di un NXD

Le architetture di un NXD si suddividono in due grandi categorie:

- *Text-Based*
- *Model-Based*

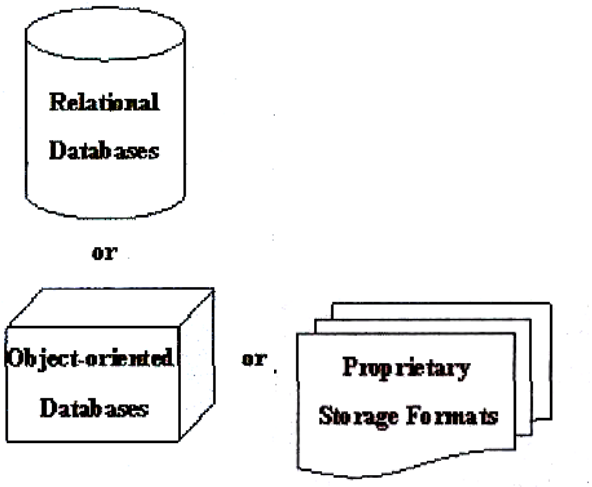
Un **Text-Based NXD** è un DB che memorizza il documento XML come testo,e utilizza qualche funzionalità del DB per accedere al documento.Una semplice strategia può essere quella di memorizzare il documento come un BLOB in un DB relazionale, o come un file in un file system, oppure utilizzare un formato testo proprietario; e poi utilizzare degli indici sul documento.



Di fatto tutti gli Text-Based NXD sono dotati di indici, inoltre consentono di avere un elevata velocità nel recuperare il documento o parte di esso, poiché effettuano un sola ricerca per indice e quindi recuperano il documento con una sola lettura del disco.

Il Text-Based NXD, come il DB gerarchico, quando recupera e ritorna dati secondo una predefinita gerarchia ha delle performance migliori di un DB relazionale ma, se deve recuperare e ritornare dati in una forma che non è quella predefinita, ad esempio in una gerarchia invertita , allora le prestazioni sono scadenti.

Un **Model-Based NXD** non memorizza il documento XML come testo ma, costruisce dal documento un object model, il quale viene poi memorizzato in un modo che dipende dal DB. Alcuni DB memorizzano l' object model in un DB relazionale, altri in un DB orientato ad oggetti, ed altri ancora utilizzano un formato proprietario di memorizzazione (come Tamino, TEXTML).



Un esempio di object model è il DOM. (Document Object Model) ; si ricorda che il DOM rappresenta un albero (tree) gerarchico all'interno del quale vengono caricati tutti gli elementi del documento XML. Attraverso poi i metodi e le proprietà del DOM diventa

possibile navigare l'albero (lo strumento preposto alla "creazione" di un DOM è il PARSER, ovvero il componente software che legge il file XML e "crea" se necessario un DOM; alcuni per ragioni di performance si limitano a scandire il documento).

Il DOM può essere memorizzato in un DB relazionale utilizzando varie tabelle, una per gli ELEMENTI un'altra per gli ATTRIBUTI, una per PCDATA , una per l'ENTITA'.

Il Model-Based NXD quindi, non memorizza l' XML nella suo forma natale (cioè testo) e non è autonomo, difatti si deve appoggiare ad un altro DB, anzi le sue performance dipendono da tale DB, d'altra parte esse sono comunque scadenti, se si richiede di restituire il documento in forma diversa da quella in cui è stato memorizzato.

Le due architetture presentate, si differenziano soprattutto per ciò che riguarda la velocità, e il *round-trip* del documento.

Il Text-Based NXD può memorizzare e ritornare (round-trip) "lo stesso" documento, quindi effettua realmente il *round-trip* del documento, invece il Model-Based NXD no, difatti è in grado di fare solamente il *round-trip* del modello del documento.

La velocità del Text-Based NXD, come preannunciato, è ovviamente maggiore nel restituire l'intero documento o parti di esso in formato testo; mentre il Model-Based NXD presenta, probabilmente, una maggiore velocità nel combinare i vari pezzi provenienti da documenti diversi, anche se però in tal caso giocano un ruolo fondamentale la lunghezza del documento, la velocità di parsing, e la velocità di recupero; il fatto poi che il Model-Based NXD sia veloce a restituire il documento come DOM trees o SAX events dipende dal singolo DB, e dal fatto che non possiamo avere contemporaneamente un elevata velocità di parsing e di recupero. In realtà i Model-Based NXD che, usano un formato proprietario di memorizzazione, presentano, probabilmente, una performance simile ai Text-Based NXD nel recuperare i dati nell'ordine in cui sono stati memorizzati; poiché usano dei puntatori fisici tra i nodi e quindi la performance è simile a quando viene recuperato testo.

2.4 Caratteristiche di un NXD

In tale paragrafo discutiamo brevemente, delle caratteristiche[18] che in generale troviamo nella maggior parte degli NXD:

Document collection

Gli NXD gestiscono le collezioni di documenti, precisamente ci consentono di interrogare e trattare dei documenti come un'insieme.

Ad esempio supponiamo che stiamo usando un NXD per archiviare degli ordini di vendita, e siamo interessati solo ad un'insieme di ordini. Bene in questo caso possiamo memorizzare i documenti di interesse in una collection ed eseguire query solo sulla collection. La nozione di collection pertanto, gioca un ruolo simile a quello di una tabella in un DB relazionale, o di una directory in un file system. Alcuni NXD consentono di definire una collection senza schema, questo consente al DB di avere una serie di flessibilità, e permette di facilitare lo sviluppo delle applicazioni. In realtà avere una collection senza schema comporta anche il rischio di avere dati inconsistenti.

Query Languages

La disponibilità di un linguaggio di interrogazione dei dati immagazzinati nei documenti XML, è di fondamentale importanza per l'esistenza stessa degli NXD.

I requisiti che si richiedono ad un query language XML sono:

- Capacità di selezionare, estrarre, combinare, eliminare, e trasformare il contenuto di un documento XML.
- Capacità di interrogare diversi documenti XML contemporaneamente.
- Capacità di supportare l'uso di funzioni di aggregazioni.
- Capacità di agire sui dati senza schema
- Capacità di considerare l'ordine degli elementi nei documenti XML.
- Capacità di essere rappresentato in XML, cioè se ci sono diverse sintassi per il query language, una di queste deve essere XML.

La comunità dei DB e quella di Internet, hanno cercato di proporre dei linguaggi che rispondessero ai requisiti menzionati, in realtà ancora oggi non esiste un Query Language Standard XML, anche se la scelta sembra orientarsi su un linguaggio come XQuery del W3C.

La mancanza di uno Standard ha spinto la maggior parte degli NXD a supportare uno o più query language. I più popolari sono XQL, XPath, e naturalmente XQuery.

XQL

L'XQL[19] (XML Query Language) fu realizzato dalla Microsoft, Texcel, e da WebMethods.La sintassi adottata è molto simile a quella del linguaggio XPath (XML Path language).

Le XQL query sono abbastanza semplici, esse estraggono i nodi e i sottorami dai documenti XML, inoltre la collezione degli elementi selezionati è sempre ritornata mantenendo l'ordine originale. Fondamentalmente una XQL query è composta da espressioni path e da espressioni di filtraggio; le espressioni path definiscono quali nodi saranno richiesti e quali nodi ritorneranno nel risultato, le espressioni di filtraggio permettono di selezionare solo i nodi che soddisfano un determinato criterio.

Le espressioni path vengono generalmente rappresentate da una serie di elementi separati dall'operatore " / " , invece le espressioni di filtraggio sono rappresentate dall'operatore " [ ] " . Le XQL query sono valutate da sinistra verso destra, rispettando i ruoli dei precedenti operatori, quindi viene ritornato l'elemento che sta più a destra.

Consideriamo ad esempio il seguente documento XML:

```
<?xml version="1.0"?>
<bibdb>
  <book isbn="_0201385902">
      <title>Principles of Database and …Systems</title>
      <publisher>
          <name>Addison-Wesley</name>
          <headquarter>Massachusetts</headquarter>
      </publisher>
      <year>1998</year>
      <price>45</price>
      <author>C. J. Date</author>
  </book>
  <book isbn = "_1-55860-452-9">
      <title>O-R DBMSs Tracking … Great Wave</title>
      <publisher>
          <name>Morgan Kaufmann</name>
          <headquarter>San Francisco</headquarter>
      </publisher>
      <year>1999</year>
      <price>42</price>
      <author>Michael Stonebraker</author>
      <author>Paul Brown</author>
  </book>
  <paper classification = "DB" ref = "_1-55860-452-9">
      <author>Pedro Soares</author>
      <title>Object-Relational Databases</title>
      <year>2000</year>
  </paper>
</bibdb>
```

fig. 1

e consideriamo la seguente espressione:

/bibdb/book

Tale espressione ritorna tutti gli elementi <book> nel "database XML" di fig.1

Una XQL query è sempre valutata in un specifico contesto, ossia in una lista di nodi del database (elementi del documento XML). Nell' espressione /bibdb/book, l'elemento <bibdb> è valutato nel contesto iniziale, mentre poi tramite l'operatore " / " siamo passati in un altro contesto, precisamente nel sottoelemento di <bibdb> , ed in tale contesto viene valutato l'elemento <book>.(l'espressione " /bibdb/book" cerca, appunto gli elementi book che sono i figli dell'elemento < bibdb>).

Gli operatori che stabiliscono un nuovo contesto sono:

- l'operatori gerarchico " / ",
- l'operatore di filtraggio " [ ] "
- l'operatore " // ", che indica un certo numero di livelli nella gerarchia degli elementi.

L'operatore " [ ] ", filtra il set di nodi che sono specificati alla sua sinistra, secondo le condizioni espresse all'interno delle parentesi [ ]. L'espressione nelle parentesi è chiamata sub-query; ogni nodo che non verifica la sub-query è omesso dal risultato, ecco perché l'operatore " [ ] " somiglia alla clausola WHERE di SQL. Per esempio consideriamo la seguente espressione:

/bibdb/book[author = "C.J Date"]/title

Tale query, ritorna i (sub)elementi <title> degli elementi <book> che contengono un (sub)elemento <author> con il valore: "C.J Date", in altre parole ritorna il titolo dei libri il cui autore è C.J Date; ciò avviene trovando prima gli elementi <book> che soddisfano la condizione di filtro sul (sub)elemento <author>, dopodiché si trovano i (sub)elementi <title> .

Infine un altro operatore che troviamo in una XQL query è l'operatore di unione:

" | " che specifica il nodo di unione tra l'espressione alla sua sinistra con i nodi alla sua destra, e consente il ritorno di elementi distinti. Per esempio l'espressione:

/bibdb/(book | paper)

trova tutti gli elementi distinti, come book e paper ,nel "db" di fig.1

XPath

XPath è un linguaggio dichiarativo [20] che, consente di indirizzare parti del documento XML, inoltre può anche essere inserito in linguaggi ospiti come XSLT 2.0 (Exestensibile Stylsesheet Language Trasformation) e XQuery.

XPath opera su una struttura logica astratta di un documento XML detta Data-Model, la quale viene descritta in XQuery 1.0 e XPath 2.0 Data-Model, in altre parole XPath modella un documento XML come un albero con sette possibili nodi:

- root nodes
- element nodes
- text nodes
- attribute nodes
- namespace nodes
- processing instruction nodes
- comment nodes

dopodiché attraverso l'uso della Path expression è possibile "indirizzare" un qualsiasi tipo, ed un qualsiasi numero di nodi.

La Path expression è il cuore di XPath, essa è composta da una sequenza di zero o più termini ,che possono essere valori di nodo (elementi,attributi, ecc) o valori atomici (numeri, stringhe, ecc) e dai seguenti operatori:

- " **.** " identifica " l'elemento corrente "
- " **..** " l'elemento padre dell'elemento corrente
- " / " passaggio all'elemento figlio di quello corrente
- " // " discendente dell'elemento corrente
- " **@** " attributo dell'elemento nodo corrente
- " * " elemento dal nome qualsiasi

- “ **[** p **]** “ predicato p per “filtrare” gli elementi individuati
- “ **[** n **]** “ l’elemento in posizione n (interroga l’ordine)

Per chiarire come effettivamente opera il linguaggio XPath riportiamo alcuni esempi [21] di path expression. Consideriamo il seguente documento XML: libri.xml

```
<?xml version="1.0"?>
<Elenco>
    <Libro disponibilità='S'>
        <Titolo>Il Signore degli Anelli</Titolo>
        <Autore>J.R.R. Tolkien</Autore>
        <Data>2002</Data>
        <ISBN>88-452-9005-0</ISBN>
        <Editore>Bompiani</Editore>
    </Libro>
    <Libro disponibilità='N'>
        <Titolo>Il nome della rosa</Titolo>
        <Autore>Umberto Eco</Autore>
        <Data>1987</Data>
        <ISBN>55-344-2345-1</ISBN>
        <Editore>Bompiani</Editore>
    </Libro>
    <Libro disponibilità='S'>
        <Titolo>Il sospetto</Titolo>
        <Autore>F. Dürrenmatt</Autore>
        <Data>1990</Data>
        <ISBN>88-07-81133-2</ISBN>
        <Editore>Feltrinelli</Editore>
    </Libro>
</Elenco>
```

fig.2

ESEMPIO 1

- Una path expression può iniziare con

  document("nome-documento")

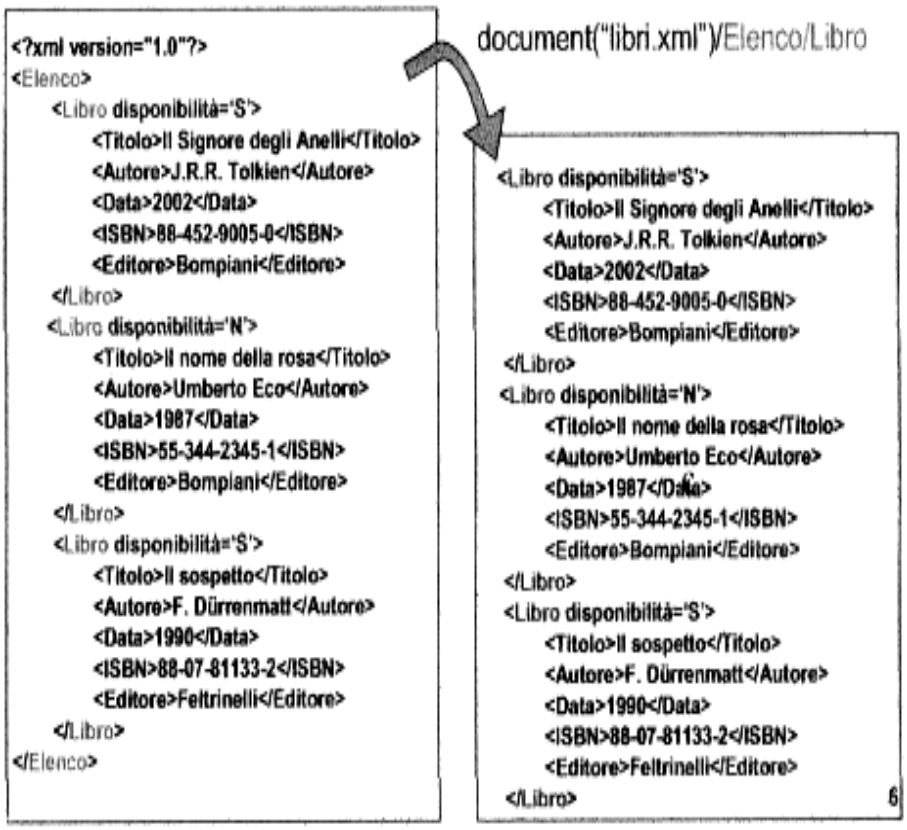
  [restituisce la radice del documento specificato]

- A partire dalla radice del documento si possono specificare dei "cammini" per "raggiungere" i frammenti XML desiderati

- Esempio:

  document("libri.xml")/Elenco/Libro

  [identifica ("restituisce", se valutata) tutti i Libro contenuti nell'Elenco all'interno del documento libri.xml]



ESEMPIO 2

document("libri.xml")/Elenco/Libro[Editore='Bompiani'AND @disponibilità= 'S']/Titolo

Restituisce l'insieme di tutti i titoli dei libri dell'editore Bompiani che sono disponibili e che si trovano nel documento libri.xml:

<Titolo> Il Signore degli Anelli</Titolo>

ESEMPIO 3

document("libri.xml")//Autore

Restituisce l'insieme di tutti gli autori che si trovanonel documento libri.xml annidati a qualunque livello:

<Autore> J.R.R Tolkien</Autore>

<Autore> Umberto Eco</Autore>

<Autore> F. Durrenmatt</Autore>

ESEMPIO 4

document("libri.xml")/Elenco/Libro[2]/*

Restituisce gli elementi contenuti nel secondo libro del documento libri.xml. Permette di interrogare gli ordini dei dati

<Titolo> Il Signore degli Anelli</Titolo>

<Autore> Umberto Eco</Autore>

<Data>1987</Data>

<ISBN>55-344-2345-1</ISBN>

<Editore> Bompiani </Editore>

Da tali esempi si evincono alcune importanti limitazioni di XPath , quali la mancanza di raggruppamento (operatori di aggregazione), e join tra documenti, ecco perché si è cercato di ottenere un linguaggio più orientato ai DB, quale XQuery.

XQuery

Il linguaggio di interrogazione XQuery [19] è stato disegnato per soddisfare i requisiti identificati, al fine di interrogare documenti XML, dal W3C XML Query

Working Group. XQuery deriva da un linguaggio di interrogazione chiamato Quilt che, a sua volta, prende alcune caratteristiche da numerosi linguaggi quali: XPath 1.0, XQL, XML-QL, SQL e OQL. XQuery è molto flessibile e permette sia interrogazioni facilmente comprensibili da un utente, che interrogazioni basate più specificamente sulla sintassi di XML e XML –Schema. XQuery, inoltre, risulta essere fortemente tipizzato ed il sistema di tipi presenti al suo interno è basato su quello fornito da XML-Schema.

Per effettuare query un pò articolate si ricorre all'uso delle espressioni FLWR ("flower"):

- **For** per l'iterazione *[from]*
- **Let** per legare variabili a insiemi
- **Where** per esprimere predicati *[where]*
- **Return** per generare il risultato *[select]*

esse sono solo un sottoinsieme della sintassi di XQuery. L' espressioni FLWR[22] permettono tramite l'impiego di parole chiave quali *for, where,* e *return,*di percorrere ricorsivamente l'albero definito dal documento XML e restituire i valori richiesti nel formato voluto.Tutto ciò è possibile per il semplice fatto che anche una espressione XQuery rappresenta, come un documento XML, un albero.

Per comprendere più a fondo quest'ultima affermazione consideriamo il seguente documento XML:

…..

```
<Libro>
    <Titolo> Xml1.0 per tutti </Titolo>
     <Autore> Marco Rossi </Autore>
          <Capitolo>
               <Argomento> Introduzione </Argomento>
                <Titolo> Introduzione </Titolo>
          </Capitolo>
          <Capitolo>
                <Argomento> XML </Argomento>
                 <Titolo> Linguaggio </Titolo>
           </Capitolo>
       …..
     </Libro>
     …….
```

Si supponga, ora, di voler porre una query , in formato XQuery, per poter trovare il titolo di ogni capitolo di ogni libro, il cui argomento sia ‘XML’.

Tale richiesta assume la forma della seguente espressione :

```
for $a in /Libro/Capitolo
    where $a/Argomento = XML
```

return $a/Titolo

si può notare come attraverso l'utilizzo delle parole chiave *for* e *where*, e l'impiego di espressioni e variabili ($a) che indichino i percorsi (path) desiderati, si esprimano le condizioni da soddisfare per il ritrovamento dell'informazione, mentre, attraverso l'uso della parola chiave *return*, si esprima il risultato da ottenere ed il formato in cui tale risultato deve comparire. La query considerata può essere visualizzata graficamente tramite una rappresentazione ad albero in cui i nodi intermedi rappresentano gli elementi XML citati, mentre le foglie sono i contenuti degli stessi. Si consideri la seguente figura:

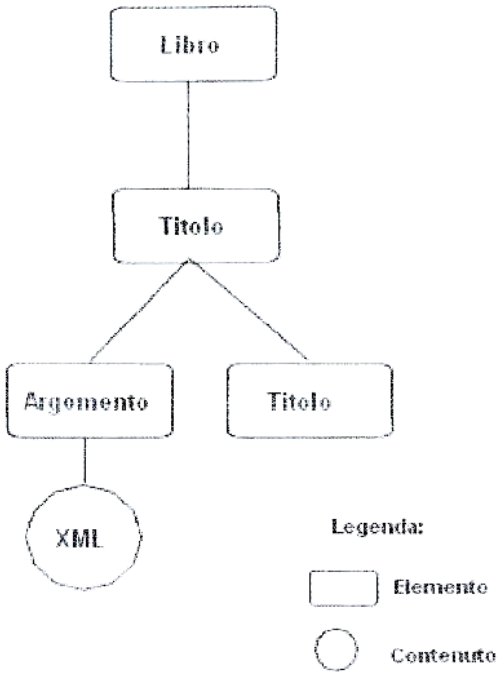


fig. 3 : Albero rappresentativo della query

Dato che, proprio per la sua struttura, anche i documenti XML sono rappresentabili tramite l'utilizzo di alberi, la risoluzione di una XQuery diviene, in realtà, un problema di confronto fra alberi e fra cammini che li compongono.

Infine proponiamo un altro esempio di XQuery , per mettere in evidenza le sue potenzialità rispetto agli altri linguaggi, difatti tramite un ciclo innestato è possibile effettuare sostanzialmente quello che, nel DB relazionale, viene identificato come *join tra due tabelle.* Consideriamo un documento XML in cui ci siano informazioni riguardanti sia clienti che gli ordini, allora se vogliamo trovare i nomi dei clienti che hanno ordinato un prodotto in cui *part-id* sia ' xx ' la richiesta assume la seguente espressione

```
for $c in clienti
 for $o in ordini
 where  $c.clien_id = $o.clien_id and $o.part_id = "xx"
return $c.nome
```

Per concludere riportiamo alcuni esempi esemplificativi delle espressioni FLWR[21]:

Espressione FOR:

```
for $x in document("libri.xml")//Libro
return $x
```

- La clausola FOR valuta l'espressione : "libri.xml" (che compare come argomento della function document(), si noti che in genere l'argomento è un URL) e itera all'interno di questa lista legando (binding) i singoli elementialla variabile $x (tante volte quante sono i componenti della lista)

- L'interrogazione restituisce poi semplicemente l'insieme di tutti i Libro nel documento libri.xml, eseguendo la sua clausola RETURN tante volte quanti sono i legami della variabile $x.

Espressioni LET:

```
let $y := document ("libri.xml")//Libro
return $y
```

- La clausola LET valuta l'espressione: "libri.xml" e assegna l'intero insieme di libri trovati alla variabile $y
- Il risultato di una clausola LET produce un singolo binding (legame) per la variabile: l'intero set viene assegnato alla variabile
- La query esegue una sols volta la sua clausola RETURN

Clausola WHERE

- La clausola WHERE esprime una condizione: solo gli assegnamenti di variabili che soddisfano la clausola sono usati per eseguire la clausola RETURN
- Le condizioni nella clausola WHERE possono contenere diversi predicati connessi da AND, OR, NOT.
- Esempio:

```
for $x in document("libri.xml")//Libro
 where $x/Editore = "Bompiani"
        AND $x/ @disponibilità = "S"
Return $x
```

Restituisce tutti I libri bompiani che sono disponibili

Clausola RETURN

- Genera l' output di un espressione FLWR che può essere:

  Un nodo: <Autore> F.Durrenmatt</Autore>

  Una lista ordinata di nodi : <Autore> F.Durrenmatt</Autore>

  <Autore> Umberto Eco</Autore>

  <Autore> J.RR.Tolkien</Autore>

  un valore : Umberto Eco

- Può contenere costruttori di elementi, riferimenti a variabili definitenelle parti FOR e LET e generiche espressioni annidate.

Updates and Deletes

Gli NXD[18] possono facilmente sostituire o cancellare documenti ma, incontrano delle serie difficoltà per quanto riguarda la modifica, che in realtà resterà problematica fino a che XQuery non prevederà una componente per l'aggiornamento.

Per il momento alcuni NXD dispongono di un proprio linguaggio di modifica (sebbene sempre un maggior numero supporta l' XUpdate language), altri invece effettuano le modifiche attraverso la manipolazione del DOM tree.

Transaction, Locking, and Concurrency

Tutti gli NXD[17,18,23] supportano il concetto di transazione (e presumibilmente anche il rollbacks) e spesso permettono di applicare il meccanismo dei lock all'intero documento; quindi non potendo "bloccarne" soltanto una parte, ci dobbiamo aspettare una gestione inefficiente della concorrenza.

In realtà tutto dipende da quello che vogliamo fare, e da come il documento è organizzato. Ad esempio, se una guida per gli utenti è suddivisa in capitoli, ognuno dei quali è un documento, allora i problemi legati alla concorrenza saranno probabilmente poco significativi, poiché sarà improbabile che due transazioni vorranno scrivere e modificare, contemporaneamente lo stesso capitolo. D'altra parte se tutti i dati di un cliente saranno memorizzati in un solo documento allora, il locking a livello di documento comporterà probabilmente, seri danni per la concorrenza.

In futuro, la maggior parte degli NXD, probabilmente, offriranno un locking a livello di frammento del documento.

Application Programmino Interfaces (APIs)

Quasi tutti gli NXD[17,18] offrono API programmatiche, in genere sono nella forma di un ODBC-like interface, con metodi per la connessione al DB, esplorazione del metadata, esecuzione di query, e restituzione dei risultati.

I risultati in genere sono restituiti come stringhe XML (un DOM tree) oppure tramite un parser SAX o XML Reader, in effetti la maggior parte degli NXD consente anche di eseguire query e ritornare i risultati attraverso http, inoltre alcuni NXD dispongono di API proprietarie.

Round-Tripping

Gli NXD[18] in genere supportano il round-tripping del documento, tale caratteristica è fondamentale per un'applicazione document-centric, poiché essa gestisce il documento integro e quindi occorre che il documento ritornato sia lo "stesso" di quello memorizzato.

Invece il round-tripping è meno importante per le applicazioni data-centric, in quanto la loro attenzione è rivolta sugli elementi, attributi, testo, e ordine gerarchico.

Si ricorda che in genere i Text-Based NXD eseguono esattamente il round-tripping del documento, mentre i Model-Based NXD eseguono il round-tripping a livello di modello.

Remote data

Alcuni NXD[18] sono in grado di memorizzare dati provenienti da altri DB. In genere questi dati sono richiamati da un DB relazionale, attraverso ODBC, OLE DB, o JDBC.

Il primo passo da compiere per trasferire dati tra documenti XML e un DB, è il mapping tra lo schema del documento XML (DTD, XML Schema, RELAX NG, ecc..) e quello del DB, tale mapping riguarda, in genere, element types, attributi, testo. Le strategie più comuni per fare il mapping sono: *table-based mapping* e *object-relational mapping*.

Il *table-based mapping* è usato in genere per trasferire dati tra un documento XML e un DB relazionale, in tal caso la struttura del documento XML sarà la seguente:

```
<database>
   <table>
      <row>
         <column1>...</column1>
         <column2>...</column2>
         ...
      </row>
      <row>
         ...
      </row>
      ...
   </table>
   <table>
      ...
   </table>
   ...
</database>
```

Ovviamente se il DB è costituito da una sola tabella non esisteranno gli elementi <database> e gli elementi aggiuntivi <table>.

L' *object-relational mapping* invece, utilizza un object model, nel quale element types con attributi, element content, o mixed content, sono generalmente modellati come classi, mentre i simple element types, attributi e PCDATA sono modellati come proprietà scalari; dopodiché utilizzando le tradizionali tecniche di object-relational mapping, le classi saranno mappate in tabelle, e le proprietà scalari saranno mappate in colonne. Infine è bene sottolineare, che l'object model utilizzato non è il DOM, difatti mentre questo è lo stesso per tutti i documenti XML, l' object model (utilizzato per il mapping) è diverso per ogni insieme di documenti conformi a un dato DTD.

Dopo il mapping tra lo schema del documento XML e lo schema del DB, il trasferimento dei dati avviene tramite software, il quale può usare un XML query language (come XPATH, XQuery, o un linguaggio proprietario), o può semplicemente trasferire i dati conformemente al mapping.

Per concludere è doveroso osservare che non sempre le modifiche apportate al documento nel NXD vengono riflesse nel DB "remoto" ma, ciò dipende dal NXD.

Indexes

Quasi tutti gli NXD[18,23] supportano, il meccanismo degli indici applicato agli elementi e ai valori degli attributi; l'utilizzo degli indici, analogamente a quanto avviene negli altri DB, ha lo scopo di velocizzare le query.

Normalization

L'utilizzo degli NXD[18] per la memorizzazione dei dati solleva, non poche discussioni sulla normalizzazione, in realtà il problema è scarsamente rilevante per i document-centric, poiché in tal caso il numero dei dati che, eventualmente sono ripetuti in ogni documento, è molto basso rispetto a tutti quanti gli altri; inoltre, analogamente a quanto avviene per i DB relazionali, non c'è nessuno che ci obbliga ad applicare la normalizzazione.

Normalizzare i dati in NXD è molto simile a quello che facciamo nei DB relazionali, quindi dobbiamo sostanzialmente progettare i nostri documenti in modo tale che i dati non si ripetano; in effetti poiché gli NXD supportano "multi-valued properties" (a differenza di molti DB relazionali) possiamo utilizzare una strategia che non è applicabile nei DB relazionali.

Per esempio consideriamo il documento che rappresenta un ordine di vendita, esso sarà in genere composto da:

- Un'intestazione, costituita ad esempio dal numero di ordine, dalla data…
- Una o più voci che riportano il codice dell'articolo, la quantità, e il prezzo totale.

Per come è strutturato il documento si pone il problema della normalizzazione, difatti possiamo avere una sola intestazione che ha più voci, e quindi si corre il rischio di ripetere per ogni voce sempre la stessa intestazione. Bene se utilizziamo un DB relazionale, il problema si risolve memorizzando l'intestazione in una tabella, separata da quella che contiene le voci, invece se utilizziamo un NXD, grazie alla natura gerarchica di XML che

,consente ad un elemento padre di avere più figli, possiamo utilizzare un solo documento per archiviare tutte le informazioni dell' ordine, compresa l'intestazione, senza alcun ridondanza.

Sfortunatamente però, non sempre la soluzione è così banale, difatti se siamo interessati ad archiviare le informazioni di un cliente, e tale cliente compare in più ordini di vendita, possiamo scegliere due strade:

1. Non normalizzare , e quindi duplicare le informazioni del cliente in ogni ordine di vendita.
2. Memorizzare l'informazione del cliente in un documento, dopodiché se XLinks è supportato, si provvederà a XLinkare il documento dell'ordine con quello del cliente; oppure se il query language supporta join si può effettuare un join tra i due documenti.

In definitiva il problema della normalizzazione non ammette un'unica risposta, anche perché, bisogna considerare la frequenza delle operazioni, difatti può accadere che, per migliorare la performance di quelle operazioni che si eseguono più spesso, decidiamo di non ricorrere alla normalizzazione.

Referential Integrity

In un NXD[18] il vincolo di integrità referenziale, consente ai "puntatori" che troviamo nel documento XML, di "validare" il documento o parte di esso.

I "puntatori" possono essere di diverso tipo:

- Attributi ID/IDREF
- Key/keyref fields
- XLinks
- Meccanismi proprietari, questi includono dei specifici languages "referencing" elements and attributes.

Il vincolo di integrità referenziale può riferirsi all'interno del documento (puntatori interni) o può riferirsi all'esterno (puntatori esterni).

Nel caso di "puntatori interni" se questi sono standard (Attributi ID/IDREF o key/keyref) allora il vincolo di integrità referenziale è supportato solo parzialmente; nel senso che la maggior parte degli NXD, esegue la validazione solo quando il documento è inserito nel DB, allora se il documento viene cancellato o viene replicato, il vincolo di integrità referenziale è assicurato dalla validazione, se invece il documento viene modificato a livello di nodo occorre un lavoro aggiuntivo (validare tutte le modifiche) che solo pochi NXD eseguono.

Invece per quanto riguarda il vincolo di integrità referenziale tra documenti diversi dello stesso DB (puntatori esterni) gli attuali NXD non sono in grado di garantirlo.

In definitiva (attualmente) dobbiamo affidarci all'applicazione, se vogliamo verificare il vincolo di integrità referenziale (sia quello interno che quello esterno)

Scalability

Molti NXD[18,23] , come i DB gerarchici e diversamente dai DB relazionali, utilizzano dei link fisici per legare i dati. In particolare i Text-Based NXD legano fisicamente insieme gruppi di dati, mentre i Model-Based NXD, che usano un sistema di memorizzazione proprietario, utilizzano dei puntatori fisici. Poiché i link fisici sono più veloci di quelli logici, l' NXD (come i DB gerarchici) può restituire i dati più velocemente del DB relazionale, quindi presenta una buona scalabilità per quanto riguarda la restituzione dei dati .

Sfortunatamente questa scalabilità è limitata poiché negli NXD (come nei DB gerarchici) possiamo applicare lo schema fisico solo ad una particolare gerarchia,

ciò significa che gli NXD sono più veloci dei DB relazionali soltanto se la restituzione dei dati avviene nelle gerarchia secondo cui sono stati memorizzati.

Ad esempio supponiamo che le informazioni di un cliente sono memorizzate all'interno di ogni documento, allora se si chiede di restituire i documenti di vendita che riguardano un

cliente, tale richiesta verrà espletata velocemente, poiché i documenti vengono restituiti nell'ordine in cui sono stati memorizzati; invece se si chiede di restituire una lista degli ordini di vendita di un dato cliente, la risposta sarà molto lenta poiché non possiamo utilizzare i links fisici.

Per rimediare in parte a questo problema, gli NXD fanno un uso intenso di indici, spesso indicizzano tutti gli elementi e gli attributi, però se da una parte il tempo di risposta si abbassa, aumenta quello per la modifica; allora se la maggior parte delle nostre applicazioni sono solo di lettura, conviene intensificare il numero degli indici, invece se facciamo soprattutto delle modifiche ci dobbiamo aspettare delle prestazioni scadenti.

D'altra parte poiché gli NXD non supportano una completa normalizzazione (a causa del fatto che il vincolo di integrità referenziale è supportato solo parzialmente) una ricerca senza indice è più scadente di quella fatta nei DB relazionali.

Per esempio, supponiamo che vogliamo ricercare per tutti gli ordini di vendita una certa data, e le date non sono indicizzate. Nel DB relazionale ciò comporta la lettura di tutti i valori della colonna Data_Ordine, mentre in un NXD si dovrà effettuare la lettura dell'intero documento che rappresenta l'ordine di vendita, quindi non solo verrà letto il contenuto di ogni elemento <order date> ma verrà anche letto il contenuto di tutti gli altri elementi; va ancora peggio se l' NXD è di tipo Text-Based, perché il testo prima che venga confrontato con il valore della data cercata, deve essere analizzato e poi convertito nel formato della data.

Per concludere osserviamo che il problema della scalabilità dipende fortemente dalle nostre applicazioni, infatti se abbiamo un applicazione che in genere ha bisogno del dato così com'è stato memorizzato allora l'NXD presenta una buona scalabilità, invece se la nostra applicazione non ha una "vista privilegiata" del dato, la scalabilità messa a disposizione dall' NXD può comportare seri problemi.

2.5 Prodotti NXD commerciali

In questo paragrafo presentiamo alcuni dei prodotti NXD[23] che troviamo sul mercato. Una lista completa è presentata da: "James Hurst- Native Xml Database Products".

Extensibile Information Server 3.1

eXtensible Information Server (XIS) di eXceJon archivia e manipola dati Xml promettendo alte prestazioni ed alta stabilità, sebbene sia un po' in ritardo rispetto ai concorrenti per quanto riguarda il supporto delle query standard. XIS Service Pack 3.1 gira su Windows, Solaris e HP-UX; i prezzi partono da 40.000 dollari Usa.
Questo database utilizza XPath come linguaggio di query - un approccio datato che non consente la funzione di join o di ordinare i risultati delle query . La versione oggi disponibile offre un supporto solo parziale a XQuery; il pieno supporto, previsto con il rilascio del Service Pack 2, sarà disponibile in breve tempo. Estensioni proprietarie di XPath consentono di trarre vantaggio dalle funzioni di ricerca a tutto testo di XIS.
Come tutti i prodotti del gruppo, tranne Tamino Xml Server , eXcelon usa un linguaggio basato su Xml per gli aggiornamenti.
Offre anche Api di programmazione basate su Java e Com, trigger sul database e trasformazioni Xslt. XIS ha capacità di importare dati da sorgenti Ole Db od Odbc, ed è l'unico prodotto del gruppo ad offrire questa funzionalità di base.
XIS ha tipi indice separati per testo e dati numerici; supporta anche le transazioni ed ha una cache di memoria molto aggressiva, a vantaggio della velocità delle query semplici. Al pari di Ipedo XML Database, questo database supporta il lock a livello nodale per migliorare le performance delle applicazioni che prevedono un numero molto elevato di aggiornamenti dei dati. Ma XIS si spinge anche oltre: come Oracle, il suo schema di concorrenza multiversione permette alle query di accedere ai dati anche mentre un nodo è in fase di revisione.
Sfortunatamente, la validazione Xml avviene solo una volta, ovvero quando i file vengono importati. Se in seguito un update altera il funzionamento dei Dtd, eXcelon non se ne accorge.

Ipedo XML database 3.0

Ipedo XML Database 3.0 si presenta come il massimo della tecnologia oggi disponibile,

offrendo il supporto alle più recenti tecnologie sia sul fronte dell'Xml sia su quello dei Web service. La versione più recente presenta strumenti di amministrazione migliorati, a linea di comando e visuali; inoltre gestisce le transazioni e offre funzioni di controllo di versione per i documenti. Ipedo XML Database gira su Linux, Windows e Solaris e costa 29.000 dollari Usa per Cpu.
Il nuovo strumento grafico di amministrazione, Ipedo Administrator, permette di creare collezioni di documenti, importare Xml, attribuire indici ed eseguire query. Purtroppo, l'importazione di dati da database relazionali è impossibile senza ricorrere ad ulteriori add-on a pagamento.
Ipedo supporta sia XPath che XQuery. Il primo è più limitato del secondo, che vanta un vero e proprio linguaggio di scrittura per query e possibilità di ordinamento, applicazione di filtri, raggruppamenti ed esecuzione di join.
La definizione di logiche di dati è possibile sia sulla base di Dtd che di Xml Schema. Se si sceglie di alterare una definizione di struttura logica, Ipedo si preoccupa di validare tutti i documenti importati ed impedisce di compiere modifiche che ne alterino la struttura stessa: un approccio che risulta indubbiamente molto utile.
Ipedo Administrator ha un wizard che crea query XPath, ma che risulta carente nel tracciare gli errori: se si sbaglia la sintassi di una query , il server restituisce solo un generico messaggio di errore interno. L'uso con XQuery può generare messaggi di errore di informazioni. Il manuale per l'amministratore è d'aiuto, e consente di portare a termine qualche esempio.
Ipedo fornisce un controllo di versione su tutti i file, ed è l'unico del gruppo a farlo. È possibile eseguire il checkout di un file, modificarlo e reinserirlo nel database con l'aggiunta di commenti;
di grande utilità il rollback su tutte le versioni precedenti del file oggetto di revisione. Questa è una caratteristica di spicco per gli sviluppatori che creano sistemi di gestione di documenti Xml.
Ipedo include tre interfacce di programmazione: Soap, Java e Com.

NeoCore XMS 2.0

Decisamente compatto - il file di installazione occupa solo 2,9 MByte- NeoCore XMS

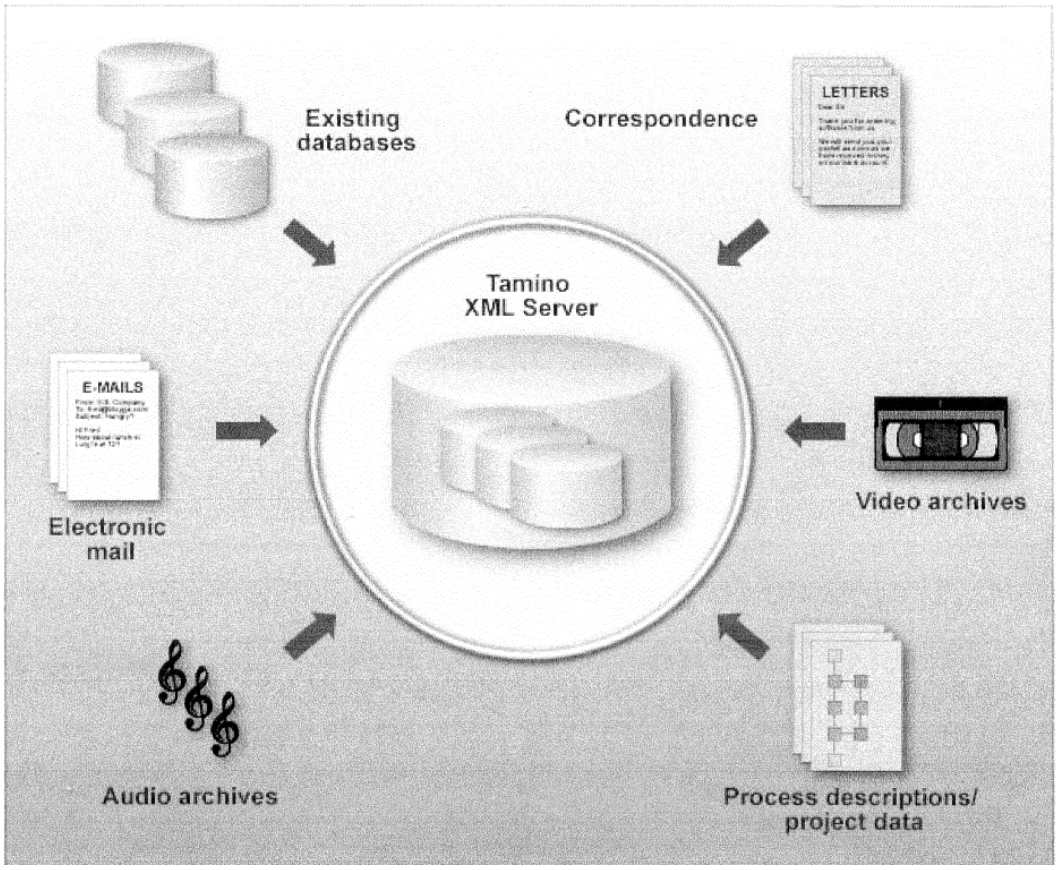
(Xml Management System) 2.0 di Neocore è un database Xml leggero ma decisamente ridotto all'osso: tra le sue prerogative di spicco troviamo le query basate su XPath, l'indicizzazione (tutti i tag sono indicizzati automaticamente) e il locking delle transazioni. Non ha però ne il supporto per XQuery ne un sistema di validazione sulla base di Dtd o Xml Schema, e nemmeno funzioni di trasformazione Xml.
In vendita a 5.000 dollari Usa, NeoCore costa molto meno dei concorrenti presentati precedentemente ma, viste le limitazioni, non ci sembra un prodotto raccomandabile se non alle organizzazioni per le quali il costo è il fattore primario di scelta.
NeoCore non usa un server standalone: richiede piuttosto gli application server iPlanet Web Server Enterprise Edition 4.1 o Bea WebLogic 6.1. Gli sviluppatori possono accedere al server tramite un'interfaccia Web che gira su Http utilizzando un protocollo di richiesta proprietario, non Soap, o mediante l'utilizzo di librerie (fornite da NeoCore) che inseriscono le chiamate Http in classi C++ o Java.
Diversamente da Ipedo e XIS, ma al pari di Tamino, NeoCore attua il lock a livello di documento; se quindi una parte qualsiasi di un documento è in fase di aggiornamento, tutte le altre risultano temporaneamente non aggiornabili. In alcune applicazioni ciò può comportare uno scadimento significativo delle prestazioni.
Lo strumento di amministrazione di Neocore, basato su Java, si chiama Access Control Manager e fornisce funzionalità decisamente spartane.
I prezzi, come accennato, partono da 5.000 dollari Usa (per un massimo di due richieste simultanee, un data base di 50 MByte e una Cpu) e arrivano fino a 75.000 dollari per Cpu.
Il software gira in ambiente Windows e Solaris; è previsto per il futuro il supporto per Linux.

CAPITOLO 3

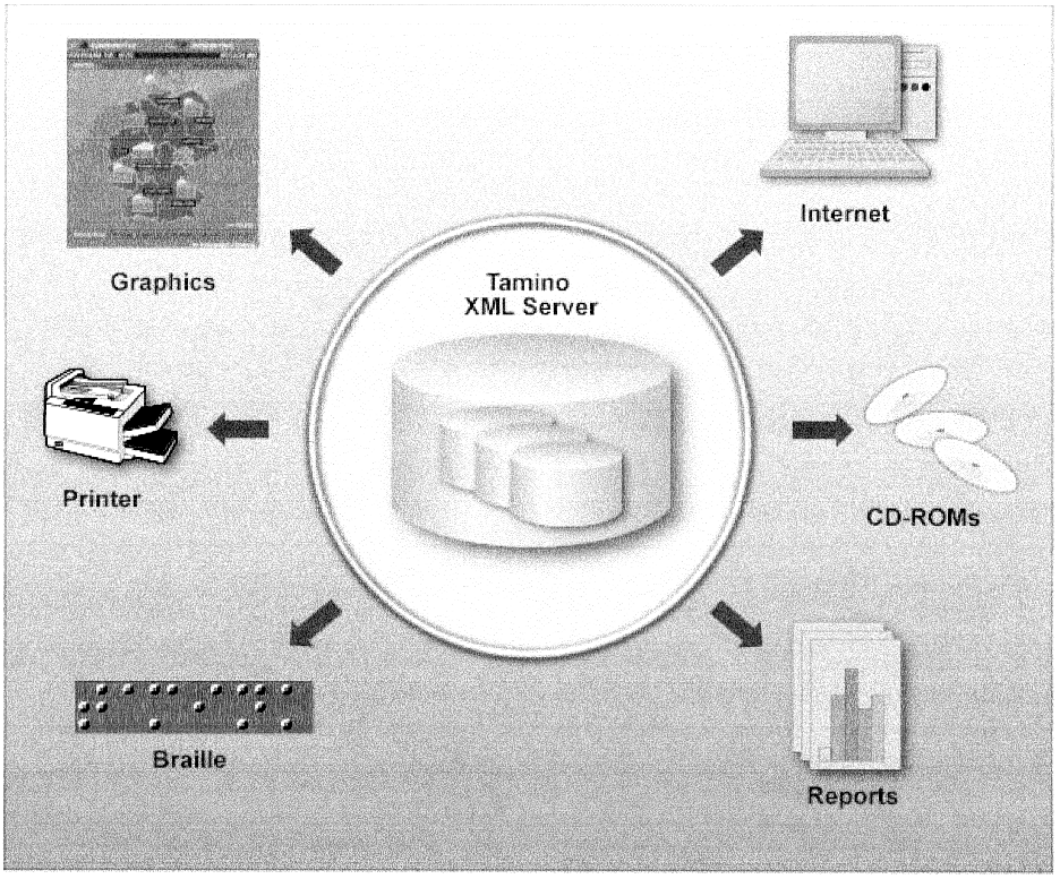
TAMINO XML SERVER 3.1

Il supporto necessario per la Total Business Integration.

In passato sistemi e processi aziendali diversi venivano integrati a livello di dati o di singola transazione, oggi XML svolge questa integrazione a livello di documento. Ciò significa che i processi aziendali possono essere tradotti in un singolo documento, capace di identificare le transazioni subordinate e interdipendenti ad essi riconducibili. L'elemento chiave che caratterizza Tamino XML Server è proprio il sistema di database management XML nativo (quindi Tamino non si presenta come uno strato software – middlware- tra DBMS e applicazioni web). Grazie a Tamino – e alla sua gamma di servizi accessori- è possibile implementare, gestire e integrare soluzioni di e-Business a vantaggio di una completa integrazione aziendale. Il server XML[24], prodotto dalla Software AG, agevola il raggiungimento di questi obbiettivi, in quanto è in grado di archiviare e di gestire tutti i dati e tutte le informazioni in formato XML.
Tali informazioni possono provenire da fonti diverse, inoltre non è detto che siano in formato XML (ad esempio in formato grafico o video), poiché Tamino è in grado di immagazzinare anche dati non in formato XML, precisamente consente di esprimere la struttura di tali dati in formato XML, in modo da poterli poi immagazzinare e trattare come oggetti XML.



Uno dei vantaggi di Tamino è quello di riuscire a sfruttare al meglio la capacità di trasformazione di XML per pubblicare, i dati immagazzinati, in formati diversi (tramite fogli di stile XSLT e XSL:FO) di modo che la stessa informazione può essere utilizzata da dispositivi diversi



Tamino è tipicamente sviluppato in ambienti comprendenti alcune combinazioni di Windows, Unix, e sistemi main-frame, e usa degli standard accessibili all'industria per la comunicazione e l'inter-operabilità.
Le funzioni principali svolte da Tamino sono:

- Immagazzinamento nativo e restituzione degli oggetti XML (XML Store e X-Machine)
- Interfaccia per le applicazioni esterne e per le fonti di dati esterne (X-Node)
- Immagazzinamento interno di dati SQL e relativa restituzione (SQL Store e SQL Engine)
- Rilevamento di dove sono immagazzinati i dettagli delle informazioni e restituzione di tali dettagli (Data Map)

- Amministrazione centrale di più nodi server Tamino su Internet attraverso i browsers Internet (Tamino Manager)

Tamino, offre inoltre, una serie di servizi e soluzioni aggiuntive quali:

- Tamino Schema Editor
- Tamino Interactive Interface
- Tamino X-Plorer
- WebDAV Server for Tamino
- Application Programming Interfaces
- Tamino X-Application

3.1 Architettuura

La seguente figura, illustra l'architettura generale di Tamino[24]

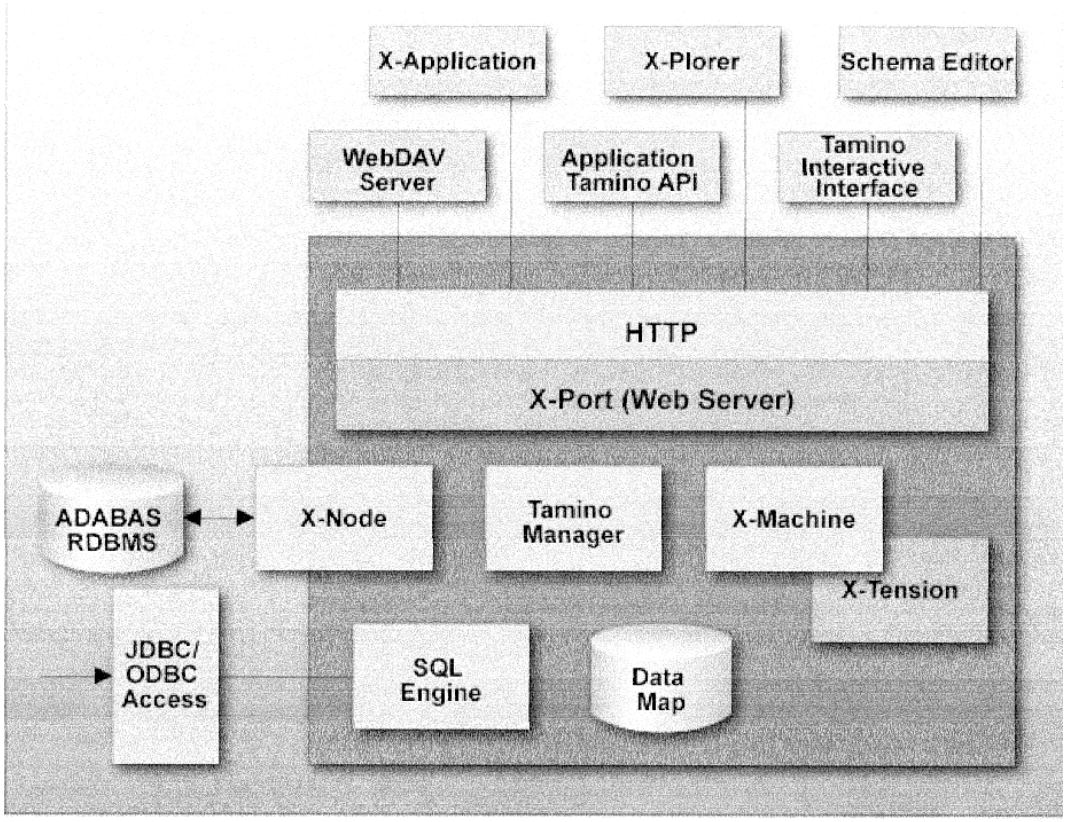


fig. 1

In effetti possiamo suddividere Tamino in due parti:

- Tamino XML Server
- Tamino Product Components

TAMINO XML SERVER

Il Tamino Xml Server[24] non è solo un archivio di dati, difatti lo possiamo suddividere in sei parti principali:

- X-Machine
- X-Node
- Data-Map
- SQL Engine
- Tamino Manager
- X-tension

X-Machine:

La componente principale di Tamino è X-Machine,il motore del DB. X-Machine ha il compito fondamentale di archiviare ed estrarre l'XML. Bisogna subito notare che Tamino consente l'archiviazione dei dati, oltre che nel proprio XML Data Store, anche in un DB relazionale interno e in DB esterni accessibili mediante ODBC e OLE-DB. L'archiviazione e l'estrazione degli oggetti XML, può essere schematizzata come in figura:

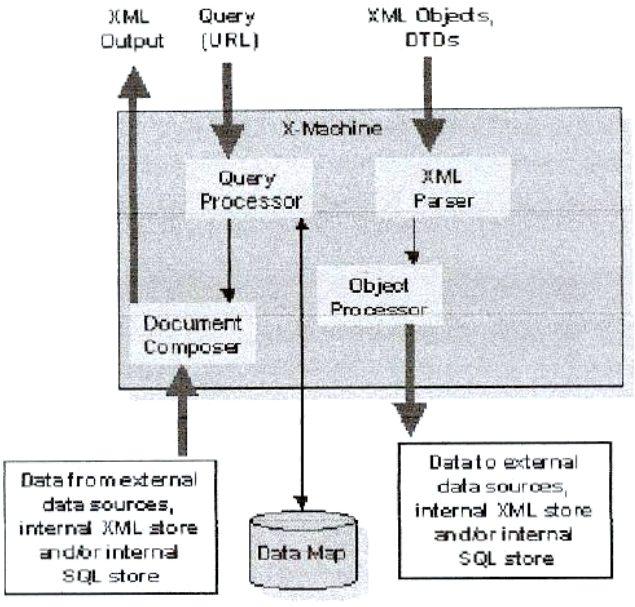


fig. 2

Come si evince dalla fig.2, X-Machine viene supportata dai seguenti sottocomponenti:

- XML Parser:

  Gli oggetti XML memorizzati dall'X-Machine, sono descritti dallo schema contenuto nel Data-Map. L'XML Parser (contenuto all'interno della X-Machine) verifica la sintassi degli schemi e assicura la corretta formattazione degli oggetti in ingresso.

- Object Processor:

  Tale componente è usato per memorizzare gli oggetti nello Store XML nativo. Ad esempio, ogni dato SQL viene memorizzato in tabelle e colonne SQL seguendo il corrispondente schema.

- Query Processor:

  Il linguaggio di query usato da Tamino è X-Query. Il Query Processor Interagisce con il modulo Document Composer per recuperare gli oggetti XML secondo lo schema memorizzato nel Data Map.

- Document Composer

  Il Document Composer costruisce l'oggetto delle informazioni secondo gli schemi del Data Map e li restituisce come documenti XML. Nel caso più semplice, vengono restituiti oggetti memorizzati all'origine in XML; nei casi più complessi, per comporre un oggetto XML da una fonte dati non-XML, è necessaria la comunicazione con SQL Engine o con il modulo X-Node.

X-Node:

Per l'accesso a data-source esterne si utlizza la componente X-Node:

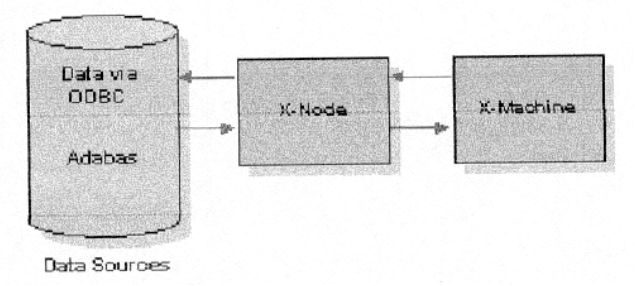


fig.3

Tale componente è nata per permettere l'integrazione di Tamino con altri mondi. L'utilizzo di X-Node è assolutamente trasparente all'utente, un singolo documento XML estratto da Tamino può essere composto da dati elementari estratti da multipli database esterni.

Data-Map:

Il Data-Map, l'analogo del Data Dictionary negli RDBMS, contiene sostanzialmente la struttura dei dati in Tamino:

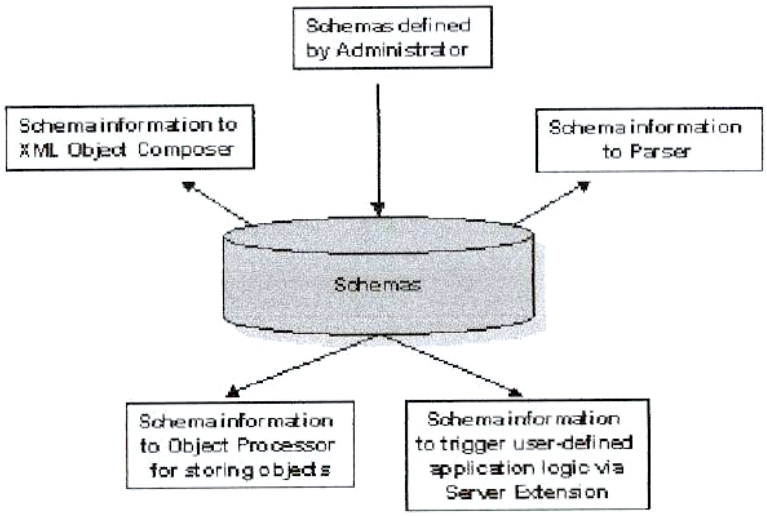


fig. 4

In particolare contiene le informazioni che sono richieste per effettuare le seguenti funzioni:

- Validazione secondo uno schema logico
- Memorizzazione e indicizzazione degli oggetti XML
- Mapping di dati in differenti strutture di dati (per esempio DB relazionali), ciò facilita l'integrazione dei dati.
- Mapping di dati dal DB Adabas

La definizione degli schemi nel Data-Map è supportata da tool grafici (Tamino Schema Editor) che ci aiutano a non commettere errori.
Si osserva infine, che lo stesso Data-Map è strutturato in formato XML, e può essere interrogato dall'utente mediante semplici query eseguite con X-Query.

SQL Engine

Tamino include anche un RDBMS, detto SQL Engine, quindi è possibile gestire anche dati in tabelle utilizzando SQL:

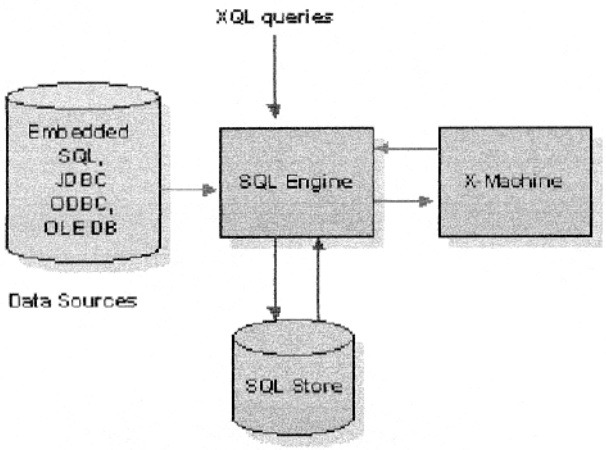


fig.5

Gli stantement SQL accettati da SQL Engine possono essere inviati a Tamino in diversi modi:

- Internamente, dal'X-Machine

- Dalle applicazioni, sia attraverso SQL embedded, sia attraverso una delle interfacce standard dei DB come ODBC, JDBC, oppure OLE DB
- Dal Tamino Manager.

Tamino Manager

Tamino Manager è un tool grafico che consente l'amministrazione di Tamino:

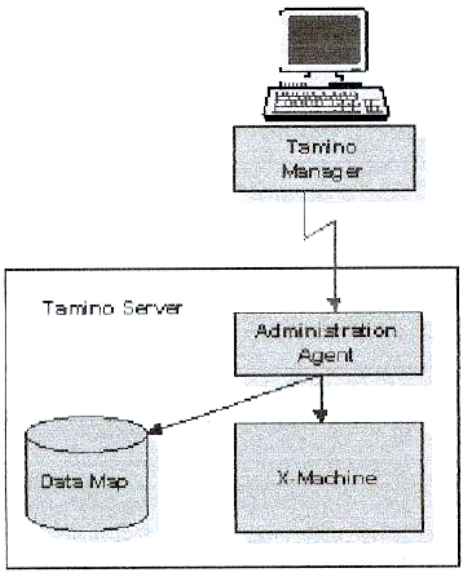


fig. 6

Tale tool è implementato come un'applicazione client-server ed è integrato nel System Management Hub (ambiente del software AG multipiattaforma per la gestione unificata dei prodotti del software AG). Attraverso il Tamino Manager è possibile svolgere le funzioni principali come: creare DB, start/stop server, backup, restore, load, ecc..

È possibile inoltre, accedere al Data Map per "adattare" le direttive del parser, aggiungere o cambiare oggetti, configurare schemi, e installare Tamino Server Extensions.

X-Tension

Un'oggetto XML (basato sullo schema definito nel Data Map) può essere "mappato" su una funzione definita dall'utente che, viene eseguita o quando inizia il processo di parsing, o quando si compone l'oggetto.

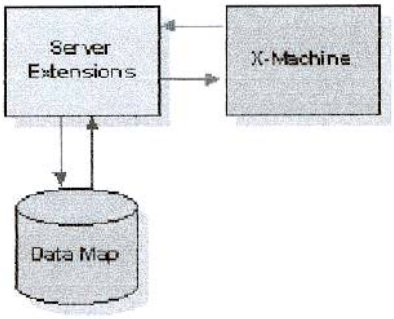


fig. 7

Le Server Extensions vengono installate in Tamino tramite il Tamino Manager, la realizzazione di tali funzioni di estensione può essere basata su COM con un wizard e un add-in per C C++, e per MS-Visual Studio. Un'altra possibilità è quella di usare Java.

Tamino Product Components

I componenti di Tamino Server[24] consentono di semplificare l'uso e lo sviluppo delle applicazioni con Tamino. Se si sceglie l'installazione tipica o full vengono installati automaticamente, altrimenti è possibile reperirli via download dalla Tamino Community Web.

- Tamino Schema Editor
- Tamino Interactive Interface
- Tamino X-Plorer
- Web Dav Server per Tamino

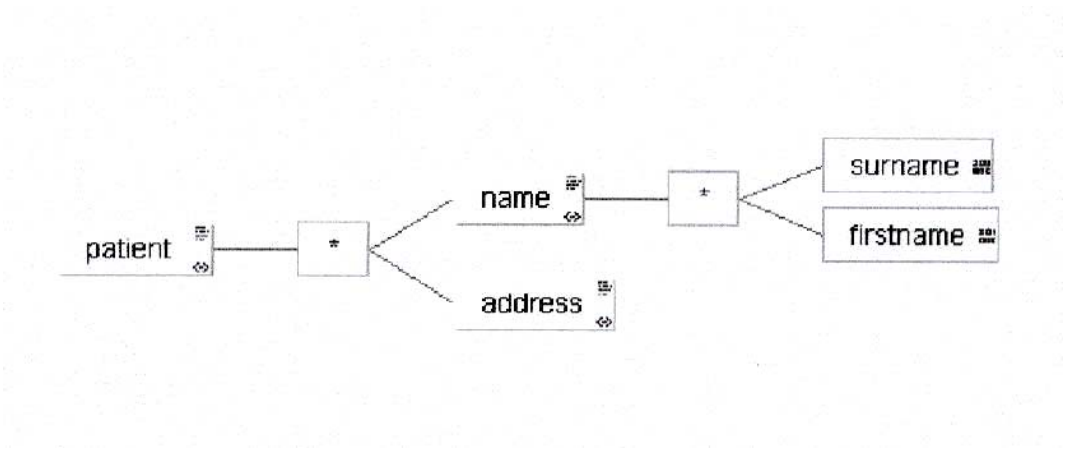
- Applications Programming Interfaces
- Tamino X-Application

Tamino Schema Editor

La descrizione dei documenti XML avviene fuori da Tamino, e può essere fatta tramite DTD (ma è già previsto il pieno supporto di XML Schema quando questo sarà perfettamente formalizzato dal W3C). invece nel DB occorre definire la struttura (schema) dei dati che viene conservata, come abbiamo visto nel Data Map.In realtà non è strettamente necessario definire uno schema, difatti Tamino consente di memorizzare Well –Formed XML object senza schema, però in tal caso non sarà ottimizzato il processo di indicizzazione, e quindi l'estrazione di XML.

Per la definizione dello schema dei dati in Tamino è possibile utilizzare un apposito linguaggio detto Schema Language (il quale è basato sullo standard XML Schema)oppure possiamo utilizzare il Tamino Schema Editor. Tale tool ci facilita il compito perché permette di definire lo schema graficamente come un albero, dopodiché la traduzione nello Schema Language di Tamino viene fatta automaticamente.

La seguente figura illustra un schema definito con il Tamino Schema Editor:



Lo Schema Editor può essere anche utilizzato, per convertire un esistente DTD nello schema utilizzato da Tamino. In tale schema i dati vengono strutturati a tre livelli: *Collection, Doctype,* e *Nodi* .Una Collection include più Doctype ognuno dei quali include più nodi.

La *Collection* è un insieme di tipi di documenti, può essere associata ad un intero DB classico.

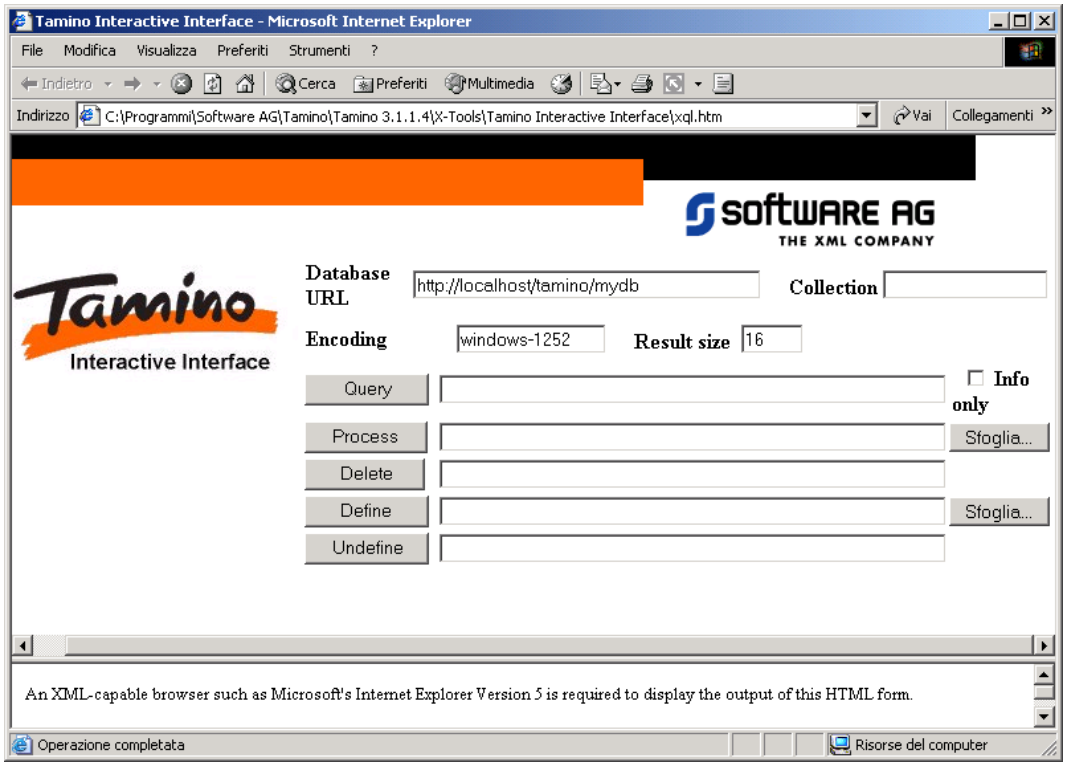
Un *Doctype* (in genere l'elemento root di un DTD) è un tipo di documento e può essere assimilato ad una tabella relazionale.

Il *Nodo* è il dato elementare e può essere associato ad una colonna dei DB relazionali. Ma un *Nodo* può essere anche composto da altri *nodi,* consentendo di creare documenti XML di qualsivoglia complessità. Tamino fornisce anche la possibilità di associare dei *Datatype* ai nodi elementari in fase di definizione dello schema. Una volta definito lo schema dei dati , Tamino memorizza ed estrae i dati secondo tale schema.

Tamino Interactive Interface:

E' un interfaccia di programmazione browser- based del server Tamino; permette di caricare e cancellare dati, eseguire query, definire e cancellare schemi dati e collection. Le stesse funzionalità possono essere ottenute anche inviando, a mezzo HTTP, dei comandi alla X-Machine.

La seguente figura illustra il Tamino Interactive Interface:



Come possiamo notare ci sono i seguenti campi:

- *Database URL*, consente di specificare la URL del Tamino DB a cui vogliamo accedere.

- *Collection,* consente di specificare il nome della Collection a cui si vuole accedere. Il Tamino Interactive Interface ignora tale campo se dobbiamo effettuare operazioni come Definire una Collection, Delete una Collection e uno Schema. Ma visto che tutte le altre operazioni richiedono comunque il nome della Collection, allora conviene comunque riempire tale campo.

- *Encoding,* in tale campo va inserito la codifica dei caratteri. Con Microsoft Internet Explorer, è inizializzato alla codifica che abbiamo, specificato quando abbiamo settato il browser .

- *Result size*, in tale campo va specificato il numero di documenti che saranno ritornati dopo la query. Il valore di tale campo deve essere un numero positivo intero , altrimenti il campo viene settato al valore di default, che è 16. Poiché tramite il Tamino Interactive Interface la query è fatta in modo interattivo, piuttosto che attraverso un'API, allora è sufficiente indicare un numero di documenti facilmente trattabili.

I restanti campi corrispondono a dei comandi con i quali comunichiamo direttamente con X-Machine di Tamino.

- *Query*, in tale campo va inserita la query di selezione, espressa tramite il linguaggio X-Query . Naturalmente prima di eseguire la query occorre popolare il DB.

- *Process,* prima di caricare gli oggetti XML bisogna caricare lo Schema, ciò avviene utilizzando il campo *Define,*dopodiché in Process andiamo a specificare il nome del file che contiene l'oggetto XML o gli oggetti XML che vogliamo caricare nel DB. Se il load degli oggetti è andato a buon fine, sarà ritornato un documento XML in cui possiamo individuare degli elementi <ino: message> in cui return value deve essere 0, e degli elementi <ino: object> in cui possiamo leggere il nome del doctype e della collection in cui l'oggetto viene caricato.

- *Delete*, tale campo ci consente di cancellare gli oggetti precedentemente caricati; precisamente utilizzando la sintassi di X-Query , possiamo specificare il contenuto o l'oggetto I D, che si vuole eliminare.

- *Define,* in questo campo dobbiamo specificare il file che contiene lo Schema definition. Uno Schema definition, descrive la struttura , ma non il contenuto, di un set di oggetti che andremo a caricare nel DB. Possiamo scrivere il path dove si trova il file, oppure possiamo utilizzare il pulsante Browser, dopodiché per caricare lo schema all'interno del DB e della collection usiamo il tasto Define. Se

eventualmente avevamo riempito il campo Colection, il nome inserito verrà ignorato; poiché la collection selezionata è quella indicata dal nome dell'attributo dell'elemento <tsd : collection>, che troviamo all'interno dello Schema definition, stessa cosa dicasi per il nome del doctype, in tal caso l'elemento di riferimento è < tsd : doctype>.

- *Undefine,* in tale campo và specificato il nome del doctype o di una collection che vogliamo cancellare. Se cancelliamo una collection, cancelliamo anche tutti i doctype e i dati in essa contenuti.

È doveroso osservare che il Tamino Interactive Interface è case-sensitive quindi ad esempio il nome "ospedale" è diverso dal nome "Ospedale".

Tamino X-Plorer

Presenta un interfaccia grafica tramite la quale è possibile eseguire le seguenti funzioni:

- Esplorazione di un DB di Tamino
- Interroga un DB di Tamino
- Cura la manutenzione delle strutture e dei contenuti di Tamino
- Mostra , crea , e prepara oggetti in Tamino
- Invoca tools che supportino l'utente nello sviluppo delle applicazioni di Tamino, come Tamino X-Application Generator, Mozquito, ecc.

Tamino WebDav Server

Il Tamino WebDavServer è basato su http e WebDav, il nuovo standard per la Web-authoring collaborative, e permette l'interrogazione con clients come quelli di Office2000 (TM). Gli utenti, usando le cartelle Microsoft Web e gli strumenti standard di Office, possono preparare risorse web con la stessa facilità con cui essi creano risorse nella directory locale. La sola differenza è che queste risorse sono subito accessibili ad altri via intranet, extranet o Internet

API

Ci sono diverse API che sono installate con l'installazione tipica di Tamino:

- Tamino API

  Trattasi di API molto flessibili e object-oriented, offrono al programmatore Java un facile accesso ai dati immagazzinati in Tamino, usando differenti object model (DOM, SAX, JDOM), oppure usando l'accesso strema-based. Nelle Tamino API per Java c'e anche Tamino EJB API che permette ai creatori di applicazioni di scrivere application business usando Tamino come gestore delle risorse in un ambiente Enterprise Java Beans.

- http Client API for ActiveX

  E' un API per le piattaforme Windows, ed usa il DOM per accedere e manipolare i dati immagazzinati in Tamino. Può essere usata per applicazioni scritte in C++ e Visual Basic.

- http Client API for Java

  E' un API per piattaforme Java, ed usa ed usa il DOM per accedere e manipolare i dati immagazzinati in Tamino.

- http Client API for JScript

  E' un API che usa il DOM per accedere e manipolare i dati immagazzinati in Tamino. Può essere usata per applicazioni in Microsoft Internet Explorer (versione 5 successive)

Tamino X-Application

Da Software AG Community Web è possibile scaricare il Tamino X-Application gratis (http://www.tamino.com/developer/x-application/default.htm). Tamino X-Application permette di accedere ai dati che sono memorizzati in Tamino usando un standard browser interface.

3.2 Qualche caso reale…

Tamino e i suoi componenti offrono un ampio range di soluzioni per lo sviluppo delle Elettronic Business Application, a tal proposito riportiamo tre esempi estratti dall' articolo: " XML dalle molte parole ai primi risultati" [25]

Trasporto sostanze pericolose

L 'utilizzo massiccio, da parte delle industrie chimiche, di prodotti tossici pone in primo piano il problema della loro gestione. Per la sicurezza dell'uso dell'accumulo e del trasporto di tali sostanze, sono in vigore numerose e complesse leggi che contengono tutte le informazioni necessarie, ma che difficilmente possono essere disponibili rapidamente, ad esempio in caso di incidente (quando è vitale avere le corrette conoscenze per prevenire o contenere i danni).

Klaus Dieter Mehler è il Direttore Generale della Magis, un'azienda di Bad Camberg specializzata in servizi ambientali. La Magis, tramite un apposito servizio basato sul Web, offre i necessari documenti di accompagnamento per il trasporto dei materiali pericolosi. Mehler conferma la mancanza di mezzi elettronici per lo scambio delle informazioni riguardanti la sicurezza delle industrie chimiche e critica l'EDI (Electronic Data Interchange) in quanto troppo complesso e soprattutto troppo costoso per le piccole e medie imprese. La Magis ha sviluppato un'applicazione pilota basata su Tamino per dimostrare come si possa effettuare il trasferimento di dati commerciali e aziendali sul Web. I dati relativi alla sicurezza di un'industria chimica sono stati memorizzati come documenti XML e spediti via web ad altre aziende. Per poter ricevere questi File, ai destinatari è richiesto soltanto di possedere un Browser XML compatibile. Se si pensa che, volendo utilizzare la soluzione EDI in una situazione analoga, si dovrebbero considerare differenti interfacce per ogni cliente, si vede chiaramente come l'XML sia il mezzo ideale per tale scambio di informazioni.

Trattamento malati HIV

Nel sistema sanitario, l'uso delle moderne tecnologie informatiche rende possibili nuovi modi di raccogliere, catalogare e distribuire le informazioni mediche. L'università di

St.Gallen ha allestito un importante progetto Web che fa parte di uno studio a lungo termine sul virus dell'AIDS. Usando la tecnologia XML, questa università ha già raccolto i dati clinici di circa 10.000 pazienti affetti dal virus HIV e li ha riuniti in un Database XML. Nel 1998 è stata progettato e realizzato un prototipo di applicazione basato su XML e conosciuta come Electronic Study Form (ESF). La disponibilità universale sul Web, l'indipendenza dalle piattaforme e dai sistemi proprietari, la buona leggibilità e l'idoneità al trattamento dei dati medici sono i motivi che hanno portato alla scelta di questo linguaggio. I dati, provenienti da varie fonti, sono stati memorizzati in formato XML su Tamino, con i seguenti risultati: accelerazione dei processi di inserimento, memorizzazione e trasmissione dei dati e riduzione del numero degli errori.
E' stata resa disponibile, in questo modo, una grande quantità di dati medici che, da un lato permette una cura migliore dei pazienti e dall'altro fornisce informazioni essenziali per la ricerca medica sull'HIV.

Gestione dell'Ambiente

La rete GEIN 2000 (German Environment Information Network) ha come obiettivo quello di consentire l'accesso ai dati disponibili sui siti Web di numerose organizzazioni del settore pubblico e di servire da informatore per quanto riguarda le problematiche ambientali. La responsabilità dello sviluppo del progetto è stata affidata al Sema Group, che ha deciso di implementarlo sulla base del nuovo standard XML usando Tamino.
Uno dei primi vantaggi di XML è stata la velocizzazione dello scambio di informazioni fra i diversi corpi dello stato, indipendentemente dalle piattaforme operative. GEIN 2000 non controlla solo lo scambio di dati fra enti esterni, ma usa XML anche per processi interni, per esempio includendo una raccolta di dati geografici che svolge le stesse funzioni di un motore di ricerca. La base per GEIN 2000 è l'applicazione XML RDF (Resource Description Framework) che permette di effettuare delle Query logiche più complesse di quanto sia possibile con l'HTML. GEIN 2000 può inoltre valutare gli indici localmente disponibili, le funzioni di ricerca di indirizzi locali e, soprattutto, i siti Web dinamici.

Concludiamo questa breve panoramica di Tamino con alcune notizie sull'installazione[]:

Requisiti ed installazione

Tamino[24] è disponibile per le seguenti piattaforme: Windows NT 4.0 (SP 6a), Windows 2000 Professional, Server ed Advanced Server (SP2), Sun Solaris 7 ed 8, IBM AIX 4.3.3, RedHat Linux 7.2, SuSE Linux 7.1, 7.2 ed Enterprise Server 7 (su piattaforma Intel e, per la Enterprise Server 7, anche S/390), HP-UX 11.0 a 32 bit, ed OS/390. Le funzionalità di personalizzazione offerte dal servizio *X-Tension non* sono disponibili per AIX 4.3.3, SuSE Enterprise Server 7 ed OS/390. Si può scaricare via Web una versione demo per Windows o Linux, il cosiddetto XML Starter Kit (http://www.xmlstarterkit.com), che comprende, oltre ad XML Server 3.1, anche il WebDAV Server, una serie di X-Tools e stumenti di amministrazione Web, ovvero il System Management Hub e l'Extended Transport Service.

La versione di valutazione è valida 90 giorni, il database può avere una grandezza massima di 20 Mb, ed il numero di accessi concorrenti è limitato a 3. È necessario fornire, per effettuare il download dello Starter Kit, un indirizzo e-mail valido, al quale verrà inviato un file "ino311.xml" contenente una chiave di registrazione del prodotto; la locazione fisica di questo file dovrà essere indicata durante la procedura di setup. È necessario che sulla macchina dove verrà installato XML Server sia presente un Web Server, e che questo sia attivo durante l'installazione; nella versione per Windows 2000 dello Starter Kit, da noi provata, viene fornito anche Apache 1.3.22. L'installazione può avvenire tramite un'interfaccia grafica, o in *batch mode*. Su piattaforma Windows, i requisiti minimi sono un Pentium II a 300 MHz con 128 Mb RAM (consigliati 256 Mb) e 270 Mb di spazio su disco, in una partizione NTFS (che comprendono anche 26 Mb necessari per l'installazione del WebDAV Server). È raccomandato l'uso di Internet Explorer 5 o Netscape Navigator 4.76, mentre è sconsigliato quello di IE 6 e Netscape 6. Se si vogliono sviluppare *plug-in* personalizzati tramite le *X-Tensions*, vengono richiesti gli appropriati ambienti di sviluppo (Visual Studio o JDK), mentre per l'uso di X-Application è richiesto un *JSP engine* come

Tomcat 3.3. Sono richiesti i privilegi di amministratore. L'installazione può essere "Typical", "Full" o "Customized": le prime due installano sia le componenti server che quelle di sviluppo, mentre la terza permette di selezionare i singoli componenti che verranno installati. È possibile aggiungere in seguito una o più unità non installate, ripetendo la procedura di installazione: curiosamente, però, la procedura di setup non è in grado di determinare quali componenti siano già installate sul sistema.

CAPITOLO 4

Casi d' Uso di Tamino

In questo capitolo, illustriamo (utilizzando la versione demo del prodotto scaricata via Web: http://www.xmlstarterkit.com) qualche semplice esempio sull'utilizzo di Tamino seguendo la procedura base, che viene comunque adottata in ogni contesto lavorativo in cui si usa Tamino. Tale procedura[24] è composta dai seguenti passi fondamentali:

- Descrizione della struttura dei dati, quindi i documenti XML dovranno essere accompagnati dai relativi DTD. Comunque Tamino accetta anche semplicemente documenti XML well formed.

- È consigliabile definire il Tamino Schema dei dati, utilizzando lo Schema Language di Tamino (basato su XML-Schema).

  Il Tamino Schema può essere scritto usando un semplice editor di testo, oppure lo Schema Editor di Tamino. La definizione del Tamino Schema è importante, poiché facilita l'individuazione del dato, e permette di ottimizzare l'indicizzazione per una ricerca più rapida. In mancanza del Tamino Schema viene, invece, applicato un'indicizzazione di default che permette di restituire full text.

- Una volta memorizzato il Tamino Schema nel Data Map, possiamo memorizzare le istanze dello schema. La memorizzazione può avvenire in diversi modi, uno di questi è quello di utilizzare il Tamino Interactive Interface, un altro invece, si basa sull'utilizzo delle API.

- Usiamo X-Query per recuperare i dati. X-Query è il linguaggio standard di query di Tamino. Possiamo chiedere i dati attraverso il browser oppure tramite il Tamino Interctive Interface.

- Costruiamo un'applicazione con i dati ritornati, usando la vasta gamma di strumenti e servizi di Tamino.

ESEMPIO 1

Il documento XML che vogliamo gestire con Tamino è un documento che contiene le informazioni degli impiegati di un azienda. Supponiamo che il documento iniziale riporti le informazioni di quattro impiegati :

<?xml version="1.0" encoding="ISO-8859-1" ?>

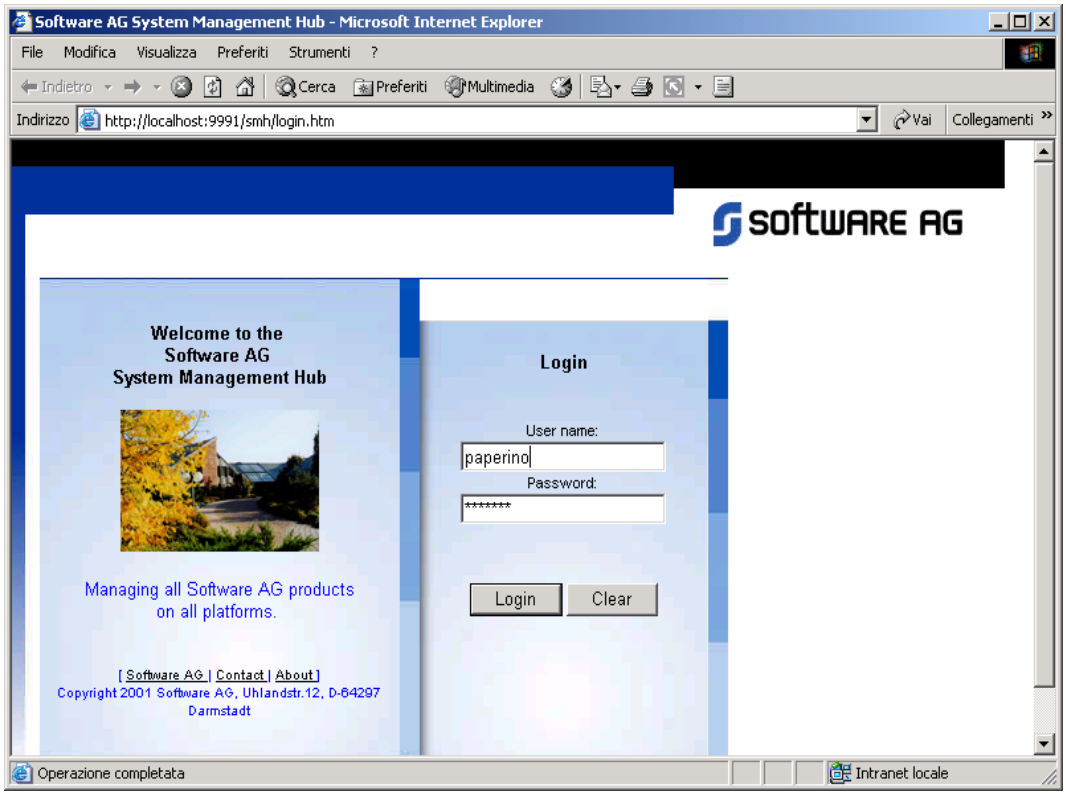
```
<!DOCTYPE impiegati SYSTEM "impiegati.dtd">
<impiegati>
 <impiegato id="M.R">
  <nome>Mario Rossi</nome>
  <anno_nascita>1950</anno_nascita>
  <dipartimento>amministrazione</dipartimento>
  <email>mrossi@foo.com</email>
 </impiegato>
 <impiegato id="F.B">
  <nome>Filippo Bianchi</nome>
  <anno_nascita>1950</anno_nascita>
  <dipartimento>amministrazione</dipartimento>
  <email>fbaldi@foo.com</email>
  </impiegato>
 <impiegato id="A.V">
  <nome>Alice Verdi</nome>
  <anno_nascita>1960</anno_nascita>
  <dipartimento>economato</dipartimento>
  <email>averdi@foo.com</email>
  <url href="http://www.foo.com/~averdi/" />
 </impiegato>
<impiegato id="A.B">
  <nome>Antonio Bisio</nome>
  <anno_nascita>1955</anno_nascita>
  <dipartimento>economato</dipartimento>
  <email>abisio@foo.com</email>
  <url href="http://www.free.com/~abisio/" />
 </impiegato>
</impiegati>
```

Le informazioni di ogni singolo impiegato sono conformi alle regole grammaticali, o vincoli, espresse dal seguente DTD,contenuto nel file “impiegati.dtd”:

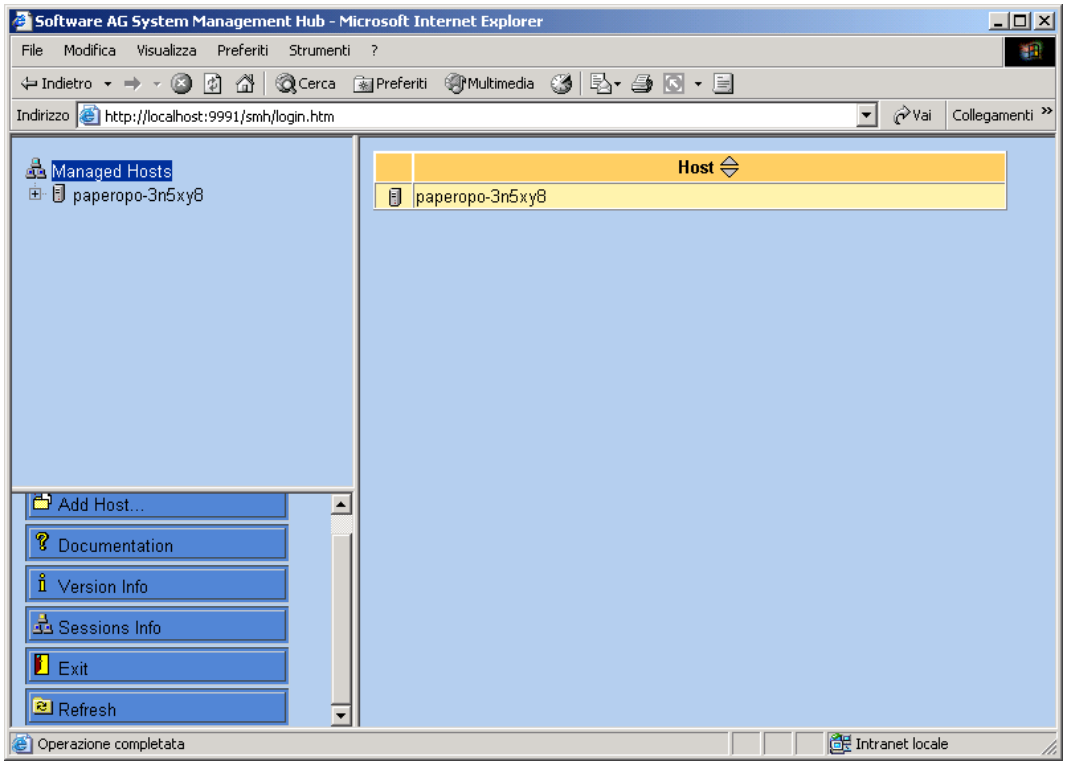
```
<!ELEMENT impiegati (impiegato)*>
<!ELEMENT impiegato (nome,anno_nascita,dipartimento, email, url?)>
<!ATTLIST impiegato id  ID  #REQUIRED>
<!ELEMENT nome (#PCDATA)>
<!ELEMENT anno_nascita (#PCDATA)>
<!ELEMENT dipartimento (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT url EMPTY>
<!ATTLIST url href CDATA #REQUIRED>
```

CREARE IL DATABASE

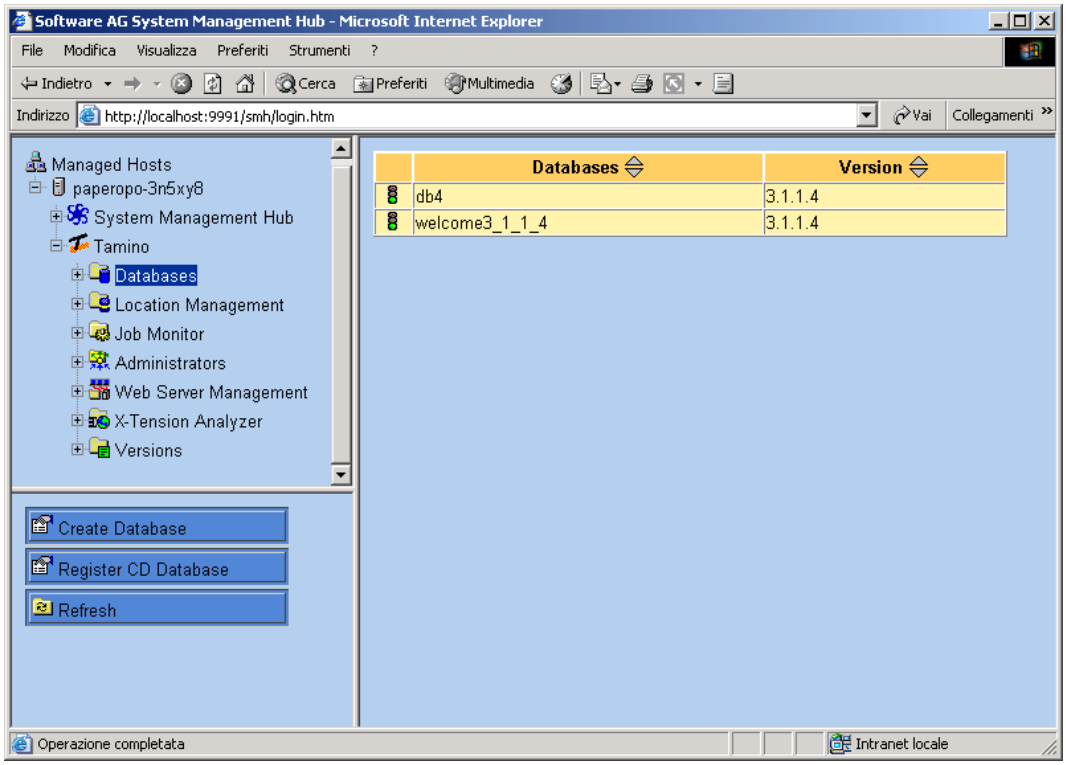
Il primo step da compiere è quello di creare un database, attiviamo quindi il *Tamino Manager* (inserendo User name e password)



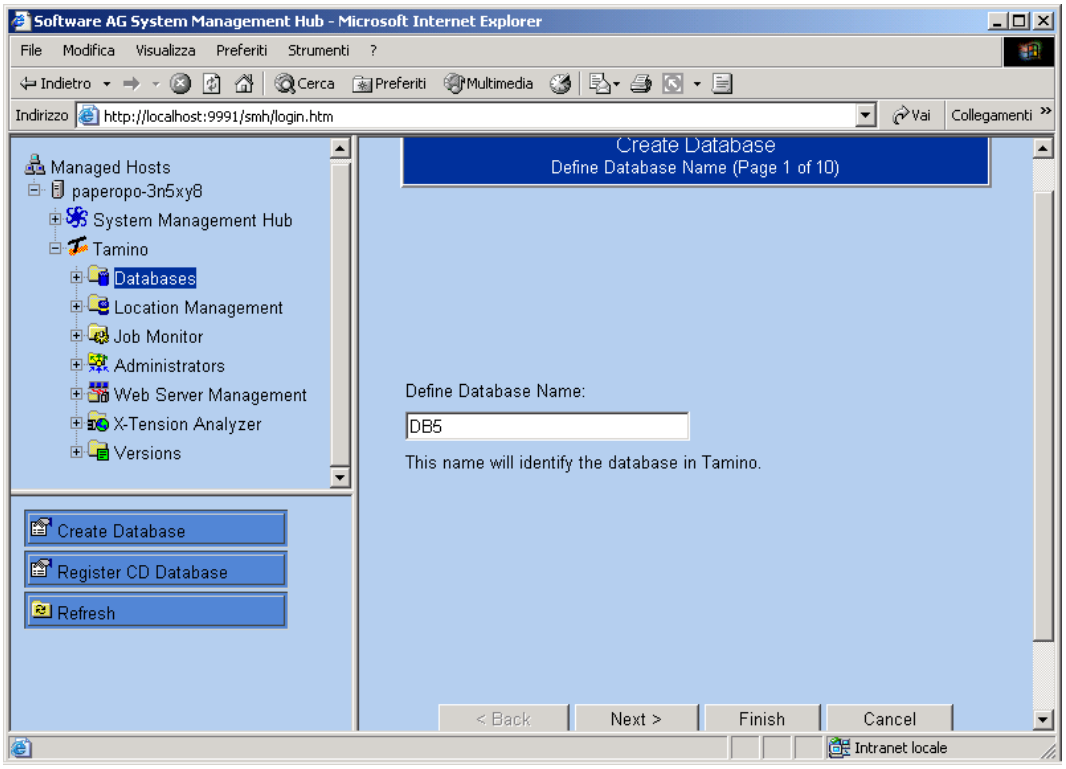
Dopodiché cliccando su Login si attiva la seguente finestra:



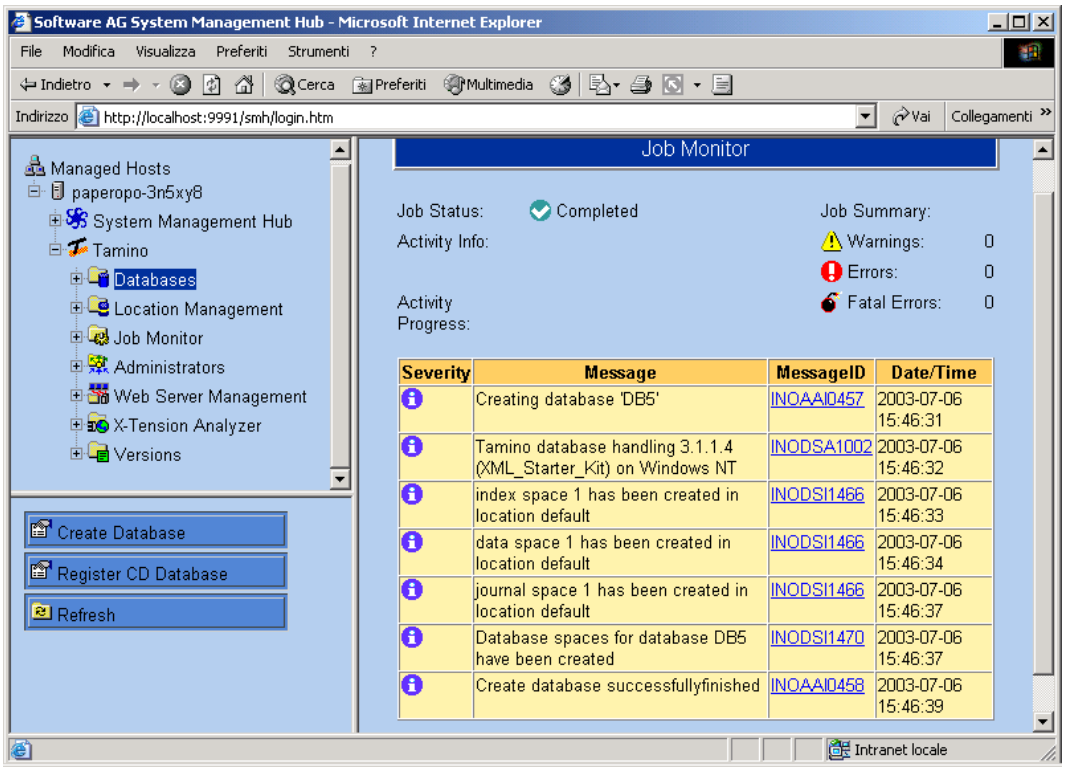
Il “Managed Host” mostra il nome dell’ “Host” e il nome del “domain”, rispettivamente pari a “paperopo” e “3n5xy8”. A questo punto, cliccando sul segno + , espandiamo il Managed Host , successivamente “espandiamo” Tamino.,dopodiché cliccando su Database con il tasto sx del mouse comparirà, in basso a sx , una serie di pulsanti, uno di questi è Create Database:



Cliccando su Create Database si aprirà un wizard che ci consente di settare una serie di parametri del Database ma, poiché il nostro esempio è puramente didattico, accettiamo i parametri di default, e quindi dopo aver specificato il nome del Database, clicchiamo su finish.

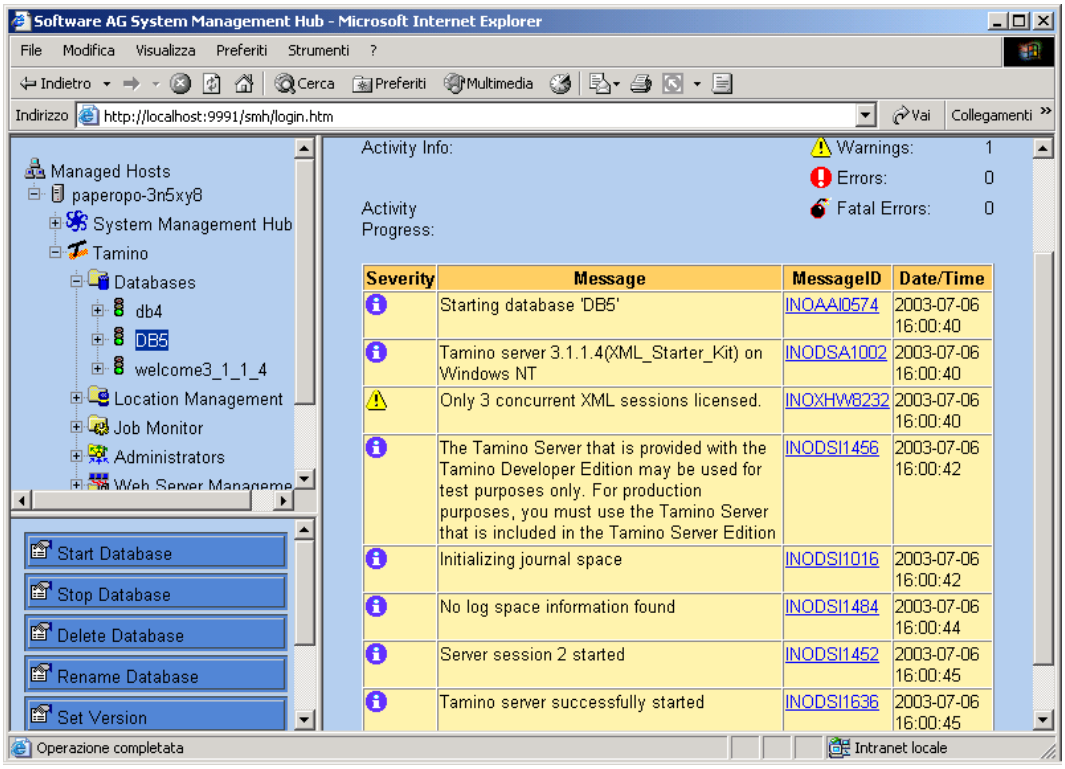


Se il processo di creazione è andato a buon fine visualizziamo quanto segue:



ATTIVARE IL DATABASE

Se ora andiamo ad "espandere" la voce Databases compariranno i nomi di tutti DB che abbiamo creato, con accanto un semaforo il quale indica lo stato del DB. Se il semaforo è rosso vuol dire che il DB non è attivo, se è verde invece vuol dire che è attivo, se è giallo allora il DB è in standby: il DB è attivo ma non può accettare transazioni. Nel nostro caso il semaforo è rosso, allora per potere utilizzare DB5 dobbiamo attivarlo. Il DB si attiva cliccando sul nome e poi su Start Database, alla fine vedremo quanto segue:

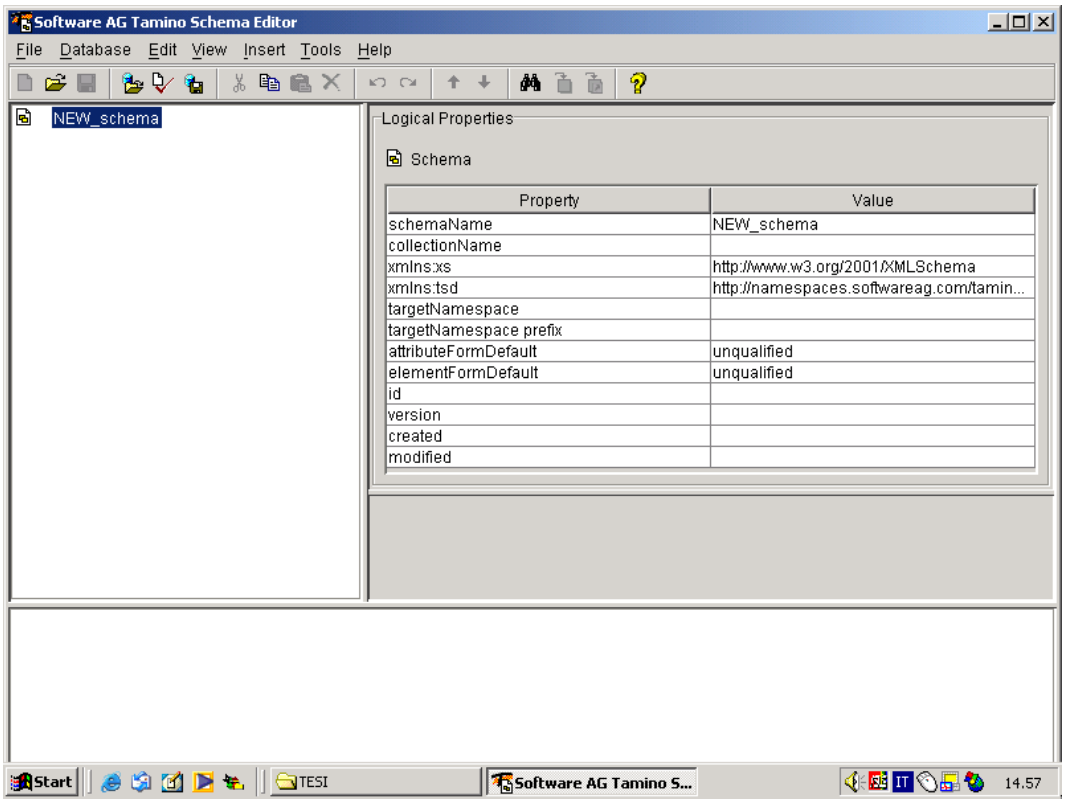


MEMORIZZAZIONE DEGLI OGGETTI XML

A questo punto, una volta che abbiamo creato il DB, occorre popolarlo, ci sono due strade per memorizzare gli oggetti XML nel DB.

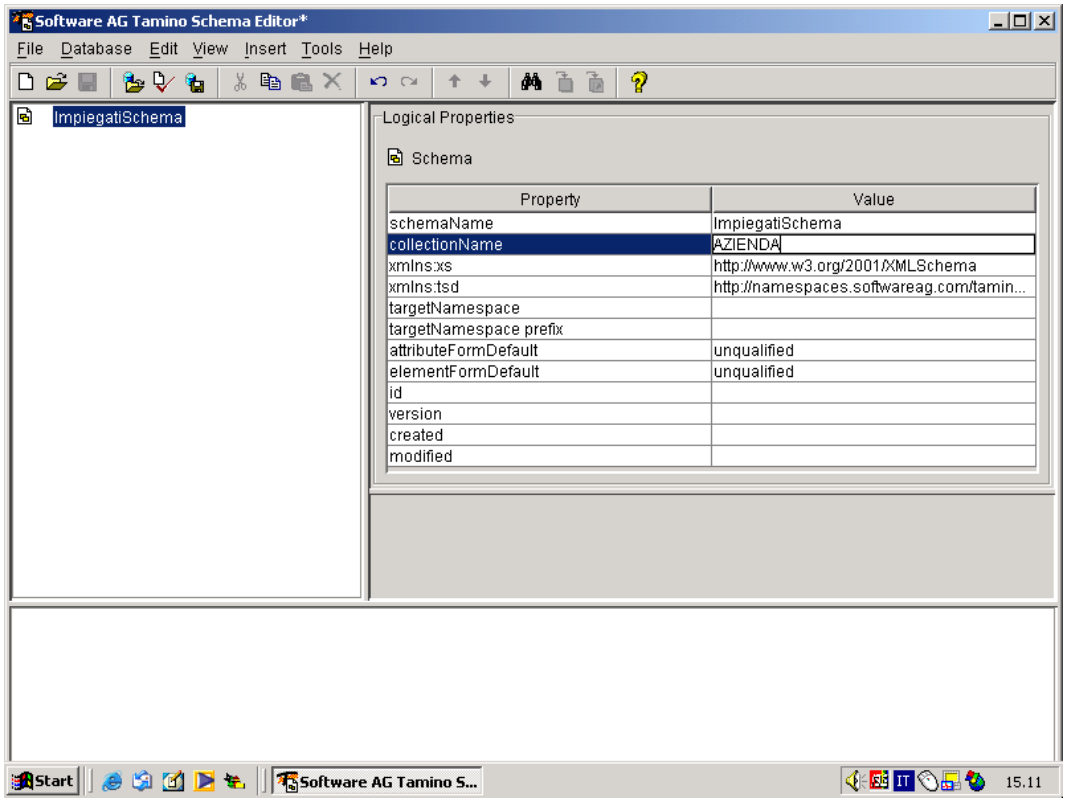
La prima è quella di definire il Tamino Schema degli oggetti XML all' interno di una Collection, l'altra è quella di memorizzare gli oggetti XML senza schema, in tal caso è sufficiente che gli oggetti siano well formed, per poi essere memorizzati da Tamino in una Collection di default di nome " ino:etc". Nel nostro caso seguiamo la prima strada, poichè il nostro documento XML è dotato di DTD, e quindi possiamo usare lo Schema Editor di Tamino per importare tale DTD e tradurlo automaticamente nel Tamino Schema. Il passo

successivo è quindi quello di attivare lo Schema Editor di Tamino, tale tool si trova in "Programmi/Software AG Tamino 3.1.1.4/ Tamino tools.

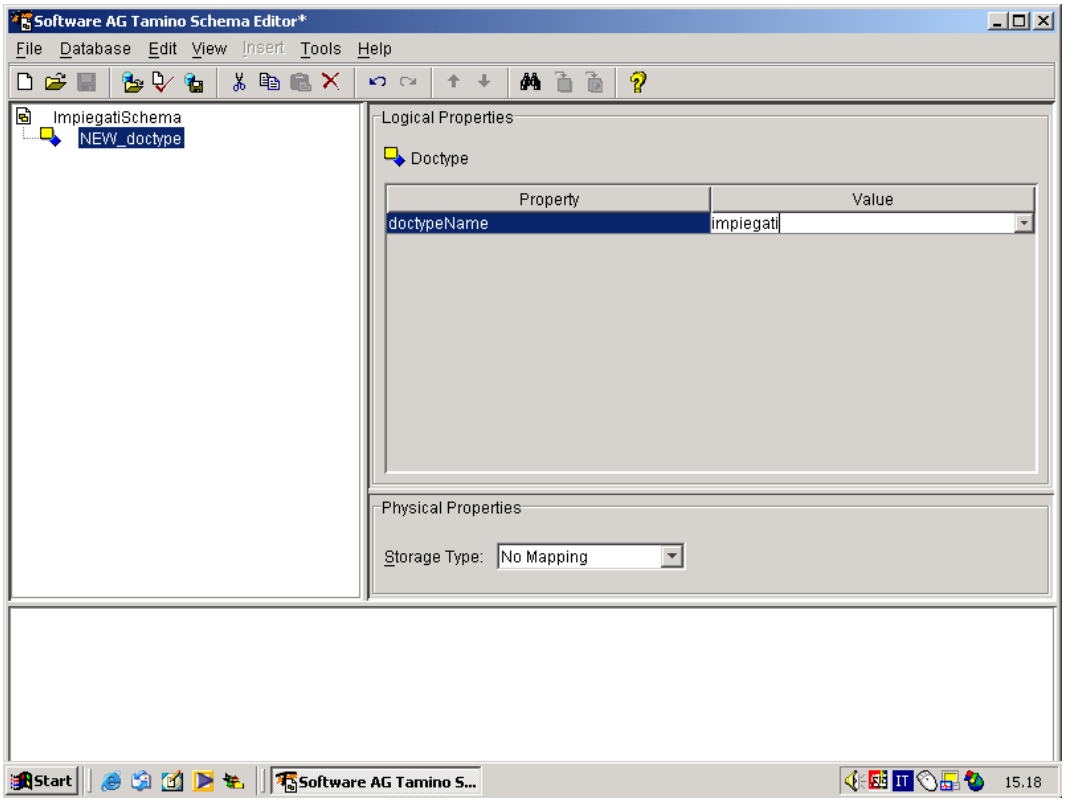


Ora occorre specificare il nome dello schema che sarà generato, inseriamo quindi il nome "ImpiegatiSchema" nel campo "value" accanto a schemaName, dopodiché specifichiamo

il nome della Collection alla quale dovrà appartenere lo schema, il nome scelto è "AZIENDA"

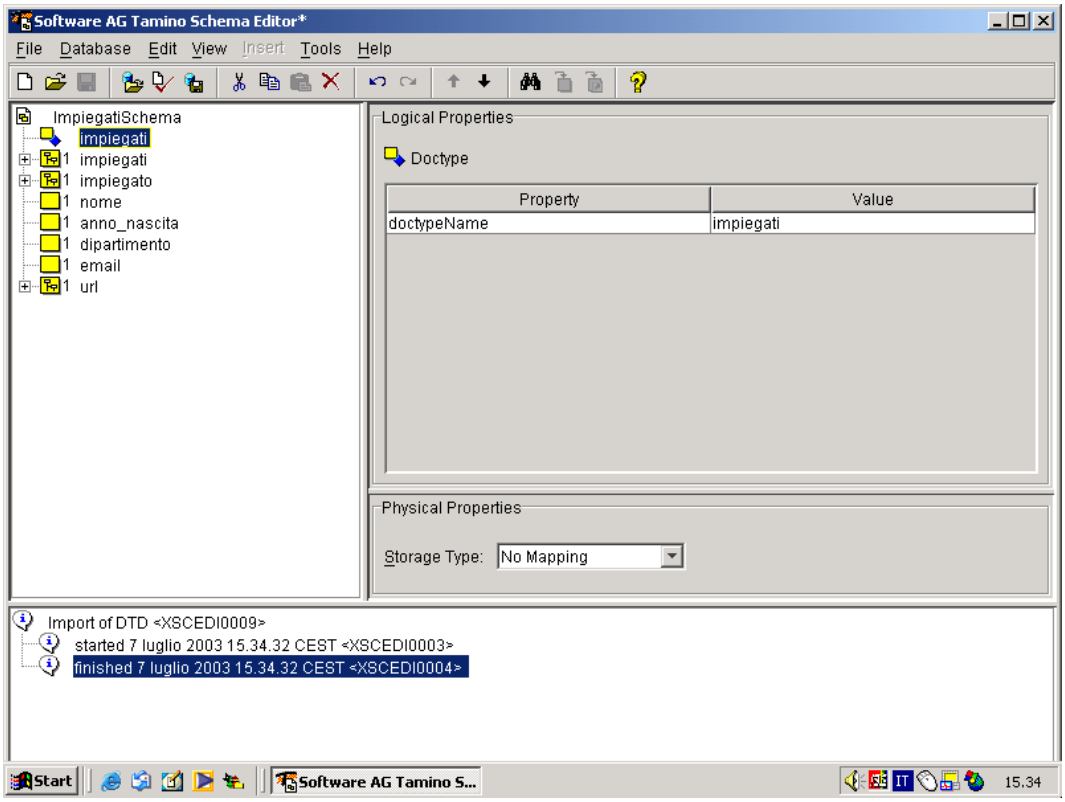


A questo punto inseriamo il doctype, precisamente dal menù Insert scegliamo doctype e diamogli poi il nome "impiegati"



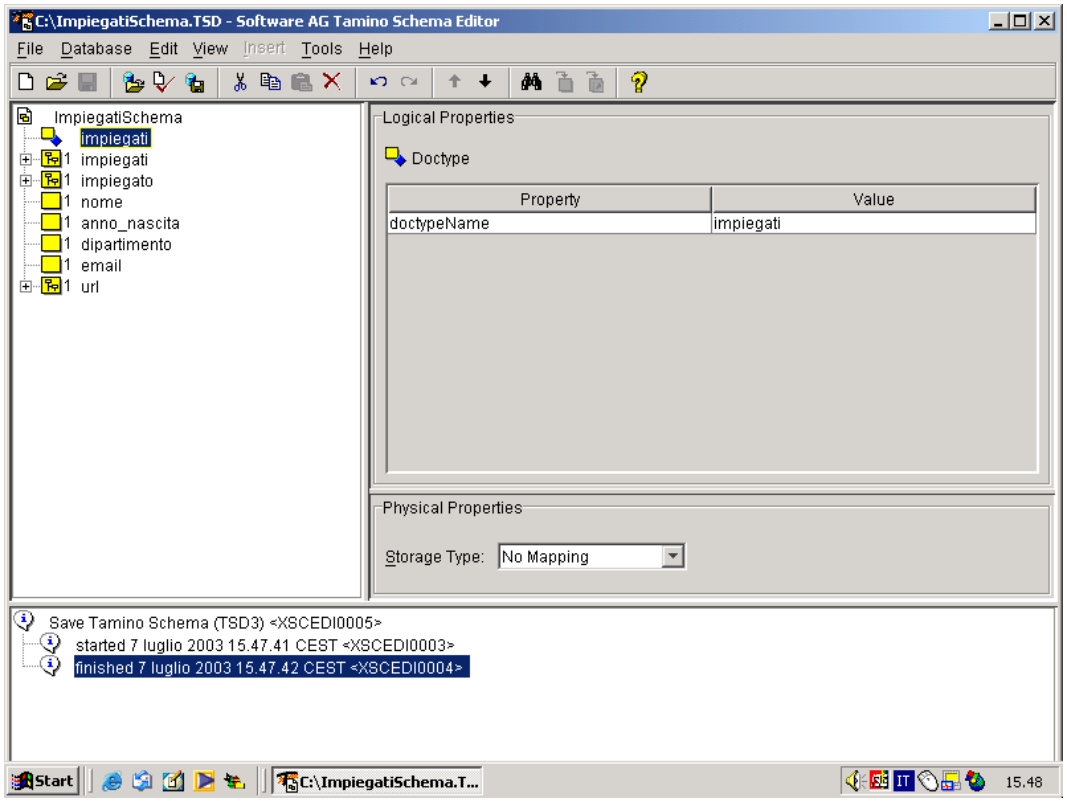
Il passo successivo è quello fondamentale, difatti ci apprestiamo ad importare il DTD, e quindi a tradurlo nel Tamino Schema.

Per importare il DTD scegliamo dal menù File il comando "Import DTD…", si aprirà una dialog box che ci consente di scegliere il file che contiene il DTD, dopodiché clicclando su "Import DTD", il DTD viene automaticamente tradotto nel formato XML Schema e caricato nello Schema Editor. A riprova di ciò possiamo osservare che nella zona a sinistra compare una vista ad albero, ed in basso a sinistra alcune informazioni che ci aggiornano sulla situazione.

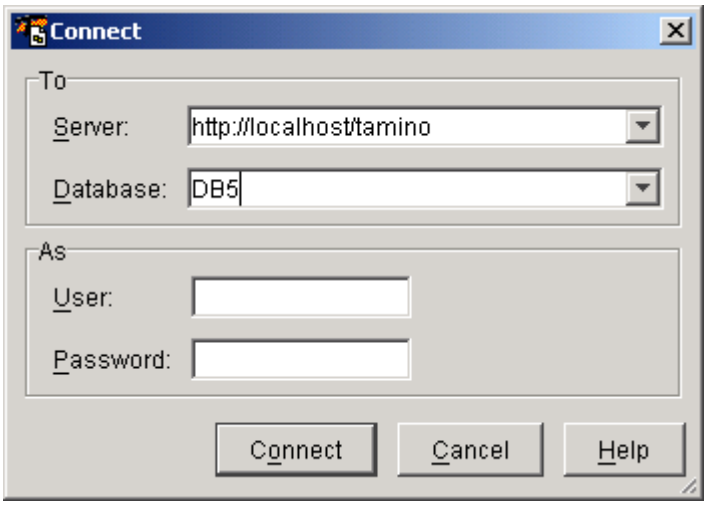


Ora occorre definire lo Schema come schema di Tamino, per fare ciò dobbiamo innanzitutto salvare lo schema, scegliamo quindi, dal menù file il comando SAVE…

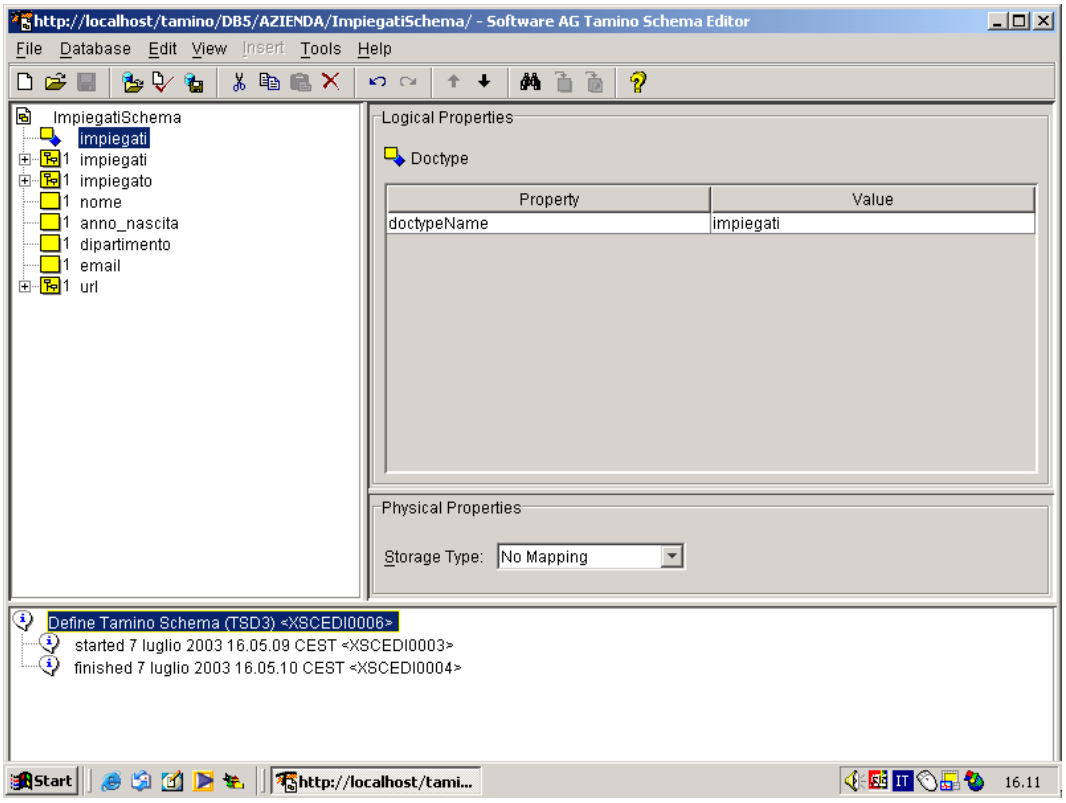
Una volta salvato compare in basso a sinistra il messaggio: ”Save Tamino Schema (TSD3) <XSCEDI0005>” che aggiorna la situazione a quella di “salvataggio” dello schema.



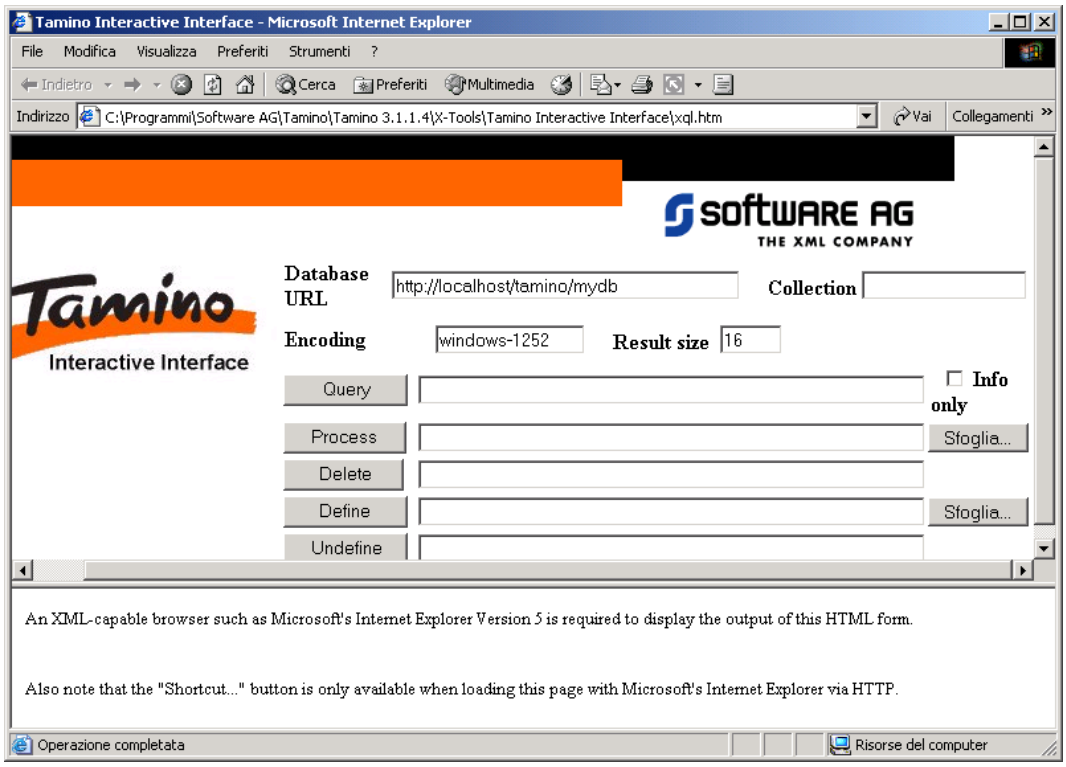
Ora occorre importare lo Schema nel server Tamino, ciò avviene iniziando a scegliere dal menu Database il comando Define Schema…, a questo punto si apre una dialog box, per la scelta del Server (ovviamente Tamino Server), a cui ci dobbiamo connettere. La connessione avviene cliccando sull'icona con il simbolo del DB con un connettore, quindi si visualizza quanto segue:



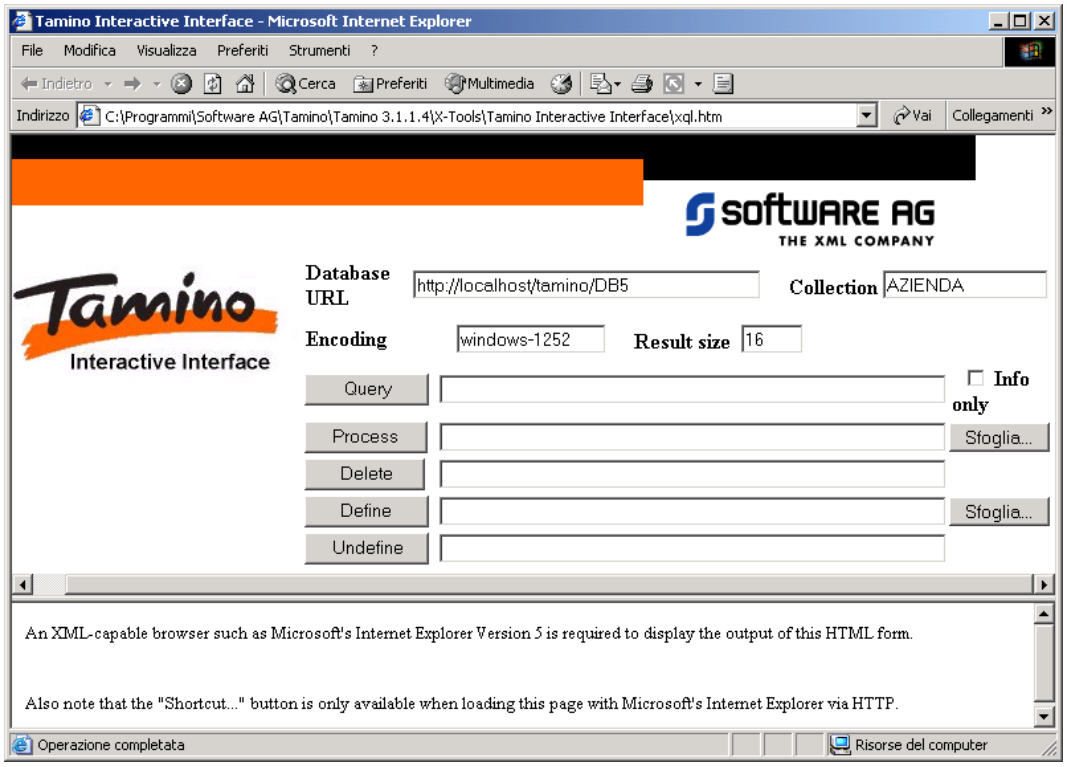
Inseriamo il nome del DB, nel nostro caso DB5, e clicchiamo su Connenct e poi su Define. Alla fine possiamo osservare in basso a sinistra dello Schema editor il messaggio: " Define Tamino Schema(TSD3) <XSCEDI0005>" che conferma l'avvenuta definizione del Tamino Schema.



A questo punto iniziamo a caricare gli oggetti XML, il modo più semplice è quello di utilizzare il Tamino Interactive Interface, tale tool si trova in “Programmi/Software AG Tamino 3.1.1.4/ Tamino tools.



Nel campo Database URL, specifichiamo la URL del DB a cui vogliamo accedere, nel nostro caso è "http//localhost/tamino/DB5". Nel campo Collection deve essere specificato il nome della Collection dove risiede il Tamino Schema dei dati che vogliamo memorizzare, nel nostro caso è "AZIENDA".



Ora il passo successivo è quello di memorizzare nella Collection AZIENDA gli oggetti XML, quindi basta inserire nel campo Process il file che contiene l' oggetto XML.

Prima di inserire il nostro file "impiegati.xml" , inseriamo un file che contiene delle informazioni che non rispettano le regole del DTD e quindi del Tamino Schema. In questo modo proviamo che il Parser di Tamino effettivamente valida il documento XML all'atto dell'ingresso.

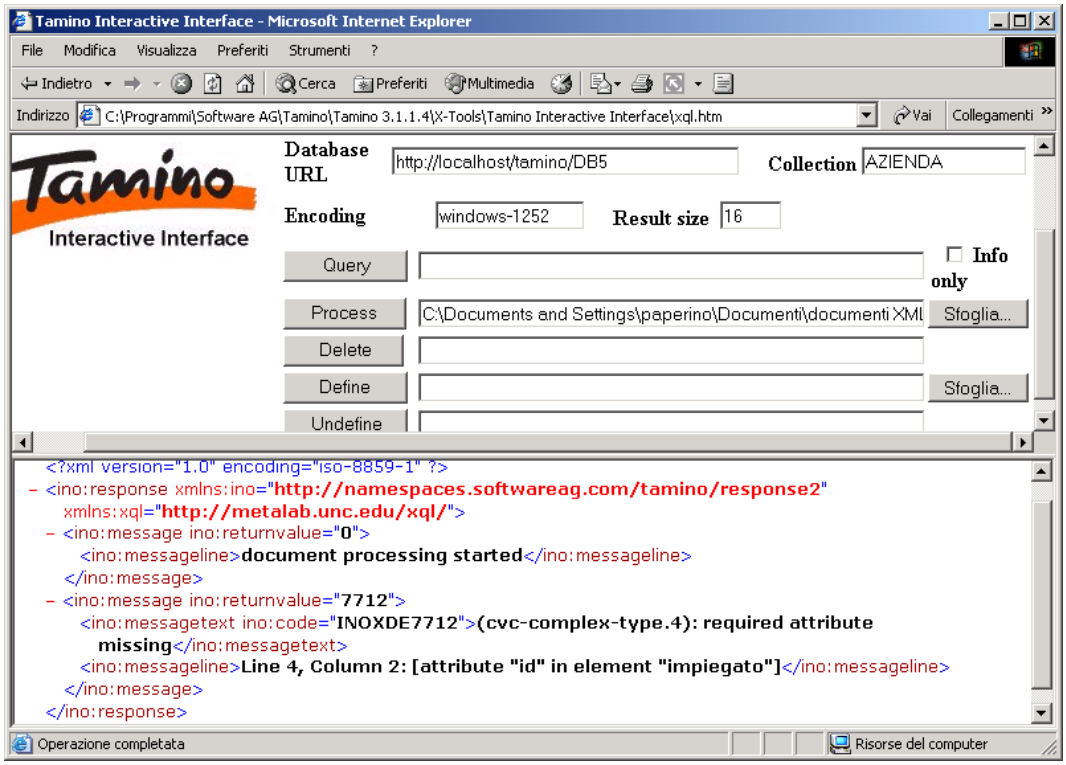
Un primo documento XML che viola il "vincolo sull'attributo ID", è il seguente:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE impiegati SYSTEM "impiegati.dtd">
```

```
<impiegati>
 <impiegato>
  <nome>Mario Alberti</nome>
  <anno_nascita>1960</anno_nascita>
  <dipartimento>amministrazione</dipartimento>
  <email>mrossi@foo.com</email>
 </impiegato>
 <impiegato id="F.R">
  <nome>Filippo Ravenna</nome>
  <anno_nascita>1963</anno_nascita>
  <dipartimento>amministrazione</dipartimento>
  <email>fbaldi@foo.com</email>
  </impiegato>
 </impiegati>
```

Difatti c'è un impiegato (il signor Mario Alberti) che non ha l'attributo ID, invece tale attributo è obbligatorio, in quanto nel DTD l'attributo ID è REQUIRED.

Bene se vogliamo memorizzare tale documento XML in TAMINO, dobbiamo inserire il nome del file, che contiene il documento, nel campo Process (ciò lo facciamo usando il tasto "Sfoglia") dopodichè se clicchiamo su Process, la risposta che otteniamo nella "vista" in basso del Tamino Interactive Interface è la seguente

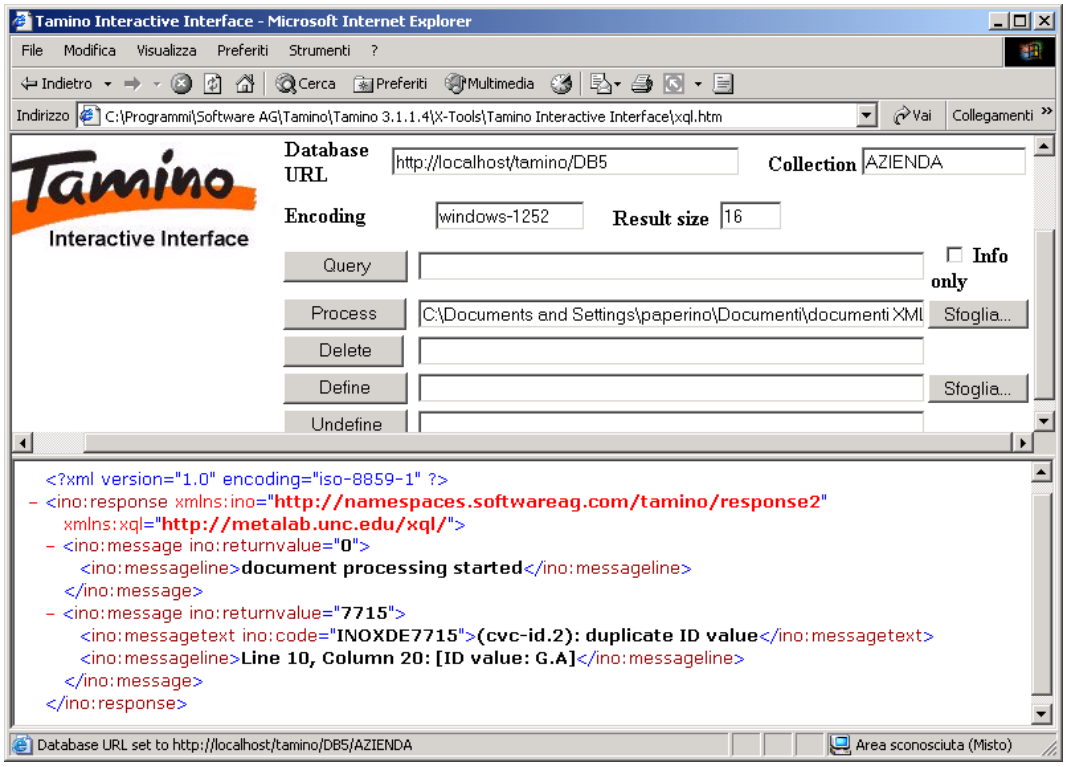


in cui in effetti si dice che la memorizzazione non è stata effettuata perché "*required attribute missing*" cioè era richiesto un attributo obbligatorio ed invece tale attributo non c'è. Una cosa analoga accade, se vogliamo memorizzare il seguente documento XML:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE impiegati SYSTEM "impiegati.dtd">
<impiegati>
 <impiegato id="G.A">
  <nome>Gianna Alberti</nome>
  <anno_nascita>1960</anno_nascita>
  <dipartimento>amministrazione</dipartimento>
  <email>galbert@foo.com</email>
 </impiegato>
 <impiegato id="G.A">
  <nome>Giacomo Acanfora</nome>
  <anno_nascita>1963</anno_nascita>
```

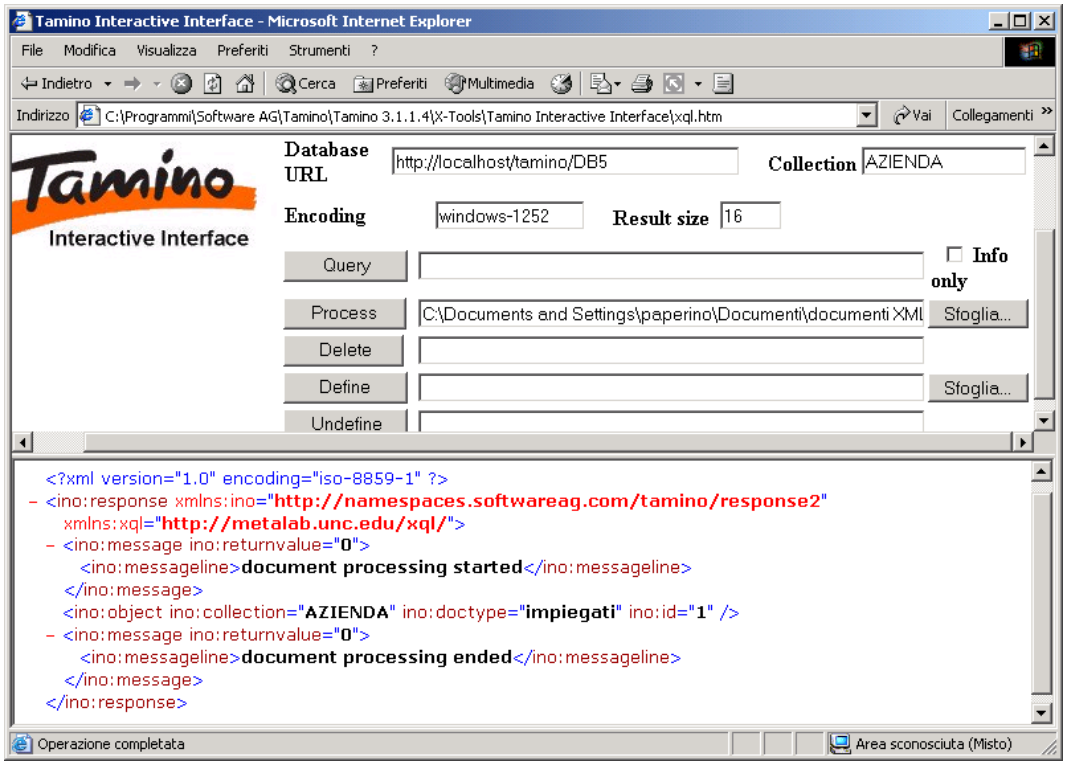
<dipartimento>amministrazione</dipartimento>
<email>gancan@foo.com</email>
</impiegato>
</impiegati>

difatti tale documento XML viola ancora il vincolo sull'attributo ID, poiché assume due volte lo stesso valore, invece il valore di un attributo di tipo ID deve essere unico.
La risposta che otteniamo in questo caso è la seguente



in cui si dice che la memorizzazione non è avvenuta perché *"duplicate ID value"* ossia nel documento compare l'attributo di tipo ID, due volte con lo stesso valore.
Con questi due documenti XML abbiamo quindi dimostrato che se definiamo nel DTD un attributo di tipo ID REQUIRED, tale attributo svolge la stessa funzione di quella di "*chiave primaria* " che conosciamo nel DB relazionale.

Ora passiamo a memorizzare il nostro documento XML iniziale, in tal caso Tamino accetta l'oggetto XML

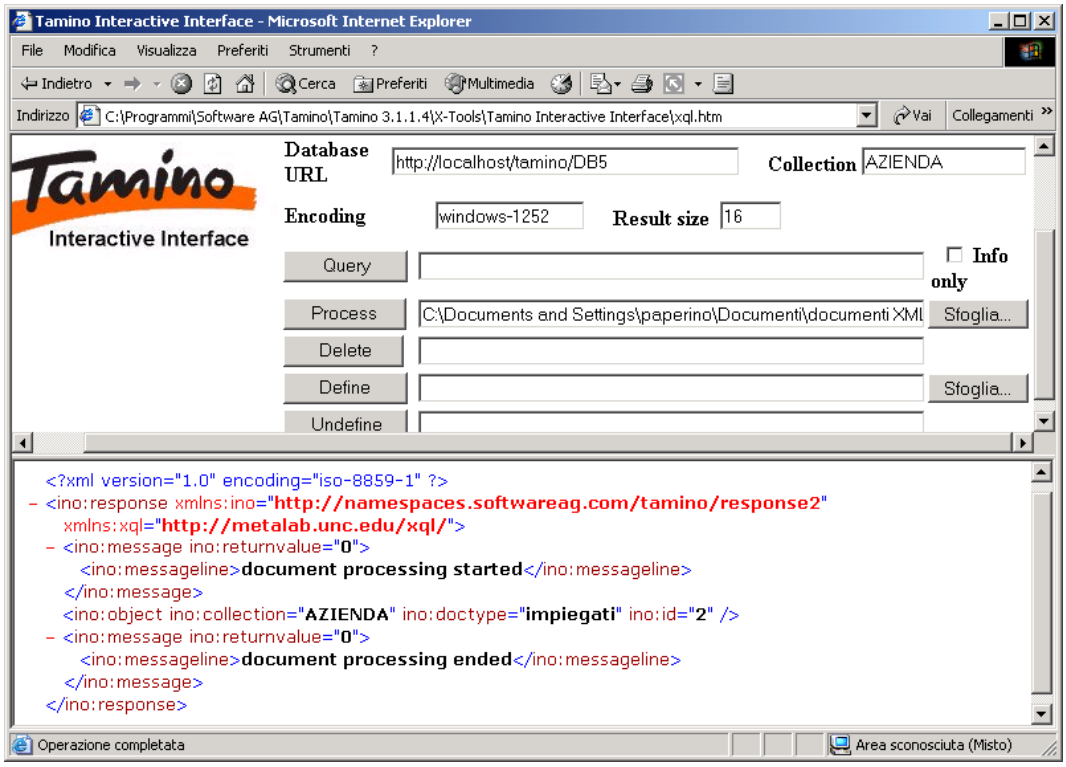


Difatti, come si può notare dalla risposta, l'oggetto XML viene memorizzato nella collection AZIENDA, il nome del doctype è impiegati, e trattasi del primo oggetto memorizzato: "ino:id=1". A questo punto se inseriamo un altro oggetto XML, sempre conforme allo schema (che è memorizzato nella Collection AZIENDA) ma che presenta un valore dell'attributo ID già esistente in un documento XML precedentemente memorizzato, Tamino non se ne accorge. Difatti se consideriamo il seguente documento XML:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE impiegati SYSTEM "impiegati.dtd">
<impiegati>
 <impiegato id="F.B">
  <nome>Francesco Basile</nome>
```

<anno_nascita>1960</anno_nascita>
<dipartimento>presidenza</dipartimento>
<email>fbas@foo.com</email>
</impiegato>
</impiegati>

tale documento presenta il valore "F.B" che già esiste nel documento che abbiamo memorizzato, se ora procediamo con la memorizzazione, in effetti possiamo osservare ancora una risposta positiva:



precisamente tale documento, viene memorizzato come secondo oggetto (ino:id =2) del doctype "impiegati".

A questo punto verifichiamo che effettivamente i documenti sono stati memorizzati nella Collection AZIENDA, relativamente al doctype "impiegati",

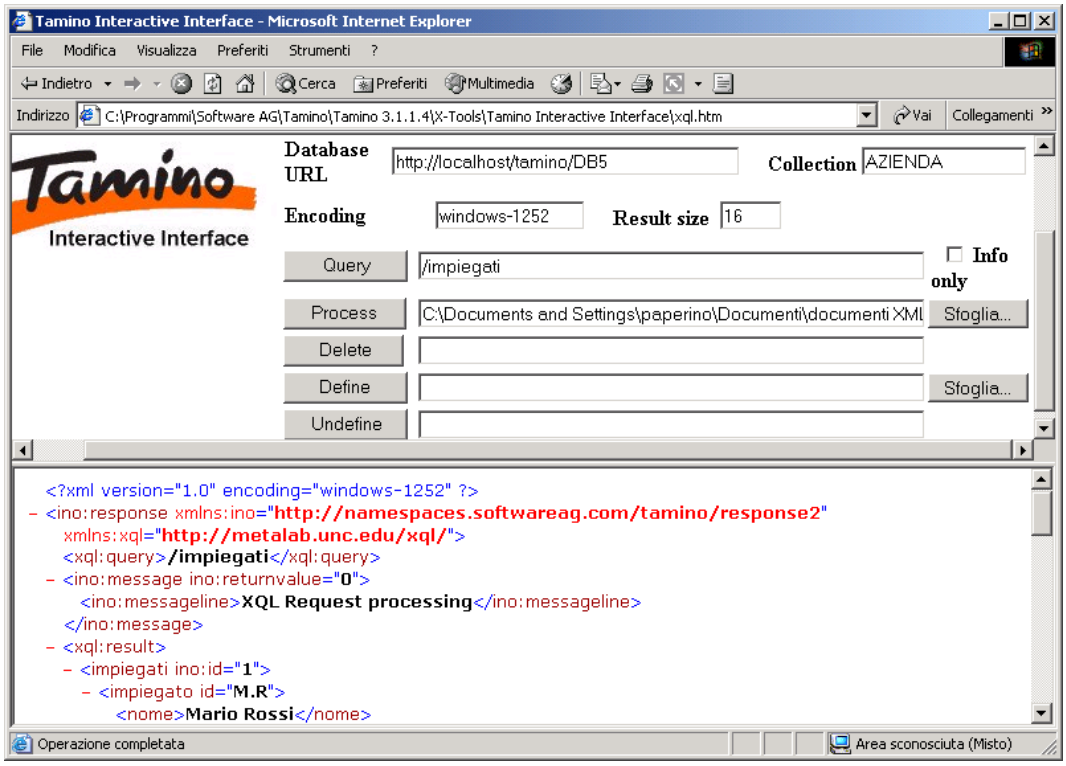
eseguiamo quindi (sempre tramite il Tamino Interactive Interface) la seguente query:

Query 1

Digitiamo nel campo Query la stringa:

/impiegati

e clicchiamo su Query



Il risultato completo è riportato qui di seguito:

<?xml version="1.0" encoding="windows-1252" ?>

- <ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2" xmlns:xql="http://metalab.unc.edu/xql/">

<xql:query>/impiegati</xql:query>

+ <ino:message ino:returnvalue="0">

<ino:messageline>XQL Request processing</ino:messageline>

</ino:message>

- <xql:result>

- <impiegati ino:id="1">

```
- <impiegato id="M.R">
  <nome>Mario Rossi</nome>
  <anno_nascita>1950</anno_nascita>
  <dipartimento>amministrazione</dipartimento>
  <email>mrossi@foo.com</email>
  </impiegato>
- <impiegato id="F.B">
  <nome>Filippo Bianchi</nome>
  <anno_nascita>1950</anno_nascita>
  <dipartimento>amministrazione</dipartimento>
  <email>fbaldi@foo.com</email>
  </impiegato>
- <impiegato id="A.V">
  <nome>Alice Verdi</nome>
  <anno_nascita>1960</anno_nascita>
  <dipartimento>economato</dipartimento>
  <email>averdi@foo.com</email>
  <url href="http://www.foo.com/~averdi/" />
  </impiegato>
- <impiegato id="A.B">
  <nome>Antonio Bisio</nome>
  <anno_nascita>1955</anno_nascita>
  <dipartimento>economato</dipartimento>
  <email>abisio@foo.com</email>
  <url href="http://www.free.com/~abisio/" />
  </impiegato>
  </impiegati>
- <impiegati ino:id="2">
- <impiegato id="F.B">
  <nome>Francesco Basile</nome>
  <anno_nascita>1960</anno_nascita>
```

<dipartimento>presidenza</dipartimento>
<email>fbas@foo.com</email>
</impiegato>
</impiegati>
</xql:result>
- <ino:cursor ino:count="2">
<ino:first ino:href="?_XQL(1,16)=/impiegati" />
</ino:cursor>
- <ino:message ino:returnvalue="0">
<ino:messageline>XQL Request processed</ino:messageline>
</ino:message>
</ino:response>

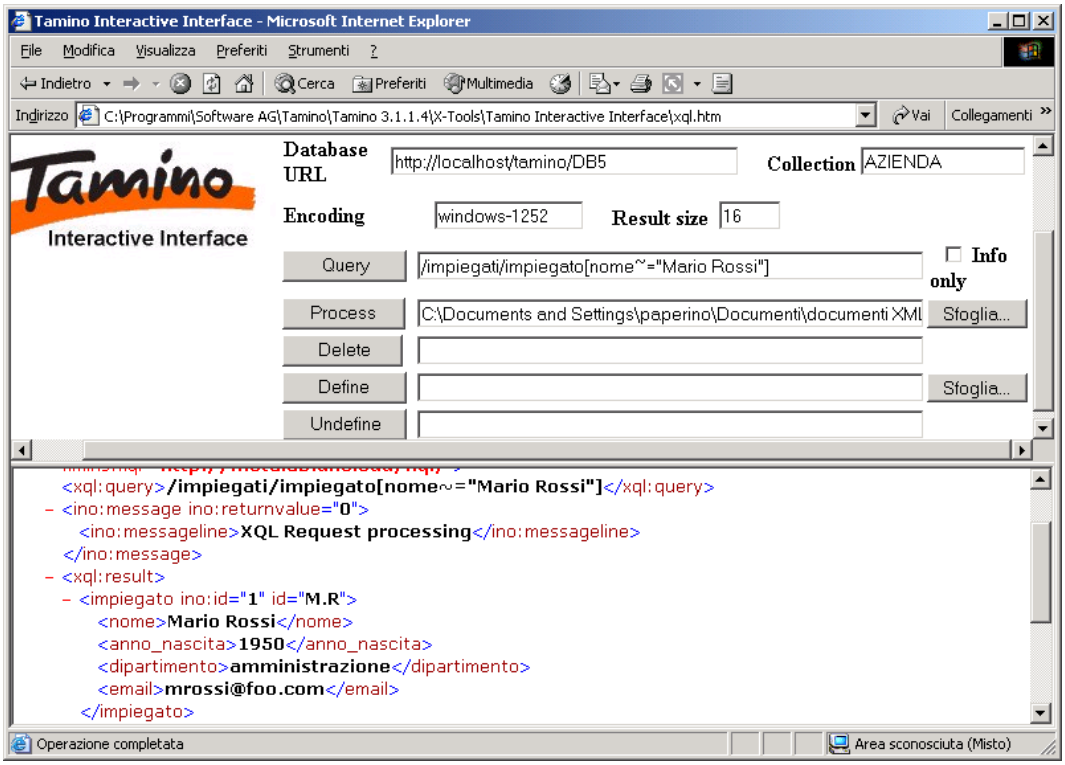
Prima di passare alle query successive, è bene sottolineare che per eseguire una query occorre semplicemente indicare il DB, e la Collection, dopodichè basta scrivere, secondo la sintassi di X-Query, la query nel campo Query del Tamino Interactive Interface.

Query 2

Digitiamo nel campo Query :

/impiegati/impiegato[nome ~=”Mario Rossi”]

tale stringa ci consente di trovare le informazioni di tutti gli impiegati di nome Mario Rossi. Notiamo che ciò si ottiene facendo uso di un particolare operatore di X-Query, ossia l’operatore CONTAINS “~=”



Il risultato per esteso è il seguente:

<?xml version="1.0" encoding="windows-1252" ?>

- <ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2" xmlns:xql="http://metalab.unc.edu/xql/">

<xql:query>/impiegati/impiegato[nome~="Mario Rossi"]</xql:query>

- <ino:message ino:returnvalue="0">

<ino:messageline>XQL Request processing</ino:messageline>

</ino:message>

- <xql:result>

- <impiegato ino:id="1" id="M.R">

<nome>Mario Rossi</nome>

<anno_nascita>1950</anno_nascita>

<dipartimento>amministrazione</dipartimento>

<email>mrossi@foo.com</email>

</impiegato>

</xql:result>

- <ino:cursor ino:count="1">

<ino:first ino:href="?_XQL(1,16)=/impiegati/impiegato[nome~="Mario Rossi"]" />

</ino:cursor>

- <ino:message ino:returnvalue="0">

<ino:messageline>XQL Request processed</ino:messageline>

</ino:message>

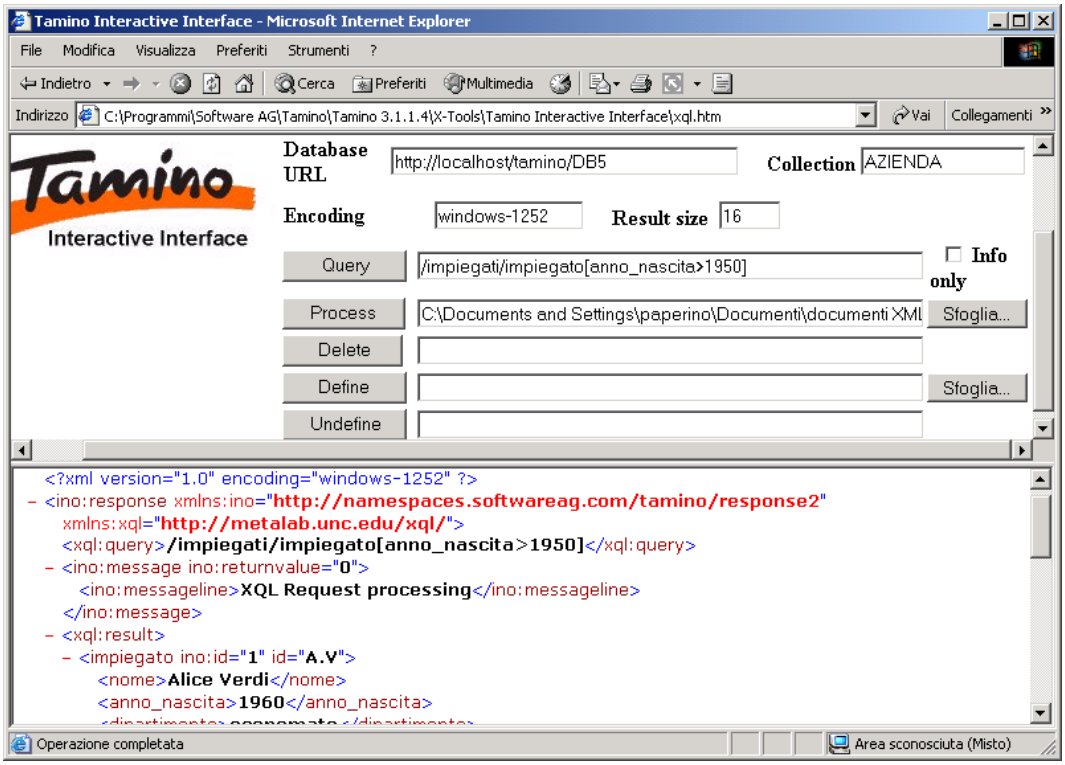
</ino:response>

Query 3

Digitiamo nel campo Query :

/impiegati/impiegato[anno_nascita>1950]

tale stringa ci consente di trovare le informazioni di tutti gli impiegati che sono nati dopo il 1950.



Il risultato per esteso è il seguente:

<?xml version="1.0" encoding="windows-1252" ?>
- <ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2" xmlns:xql="http://metalab.unc.edu/xql/">
<xql:query>/impiegati/impiegato[anno_nascita>1950]</xql:query>
- <ino:message ino:returnvalue="0">
<ino:messageline>XQL Request processing</ino:messageline>
</ino:message>
- <xql:result>
- <impiegato ino:id="1" id="A.V">
<nome>Alice Verdi</nome>
<anno_nascita>1960</anno_nascita>
<dipartimento>economato</dipartimento>
<email>averdi@foo.com</email>
<url href="http://www.foo.com/~averdi/" />
</impiegato>
- <impiegato ino:id="1" id="A.B">
<nome>Antonio Bisio</nome>
<anno_nascita>1955</anno_nascita>
<dipartimento>economato</dipartimento>
<email>abisio@foo.com</email>
<url href="http://www.free.com/~abisio/" />
</impiegato>
- <impiegato ino:id="2" id="F.B">
<nome>Francesco Basile</nome>
<anno_nascita>1960</anno_nascita>
<dipartimento>presidenza</dipartimento>
<email>fbas@foo.com</email>
</impiegato>
</xql:result>
- <ino:cursor ino:count="3">

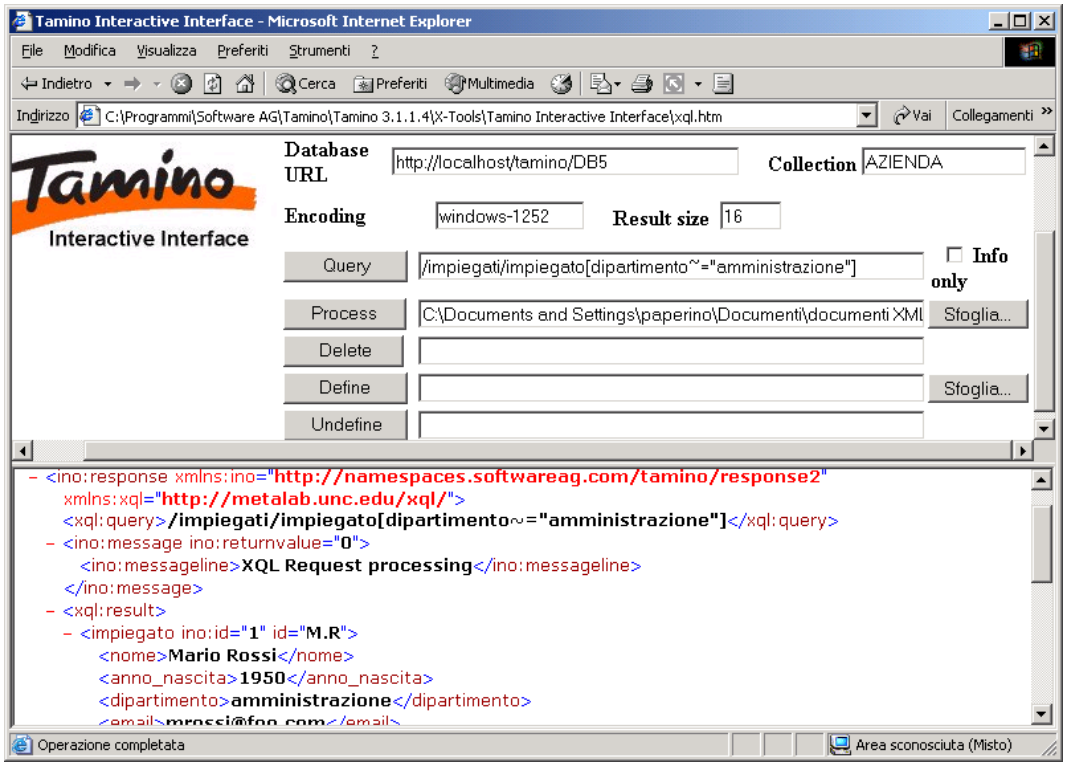
<ino:first ino:href="?_XQL(1,16)=/impiegati/impiegato[anno_nascita>1950]" />
</ino:cursor>
- <ino:message ino:returnvalue="0">
<ino:messageline>XQL Request processed</ino:messageline>
</ino:message>
</ino:response>

Query 4

Digitiamo nel campo Query :

/impiegati/impiegato[dipartimento ~=”amministrazione”]

tale stringa ci consente di trovare le informazioni di tutti gli impiegati che lavorano nel dipartimento “amministrazione”.

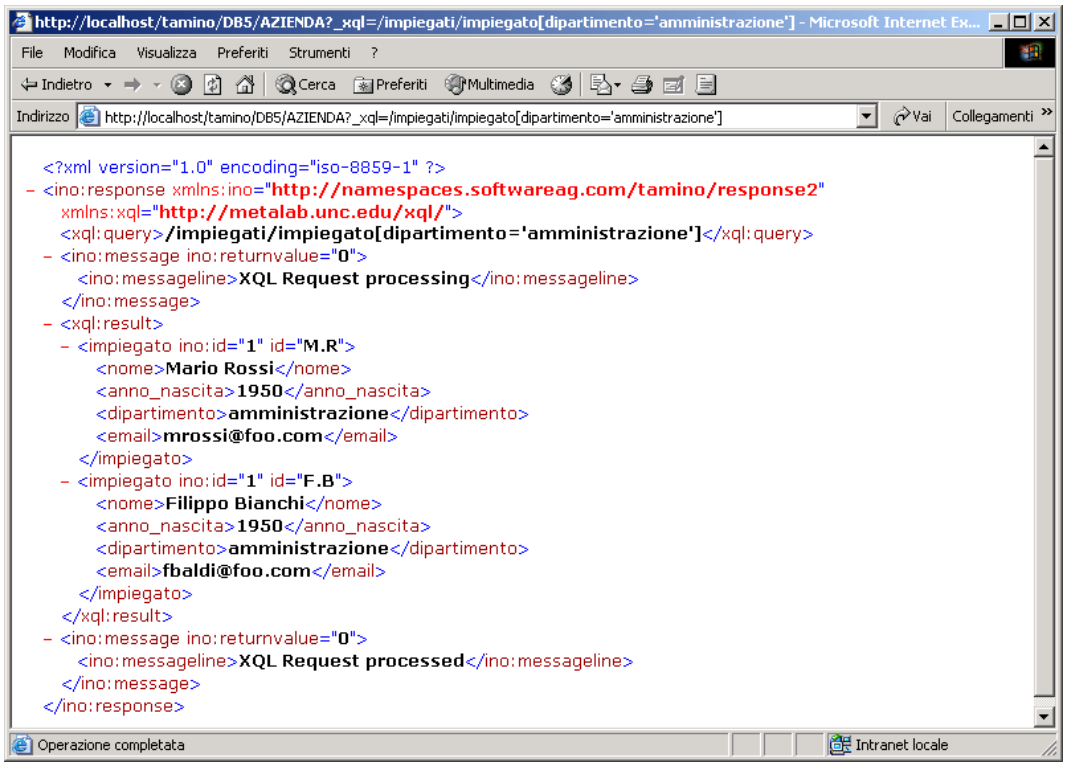


Il risultato per esteso è il seguente:

<?xml version="1.0" encoding="windows-1252" ?>
- <ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2" xmlns:xql="http://metalab.unc.edu/xql/">
<xql:query>/impiegati/impiegato[dipartimento~="amministrazione"]</xql:query>
- <ino:message ino:returnvalue="0">
<ino:messageline>XQL Request processing</ino:messageline>
</ino:message>
- <xql:result>
- <impiegato ino:id="1" id="M.R">
<nome>Mario Rossi</nome>
<anno_nascita>1950</anno_nascita>
<dipartimento>amministrazione</dipartimento>
<email>mrossi@foo.com</email>
</impiegato>
- <impiegato ino:id="1" id="F.B">
<nome>Filippo Bianchi</nome>
<anno_nascita>1950</anno_nascita>
<dipartimento>amministrazione</dipartimento>
<email>fbaldi@foo.com</email>
</impiegato>
</xql:result>
- <ino:cursor ino:count="2">
<ino:first ino:href="?_XQL(1,16)=/impiegati/impiegato[dipartimento~="amministrazione"]" />
</ino:cursor>
- <ino:message ino:returnvalue="0">
<ino:messageline>XQL Request processed</ino:messageline>
</ino:message>
</ino:response>

Per concludere, osserviamo che le query possono essere anche eseguite a mezzo http, inserendo l'URL completo nell'indirizzo del web browser. Ad esempio possiamo rifare la query4, se digitiamo il seguente URL:

http://localhost/tamino/DB5/AZIENDA?_xql=/impiegati/impiegato[dipartimento='amministrazione']

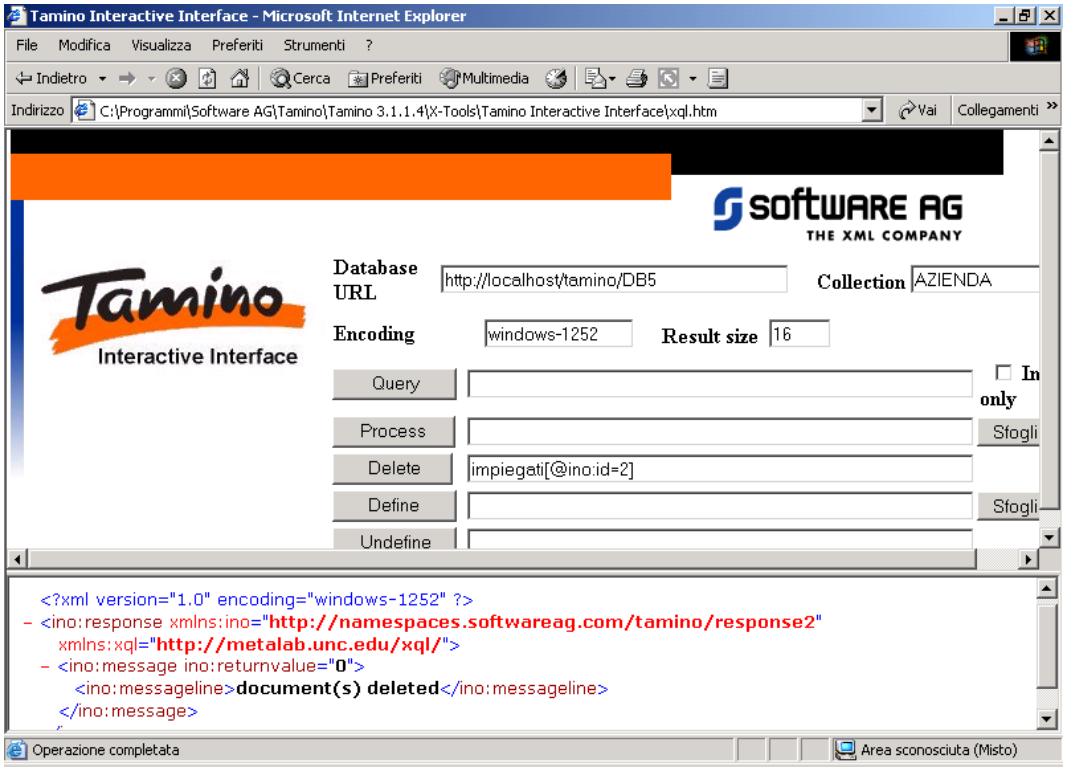


CANCELLAZIONE DEGLI OGGETTI XML

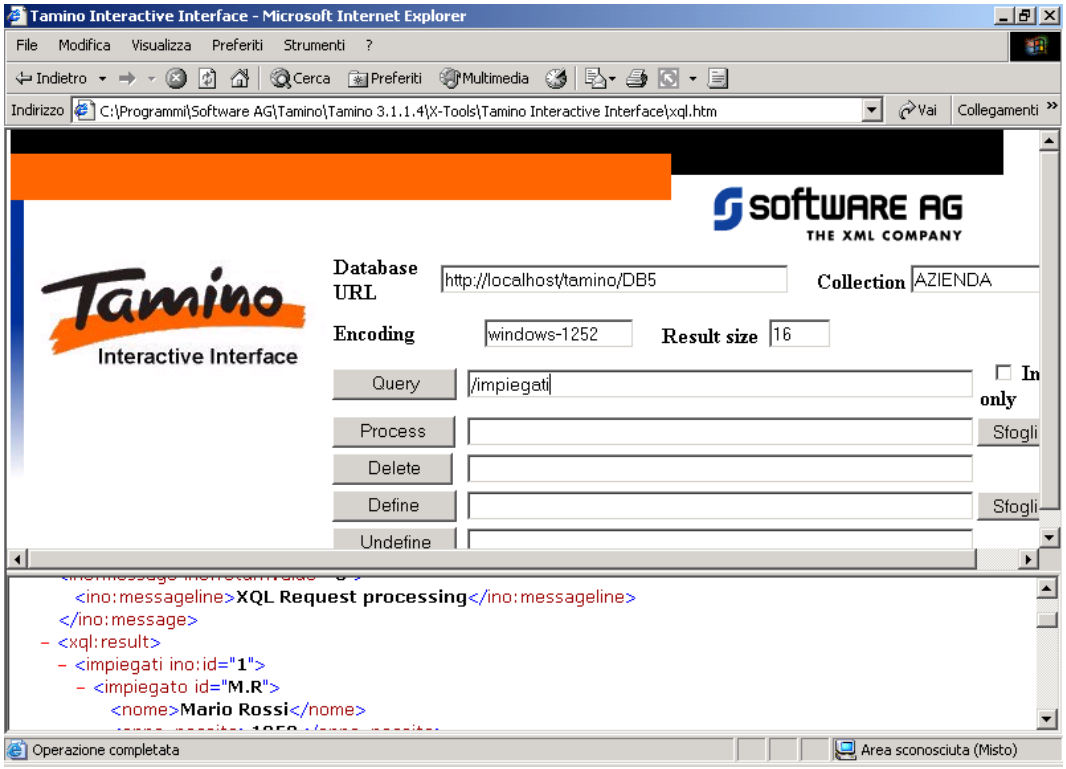
Per cancellare gli oggetti XML, abbiamo due alternative una di queste è quella di voler cancellare un'intero oggetto, ad esempio volendo cancellare un oggetto XML, dobbiamo conoscere il relativo "ino:id", quindi se vogliamo cancellare l'oggetto XML con "ino:id=2" dobbiamo digitare nel campo Delete la seguente stringa:

impiegati[@ino:id=2]

e cliccare su Delete.



La risposta è affermativa, ma per verificare che effettivamente non c'è più nella Collection AZIENDA il secondo oggetto (che abbiamo inserito), riseguiamo la Query 1



il risultato esteso è il seguente:

<?xml version="1.0" encoding="windows-1252" ?>

- <ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2" xmlns:xql="http://metalab.unc.edu/xql/">

<xql:query>/impiegati</xql:query>

- <ino:message ino:returnvalue="0">

<ino:messageline>XQL Request processing</ino:messageline>

</ino:message>

- <xql:result>

- <impiegati ino:id="1">

- <impiegato id="M.R">

<nome>Mario Rossi</nome>

<anno_nascita>1950</anno_nascita>

<dipartimento>amministrazione</dipartimento>

<email>mrossi@foo.com</email>

```
  </impiegato>
- <impiegato id="F.B">
  <nome>Filippo Bianchi</nome>
  <anno_nascita>1950</anno_nascita>
  <dipartimento>amministrazione</dipartimento>
  <email>fbaldi@foo.com</email>
  </impiegato>
- <impiegato id="A.V">
  <nome>Alice Verdi</nome>
  <anno_nascita>1960</anno_nascita>
  <dipartimento>economato</dipartimento>
  <email>averdi@foo.com</email>
  <url href="http://www.foo.com/~averdi/" />
  </impiegato>
- <impiegato id="A.B">
  <nome>Antonio Bisio</nome>
  <anno_nascita>1955</anno_nascita>
  <dipartimento>economato</dipartimento>
  <email>abisio@foo.com</email>
  <url href="http://www.free.com/~abisio/" />
  </impiegato>
  </impiegati>
  </xql:result>
- <ino:cursor ino:count="1">
  <ino:first ino:href="?_XQL(1,16)=/impiegati" />
  </ino:cursor>
- <ino:message ino:returnvalue="0">
  <ino:messageline>XQL Request processed</ino:messageline>
  </ino:message>
  </ino:response>
```

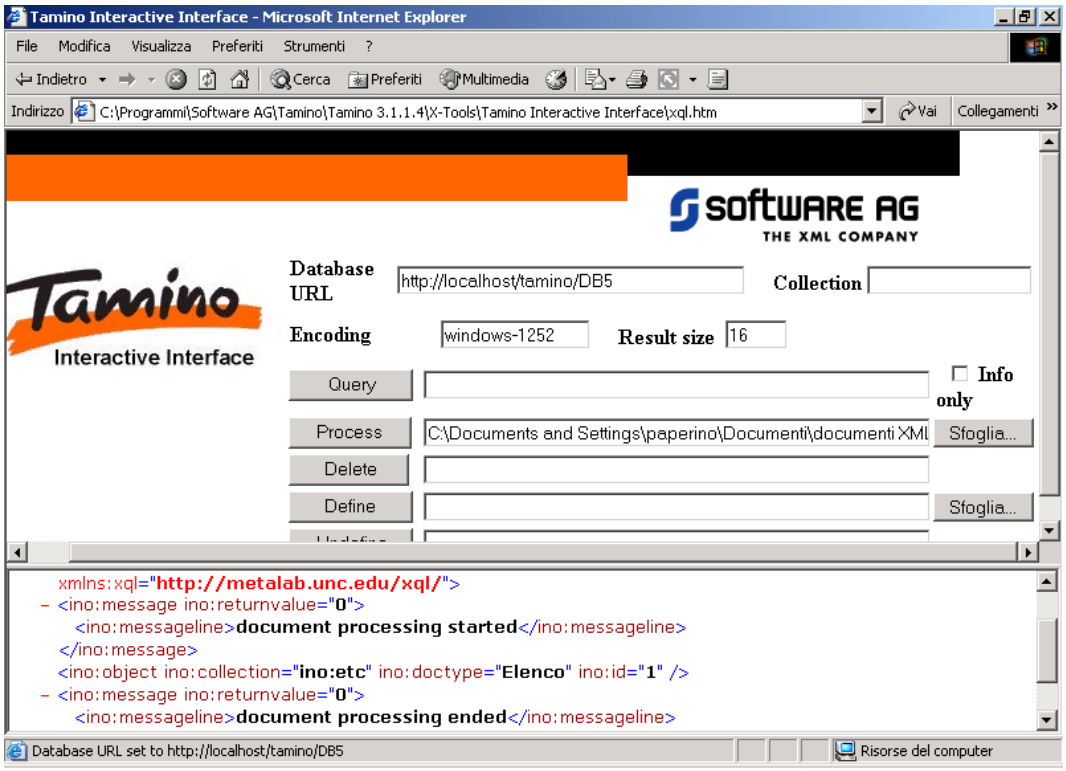
Come si può vedere non c'è il secondo oggetto XML, cioè quello che conteneva l'informazione del solo impiegato Francesco Basile. L'altro modo per cancellare gli oggetti è quello di utilizzare la sintassi di X-Query.

ESEMPIO 2

Il documento XML che vogliamo gestire con Tamino è un documento che contiene le informazioni di una libreria. Supponiamo che il documento iniziale riporti le informazioni di tre libri:

```
   <?xml version="1.0" encoding="ISO-8859-1"?>
<Elenco>
  <Libro disponibilità='S'>
   <Titolo>Basi di Dati</Titolo>
   <Autore>Atzeni</Autore>
   <Data>1999</Data>
   <Editore>McGraw-Hill</Editore>
  </Libro>
  <Libro disponibilità='N'>
   <Titolo>Fisica 1</Titolo>
   <Autore>C.Mencuccini</Autore>
   <Data>1996</Data>
   <Editore>Liguori</Editore>
  </Libro>
  <Libro disponibilità='S'>
   <Titolo>Esercizi di Elettronica Generale</Titolo>
   <Autore>Cupido</Autore>
   <Data>1990</Data>
   <Editore>Liguori</Editore>
  </Libro>
</Elenco>
```

In tale esempio vogliamo semplicemente eseguire altre query, e verificare quanto detto nell'esempio precedente, per quel che riguarda la memorizzazione di un oggetto XML senza schema. Scegliamo come DB quello usato nell'esempio precedente.



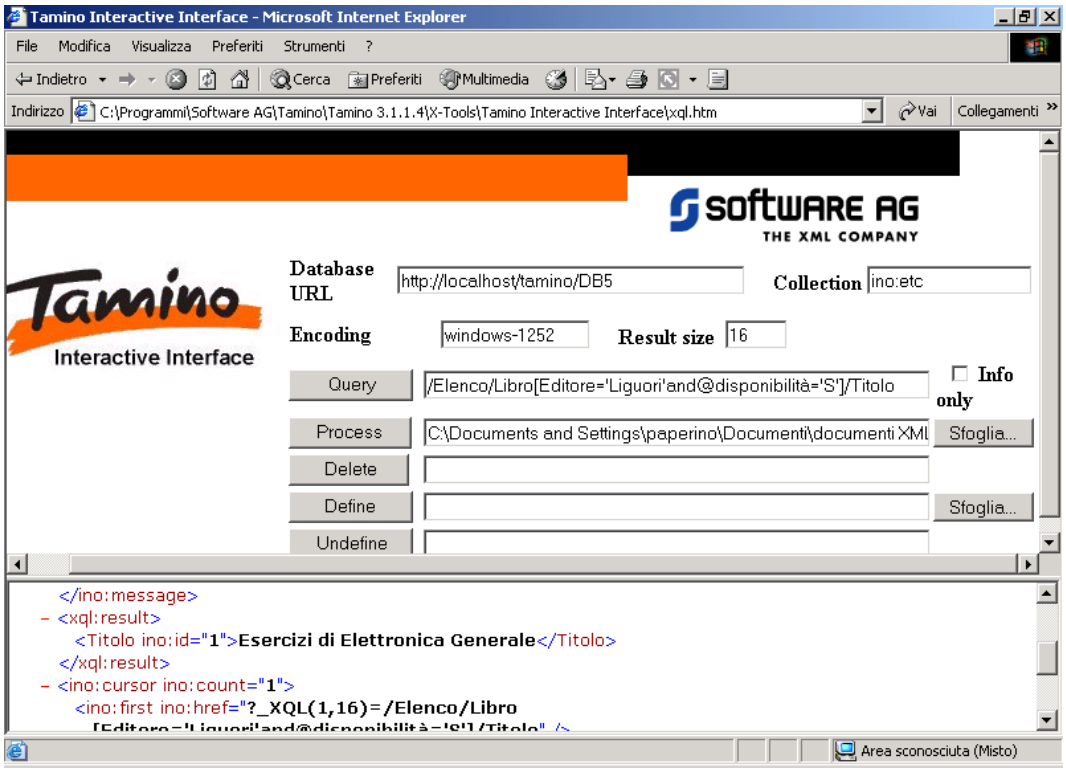
Come possiamo osservare dalla risposta, l'oggetto è stato effettivamente memorizzato nella Collection di default "ino:etc", e il nome del doctype è quello dell' elemento root dell'oggetto XML considerato.

Query 1

Si ricordi che per eseguire una Query occorre specificare la Collection in cui l'oggetto XML è stato memorizzato, e bisogna sempre partire dal doctype. La seguente query:

/Elenco/Libro[Editore=’Liguori’and@disponibilità=’S’]/Titolo

Restituisce l’insieme di tutti i titoli dei libri dell’editore Liguori che sono disponibili.



Il risultato esteso è il seguente:

<?xml version="1.0" encoding="windows-1252" ?>

- <ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2" xmlns:xql="http://metalab.unc.edu/xql/">

<xql:query>/Elenco/Libro[Editore='Liguori'and@disponibilità='S']/Titolo</xql:query>

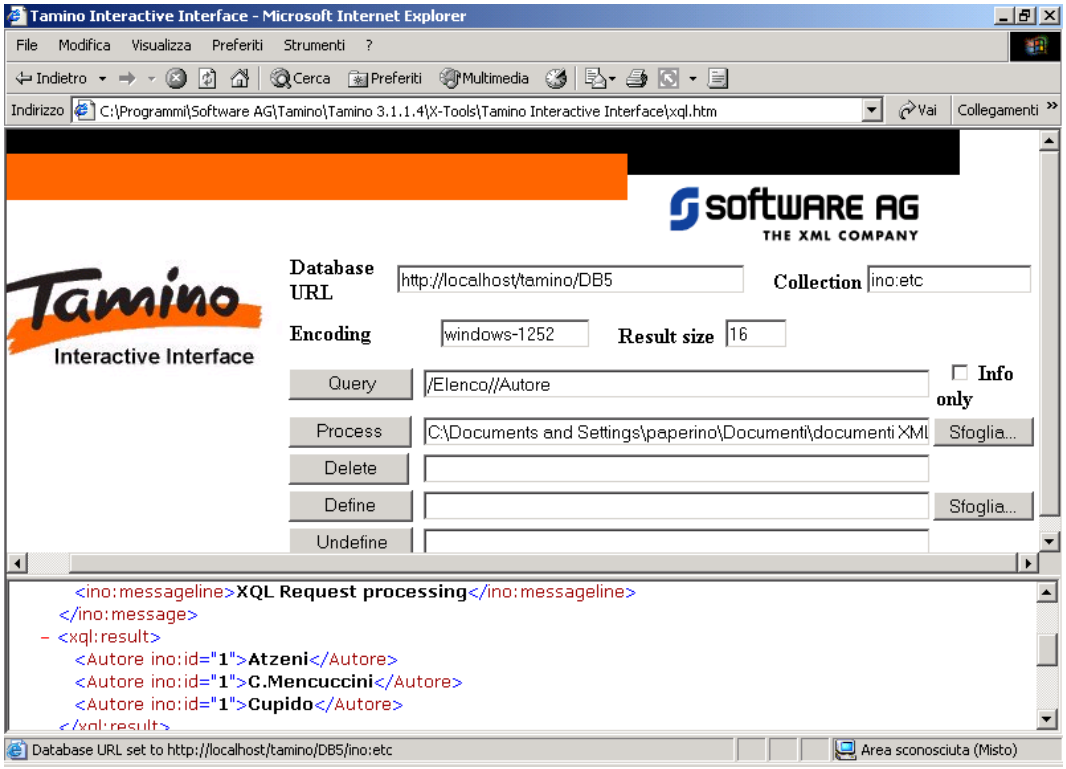
- <ino:message ino:returnvalue="0">

<ino:messageline>XQL Request processing</ino:messageline>
</ino:message>
- <xql:result>
<Titolo ino:id="1">Esercizi di Elettronica Generale</Titolo>
</xql:result>
- <ino:cursor ino:count="1">
<ino:first
ino:href="?_XQL(1,16)=/Elenco/Libro[Editore='Liguori'and@disponibilità='S']/Titolo" />
</ino:cursor>
- <ino:message ino:returnvalue="0">
<ino:messageline>XQL Request processed</ino:messageline>
</ino:message>
</ino:response>

Query 2

Se vogliamo individuare tutti gli autori che si trovano nell'oggetto XML memorizzato, dobbiamo considerare la seguente stringa:

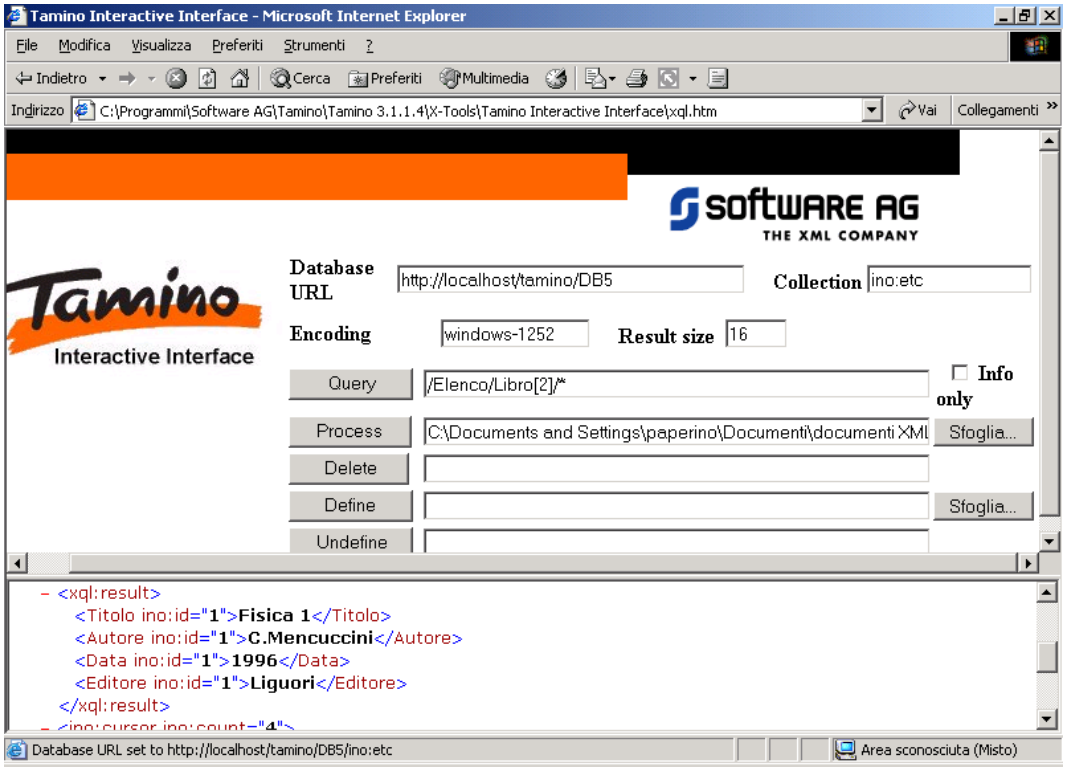
Elenco//Autore



Query 3

Concludiamo con una Query che non può essere eseguita con SQL, difatti è una Query che ci consente di interrogare l'ordine dei dati. Se vogliamo l'ordine dei dati contenuti nel secondo libro dobbiamo considerare la seguente stringa:

/Elenco/Libro[2]/*



Il risultato esteso è il seguente:

<?xml version="1.0" encoding="windows-1252" ?>

- <ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2" xmlns:xql="http://metalab.unc.edu/xql/">

<xql:query>/Elenco/Libro[2]/*</xql:query>

- <ino:message ino:returnvalue="0">

<ino:messageline>XQL Request processing</ino:messageline>

</ino:message>

- <xql:result>

<Titolo ino:id="1">Fisica 1</Titolo>

<Autore ino:id="1">C.Mencuccini</Autore>

<Data ino:id="1">1996</Data>

<Editore ino:id="1">Liguori</Editore>

</xql:result>

- <ino:cursor ino:count="4">

<ino:first ino:href="?_XQL(1,16)=/Elenco/Libro[2]/*" />

</ino:cursor>
- <ino:message ino:returnvalue="0">
<ino:messageline>XQL Request processed</ino:messageline>
</ino:message>
</ino:response>

Conclusioni

L'obbiettivo di questa Tesi è stato quello di esplorare degli oggetti ancora non ben definiti i " Database XML Nativi (NXD)".

Una delle cose certe è che tali NXD consentono di archiviare direttamente documenti XML, senza alcuna fase di " traduzione ".

L'esigenza di gestire documenti XML nasce dal fatto che XML si propone come formato universale per lo scambio dei dati, abbattendo quindi, i "costi " per fare e-Commerce .

Fare e-Commerce significa completare con successo delle transazioni elettroniche tra aziende diverse usando il WEB: requisiti fondamentali sono una completa indipendenza dal tempo, dal luogo, e dall'infrastruttura sottostante.

Una volta compreso l'importanza di gestire documenti XML, sono state prese in considerazione due possibilità, una appunto è quella degli NXD, l'altra è quella dei DB XML-Enabled, cioè di quelli che offrono un supporto ad XML.

La sfida tra le due soluzioni è ancora aperta, anche se le limitazioni dei linguaggi di interrogazione dei documenti XML (che in effetti sono quasi tutti read-only) sembra avvantaggiare quelli Enabled, che invece usano un linguaggio ben assestato e completo come SQL. D'altra parte però, gli NXD non presentando alcuna fase di traduzione (o meglio di mapping), sono decisamente più veloci, più affidabili, e richiedono una manutenzione minore. Quindi ciascuna azienda sceglierà quale soluzione adottare in base al volume e alla complessità delle transazioni di e- Business, se la mole di transazioni è notevole, in tal caso potrebbe essere vantaggioso adottare gli NXD.

Bibliografia

[1] Andrea Chiarelli "Da SGML a XML passando per HTML" , DEV nr. 70, Gennaio 2000.

[2] http:// www.liberliber.it/bibloteca/c/calvo/internet98

[3] Appunti Corso di Tecnologie biomediche. Prof Alesando Pepino

[4] http:// www.html.it/xml/guida

[5] Massimo Autiero "Elaborare doc. XML in VB", IO Programmo nr.44, Febbraio 2001

[6] Lorenzo Vandoni "Modellare i dati con XML", DEV nr. 103,Gennaio 2003

[7] Gianluca Nuscis " Gli attributi XML", IO Programmo nr.31, Dicembre 1999

[8] Massimo Ruocchio " XML Schema ai mondiali di calcio", Computer Programmino nr.109, Gennaio 2002

[9] http:// Nike.psv.edu/ publications.html

[10] Corrado Mencar "Le insidie del data modeling con XML", Computer Programmino nr.109, Gennaio 2002

[11] Shanmugasundarem,J,Tufte,DeWitt " Relational Databases for Querrying XML Documents: Limitiations and oppurtunities"

[12] ACM Transaction an Databases System, vol 27, nr.4, December 2002

[13] Lorenzo Vandoni " XML e i DBMS", DEV nr. 104, Febbraio 2003

[14] http:// www.nwi,it

[15] http:// www.docflow.com/risorse2.htm

[16] http://della penna.univaa.it

[17] http:// www.xml.com/pub/a/2001/10/31/nativexmldb.html

[18] R.Bourret "XML and Databases", www.rpbourret.com/xml/index.htm,2002

[19] Artur Afonso de Sousa, Josè Luis Pereira and Jaao Alvano Corvallo "Querying XML Databases", Computer Science Society

[20] http:// www.w3.org/tr/2003/WD-xpath

[21] http:// www.xml.com/pub/a/2002/10/16/xquery.html

[22] Daniele Miselli "Riscrittura di interrogazioni XML: un'approccio basato sull'analisi semantica degli schemi", Tesi di laurea presso la Facoltà di ingegneria dell'Università degli Studi di Modena e Reggio Emilia

[23] Maurizio Bergami e Michael Floyd "Database XML Nativi" , PC Professional, Gennaio 2002

[24] Software AG "Tamino 2.3.1" , Documentazione tecnica, 2003

[25] http:// www.itware.com/tool9910/47_48.htm