UNIVERSITA' DEGLI STUDI DI MODENA
E REGGIO EMILIA

Facoltà di Ingegneria - Sede di Modena

Corso di Laurea in Ingegneria Informatica

Query OQL e XQUERY a confronto

Relatore
Prof. Sonia Bergamaschi

Tesi di Laurea di
Campana Tiziana

Correlatore
Ing. Francesco Guerra

Anno Accademico 2002 - 2003

Indice

Introduzione

Intuitivamente un modello dei dati è uno strumento concettuale che consente al progettista di attribuire un certo signifcato (o interpretazione) ai dati e di manipolare i dati stessi. Si può affermare che la vera e propria teoria delle basi di dati come disciplina informatica è nata con la nozione di modello dei dati.

DBMS Orientati agli Oggetti:

- **Modello Logico:**

- I dati sono organizzati in classi
- Ogni classe genera un insieme di oggetti
- Con un insieme di propietà e di metodi
- Oggetti diversi sono identificati sulla base di correlatori

Sistemi basati su XML

- **Modello Logico:**

- I dati sono organizzati in strutture gerarchiche (alberi)
- Ogni albero ha un insieme di nodi (elementi)
- Oggetti diversi sono correlati sulla base di relazioni di contenimento

Linguaggio per Basi di Dati

- **DBMS a oggetti: OQL**
  XML: XPATH (prima), Xquery (adesso)

Capitolo I
XML

1.1 Introduzione

L'Extensible Markup Language (XML) è un metalinguaggio, ovvero un insieme di regole base utilizzate per creare altri linguaggi. Dalla sua nascita avvenuta nel 1998, ha assunto un ruolo centrale per quanto riguarda lo sviluppo di nuove tecnologie in ambito Web e non solo. XML permette di creare nuovi linguaggi per rappresentare informazione strutturata in formato testuale.

Il suo successo è dovuto a molti motivi e tra i principali possiamo sicuramente includere la sua semplicità, la sua espandibilità e la sua portabilità.

1.2 Cosa è XML?

- XML sta per E**X**tensible **M**arkup **L**anguage.
- XML è un linguaggio di markup come HTML.
- XML è stato progettato per descrivere i dati.
- I *tags* XML non sono predefiniti. È possibile definire dei propri *tags.*
- XML usa un **Document Type Definition** (DTD) o un **XML schema** per descrivere i dati.
- XML con un DTD o XML schema è stato progettato per essere auto-descrittivo.

Le principali differenze tra XML e HTML

Una caratteristica fondamentale di XML è quella di occuparsi del contenuto dell'informazione e non la sua rappresentazione. La modalità di rappresentazione dell'informazione può essere scelta in un secondo momento e, partendo dallo stesso file XML, possiamo rappresentare l'informazione contenuta al suo interno in differenti modi, come ad esempio in HTML, XHTML, SVG, etc.

XML non è una sostituzione di HTML.
XML e HTML sono stati quindi progettati con scopi differenti.

Tags XML

I tags XML non sono predefiniti. Ognuno puoò "inventare" i propri tags.

La specifica di XML descrive un insieme di regole per definire linguaggi basati su tag (marcatori) all'interno dei quali inserire in maniera strutturata il contenuto informativo da rappresentare.

Nel caso di HTML: I tags sono usati per valorizzare (contrassegnare) un documento HTML e le strutture dei documenti HTML sono predefinite. Gli autori dei documenti HTML possono solo usare tags che sono definiti negli standard HTML (like <p>, <h1>, etc.).

XML permette agli autori di definire dei suoi tags e una propria struttura del documento.

I tags non sono definiti in alcun standard XML. Questi tags vengono "inventati" dagli autori del documento.

I tag devono essere inseriti correttamente l'uno dentro l'altro, ad ogni tag di apertura deve corrispondere un tag di chiusura e gli attributi dei tag devono essere racchiusi tra apici.

Tutti i documenti XML devono contenere una singola coppia di tag che definiscono una root element.

Utilizzando queste semplici ed essenziali regole l'utente ha la possibilità di creare un nuovo linguaggio definendo i tag e gli attributi più appropriati a memorizzare l'informazione che si vuole trattare.

XML è un complemento ad HTML
XML non è un sostituto di HTML.

Nel futuro sviluppo del Web XML sarà utilizzato per descrivere i dati, mentre HTML sarà utilizzato per il formato e la visualizzazione di alcuni dati.

Inoltre XML favorisce l'interoperabilità in quanto è un formato testuale, quindi facilmente trasferibile ed elaborabile su differenti piattaforme hardware e software.

1.3 Sintassi XML

Per **elemento XML** si intende qualsiasi cosa contenuta tra il tag d'inizio e quello di chiusura. Gli elementi vengono suddivisi in:

- Element content : contiene all'interno altri elementi
- Mixed content: contiene sia testo che altri elementi
- Simple content: contiene solo testo
- Empty content: non porta informazioni (vuoto)

Gli elementi XML possono avere sttributi, sono posizionati all'interno del tag d'inizio e forniscono informazioni aggiuntive sull'elemento non sui dati (i datiche vengono descritti dall'elemento).

1.4 ESEMPIO

Come esempio vediamo adesso di memorizzare in formato xml le informazioni relative ad una rubrica. Tipicamente in una rubrica, sono conservati alcuni dati (quali ad esempio nome, cognome, indirizzo, numero di telefono) riguardanti una determinata persona e tutto questo in XML può essere espresso con:

```
<?xml version="1.0"?>
<rubrica>
  <persona>
    <nome>Mario</nome>
    <cognome>Rossi</cognome>
    <indirizzo>
      <via>via bianchi 1</via>
      <cap>00000</cap>
      <citta>Roma</citta>
    </indirizzo>
    <telefono>
      <telefono_fisso>123456</telefono_fisso>
      <cellulare>987656412</cellulare>
    </telefono>
  </persona>
</rubrica>
```

In questo esempio abbiamo descritto in formato XML le informazioni relative ad una rubrica, andando a creare gli opportuni tag ed annidandoli in modo tale da rappresentare la struttura dati che avevamo in mente.

La prima linea del documento – the XML declaration- definisce la versione di XML, in questo caso la 1.0, e la character encoding che in questo caso è stata omessa.

La seconda linea descrive l'elemento root (e come se stesse dicendo: " questo documento è una rubrica").

Le successive linee rappresentano gli elementi figli dell'elemento root , mentre l'ultima linea definisce la fine dell'elemento root.

Tutti gli elementi XML devono avere un tag di chiusura e differentemente da HTML i tags XML sono sensibili al carattere scritto in maiuscolo o in minuscolo.

Proprio a causa delle caratteristiche descritte, XML ha incontrato molto successo come formato per lo scambio di dati tra applicazioni differenti.

Affinché due applicazioni possano scambiarsi dei dati in formato XML è necessario però che queste conoscano come viene strutturata l'informazione all'interno del file, ovvero quali sono e come sono chiamati i tag e gli attributi che costituiscono il file XML.

Per questo scopo sono state sviluppate alcune tecnologie quali XML Namespace, DTD e XML Schema.

1.5 XML NameSpace

Serve per evitare conflitti tra i nomi degli elementi di diversi documenti XML.

Siccome i nomi degli elementi in XML non sono fissati potrebbero esserci dei conflitti tra due differenti documenti che utilizzano gli stessi nomi per descrivere due differenti tipi di elementi.

1.6 ESEMPIO Namespace

Questo documento porta informazioni di una tabella

```
<table>
  <tr>
  <td>Apples</td>
  <td>Bananas</td>
  </tr>
</table>
```

Quest'altro documento porta informazioni su un'altra tabella

```
<table>
  <name>African Coffee Table</name>
  <width>80</width>
  <length>120</length>
</table>
```

Se questi due documenti sono aggiunti insieme, ci sarà un conflitto di nomi perché entrambi documenti contengono un <table> elemento con differenti contenuti e definizioni.
Utilizzando un prefisso, noi possiamo creare due differenti tipi di <table> elementi in questo modo:

```
<h:table>
  <h:tr>
  <h:td>Apples</h:td>
  <h:td>Bananas</h:td>
  </h:tr>
</h:table>
```

```
<f:table>
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>
```

Ora non ci sono conflitti sui nomi degli elementi perché i due documenti usano due differenti nomi per l'elemento <table> (< h:table> e < f:table>)

Quindi XML Namespace è un insieme di nomi, utilizzato per definire gli elementi e gli attributi di un linguaggio XML, a cui viene associato un URI (Universal Resource Identifier) di riferimento tramite il quale identificare in maniera assoluta il namespace. Un URI (Universal Resource Identifier) è una stringa di caratteri che definisce il nome di una risorsa sul web ed è per definizione unica. A questo insieme di nomi viene associato un prefisso che utilizzato in abbinata con il nome dell'elemento, permette di riferire univocamente tutti gli elementi.

Come per gli elementi è possibile definire degli namespace attributi.

Come esempio vediamo di definire un namespace per l'insieme di elementi utilizzati per descrivere la rubrica.

```
<?xml version="1.0"?>
<mia_rubrica:rubrica xmlns:mia_rubrica="http://indirizzo_del_sito/mia_rubrica_ns">
  <mia_rubrica:persona>
    <mia_rubrica:nome>Mario</mia_rubrica:nome>
    <mia_rubrica:cognome>Rossi</mia_rubrica:cognome>
    <mia_rubrica:indirizzo>
      <mia_rubrica:via>via bianchi 1</mia_rubrica:via>
      <mia_rubrica:cap>00000</mia_rubrica:cap>
      <mia_rubrica:citta>Roma</mia_rubrica:citta>
    </mia_rubrica:indirizzo>
```

```
    <mia_rubrica:telefono>
      <mia_rubrica:telefono_fisso>123456</mia_rubrica:telefono_fisso>
      <mia_rubrica:cellulare>987656412</mia_rubrica:cellulare>
    </mia_rubrica:telefono>
  </mia_rubrica:persona>
</mia_rubrica:rubrica>
```

Attraverso l'attributo xmlns (inserito nel primo tag del nostro file) definiamo l'URI di riferimento associato al nostro namespace e gli assegniamo un prefisso (mia_rubrica) che sarà utilizzato in abbinata con il nome dell'elemento per identificare in maniera univoca gli elementi del nostro file.

Capitolo II

DTD

Per quanto riguarda la descrizione della struttura di un file XML, esistono due tecnologie: DTD e XML Schema.

Document Type Definition (DTD) è un linguaggio utilizzato per definire la struttura di un file XML ed è storicamente il primo metodo utilizzato per tale scopo. È caratterizzato da una sintassi complessa, difficilmente estendibile e totalmente estranea al mondo XML.

DTD definisce gli elementi legali di un documento XML.

Per vedere un esempio di DTD proviamo a definire la struttura del nostro file XML che descrive la rubrica.

```
<!DOCTYPE rubrica [
<!ELEMENT rubrica (persona)>
<!ELEMENT persona (nome, cognome, indirizzo, telefono)>
<!ELEMENT nome (#PCDATA)>
<!ELEMENT cognome (#PCDATA)>
<!ELEMENT nome (#PCDATA)>
<!ELEMENT indirizzo (via, cap, citta)>
<!ELEMENT via (#PCDATA)>
<!ELEMENT cap (#PCDATA)>
<!ELEMENT citta (#PCDATA)>
<!ELEMENT telefono (telefono_fisso, cellulare)>
<!ELEMENT telefono_fisso (#PCDATA)>
<!ELEMENT cellulare (#PCDATA)>
]>
```

Per ovviare alle limitazioni della DTD è nato XML Schema, ovvero un linguaggio, basato su XML, per definire la struttura di un documento XML.

XML Schema è uno strumento molto più potente di DTD per descrivere la struttura di un file XML in quanto è più espandibile, permette di ottenere una migliore caratterizzazione dei tipi di dati e ha una sintassi basata su XML, quindi possiamo utilizzare gli strumenti XML per lavorare con gli Schema. Ma noi per adesso poniamo attenzione solo sul DTD.

Capitolo III

XPATH

Iniziamo con questo capitolo, la panoramica sulle tecnologie della famiglia XML andando a parlare di XPath. XPath è un linguaggio tramite il quale è possibile esprimere delle espressioni per indirizzare parti di un documento XML.

È un linguaggio ideato per operare all'interno di altre tecnologie XML (esempio XQUERY), ed è caratterizzato dal fatto di avere una sintassi non XML.In questo modo può essere meglio utilizzato all'interno di URI o come valore di attributi di documenti XML.

XPath opera su una rappresentazione logica del documento XML, che viene modellato con una struttura ad albero ed XPath definisce una sintassi per accedere ai nodi di tale albero. Oltre a questo XPath mette a disposizione una serie di funzioni per la manipolazione di stringhe, numeri e booleani, da utilizzare per operare sui valori o sugli attributi dei nodi.
Le espressioni definite da XPath per accedere ai nodi dell'albero prendono il nome di Location Path (percorsi di localizzazione).
La struttura di un location path è la seguente: *axis::node-test[predicate]*.
La componente *axis* esprime la relazione di parentela tra il nodo cercato ed il nodo corrente; la componente *node-test* specifica il tipo o il nome del nodo da cercare; mentre *predicate* contiene zero o più filtri (espressi tra parentesi quadre) per specificare delle condizioni più selettive da applicare alla ricerca.
Le relazioni di parentela principali che possono essere contenute in *axis* sono:

- ancestor: indica tutti i nodi antenati del nodo corrente, ovvero tutti i nodi che lo precedono nell'albero associato al documento XML;
- attribute: indica tutti gli attributi del nodo corrente;
- child: indica i nodi figli del nodo corrente;
- descendant: indica tutti i discendenti del nodo corrente, ovvero tutti i nodi che hanno seguono il nodo corrente nell'albero XML;
- parent: indica il nodo genitore del nodo corrente, ovvero quello che lo precede nell'albero;
- self: indica il nodo corrente.

Vediamo qualche esempio per capire meglio come utilizzare i Location Path per accedere agli elementi di un documento XML, utilizzando l'esempio della rubrica introdotto nel capitolo precedente, anche se con qualche piccola modifica.

```
<?xml version="1.0"?>
<rubrica>
  <persona>
    <nome>Mario</nome>
    <cognome>Rossi</cognome>
    <indirizzo>
      <via>via bianchi 1</via>
      <cap>00000</cap>
      <citta>Roma</citta>
    </indirizzo>
    <telefono>
      <telefono_fisso gestore="Abc">123456</telefono_fisso>
      <telefono_cellulare gestore="Def">987656412</telefono_cellulare>
    </telefono>
  </persona>
</rubrica>
```

child::nome
Questa espressione seleziona tutti i nodo chiamati 'nome' che sono figli del nodo corrente.
*child::**
Seleziona tutti i nodi figli del nodo corrente.
attribute::gestore
Seleziona l'attributo di nome 'gestore' del nodo corrente.
descendant::cognome[cognome='Rossi']
Seleziona tutti i nodi chiamati 'cognome' tra i nodi discendenti del nodo corrente, il cui valore è Rossi.

In XPath è possibile accedere ai nodi dell'albero utilizzando delle espressioni abbreviate dei Location Path. Le espressioni abbreviate, come suggerisce il nome stesso, sono una versione semplificata e compatta dei Location Path ed offrono un meccanismo più veloce, ma al tempo stesso meno potente per accedere ai nodi dell'albero. Queste espressioni sono costituite da una lista di nomi di elementi del documento XML, separati da uno slash(/), e tale lista descrive il percorso per accedere all'elemento desiderato. È un meccanismo molto simile a quello usato per identificare i file e le directory nel filesystem. Ad esempio per indicare il file (chiamato mio-file) memorizzato all'interno di una directory (chiamata mia-directory) del vostro hard disk, utilizzate la seguente sintassi: c:\directory1\file1. Le espressioni abbreviate di XPath utilizzano un meccanismo concettualmente simile: vediamo come, utilizzando il solito file XML d'esempio.

/rubrica/persona/nome
Questa espressione abbreviata permette di recuperare i nodi chiamati 'nome' indicando

il percorso assoluto per raggiungere il nodo desiderato, attraverso una lista di nodi separata da slash.

//nome
Con il doppio slash ricerchiamo i nodi chiamati 'nome', in tutto il documento, indipendentemente dalla loro posizione e dal loro livello sull'albero associato al documento XML.

/rubrica//via
Ricerchiamo tutti i nodi chiamati 'via' a qualsiasi livello dell'albero purchè contenuti all'interno del nodo chiamato 'rubrica'.

*/rubrica/persona/**
Ricerca qualsiasi elemento figlio del nodo chiamato 'persona'.

//nome/@valuta
Ricerca l'attributo 'valuta' dell'elemento 'nome'.

Anche all'interno delle espressioni abbreviate possiamo inserire i predicati visti nel caso dei Location Path. Ad esempio:

//persona[nome='Mario']
questa espressione ricerca tutti i nodi 'persona' che hanno il tag 'nome' il cui valore è Mario.

Come detto all'inizio del capitolo, XPath mette a disposizione anche delle funzioni per gestire i nodi, le stringhe, i numeri e i booleani. Vediamo adesso di elencare brevemente e schematicamente alcune funzioni principali:

- count(node-set): restituisce il numero di nodi contenuti nell'insieme di nodi passato come argomento della funzione;
- name(nodo): restituisce il nome di un nodo;
- position(): determina la posizione di un elemento all'interno di un insieme di nodi;
- last(): indica la posizione dell'ultimo nodo di un'insieme di nodi;
- id(valore): seleziona gli elementi in funzione del loro identificatore;
- concat(s1,...,sn): restituisce una stringa risultato della concatenazione delle stringhe specificate tra gli argomenti di una funzione;
- string(valore): converte il valore dell'argomento in una stringa;
- string-length(stringa): ritorna la lunghezza della stringa passata come parametro;

- substring(stringa,inizio,lunghezza): restituisce una sotto-stringa della stringa passata come argomento;
- ceiling(numero): arrotonda il numero per eccesso;
- floor(numero): arrotonda il numero per difetto;
- number(valore): converte il valore dell'argomento in un numero;
- sum(node-set): esprime la somma di un insieme di valori numerici contenuti in un insieme di nodi.

Come esempi di espressioni XPath che fanno uso di queste funzioni consideriamo:

//persona[last()]
Questa espressione restituisce l'ultimo nodo 'persona' contenuto all'interno del file XML.
//persona[position()= 3]
Restituisce il terzo nodo 'persona' contenuto all'interno del file XML.
count(/rubrica/persona)
Indica il il numero di nodi chiamati 'persona' contenuti all'interno del documento XML.

Capitolo IV
XQUERY

4.1 Introduzione

Negli ultimi anni la RETE è diventata protagonista di una metamorfosi che la farà sempre più simile a una sterminata struttura in cui i dati sono tra loro totalmente interconnessi e quindi agevolmente manipolabili.

L'ondata di novità copre in prevalenza ambiti operativi già attivi in seno al World Wide Web. In particolare, riguarda XPath, il metalinguaggio che offre la possibilità di contrassegnare le parti rilevanti di un documento Xml; XQuery, che definisce i modi per ricercare documenti Xml; XSlt, che si occupa di tradurre documenti scritti in linguaggi Xml differenti e tutta una serie di loro sottoinsiemi.
Prima dell'avvento di XQUERY, l'unico modo per interrogare un database che conteneva dati nel formato XML era quello di utilizzare XPath, un linguaggio nato per la navigazione all'interno dei dati in formato XML, che offre soltanto un sottoinsieme delle funzionalità di XQuery.

XQuery è basilare come meccanismo per la ricerca e l'accesso ai dati XML. XQuery è stato progettato per supportare completamente il paradigma XML, il che vuol dire che per le applicazioni che intendono utilizzare la tecnologia XML, sarà molto più semplice utilizzare questo linguaggio di interrogazione, che incrementerà la loro produttività, che non XPath che invece è progettato pensando alla navigazione.

"Sono sempre di più le grandi organizzazioni che scelgono di memorizzare le proprie informazioni nel formato XML e da qui nasce la necessità di potere disporre di un potente linguaggio di interrogazione. Molti paragonano XQuery a SQL proprio per la sua capacità espressiva. Inizialmente si era pensato di usare SQL per effettuare le ricerche nei documenti XML, ma questo linguaggio non è adatto per cercare in modo gerarchico mentre invece XML è essenzialmente un formato gerarchico."

In questo capitolo andiamo ad analizzare una tecnologia ideata per il recupero delle informazioni memorizzate all'interno di un file XML.

È molto importante per la diffusione e l'utilizzo di XML nell'ambito di documenti contenenti grandi quantità di dati, avere a disposizione uno strumento relativamente facile e potente per poter recuperare l'informazione presente in un file XML.

Questo strumento deve permettere di realizzare delle query (interrogazioni) sul documento proprio come avviene ad esempio con il linguaggio SQL nel caso dei database relazionali.

XML Query languge (XQuery) nasce proprio con l'intento di realizzare un linguaggio per recuperare agevolmente le informazioni da un documento XML . **XQuery non è un linguaggio basato su XML ed è costituito da una sintassi semplice e facilmente leggibile per formulare, nel modo più agevole possibile, le query sui dati.** Il working group del W3C ha sviluppato anche una versione di XQuery con sintassi XML, chiamata XQueryX ma io farò in questa tesi un'analisi su XQUERY.

Una query in XQuery è costituita da un'espressione che legge una sequenza di nodi XML od un singolo valore e restituisce come risultato una sequenza di nodi od un singolo valore. Le espressioni XQuery sono composte da espressioni XPath per individuare i nodi da analizzare e da delle funzionalità aggiuntive specifiche di XQuery per il recupero delle informazioni.

4.2 Struttura XQUERY

- XQuery è un linguaggio funzionale

  - Una Query è una espressione.

  - Le espressioni possono essere liberamente combinate.

- Struttura di una Query

  - Dichiarazione Namespace (opzionale)

  - Definizione funzioni (opzionale)

  - Importo schema (opzionale)

  - L'espressione della Query - spesso si compone di molte spressioni -

Query Prolog

Expr Sequence

In questa tesi analizzerò :

4.3 L'espressione della Query

L'espressione della Query è formata dai seguenti elementi:

- Path expressions
- FLWR espressions
- Element constructors: <a>….
- Variabili e costanti: $x, 5
- Operatori e chiamate di funzioni: x+y, -z, foo(x,y)
- Conditional expressions: IF….THEN….ELSE
- Quantifiers: EVERY var, IN expr, SATISFIES expr
- Sorted expressions: expr SORTBY (expr ASCENDING, …)

4.3.1 Path expression

- "**Navigare**" nella struttura ad albero del documento.
- "**Identificare**" liste (*ordinate*) di "frammenti" XML
-

Costrutti delle path expression

- . 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)

Esempi di path expression

- Una path expression può ***iniziare*** con

document("*nome-documento*")

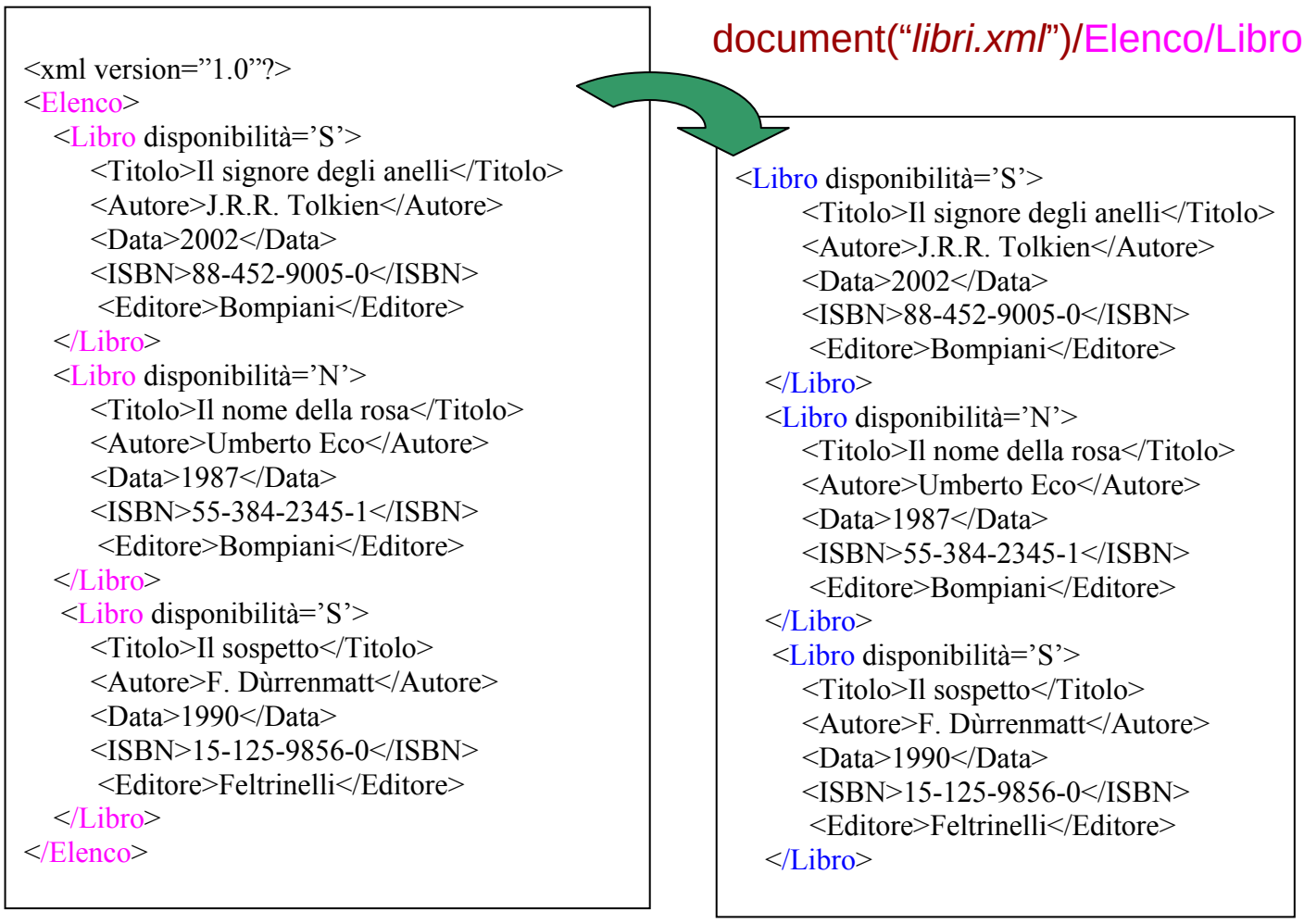
[restituisce la radice del documento specificato]

- A partire dalla radice è possibile 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*]

Documento



- 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>
```

- document(“libri.xml”)//Autore

  Restituisce l’insieme di tutti gli autori che si trovano nel documento libri.xml annidati a qualunque livello

```
<Autore>J.R.R. Tolkien</Autore>
<Autore>Umberto Eco</Autore>
<Autore>F. Dùrrenmatt</Autore>
```

- document(“libri.xml”)/Elenco/Libro**[2]**/*

  Restituisce gli elementi contenuti nel **secondo** libro del documento libri.xml

  Permette di **INTERROGARE L’ORDINE** dei dati

```
<Titolo> Il nome della rosa </Titolo>
<Autore> Umberto Eco </Autore>
<Data> 1987 </Data>
<ISBN> 55-384-2345-1 </ISBN>
<Editore> Bompiani </Editore>
```

NON esiste analogo **SQL!**

4.3.2 XQUERY: Espressioni FLWR (“flower”)

- for per l’iterazione
- let per legare variabili a insiemi
- where per esprimere predicati
- return per generare il risultato

4.3.2.1 Espressioni FOR

- Esempio:

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

- La clausola FOR valuta l’espressione e ***itera*** all’interno di questa *lista* legando [binding] i singoli elementi alla 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
- Le espressioni FOR possono essere annidate:

```
for $x in document(“libri.xml”)//Libro
    $a in $x/Autore
return $a
```

Come in OQL si scrive per estrarre i cugini (uno alla volta) :
FROM Z in Zio, C in Z.Figli

```
<Elenco>
  <Libro disponibilità=’S’>
    <Titolo>Designing Data Intensive
            Web Applications</Titolo>
    <Autore>Stefano Ceri</Autore>
    <Autore>Pietro Fraternali</Autore>
    <Autore>Aldo Bongio</Autore>
    <Autore>Marco Brambilla</Autore>
    <Autore>Sara Comai</Autore>
    <Autore>Maristella Matera</Autore>
     …
  </Libro>
…
</Elenco>
```

4.3.2.2 Espressione LET

- Esempio:

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

- La clausola LET valuta l'espressione 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** sola volta la sua clausola RETURN

4.3.2.3 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

4.3.2.4 Clausola RETURN

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

–Un nodo

```
<Autore>StefanoCeri</Autore>
```

–Una lista ordinata di nodi

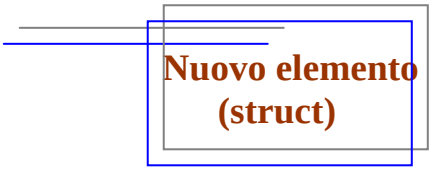
```
<Autore>Stefano Ceri</Autore>
<Autore>Umberto Eco</Autore>
<Autore>Sara Comai</Autore>
```

–Un valore

```
Stefano Ceri
```

- Può contenere *cosruttori* di elementi, *riferimenti* a variabili definite nelle parti FOR e LET e generiche *espressioni annidate*
- Un ***costruttore*** di un elemento consiste in un tag iniziale e di un tag finale che racchiudono una lista (opzionale) di espressioni **ARBITRARIE** che, valutate, determinano il contenuto dell'elemento costruito
- Esempio:

```
for $x in document("libri.xml")//Libro
where $x/Editore="Bompiani"
retun <Libro-Bompiani>
          {$x/Titolo}
      </Libro-Bompiani>
```




```
<Libro-Bompiani><Titolo>Il signore degli anelli</Titolo></Libro-Bompiani>
<Libro-Bompiani><Titolo>Il nome della rosa</Titolo></Libro-Bompiani>
```

4.3.2.5 Funzioni Aggregate

- Esempio:

```
for $e in document("libri.xml")//Libro/Editore
let $x:= document("libri.xml")//Libro[Editore = $e]
where count($x) > 100
return $eù
```

- Restituisce gli editori che nell'elenco hanno almeno 100 libri
- Elenco delle funzioni maggiormente utilizzate:
  - max(), min(), sum(), count(), avg()
  - distinct-values(), empty(), contains()

4.3.2.6 Ordinare il risultato

- Esempio:

```
for $x in document("libri.xml")//Libro
return
    <Libro>
        $x/Titolo,
         $x/Editore
    </Libro>
 ORDER BY Titolo ASCENDING
```

- I libri vengono (ovviamente) ordinati rispetto al titolo

4.3.2.7 Analogia tra XQUERY e OQL

for	per l'iterazione	→	**[from]**
let	per legare variabili a insiemi	→	**[group by]**
where	per esprimere predicati	→	**[where]**
return	per generare il risultato	→	**[select]**

Capitolo IV
Il linguaggio OQL

5.1 Introduzione

I modelli di dati orientati ad oggetti nascono come risposta ad una serie di esigenze che i modelli precedenti non riuscivano a soddisfare: maggiore uniformità di trattazione di dati ed operazioni, maggiore separazione tra livello implementativo e livello di interfaccia, costituzione di librerie di oggetti che rendono semplice ed effciente il riutilizzo del software, semplificare la realizzazioni di applicazioni complesse che trattano grandi quantità di dati. Questi modelli hanno trovato nei sistemi di gestione di basi di dati un importante campo applicativo.
In questo capitolo viene descritto il linguaggio Object Query Language (OQL), il componente dello standard ODMG-93 utilizzato per l'interrogazione di basi di dati orientate ad oggetti.
Di seguito vengono illustrate le caratteristiche principali di OQL.

5.2 Aspetti principali del linguaggio

Il linguaggio OQL si basa sui seguenti principi ed assunzioni:

- OQL _e basato sul modello ad oggetti definito da ODMG.
- OQL utilizza una sintassi simile a quella definita per SQL 92. Rispetto a questa presenta delle estensioni finalizzate alla gestione degli aspetti object-oriented, in particolare le espressioni di percorso, il polimorfismo e l'invocazione delle operazioni.
- OQL _e un linguaggio funzionale in cui gli operatori possono essere liberamente composti, a patto di rispettare la compatibilità tra tipi, secondo quanto previsto dal sistema dei tipi di ODMG-93.
- OQL non è completo dal punto di vista computazionale.

.

- OQL non fornisce in modo esplicito operatori per l'aggiornamento della base di dati, lasciando questo compito a operazioni opportune che fanno parte delle caratteristiche di ogni oggetto che popola il database. In questo modo si rispetta la semantica propria degli ODBMS che, per definizione, vengono gestiti attraverso i metodi definiti sugli oggetti.

5.3 Input e Output di una query OQL

Utilizzato come un linguaggio indipendente, OQL consente di estrarre gli oggetti da una base di dati attraverso il loro nome, che agisce come entry-point. Un nome denota un qualunque tipo di oggetto, sia esso atomico,

strutturato, letterale, collezione ecc. Questo è il principale modo di impiego del linguaggio.
Utilizzato invece in maniera embedded, OQL permette di utilizzare le query, all'interno del linguaggio di programmazione, come funzioni che restituiscono oggetti.

5.4 Aspetti principali del linguaggio

Fatte le precedenti precisazioni possiamo osservare che, in generale, un'interrogazione OQL estrae oggetti da una collezione che corrisponde all'estensione di una classe.
Facendo riferimento allo schema presente nel capitolo Applicazione, quindi, la seguente interrogazione in linguaggio OQL

select * from Ordine

non sarebbe intesa sui dati (come avviene generalmente nei sistemi relazionali) ma sui metadati: infatti restituirebbe in uscita la struttura della classe Ordine.
Per poter introdurre degli oggetti nelle classi definite nello schema presente nel capitolo Applicazione e quindi effettuare in seguito delle interrogazioni su tali oggetti, occorre aggiungere a tale schema le dichiarazioni dei nomi delle estensioni delle classi:

```
interface Ordine
(extent ordini
key(ID_numero))
{
    attribute integer ID_numero;
    attribute integer Data_ordine;
    attribute integer Data_richiesta;
    attribute set<Spedizione> trasportato;
    attribute set<DettagliOrdine> contiene;

};
```

Supponiamo ad esempio, di istanziare un oggetto di tipo Ordine:

Ordine(ID_numero: "8888", Data_ordine: “23/10/2003”, Data_richiesta: "19/09/2003").

select * from Materials where name = "M1"
Volendo estrarre dal database l'oggetto istanziato si potrebbe ricorrere all'interrogazione seguente:

Select * from Ordine where ID_numero = ”8888”.

Il cui risultato sarebbe appunto l’oggetto 8888.

5.5 Selezione di oggetti esistenti

Le espressioni che estraggono oggetti dal database possono restituire:

- una collezione di oggetti con OID. Ad esempio l'interrogazione:

  select C from Cliente as C where C.name = "Rita"

  restituisce una collezione di clienti di nome "Rita".
- un oggetto con identità. Ad esempio la query seguente:

  select C from Cliente as C where M.ID_cliente = "356464"

  restituisce l'unico cliente il cui codice è "356464".
- una collezione di letterali. Ad esempio l'interrogazione:

  select C.ID_cliente from Cliente as C where C.name = "Rita"

  restituisce una collezione di stringhe di caratteri corrispondenti al codice di ogni cliente di nome "Rita".
- un letterale. Ad esempio la query seguente:

  select C.età from Cliente C where Cliente.ID_cliente = "356464"

  restituisce il numero intero corrispondente all’età dell'unico cliente il cui codice è "356464".

Una query può quindi restituire un oggetto (o un insieme di oggetti) con o senza identità; alcuni di questi oggetti possono essere generati dall'interprete del linguaggio, se l'interrogazione fa uso di determinati costruttori, altri possono essere estratti direttamente dalla base di dati.

5.6 Espressioni di percorso

L'accesso alla base di dati può avvenire tramite il nome degli oggetti. In generale emerge però la necessità di navigare attraverso la gerarchia delle classi nello schema, per raggiungere altri oggetti che costituiscono l'obiettivo dell'interrogazione.
OQL prevede quindi la notazione “." (o indifferentemente “->") per accedere alle proprietà di oggetti complessi e per attraversare semplici associazioni.

Ad esempio, la query seguente parte dalla variabile P di tipo Persona, accede, all'interno del complesso attributo Indirizzo di cui ottiene la città (non facciamo riferimento all'esempio presente nel capitolo Applicazione):

Esempio:

P.Indirizzo.Città

Questo esempio presenta una relazione di tipo 1-1 tra Persona e Indirizzo. Occore sottolineare che in OQL un'espressione di percorso può essere usata solo per associazioni di molteplicità 1-1 oppure m-1, poichè per le associazioni di tipo 1-m oppure m-n la notazione fornita sarebbe ambigua.
Se, ad esempio, l'interrogazione P.Figli.Nome fosse ammessa, il risultato della query sarebbe indefinito dato che P.Figli rappresenta una collezione. Possono però presentarsi dei casi meno intuitivi, in cui non è chiaro a quali elementi del percorso è dovuta la molteplicità del risultato e per i quali è necessaria una notazione non ambigua. Per navigare attraverso associazioni multiple di questo tipo OQL estende, rispetto all'SQL relazionale, la sintassi della clausola from nel costrutto select-from-where.

Esempio

```
select C.Nome
from P.Figli as C
```

Per navigare attraverso percorsi che fanno uso di più attributi multivalore OQL consente di specificare nella parte from dell'operatore select-from--where più collezioni, ognuna delle quali è rappresentata da un cammino che deriva da quelli che lo precedono nella lista dichiarativa.

Esempio

```
select C.Indirizzo
from Persona as P,
        P.Figli as C
```

5.7 Predicati

La parte where dell'operatore select-from-where, definendo una serie di predicati booleani, permette di effettuare una restrizione degli oggetti appartenenti alle collezioni interrogate. Vengono quindi selezionati solo i dati che soddisfano l'espressione specificata. L'interrogazione seguente, ad esempio, seleziona tra tutti i clienti soltanto quelli con posizione "media" e ne riporta il nome:

```
Select c.Nome
from Cliente as c
where c.Posizione = “media”
```

5.8 Join

In OQL è possibile formulare interrogazioni che riguardano più classi indirettamente correlate da un cammino, effettuando una operazione di join in maniera simile a quanto avviene in SQL. Questo esempio seleziona le persone che hanno lo stesso nome di un fiore, assumendo che esiste un set di tutti i fiori chiamato Fiori.

Esempio

```
select P
from Persona as P,
Fiori as F
where P.name = F.name
```

5.9 OQL / SQL

OQL è un linguaggio puramente funzionale, in cui ogni operatore viene visto come una interrogazione che restituisce un risultato di un determinato tipo.
Per questo motivo tutti gli operatori possono essere composti in qualunque modo purchè venga sempre rispettato il sistema dei tipi. OQL è quindi, a differenza dell'SQL relazionale, un linguaggio completamente ortogonale.
Questa caratteristica permette di non limitare la potenza delle espressioni e rende il linguaggio più semplice da apprendere, poichè mantiene la stessa sintassi di SQL per le interrogazioni più semplici. Le espressioni SQL che non rientrano in una categoria puramente funzionale, e che quindi non rientrano nella filosofia di OQL, sono viste come variazioni sintattiche dei costrutti OQL equivalenti.

Capitolo VI

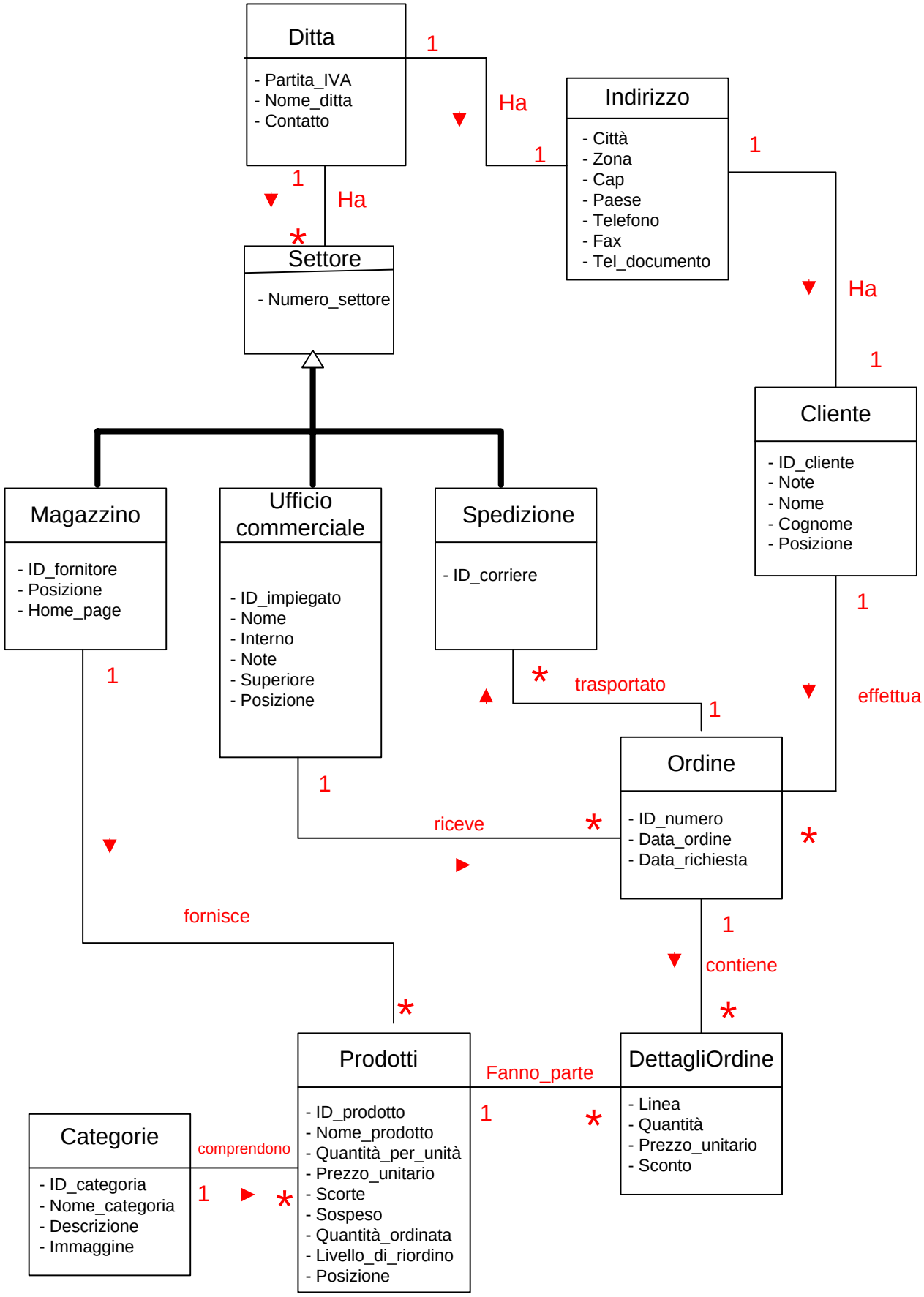
Applicazione

6.1 Introduzione

In questo capitolo ho elaborato un esempio in cui:

- Ho costruito un modello logico di un DBMS orientato agli oggetti ossia un diagramma delle classi che ho chiamato ORDINI.
- Mediante il linguaggio ODL ho validato lo schema per la mia applicazione, ODL è uno specifico linguaggio usato per definire le specificazioni di tipi di oggetti conformi all' ODMG Object Model.
- Ho costruito un modello logico per XML organizzando i dati in modo gerarchico (albero), quindi ho cercato di rappresentare il diagramma delle classi di cui al punto precedente mediante una struttura gerarchica.
- Mediante il DTD ho descritto la struttura del file XML come avevo in precedenza fatto con l'ODL.
- Ho costruito un documento XML basato sul DTD definito precedentemente in cui ho inserito dei dati. Lo scopo è quello di far vedere come è formato un documento XML ed evidenziare il legame che esiste con il suo DTD.
- Ho effettuato delle query in OQL e le ho tradotte in XQUERY basandomi sui rispettivi modelli logici e cercando di produrre gli stessi risultati.
Ho diviso le Query in quattro settori: predicati semplici, predicati di join, interrogazioni innestate e infine funzioni aggregate e raggruppamenti.
- Ho enunciato le regole di traduzione per passare da un query in OQL in una query in XQUERY e viceversa, basandomi sui rispettivi BNF in cui viene descritta la grammatica.

ORDINI Database



6.3 Classi

Il sistema Ordini Database descrive la gestione degli Ordini di una determinata Ditta. Tale Ditta ha tre settori: Ufficio Commerciale, Magazzino e Spedizione.
Descriviamo brevemente le classi facenti parte di questo sistema:

- Classe **Ditta** descrive le generalità della ditta che si occupa della gestione degli Ordini.

- Classe **Settore** descrive i settori che compongono la ditta, ha tre sottoclassi quindi sono previsti tre settori.

- Classe **Ufficio Commerciale** è una sottoclasse della classe *Settore*, si occupa di ricevere gli *Ordini* effettuati dai *Client*i.

- Classe **Magazzino** è una sottoclasse della classe *Settore*, fornisce i Prodotti richiesti dal *Cliente* e contenuti *nell'Ordine*.

- Classe **Spedizione** è una sottoclasse della classe *Settore*, si occupa di trasportare i *Prodotti* contenuti *nell'Ordine al Cliente* .

- Classe **Cliente** descrive le generalità dei Clienti della *Ditta*.

- Classe **Indirizzo** contiene gli indirizzi dei *Clienti* della Ditta e della *Ditta* stessa.

- Classe **Ordine** contiene gli elementi dell'Ordine effettuato dal *Cliente*, e ricevuto dal *Ufficio Commerciale* e trasportato dal *Settore Spedizione*.

- Classe **Dettagli Ordine** rappresenta le linnee dell'*Ordine.*

- Classe **Prodotti** contiene le informazioni sui prodotti che compongono l'*Ordine* e che la *Ditta* tramite il suo *Settore Magazzino* fornisce.

- Classe **Categorie** contiene informazioni ulteriori sui *Prodotti* come ad esempio la loro classificazione all'interno della *Ditta*.

6.4 Linguaggio ODL

```
 interface Indirizzo
(extent indirizzi
key (Telefono))
     {
    attribute integer Telefono;
    attribute string Citta;
    attribute string Zona;
    attribute unsigned short Telefono;
    attribute unsigned short Tel_domicilio;
    attribute string Nazione;
    attribute integer Cap;
    attribute unsigned short Fax;
    relationship Ditta e_della
        inverse Ditta::ha;
    relationship Cliente e_del
        inverse Cliente::ha;

    };

interface Ditta
(extent ditte
key (Partita_IVA))
{
    attribute integer Partita_IVA;
    attribute string Nome_ditta;
    attribute string Contatto;
    attribute set<Settore> contiene;
    relationship Indirizzo ha
       inverse Indirizzo::e_della;


      };

interface Settore
(extent settori
key(ID_settore))
{
    attribute integer ID_settore;
};
```

```
interface Spedizione: Settore
(extent spedizioni
key(ID_corriere))
{
    attribute integer ID_corriere;
};

interface Ordine
(extent ordini
key(ID_numero))
{
    attribute integer ID_numero;
    attribute integer Data_ordine;
    attribute integer Data_richiesta;
    attribute set<Spedizione> trasportato;
    attribute set<DettagliOrdine> contiene;

};

interface DettagliOrdine
(extent dettagliOrdini
key (Linea))
{
    attribute integer Linea;
    attribute integer Quantita;
    attribute integer Sconto;
    attribute real Prezzo_unitario;

};

interface Prodotto
(extent prodotti
key(ID_prodotti))
{
    attribute integer ID_prodotto;
    attribute string Nome_prodotto;
    attribute integer Quantita_per_unita;
    attribute real Prezzo_unitario;
    attribute string Scorte;
    attribute string Sospeso;
    attribute integer Quantita_ordinata;
    attribute string Livello_di_riordino;
    attribute set<DettagliOrdine> fanno_parte;
};
```

```
interface Magazzino: Settore
(extent  magazzini
key (ID_fornitore))
{
    attribute integer ID_fornitore;
    attribute string Posizione;
    attribute string Home_page;
    attribute set<Prodotto> fornisce;
};


interface Commerciale: Settore
(extent Commerciali
key(ID_ufficio))
{
    attribute integer ID_ufficio;
    attribute string Note;
    attribute string Superiore;
    attribute string Posizione;
    attribute integer Interno;
    attribute set<Ordine> riceve;
};


interface Cliente
(extent clienti
key (ID_cliente))
{
     attribute integer ID_cliente;
     attribute string Cognome;
     attribute string Nome;
     attribute string Posizione;
     attribute string Note;
     attribute string sesso;
     attribute set<Ordine> effettua;
     relationship Indirizzo ha
          inverse Indirizzo::e_del;
};


interface Categoria
(extent categorie
key (ID_categoria))
```

```
{
    attribute integer ID_categoria;
    attribute string Nome_categoria;
    attribute string Descrizione;
    attribute string Immaggine;
    attribute set<Prodotto> comprendono;
};
```

6.5 Specifiche Albero XML

Ho rappresentato il database ordini tramite l'albero XML in cui ho utilizzato le seguenti notazioni simboliche:

- /Ordini — ho indicato la radice;
- Elemento - mixed — ho indicato gli elementi mixed (che contengono sia testo che altri elementi)
-  Elemento - semplice — ho indicato gli elementi semplici (che contengono solo testo)

Nel passaggio all'albero di XML dal Diagramma delle Classi ho effettuato le seguenti scelte:

♦ La relazione di **Part of** nel Diagramma delle Classi è diventata una **relazione tra Padre e Figlio** nell'albero di XML.

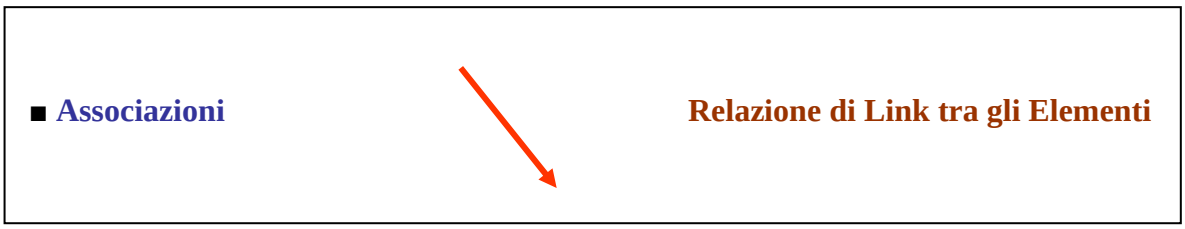
♦La relazione di **Generalizzazione** nel Diagramma delle Classi è stata tradotta come una **relazione di Link** tra gli elementi dell'albero di XML (IDREF è il valore ID di un altro elemento).

♦Le **Associazioni** nel Diagramma delle Classi sono diventate **relazione di Link** tra gli elementi dell'albero di XML.

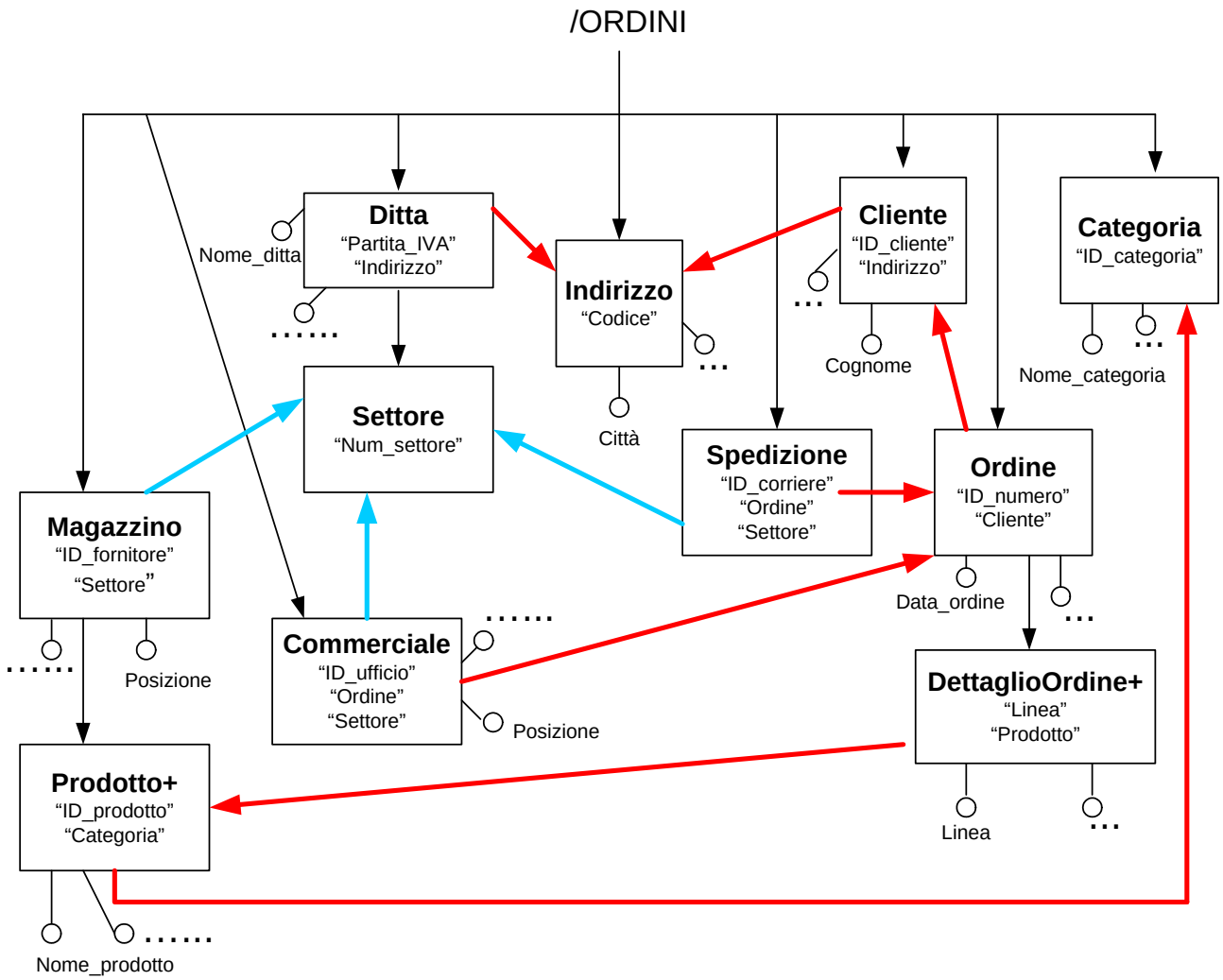
Simbolicamente Abbiamo:

■ Relazione di **Part of**  **Relazione tra Padre e Figlio**

■ Relazione di **Generalizzazione**  **Relazione di Link tra gli Elementi**



6.6 Albero XML



6.7 DTD

<!DOCTYPE ORDINI [
<!ELEMENT ORDINI (**Ditta, Cliente, Indirizzo+, Ordine, Categoria+, Magazzino, Commerciale, Spedizione**)>
<!ELEMENT **Ditta** (Settore+, Nome_ditta, Contatto)>
<!ELEMENT Nome_ditta (#PCDATA)>
<!ELEMENT Contatto (#PCDATA)>
<!ELEMENT Settore(Empty)>
<!ELEMENT **Magazzino** (Posizione, Home_page, Prodotto+)>
<!ELEMENT Posizione (#PCDATA)>
<!ELEMENT Home_page (#PCDATA)>
<!ELEMENT Prodotto (Nome_prodotto, Quantità_per_unità, Prezo_unitario, Scorte, Sospeso, Quantità_ordinata, Livello_di_riordino, Posizione)>
<!ELEMENT Nome_prodotto (#PCDATA)>
<!ELEMENT Quantità_per_unità (#PCDATA)>
<!ELEMENT Prezzo_unitario (#PCDATA)>
<!ELEMENT Scorte (#PCDATA)>
<!ELEMENT Sospeso (#PCDATA)>
<!ELEMENT Quantità_ordinata (#PCDATA)>
<!ELEMENT Livello_di_riordino (#PCDATA)>
<!ELEMENT Posizione (#PCDATA)>
<!ELEMENT **Commerciale** (Interno, Note, Superiore, Posizione)>
<!ELEMENT Interno (#PCDATA)>
<!ELEMENT Note (#PCDATA)>
<!ELEMENT Superiore (#PCDATA)>
<!ELEMENT Posizione (#PCDATA)>
<!ELEMENT **Spedizione**(Empty)>
<!ELEMENT **Cliente** (Note, Nome, Cognome, Posizione)>
<!ELEMENT Note (#PCDATA)>
<!ELEMENT Nome (#PCDATA)>
<!ELEMENT Cognome (#PCDATA)>
<!ELEMENT Posizione (#PCDATA)>
<!ELEMENT **Indirizzo** (Citta, Zona, Cap, Paese, Telefono, Fax, Tel_domicilio)>
<!ELEMENT Citta (#PCDATA)>
<!ELEMENT Zona (#PCDATA)>
<!ELEMENT Cap (#PCDATA)>
<!ELEMENT Paese (#PCDATA)>
<!ELEMENT Telefono (#PCDATA)>
<!ELEMENT Fax (#PCDATA)>
<!ELEMENT Tel_domicilio (#PCDATA)>
<!ELEMENT **Ordine** (Data_ordine, Data_richiesta, DettegliOrdine +)>
<!ELEMENT Data_ordine (#PCDATA)>

```
<!ELEMENT Data_richiesta (#PCDATA)>
<!ELEMENT DettagliOrdine (Quantità, Prezzo_unitario, Sconto)>
<!ELEMENT Quantità (#PCDATA)>
<!ELEMENT Prezzo_unitario (#PCDATA)>
<!ELEMENT Sconto (#PCDATA)>
<!ELEMENT Categoria (Nome_categoria, Descrizione, Immaggine)>
<!ELEMENT Nome_categoria (#PCDATA)>
<!ELEMENT Descrizione (#PCDATA)>
<!ELEMENT Immaggine (#PCDATA)>


<!ATTLIST   Indirizzo   Codice   ID   #REQUEST>

<!ATTLIST   Settore   Num_settore   ID   #REQUEST>


<!ATTLIST  Ordine
  ID_numero     ID          #REQUEST
  Cliente          IDREF    #REQUEST>

<!ATTLIST  DettaglioOrdine
Linea             ID          #REQUEST
 Prodotto        IDREF     #REQUEST>

<!ATTLIST  Categoria   ID_categoria   ID   #REQUEST>

<!ATTLIST  Ditta
Partita_IVA     ID        #REQUEST
Indirizzo         IDREF  #REQUEST>

<!ATTLIST Prodotto
ID_prodotto    ID          #REQUEST
Categoria       IDREF      #REQUEST>

<!ATTLIST Cliente
ID_cliente      ID          #REQUEST
Indirizzo       IDREF      #REQUEST>

<!ATTLIST  Magazzino
ID_fornitore     ID    #REQUEST
Settore       IDREF       #REQUEST >
```

<!ATTLIST Commerciale
ID_ufficio ID #REQUEST
Settore IDREF #REQUEST
Ordine IDREF #REQUEST>

<!ATTLIST Spedizione
ID_corriere ID #REQUEST
Settore IDREF #REQUEST
Ordine IDREF #REQUEST>

6.8 Esempio XML

<Ordini>
<Ditta Indirizzo=”111” Partita_IVA=”2654499”>
<!—Commento: Ditta ha come attributo il valore della partita IVA e come IDREF Indirizzo→
<Nome_ditta> Stanguellini </Nome_ditta>
<Contatto> nessuno </Contatto>
<Settore Num_settore = ‘1’/>
<Settore Num_settore = ‘2’/>
<Settore Num_settore = ‘3’/>
</Ditta>
<Magazzino ID_fornitore=”11” Settore=”1”>
<!—Commento GENERALIZZAZIONE: Magazzino ha come attributo IDREF il valore del Num_settore di Settore oltre al suo ID→
<Posizione> centrale </Posizione>
<Home_page> lijfhuytn </Home_page>
<Prodotto ID_prodotto =”222” Categoria=”02>
<!—Commento: Prodotto ha come attributo IDREF il valore dell’ID dell’elemento Categoria a cui appartiene oltre al suo ID→
<Nome_prodotto> Spazzola </Nome_prodotto>
<Quantità_per_unità> 5 <Quantità_per_unità>
<Prezzo_unitario> 10 <Prezzo_unitario>
<Scorte> nessuno </Scorte>
<Sospeso> no </Sospeso>
<Quantità_ordinata> 20 <Quantità_ordinata>
<Livello_di_riordino> buono <Livello_di_riordino>
<Posizione> centrale </Posizione>
</Prodotto>
<Prodotto ID_prodotto =”333” Categoria=”02”>
<Nome_prodotto> Pettin e</Nome_prodotto>
<Quantità_per_unità> 8 <Quantità_per_unità>
<Prezzo_unitario> 5 <Prezzo_unitario>

<Scorte> 4 </Scorte>
<Sospeso> n o</Sospeso>
<Quantità_ordinata> 4 <Quantità_ordinata>
<Livello_di_riordino> buono <Livello_di_riordino>
<Posizione> alta </Posizione>
</Prodotto>
</Magazzino>
<Commerciale ID_ufficio=”36” Ordine=”8888” Settore=”2”>
<!—Commento GENERALIZZAZIONE E LINK a ORDINE: Commerciale ha come attributo IDREF il valore del Num_settore di Settore e il valore di ORDINE oltre al suo ID→
<Interno> 5 </Interno>
<Note> kjyytg </Note>
<Superiore> </Superiore>
<Posizione> addetto al rapporto con i clienti </Posizione>
</Commerciale>
<Spedizione ID_corriere=”7777” Ordine=”8888” Settore=”3”>
<!—Commento GENERALIZZAZIONE E LINK a ORDINE: Spedizione ha come attributo IDREF il valore del Num_settore di Settore e il valore di ORDINE oltre al suo ID→
</Spedizione>
<Cliente ID_cliente=”356464” Indirizzo=”222”>
<!—Commento: Cliente ha come attributo il suo ID e come IDREF Indirizzo→
<ID_cliente> 356464 </ID_cliente>
<Note> nessuna </Note>
<Nome> Rita </Nome>
<Cognome> Rossi </Cognome>
<Posizione> medi a</Posizione>
</Cliente>
<Indirizzo Codice=”111>
<Telefono> 2599875 </Telefono>
<Città> Milano </Città>
<Zona> Altri </Zona>
<Cap> 41256 </Cap>
<Paese> Italia </Paese>
<Fax> 256897469 </Fax>
<Tel_domicilio> 0259716598 </Tel_domicilio>
</Indirizzo>
<Indirizzo Codice=”222”>
<Telefono> 2356984 </Telefono>
<Città> Parma </Città>
<Zona> non specificata </Zona>
<Cap> 4 1256 </Cap>
<Paese> Italia </Paese>

```
            <Fax> 222222 </Fax>
             <Tel_domicilio> 26444444 </Tel_domicilio>
        </Indirizzo>
      <Ordine ID_numero=”8888”  Cliente=”356464” >
    <!—Commento: Ordine ha come atributo il suo ID e come IDREF Cliente
     (ID_cliente)→
        <Data_ordine> 23/10/2003 </Data_ordine>
        <Data_richiesta> 19/09/2003 </Data_richiesta>
        <DettagliOrdine Linea=”1”  Prodotto=”222”>
          <Quantità> 2 </Quantità>
          <Prezzo_unitario> 10 </Prezzo_unitario>
          <Sconto> 1 </Sconto>
        </DettagliOrdine>
           <DettagliOrdine Linea=”2” Prodotto=”333”  >
          <Quantità> 3 </Quantità>
          <Prezzo_unitario> 5 </Prezzo_unitario>
          <Sconto> 2 </Sconto>
        </DettagliOrdine>
      </Ordine>
      <Categoria ID_categoria=”02”>
        <Nome_categoria> Cv </Nome_categoria>
         <Descrizione> nessuna </Descrizione>
      <Immagine> cerchio </Immagine>
      </Categoria>
</Ordini>
```

6.9 Query

Iniziamo ad interrogare con delle query rispettivamente in OQL e in XQUERY i nostri database.
Riporterò due esempi per ciascuna delle seguenti categorie:

6.9.1 Predicati semplici

Partiamo dalle query più semplici utilizzando due dei seguenti operatori relazionali { = , <>, <, >, >=, <=}. In OQL interrogheremo una singola classe, in XQUERY un singolo elemento mixed.

Esempi:

1) Seleziona tra tutte le Ditte soltanto quella con Partita IVA "265449" e ne riporta il nome:

OQL:

```
select   d.Nome_ditta
from     Ditta as d
where    d.Partita_IVA = "265449"
```

XQUERY:

```
for      $x in dcument("Ordini.xml")//Ditta
where    $x/@Partita_IVA = "265449"
return
   <Ditta>
      {
        $x/Nome_ditta
      }
```

2) Seleziona tra tutti i Prodotti soltanto quelli con quantità per unità maggiore o uguale a zero e al più uguale a cinque e ne riporta il prezzo unitario:

OQL:

```
select   p.Prezzo_unitario
from     Prodotto as p
where    p.Quantità_per_unità >= 0
and      p.Quantità_per_unità <= 5
```

XQUERY:

```
for      $y in document("Ordini.xml")//Prodotto
where    $y/Quantità_per_unità >= 0
and      $y/Quantità_per_unità <= 5
return   <Prodotti>
         {$y/Prezzo_unitario}
         </Prodotti>
```

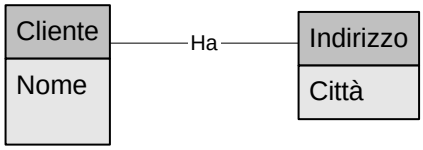
6.9.2 Predicati di Join

E' un importante tipo di predicato in quanto combina dati da multiple sorgenti in un singolo risultato.

Esempi:

3) Seleziona il cognome e la città di provenienza di tutti i clienti:

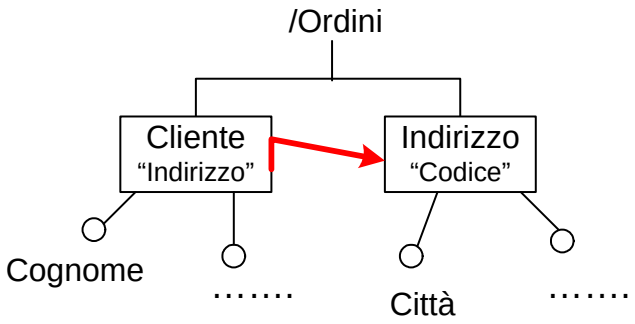
OQL:



c.Ha ⟶ Indirizzo

```
Select   c.Cognome, i.Città
from     Cliente as c,
         c.Ha as i
```

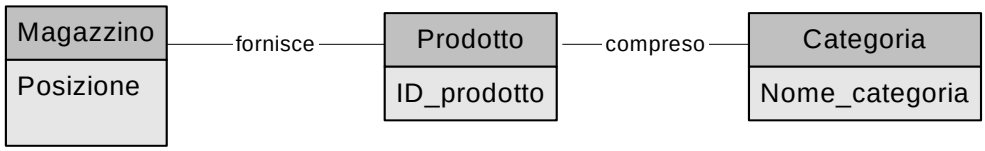
XQUERY:




```
for   $x in document("Ordini.xml")
      $b in $x/Cliente
      $a in $x/Indirizzo[@Codice = $b/@Indirizzo]
return  <Informazioni>
          {
              $a/Cognome,
              $b/Città
              }
        </Informazioni>
```

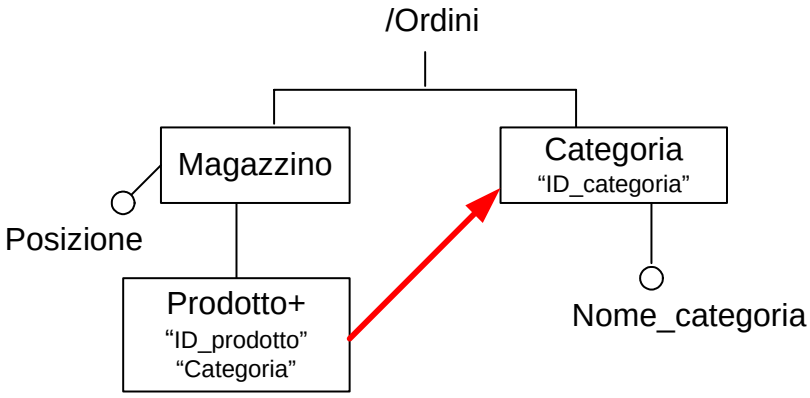
4) Seleziona il codice prodotto e il nome della sua categoria dei Magazzini con l'attributo posizione uguale a "centrale":

OQL:




```
Select  p.ID_prodotto,  p.compreso.Nome_categoria
from    Magazzino as m
        m.fornisce as p
where   m.Posizione ="centrale"
```

XQUERY:



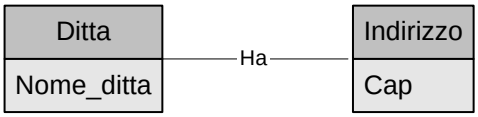

```
for     $x in document("Ordini.xml")
        $c in $x/Magazzino
        $p in $c/Prodotto
        $y in $x/Categoria[@ID_categoria = $p/@Categoria]
where   $c/Posizione = "centrale"
return   <Legame>
          {
             $p/ID_prodotto,
             $y/Nome_categoria
           }
         </Legame>
```

6.9.3 Iterrogazioni Innestate

Nella prima QUERY in OQL ho utilizzato il quatificatore esistenziale EXISTS invece nella seconda QUERY in OQL ho utilizzato l'operatore quantificato ALL insieme all'operatore di set IN. Il significato di questi costrutti verrà spiegato successivamente nel paragrafo *Regole di Traduzione*. Ho cercato di riprodurre delle QUERY analoghe in XQUERY.

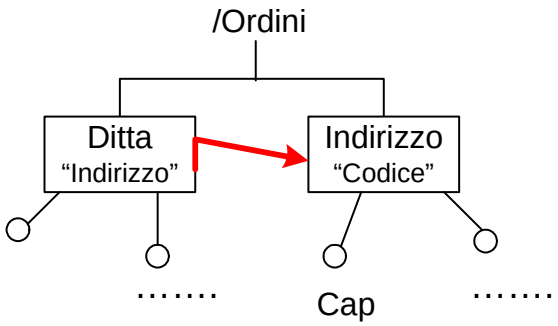
5) Selezionare il nome della ditta dove esiste l'attributo CAP uguale a " 41013 " e ordinare il risultato in base al nome della ditta:

OQL:




```
Select    d.Nome_ditta
from      Ditta as d
where EXISTS s in d.Ha : s.Cap = " 41013 "
Order by d.Nome_ditta
```

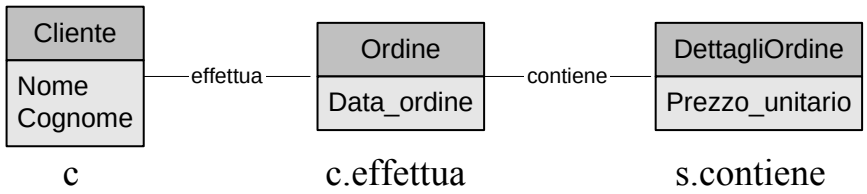
XQUERY:




```
for   $a in document("Ordini.xml")//Ditta
let    $p := document("Ordini.xml")//Indirizzo[@Codice =
$a/@Indirizzo AND Cap = "41013 "]
Order by  $a/Nome_ditta
return
      if (exists($p)) then
                  <Risultato>
                {
                 $a/Nome_ditta
                 }
                  </Risultato>
    else()
```

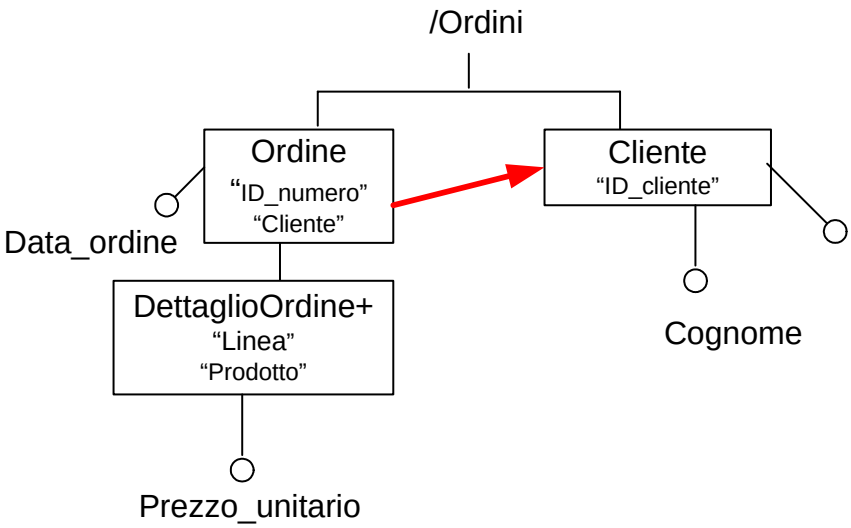
6) Selezionare i nomi e cognomi dei cliente dove il prezzo unitario è uguale a 10 per tutti gli ordini di data “10/12/2003”, ordinare il risultato in base al cognome del cliente:

OQL:




```
Select    c.Cognome, c.Nome
from      cliente as c
where FOR ALL  be  IN (select s.contiene  from c.effettua as s
where s.Data_ordine = “10/12/2003”) : be.Prezzo_unitario = 10
Order by c.Cognome
```

XQUERY:



```
for    $a in document("Ordini.xml")//Ordine[Data_ordine = "10/12/2003"]
       $b in document("Ordini.xml")//Cliente[@ID_cliente  = $a/@Cliente]
let    $p := $a/DettaglioOrdine[Prezzo_unitario = "10"]
Order by  $b/Cognome
return
          if (exists($p)) then
                    <Risultato>
                     {
                      $b/Nome,
                      $b/Cognome
                      }
                    </Risultato>
        else()
```

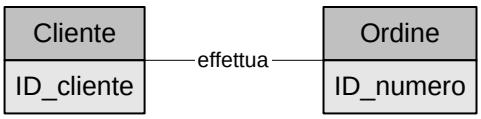
6.9.4 Funzioni Aggregate e Raggruppamenti

Utilizziamo una serie di funzioni per elaborare i valori di un attributo (MAX, MIN, AVG, SUM) e per contare le tuple che soddisfano una condizione (COUNT). In una istruzione SELECT è possibile formare dei gruppi di tuple che hanno lo stesso valore di specificati attributi, tramite la clausola GROUP BY. Invece l'istruzione FLWR utilizza l'istruzione let per legare variabili ad insieme.

Esempio:

7) Questa query trova l'ID_cliente e l'ID_numero degli Ordini che hanno almeno tre Cliente:

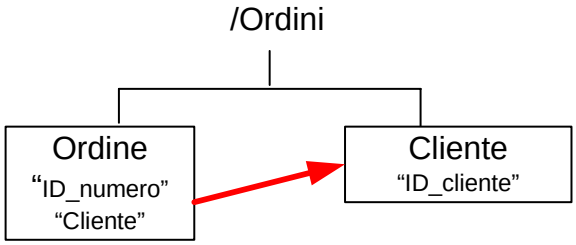
OQL:



c c.effettua as o

```
Select distinct   o.ID_numero, c.ID_cliente
from          Cliente as c
              c.effettua as o
group by   c.ID_cliente
having count(*) > 3
```

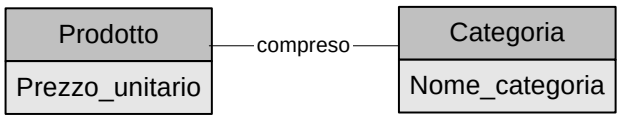
XQUERY:




```
for $pn in distinct-values(document("Ordini.xml")//Ordine)
let $i := document("Ordini.xml")//Cliente[@ID_cliente = $pn/@Cliente]
where count($i) >= 3
order by $pn
return
      <Risultato>
         {$pn/@ID_numero,
          $i/@ID_cliente
          }
      </Risultato>
```

8) Questa query trova la media dei Prezzi_unitari dei Prodotti che fanno parte della Categoria "CV":

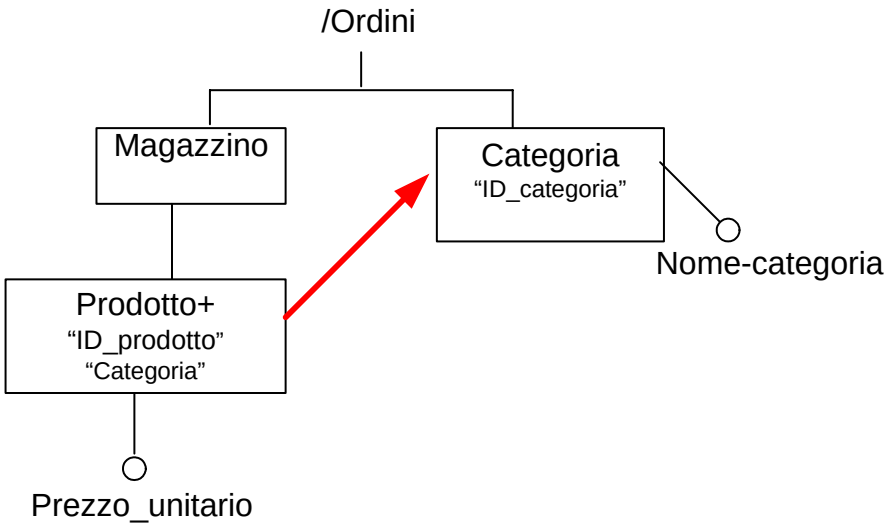
OQL:



p p.compreso as c

```
Select  avg(p.Prezzo_unitario)
from   Prodotto as p
         p.compreso as c
where  c.Nome_categoria ="CV"
```

XQUERY:




```
for $z in document("Ordini.xml")//Categoria
let $i := document("Ordini.xml")//Magazzino/Prodotto[@Categoria=$z/ID_ca-
tegoria]
where $z/Nome_categoria = "CV"
return
      <Risultato>
        <Media>    {avg($i/Prezzo_unitario)}    <Media>
      <Risultato>
```

CAPITOLO VII
Regole di Traduzione

In questo paragrafo effettuerò le regole di traduzione da una query in OQL a una query in XML.
Indicherò le classi in OQL e gli elementi- mixed in XQUERY con le lettere dell'alfabeto maiuscole (A as a, B as b….), invece con una combinazione di lettere e numeri indicherò rispettivamente gli attributi e gli elementi-semplici (o1, o2, o3……).

Tradurrò quattro tipologie di query nei seguenti quattro campi:

- ♦Predicati semplici
- ♦Predicati di JOIN
- ♦Funzioni aggregate e Raggruppamenti
- ♦Interrogazioni Innestate

Naturalmente nel caso di OQL considererò come modello logico quello organizzato in Classi e per XQUERY considererò come modello logico un Albero in cui da una root si diramano i vari elementi semplici e mixed (similarmente all'esempio riportato sempre in questa tesi.).

7.1 Predicati semplici

Tradurrò il seguente semplice formato di QUERY che prevede un' interrogazione su una singola classe (in OQL) o singolo elemento-mixed (in XQUERY) utilizzando l'operatore relazionale " = " e il predicato AND.

Formato:

In OQL

```
Select  a.o1
from    A as a
where   a.o2 = "….."
and a.o3 >= 0
```

In XML:

```
for     $a in document("…….")//A
where   $a/o2 = "……."
and     $a/o3 >= 0
retun   <Costruttore>
          { $a/o1}
        <Costruttore>
```

Questa query non prevede l'utilizzo della clausola "let" (assegna l'intero set alla variabile) che in XQUERY principalmente viene utilizzata quando abbiamo funzioni aggregate.

Prima di enunciare le regole di traduzione consideriamo il seguente ESEMPIO:

1) Seleziona il Nome dei prodotti che hanno scorte maggiore o uguale a zero e un Livello_ di_ riordino " buono"

In OQL:

```
select   p.Nome_prodotto
from     Prodotto as p
where    p.Livello_di_riordino = "Buono"
 and     p.Scorte >= 0
```

In XML:

```
for   $a in document("Ordini.xml)//Prodotto
 where     $a/Livello_di_riordino = "Buono"
  and      $a/Scorte >= 0
 return     <Risultato>
          { $a/Nome_prodotto }
            </Risultato>
```

7.2 Regole di traduzione predicati semplici

Select Projection List	return Element Costructor

Projection List → Identifier → char/char string

Path → ./->

Identifier → char/char string

ElementConstructor → "<" Qname AttributeList ("/>" | ">")
ElementContent*
"</"Qname AttributeList ">"

from Identifier as Identifier	for $Varname in ExprSingle

Le espressiono "for" possono essere annidate.

Identifier as Identifier → char/char string, char/char string

Varname → Qname

ExprSingle → document (Expr) // Relative path Expr → StepExpr (("/" | "//") Step Expr)*

where Accessor ComparisonOperator StringLiteral	where Expr

Accessor → Identifier path Identifier

ComparisonOperator → "="

StringLiteral → "String"

Expr → $Varname/Relative path Expr GeneralComp "String"

GeneralComp → "="

and Accessor ComparisonOperator IntegerLiteral	and Expr

Accessor → Identifier path Identifier

ComparisonOperator → " >= "

ItegerLiteral → UnsignedLong → Digit → 0 | 1 | ….. | 9

Expr → $Varname/Relative path Expr GeneralComp IntegerLiteral

GeneralComp → " >="

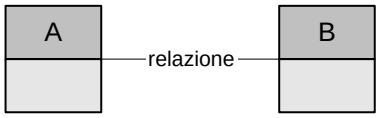
IntegerLiteral → Digits → [0 – 9]+

7.3 Predicati di JOIN

Il seguente formato di QUERY prevede un' interrogazione su più classi (in OQL) o più elementi-mixed (in XQUERY) e ORDINA il risultato in base al valore di un attributo (in OQL) o di un elemento semplice (in XQUERY).
In OQL con il termine *relazione* indico la relazione esistente tra due classi.

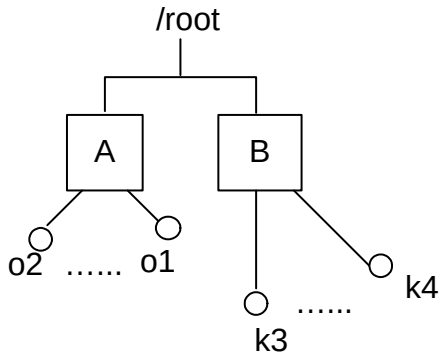
ESEMPIO:

OQL:



a.relazione → B
corrisponde

In XQUERY:



Per il Join abbiamo ipotizzato che l'elemento semplice o2 di A contenga la stessa tipologia di dati dell'elemento semplice k3 di B.

Formato:

In OQL

```
Select  a.o1, b.K4
from    A as a
        a.relazione as b
order by a.o1
```

In XML:

```
for         $a in document("……")//A
            $b in  document("……")//B[ k3 = $a/o2 ]
order by    $a/o1
retun     <Costruttore>
              { $a/o1,
                $b/k4 }
           </Costruttore>
```

Questa query non prevede l'utilizzo della clausola "let".

Prima di enunciare le regole di traduzione consideriamo il seguente ESEMPIO:

2) Seleziona il Nome della categoria e il nome dei prodotti in essa contenuti, ordinando risultato per Nome_categoria.

In OQL:

```
select    a.Nome_categoria, p.Nome_prodotto
from      Categoria as c
          c.comprendono as p
order by  a.Nome_categoria
```

In XML:

```
for       $a in document("Ordini.xml)//Categoria
          $b in document("Ordini.xml)//Prodotto[Categoria = $a/ID_categoria]
order by     $a/Nome_categoria
return     <Risultato>
             { $a/Nome_categoria
               $b/Nome_prodotto }
           </Risultato>
```

7.4 Regole di traduzione predicati di JOIN

Select Projection List	return Element Costructor

Projection List+ → Identifier → char/char string

Path → ./->

Identifier → char/char string

ElementConstructor → "<" Qname AttributeList ("/>" | ">")
ElementContent*
"</"Qname AttributeList ">"

from (Identifier as Identifier)+	for $Varname in ExprSingle

- Prima Espressione for:

(Identifier as Identifier)+

↓ ↓

char/char string char/char string

↓

(a | b |...| A |…| Z).(string)

Varname ExprSingle

↓ ↓

Qname document (Expr) // Relative path Expr

↓

StepExpr ((“/” | “//”) Step Expr)*

• Seconda Epressione for:

Varname ExprSingle

↓ ↓

Qname document (Expr) // Relative path Expr

↓

StepExpr ((“/” | “//”) Step Expr)*

↓

[StringLiteral GeneralComp Expr]

↓ ↓

“=” ↓

$Varname/Relative path Expr

Order by Identifier Path Identifier	order by Expr

Expr

↓

$Varname/Relative path Expr

7.5 Funzioni Aggregate

Questa query prevede l'utilizzo delle funzione aggregata AVG(), anche in questa query abbiamo il JOIN tra più classi (OQL) o tra più elementi mixed (XQUERY), ho riportato esattamente il Join dell'esempio precedente.

Formato:

In OQL

```
Select  a.o1, b.K4, avg(b.k5)
from    A as a
        a.relazione as b
```

In XML:

```
for         $a in document(".....")//A
let         $b :=  document(".....")//B[ k3 = $a/o2 ]

retun     <Costruttore>
    {
        <o1>  $a/o1  </o1>
        <b4>  $b/k4  </b4>
        <media>  avg($b/k5)  </media>
         }
  </Costruttore>
```

In questa query abbiamo utilizzato la clausola "let" (assegna l'intero set alla variabile) che in XQUERY principalmente viene utilizzata con le funzioni aggregate.

Prima di enunciare le regole di traduzione consideriamo il seguente ESEMPIO:

3) Seleziona il Nome della categoria e il nome dei prodotti in essa contenuti con la media dei prezzi unitari.

In OQL:

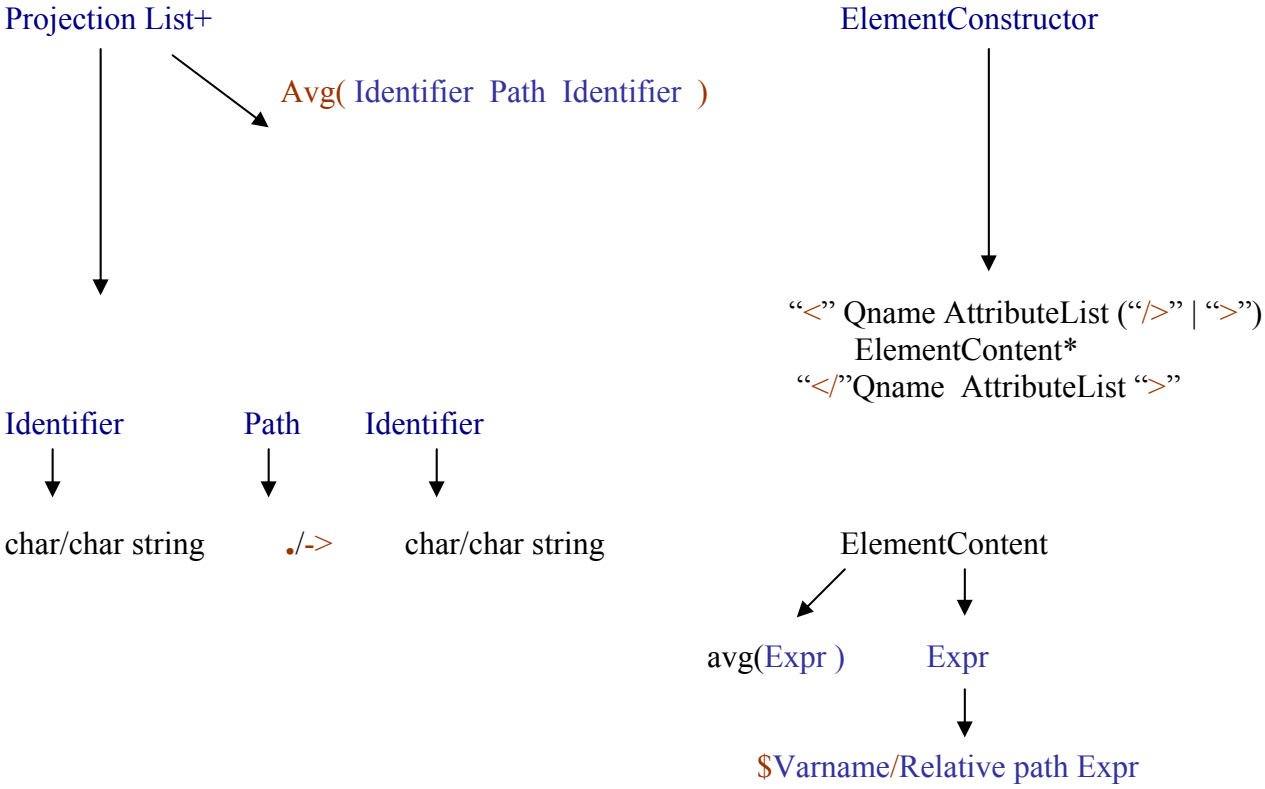
```
select      a.Nome_categoria, p.Nome_prodotto, avg(p.Prezzo_unitario)
from        Categoria as c
            c.comprendono as p
```

In XML:

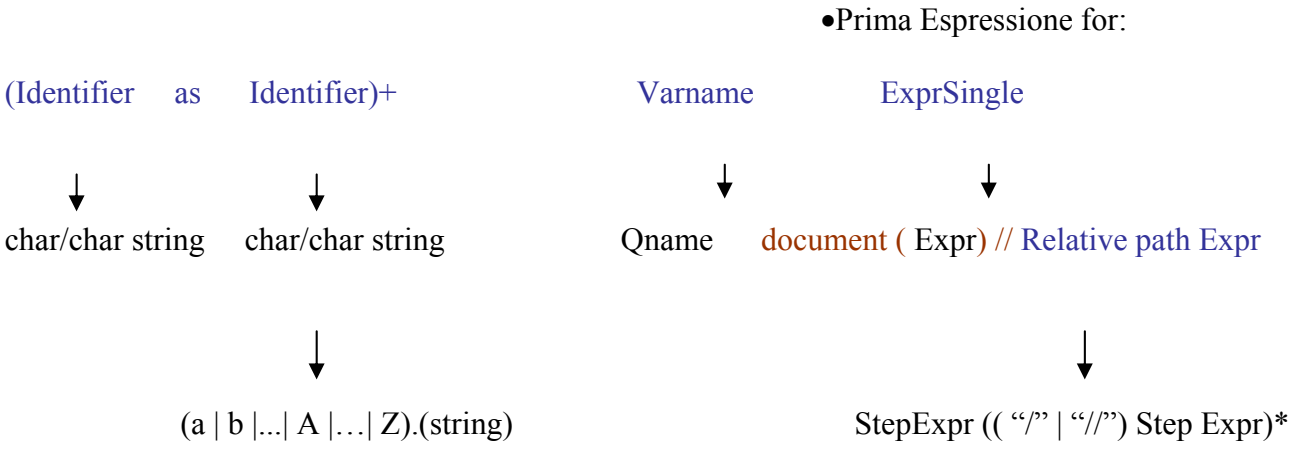
```
for         $a in document("Ordini.xml)//Categoria
let         $b :=  document("Ordini.xml)//Prodotto[Categoria = $a/ID_categoria]
return      <Risultato>
             {
             <Categoria> $a/Nome_categoria </Categoria>
             <Nome>  $b/Nome_prodotto </Nome>
             <media> avg($b/Prezo_unitario) </media>
             }
            </Risultato>
```

7.6 Regole di traduzione funzioni Aggregate

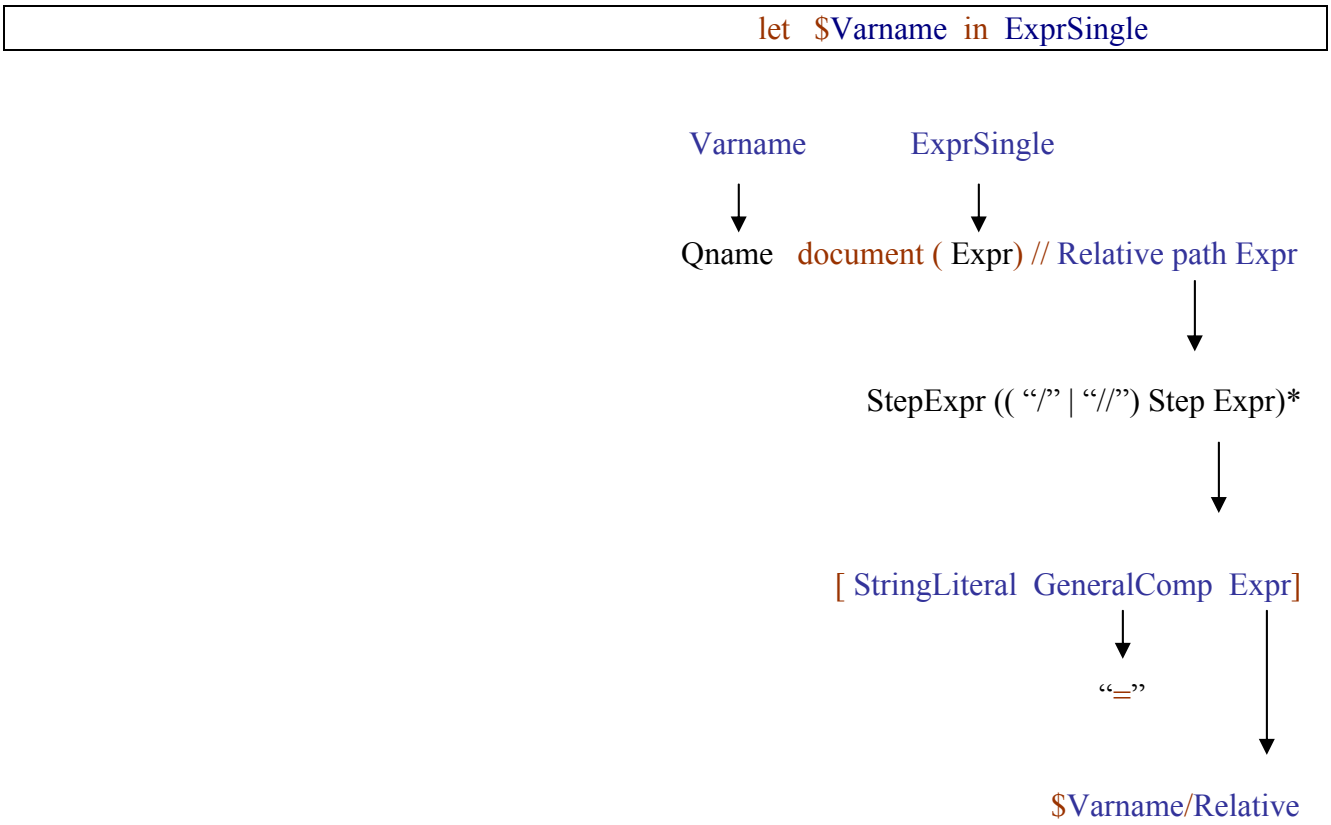
Select Projection List	return Element Costructor



from (Identifier as Identifier)+	for $Varname in ExprSingle



Solo XQL:



7.7 Interrogazioni Innestate

In OQL abbiamo utilizzato il seguente Quantificatore Esistenziale:

exists x in **e1** : **e2** relazione x

Se x è il nome di una variabile, e se **e1** ed **e2** sono espressioni, **e1** denota una collezione ed **e2** è una espressione di tipo booleana, allora esiste x in **e1** : **e2** è una espressione di tipo booleana.
Il risultato di questo predicato è vero se c'è almeno un elemento della collezione **e1** che soddisfa **e2** e falso altrimenti.
Invece con:

for all x in **e1** : **e2** relazione x

Risulta vero se tutti gli elementi della collezione **e1** soddisfano **e2** e falso altrimenti.

Formato:

In OQL:

```
Select  a.o1
from   A as a
where exists  b in a.relazione : b.k4 = "…"
```

In XML:

```
for          $a in document(".....")//A
let          $b :=  document(".....")//B[ k3 = $a/o2 and k4 = "…." ]
             retun
                                  if(exists($b)) then

                                        <Costruttore>
                                           {
                                                $a/o1
                                                }
                                         </Costruttore>

                                  else()
```

Prima di enunciare le regole di traduzione consideriamo il seguente ESEMPIO:

4) Seleziona il Nome delle Ditte che sono situate nella città di " Milano "

In OQL:

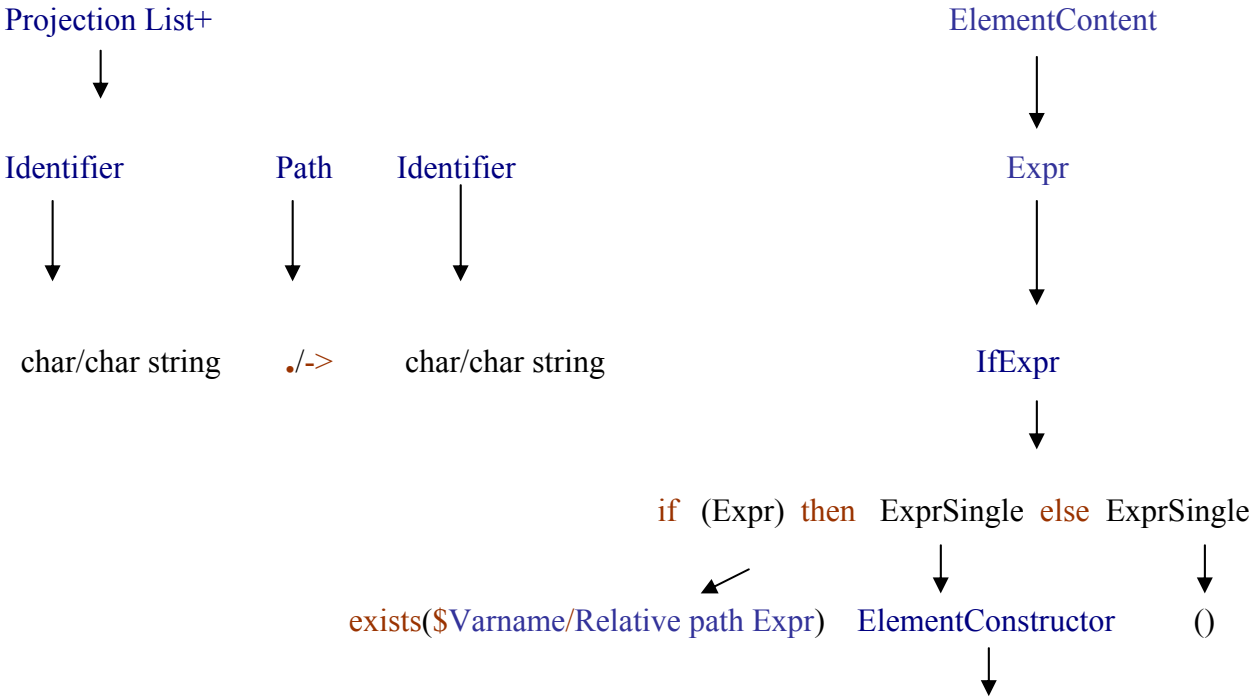
```
Select d.Nome_ditta
from Ditta as d
where EXISTS s in d.Ha : s.Città = " Milano "
```

In XML:

```
for  $a in document("Ordini.xml")//Ditta
let $p := document("Ordini.xml")//Indirizzo[@codice = $a/@indirizzo AND Città =
"Milano"]
return
          if (exists($p)) then
                <Risultato>
                  {
                    $a/Nome_ditta
                   }
                 </Risultato>
          else()
```

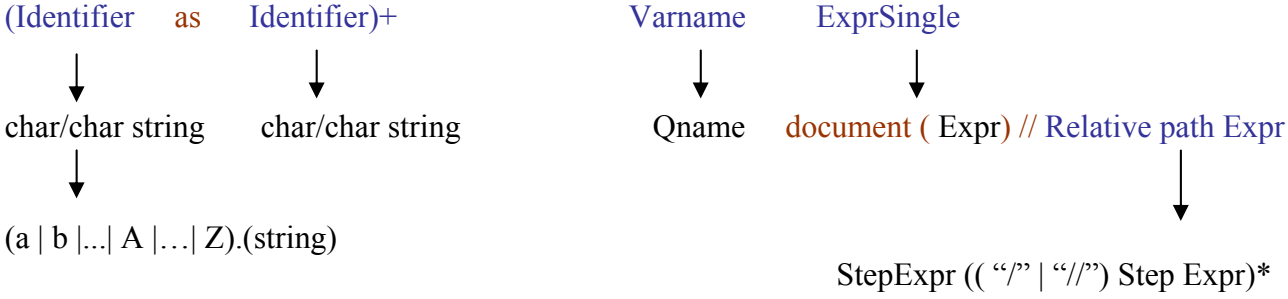
7.8 Regole di traduzione interrogazioni innestate

Select Projection List	return Element Content

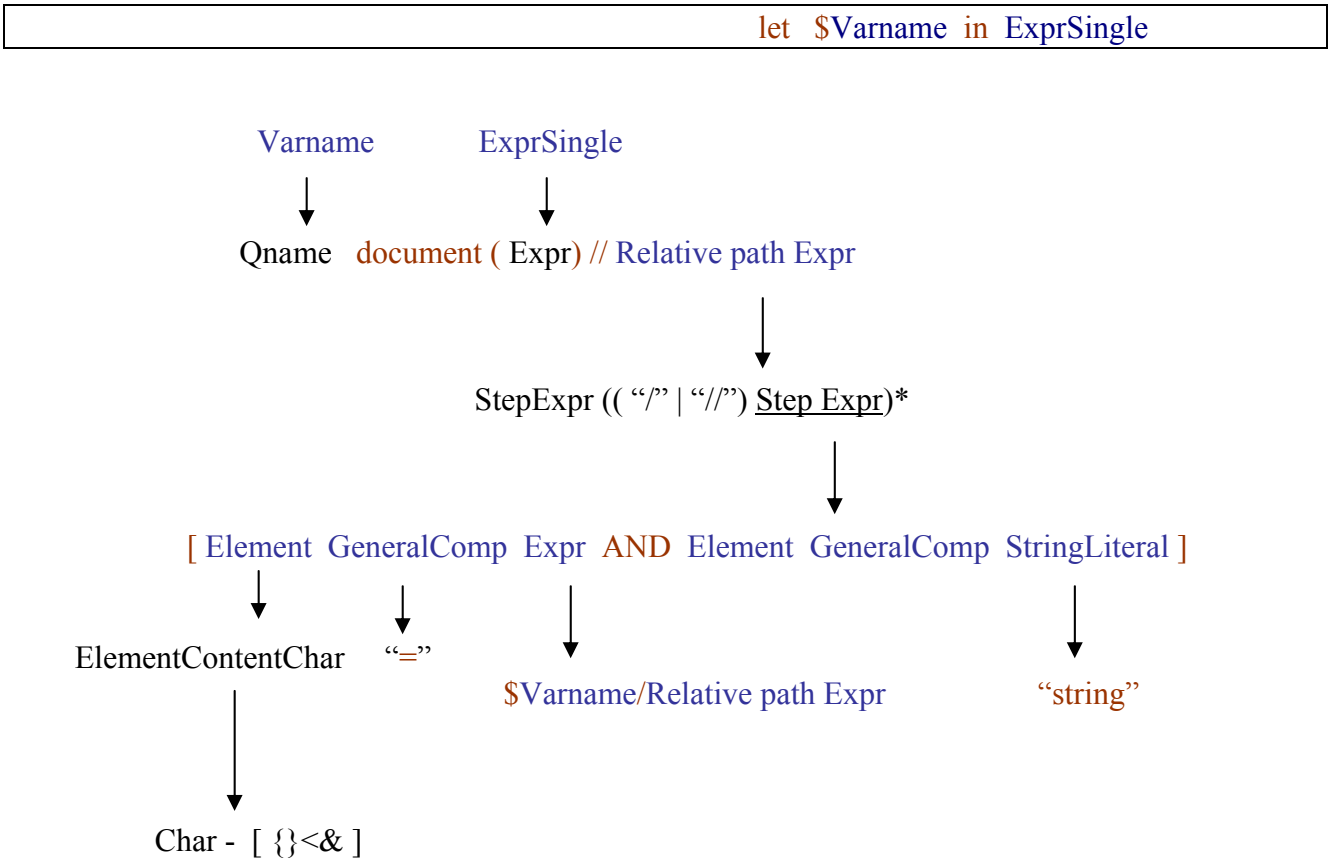


from (Identifier as Identifier)+	for $Varname in ExprSingle

•Prima Espressione for:



Solo XQUERY:



Solo XQL: