Parole chiave:

DBMS
QUERY
SQL92
XML
XQUERY

# INDICE DEGLI ARGOMENTI TRATTATI:


**INTRODUZIONE** 6

**1 - SQL E IL MODELLO RELAZIONALE** 6

**1.1 COME NASCE SQL** 6
**1.2 IL MODELLO RELAZIONALE** 7
**1.2.1** RELAZIONI E ISTANZE DI RELAZIONE 7
**1.2.2** I VANTAGGI DEL MODELLO RELAZIONALE 8
**1.2.3** I DATABASE RELAZIONALI 8

**2 – XML, XQUERY 1.0 E IL MODELLO SEMI-STRUTTURATO** 9

**2.1 COME NASCE XQUERY** 9
**2.1.2** IL LINGUAGGIO XML 9
**2.1.2.a** XML E HTML A CONFRONTO 9
**2.1.2.b** I NAMESPACE IN XML 10
**2.1.2.c** IL DTD (DOCUMENT TYPE DEFINITION) 10
**2.1.2.c1** DICHIARAZIONE DI ELEMENTI 11
**2.1.2.c2** DICHIARAZIONE DI ATTRIBUTI 11
**2.1.3** VANTAGGI DI XQUERY 14
**2.1.4** SVANTAGGI DI XQUERY 14

**3 - I LINGUAGGI SQL92 E XQUERY-1.0 A CONFRONTO** 15

**3.1 PREMESSA AI DUE LINGUAGGI** 15
**3.1.1** COSTRUTTORE DI ELEMENTI IN XQUERY 1.0 15
**3.1.1.a** ATTRIBUTI NEI COSTRUTTORI DI ELEMENTI 16
**3.1.1.b** CONTENUTO DI UN COSTRUTTORE DI ELEMENTI 17
**3.1.1.c** SPAZI BIANCHI NEL CONTENUTO DI UN COSTRUTTORE DI ELEMENTI1 17
**3.1.2** ESPRESSIONI DA VALUTARE 18
**3.1.3** CARATTERI CON SIGNIFICATI PARTICOLARI 19
**3.1.4** NOTAZIONI PARTICOLARI 19
**3.1.5** FORZARE L'ORDINE DI VALUTAZIONE DI ESPRESSIONI 19
**3.1.6** ALIASES IN SQL 92 20
**3.1.6.a** COLUMN NAME ALIASES 20
**3.1.6.b** TABLE NAME ALIASES 21
**3.2 STRUTTURA BASE DI QUERY** 21
**3.2.1** IN SQL 92 21
**3.2.2** IN XQUERY 1.0 22
**3.3 SQL 92: SELECT E FROM** 23
3.3.1 SELEZIONE DI COLONNE (ATTRIBUTI) 23
**3.4 IN E LE ESPRESSIONI DI PATH IN XQUERY 1.0** 24
**3.5 FOR E LE ESPRESSIONI FLWOR IN XQUERY 1.0** 26
**3.5.1** FOR 27
**3.5.2** LET 27
**3.5.3** RETURN 29
**3.5.4** WHERE 29
**3.5.4.a** LIKE IN SQL 92 31
**3.5.4.b** AND E OR 32
**3.5.4.c** BETWEEN … AND IN SQL 92 32
**3.5.4.d** EMPTY E IS NULL 33
**3.5.4.e** SOME | EVERY…IN…SATISFIED IN XQUERY 1.0 34

**3.5.4.f GLI OPERATORI DI SEQUENZA** « >> » E « << » IN XQUERY 1.0 36
**3. 6 ORDER BY 36**
**3.6.1** ORDER BY IN SQL 92 36
**3.6.2** ORDER BY IN XQUERY 1.0 38
**3.7 DISTINCT E DISTINCT-VALUES 39**
**3.8 UNION 40**
**3.9 SOTTOQUERY IN SELECT 40**
**3.9.1** SOTTOQUERY CON L'OPERATORE DI SET [NOT] IN 41
**3.9.2** SOTTOQUERY CON INTERSECT ED EXCEPT IN XQUERY 1.0 41
**3.9.3** [NOT] EXISTS IN SQL 92 42
**3.9.4** EXISTS IN XQUERY 1.0 42
**3.10 RAGGRUPPAMENTI IN SQL 92 43**
**3.10.1** GROUP BY 43
**3.10.2** HAVING 43
**3.11 RAGGRUPPAMENTI IN XQUERY 1.0 44**
**3.12 FIRST E «[ ]» 45**
**3.13 ESPRESSIONI ARITMETICHE 46**
**3.14 ESPRESSIONI CONDIZIONALI (IF…THEN…ELSE) IN XQUERY 1.0 46**
**3.15 JOIN IN SQL 92 47**
**3.16 JOIN IN XQUERY 1.0 48**
**3.17 DIVERSI TIPI DI JOIN 49**
**3.17.1** INNER JOIN IN SQL 92 49
**3.17.2** INNER JOIN IN XQUERY 50
**3.17.3** SELF JOIN IN SQL 92 50
**3.17.4** SELF JOIN IN XQUERY 1.0 51
**3.17.5** (LEFT | RIGHT) OUTER JOIN IN SQL 92 51
**3.17.6** (LEFT | RIGHT) OUTER JOIN IN XQUERY 1.0 53
**3.17.7** FULL OUTER JOIN IN SQL 92 53
**3.17.8** FULL OUTER JOIN IN XQUERY 1.0 54
**3.17.9** JOIN MULTIPLI IN SQL 92 54
**3.17.10** JOIN MULTIPLI IN XQUERY 1.0 55
**3.18 FUNZIONI IN SQL 92 55**
**3.18.1** TIPI DI FUNZIONI 55
**3.18.2** FUNZIONI DI AGGREGAZIONE 56
**3.18.3** FUNZIONI SCALARI 56
**3.18.3.a** FUNZIONI DI TEMPO 56
**3.18.3.b** FUNZIONI DI CONVERSIONE DI DATE 57
**3.18.3.c** FUNZIONI CHE AGISCONO SUI CARATTERI 57
**3.18.4** LE VISTE IN SQL 92 57
**3.19 FUNZIONI IN XQUERY 1.0 58**
**3.19.1** FUNZIONI PREDEFINITE IN XQUERY 1.0 58
**3.19.1.a** FUNZIONI DI AGGREGAZIONE 59
**3.19.1.b** FUNZIONI DI TEMPO 59
**3.19.1.c** FUNZIONI BOOLEANE 59
**3.19.1.d** FUNZIONI DI ERRORE 59
**3.19.1.e** FUNZIONI SULLE STRINGHE 59
**3.19.1.f** ALTRE FUNZIONI 59
**3.19.2** FUNZIONI DEFINIBILI DALL'UTENTE 60
**3.20 COMMENTI IN XQUERY 1.0 62**
**3.21 COMMENTI IN SQL 92 62**

**4 - QUADRO SINOTTICO RIASSUNTIVO 63**

**5 – CONCLUSIONI 64**

**6 - NOTAZIONE BNF E BASIC EBNF 65**

**BIBLIOGRAFIA** **66**

# INDICE DEGLI ESEMPI:


Esempio 1: tabella nel modello di dati relazionale 7
Esempio 2: DTD 11
Esempio 3: dati che rispettano il DTD dell'Esempio 2 12
Esempio 4: formato XML 13
Esempio 5: rappresentazione gerarchica di un documento XML 14
Esempio 6: costruttore di elementi in XQuery 1.0 16
Esempio 7: attributi in un costruttore di elementi in XQuery 1.0 16
Esempio 8: rappresentazione gerarchica del risult. di un costruttore di elementi con attributi 17
Esempio 9: spazi bianchi nel contenuto di un costruttore di elementi 17
Esempio 10: spazi bianchi preservati in un costruttore di elementi 18
Esempio 11: espressioni da valutare in XQuery 1.0 18
Esempio 12: alias di nomi di colonna 20
Esempio 13: alias di nomi di tabella 21
Esempio 14: query semplice in SQL 92 21
Esempio 15: query semplice in XQuery 1.0 22
Esempio 16: espressione di path (I) 25
Esempio 17: espressione di path (II) 25
Esempio 18: espressione di path (III) 25
Esempio 19: espressione di path (IV) 26
Esempio 20: clausola "for" in XQuery 1.0 27
Esempio 21: clausola "let" in XQuery 1.0 28
Esempio 22: clausole "let" e "for" a confronto 28
Esempio 23: clausola "where" in SQL 92 e in XQuery 1.0 30
Esempio 24: alternative al simbolo "=" nella clausola "where" in SQL 92 e in XQuery 1.0 31
Esempio 25: esempio di simbolo "%" come wildcard in SQL 92 (I) 32
Esempio 26: esempio di simbolo "%" come wildcard in SQL 92 (II) 32
Esempio 27: esempio di simbolo "%" come wildcard in SQL 92 (III) 32
Esempio 28: operatori "and" e "or" 32
Esempio 29: "between…and" in SQL 92 33
Esempio 30: "empty" in XQuery e "is null" in SQL 92 34
Esempio 31: utilizzo di "some-in-satisfies" in XQuery 1.0 35
Esempio 32: utilizzo di "every-in-satisfies" in XQuery 1.0 35
Esempio 33: operatore di sequenza ">>" in XQuery 1.0 36
Esempio 34: clausola "order by" in SQL 92 37
Esempio 35: clausola "order by" in SQL 92 su più campi 37
Esempio 36: chiave di raggruppam. della clausola "order by" in SQL 92 con n. identificativo 37
Esempio 37: clausola "order by" in XQuery 1.0 38
Esempio 38: clausola "order by" in XQuery 1.0 su più campi 38
Esempio 39: confronto fra lista ottenuta con e senza la clausola "order by" in XQuery 1.0 39
Esempio 40: clausola "order by" in XQuery 1.0 con chiave di ordinam. non implicita nel ris. 39
Esempio 41: costrutti "select" annidati in SQL 92 40
Esempio 42: parole chiave "not in" in SQL 92 41
Esempio 43: quantificatore esistenziale "exists" in SQL 92 42
Esempio 44: quantificatore esistenziale "exists" in XQuery 1.042

Esempio 45: clausola "group by" in SQL 92 43
Esempio 46: confr. fra raggr. con "let" e "order by" in XQuery 1.0 e "group by" in SQL 92 44
Esempio 47: raggruppamenti in base a più valori in XQuery 1.0 44
Esempio 48: operatori "[" e "]" in XQuery 1.0 45
Esempio 49: espressione condizionale "if then else" in XQuery 1.0 (I) 47
Esempio 50: espressione condizionale "if then else" in XQuery 1.0 (II) 47
Esempio 51: prodotto cartesiano in SQL 92 48
Esempio 52: join con condizione in SQL 92 48
Esempio 53: inner join in SQL 92 49
Esempio 54: inner join semplificato in SQL 92 49
Esempio 55: inner join in XQuery 1.0 50
Esempio 56: self join in SQL 92 (I) 50
Esempio 57: self join in SQL 92 (II) 51
Esempio 58: self join in XQuery 1.0 51
Esempio 59: confronto fra inner join e outer join in SQL 92 52
Esempio 60: outer join in XQuery 1.0 53
Esempio 61: full outer join in XQuery 1.0 54
Esempio 62: join multiplo in SQL 92 54
Esempio 63: join multiplo in XQuery 1.0 55
Esempio 64: semplificazione di query mediante l'uso di una vista 58
Esempio 65: dichiarazione di funzione in XQuery 1.0 (I) 60
Esempio 66: dichiarazione di funzione in XQuery 1.0 (II) 61
Esempio 67: dichiarazione di funzione ricorsiva in XQuery 1.0 61

# INTRODUZIONE

Per ricercare dati e interrogare database sono attualmente disponibili diversi linguaggi, fra i quali troviamo SQL 92 e XQuery 1.0; il primo viene utilizzato in modelli di dati di tipo relazionale, il secondo, che come si può notare dal numero della versione è stato introdotto di recente ed è ancora in fase di definizione, in modelli di tipo semi-strutturato.

I due linguaggi presentano analogie, ed è perciò possibile confrontarne le query e ricavare principi di traduzione da uno all’altro.

Di seguito questi due linguaggi di interrogazione verranno per prima cosa presentati a livello generale, mostrandone le differenze e le somiglianze e i rispettivi vantaggi e svantaggi, mentre si entrerà più nel dettaglio nella sezione successiva, dove saranno riportate e messe a confronto le specifiche della sintassi dei due linguaggi e saranno presentati alcuni esempi di queries (interrogazioni) SQL 92 tradotte in XQuery 1.0.

# 1 - SQL 92 E IL MODELLO RELAZIONALE:

## 1.1 - COME NASCE SQL:

SQL (Structured Query Language) è un linguaggio creato inizialmente dalla IBM: la sua funzione principale è quella di accedere a database al fine di ricercare o aggiornare dati. Per ragioni di performance o per trarre vantaggio a livello di competizione, sono state create diverse implementazioni di SQL, ognuna delle quali differiva in piccole parti dalle altre e dalla versione IBM del linguaggio. Per garantire che queste differenze rimanessero minime, nei primi anni Ottanta è stata istituita una commissione per definire uno standard; la commissione X3H2, sponsorizzata dall’ANSI (American National Standards Institute), ha definito nel 1986 lo standard, con il quale ha obbligato tutte le versioni a supportare le stesse parole chiave (come SELECT, UPDATE, DELETE, INSERT, WHERE), e a far sì che queste producessero i medesimi risultati.

SQL è un linguaggio molto diffuso: infatti viene utilizzato con programmi database come MS Access, DB2, Informix, MS SQL Server, Oracle, Sybase, e molti altri.

La versione di SQL di cui di seguito verranno analizzate le specifiche è quella del 1992 (SQL 92).

Come scritto poco sopra, SQL viene utilizzato per compiere interrogazioni su dati di tipo relazionale: vediamo ora in dettaglio questo modello di dati, introdotto nel 1970 da E.F.Codd, ma che nonostante la sua semplicità di comprensione si è diffuso soltanto a metà degli anni ottanta per la mancanza all'epoca di soluzioni efficienti per implementare le strutture relazionali.

# 1.2 - IL MODELLO RELAZIONALE:

Alla base del modello relazionale (1) sta il concetto dell'evitare agli utenti di banche dati la necessità di conoscere come i dati sono organizzati nella macchina.

Il modello relazionale si basa sul concetto matematico di relazione tra insiemi:

## 1.2.1 - RELAZIONI E ISTANZE DI RELAZIONE:

La struttura fondamentale del modello relazionale è appunto la "relazione", cioè una tabella bidimensionale costituita da righe (tuple) e colonne (attributi), che rappresentano le entità presenti nel database. Ogni istanza dell'entità troverà posto in una tupla della relazione, mentre gli attributi della relazione rappresenteranno le proprietà dell'entità`. Ad esempio, se nel database si dovranno rappresentare delle persone, si potrà definire una relazione chiamata "Person", i cui attributi descrivono le caratteristiche delle persone. Ciascuna tupla della relazione "Person" rappresenterà una particolare persona.

Esempio 1: tabella nel modello di dati relazionale

**PERSON**

| NAME | TEL | EMAIL |
|---|---|---|
| Mario Rossi | 059 121314 | rossi.mario@tin.it |
| Paola Bianchi | 059 222214 | p.bianchi@tin.it |
| Michele Neri | 059 454554 | m.neri@virgilio.it |
| Piero Verdi | 059 677272 | verdi.p@tin.it |

In realtà, volendo essere rigorosi, una relazione è solo la definizione della struttura della tabella, cioè il suo nome e l'elenco degli attributi che la compongono. Quando essa viene popolata con delle tuple, si parla di "istanza di relazione".

Di seguito vediamo in sintassi BNF come viene definito un elemento tabella in SQL 92:

```
[1] <table element> ::=
      <column definition> | <table constraint definition>
```

dove

```
[2] <column definition> ::=
    <column name> { <data type> | <domain name> }
    [ <default clause> ]
    [ <column constraint definition>... ]
    [ <collate clause> ]
```

```
[3] <column name> ::= <identifier>

[4] <data type> ::=
      <character string type>
           [ CHARACTER SET <character set specification> ]
    | <national character string type>
    | <bit string type>
    | <numeric type>
    | <datetime type>
    | <interval type>

[5] <character string type> ::=
      CHARACTER [ <left paren> <length> <right paren> ]
    | CHAR [ <left paren> <length> <right paren> ]
    | CHARACTER VARYING <left paren> <length> <right paren>
    | CHAR VARYING <left paren> <length> <right paren>
    | VARCHAR <left paren> <length> <right paren>

[6] <length> ::= <unsigned integer>
```

### 1.2.2 - I VANTAGGI DEL MODELLO RELAZIONALE:

Uno dei grandi vantaggi del modello relazionale è che esso definisce anche un algebra, chiamata appunto "algebra relazionale". Tutte le manipolazioni possibili sulle relazioni sono ottenibili grazie alla combinazione di cinque soli operatori: RESTRICT, PROJECT, TIMES, UNION e MINUS.
I database relazionali compiono tutte le operazioni sulle tabelle utilizzando l'algebra relazionale, anche se normalmente non permettono all'utente di utilizzarla; infatti l'utente interagisce con il database attraverso un'interfaccia differente, il linguaggio SQL. Le istruzioni SQL vengono poi scomposte dal DBMS in una serie di operazioni relazionali.

### 1.2.3 - I DATABASE RELAZIONALI:

I database relazionali sono attualmente molto diffusi: le caratteristiche che fanno del database relazionale uno dei modelli di preferenza sono la capacità di evitare la ridondanza dei dati e la velocità con cui è possibile effettuare interrogazioni complesse. Inoltre forniscono sistemi semplici ed efficienti per rappresentare e manipolare i dati e si basano su un modello (quello relazionale) con solide basi teoriche.

I database relazionali si utilizzano al meglio nei casi in cui si debbano associare e correlare grandi quantitativi di record per produrre risultati sintetici.

# 2 - XML, XQUERY 1.0 E IL MODELLO SEMI-STRUTTURATO:

## 2.1 - COME NASCE XQUERY:

La nascita di XQuery si deve alla sempre più crescente necessità di compiere interrogazioni su dati non più basati sul modello relazionale, ma di tipo semi-strutturato: infatti questo tipo di dati sta prendendo sempre più piede, grazie soprattutto all'ampia diffusione di XML a scapito di HTML.
XQuery è un linguaggio funzionale, il che significa che si compone di espressioni che ritornano valori e non hanno effetti collaterali e che le espressioni possono essere innestate con piena generalità (sebbene diversamente dai linguaggi funzionali puri non permetta la sostituzione di variabili, se la dichiarazione delle variabili contiene la costruzione di nuovi nodi).
XQuery è inoltre un linguaggio fortemente tipizzato, ovvero gli operandi delle varie espressioni, gli operatori e le funzioni devono essere conformi ai tipi attesi.

### 2.1.2 - IL LINGUAGGIO XML:

XML (Extensible Markup Language) è un linguaggio di markup definito dal consorzio W3C (World Wide Web Consortium, l'organismo che stabilisce gli standard per il Web) nel 1996, derivato dal linguaggio SGML (Standard Generalized Markup Language, un linguaggio per la specifica dei linguaggi di markup, nonché padre del ben noto HTML).
È un'evoluzione rispetto agli altri linguaggi di questo tipo: uno dei punti di forza dell'XML rispetto all'HTML è proprio quello di essere estensibile, quindi di permettere ai progettisti di documenti di gestire fonti multiple di dati e di creare ambienti personalizzati per tipi speciali di dati.
Tramite XML è possibile infatti definire dei propri tag, a patto di rispettare le specifiche del W3C.

#### 2.1.2.a - XML E HTML A CONFRONTO:

Sono diverse le caratteristiche positive del linguaggio XML: una delle più importanti, soprattutto per quanto riguarda i database, è che XML facilita la gestione e lo scambio dei dati, visto che per sua natura è orientato alla descrizione degli stessi.
XML permette di superare il grosso limite del linguaggio HTML, nato per condividere le informazioni su sistemi differenti, ma che siano informazioni semplici, composte da testo con al più immagini e collegamenti ipertestuali.
Attualmente però, le informazioni sul Web sono sempre più articolate, in quanto composte da immagini, suoni, video e database di testo, quindi da tipi differenti di informazioni e relazioni complesse fra documenti.
Per gestire questo tipo di informazioni, XML consente di descrivere la struttura del documento in modo gerarchico, (ed è questo aspetto che ha generato l'esigenza di un nuovo linguaggio per effettuare queries, come XQuery) nidificando i tag per ogni livello di complessità`: mentre HTML prende in considerazione soprattutto il modo in cui le

informazioni vengono presentate, XML pone un accento sul tipo o sulla struttura di tali informazioni.

In altre parole, quando un documento HTML viene processato dal browser, la semantica viene ignorata, la macchina non comprende quale tipo di informazione deve essere resa, il contenuto informazionale del testo non è oggetto di indagine.
Viceversa una codifica XML presta attenzione non all'aspetto degli elementi testuali, cioè alla loro distribuzione fisica, ma al contenuto di ogni singola partizione; esprime quindi, tramite il ricorso a marcatori alfabetici, il significato della stringa di caratteri cui il tag è associato. XML focalizza la codifica sulla semantica e sulla struttura dei dati, mantenendo altresì l'ordinamento gerarchico che sovrintende l'organizzazione degli elementi della fonte. Ne deriva un assoluto parallelismo con la distribuzione dei record all'interno di una base di dati.

Inoltre il linguaggio XML permette di creare il proprio linguaggio di markup (da qui la denominazione "Extensible"), specifico per il tipo di informazione trattata, senza le limitazioni derivanti dall'utilizzo di tag predefiniti come invece avviene in HTML; il codice risultante è quindi più adattabile alle proprie esigenze e senza dubbio permette di comprendere il significato dei dati in modo semplice e più intuitivo.

### 2.1.2.b - I NAMESPACE IN XML:

Come descritto in precedenza, in XML i tipi di documenti si mescolano e si fondono tra loro in maniera complessa. Lo stesso documento potrebbe avere alcuni elementi definiti in un vocabolario ed altri in un altro.
Un esempio comune è un documento XML di valori di borsa che adopera i tag di HTML per definire gli elementi di testo, ed un insieme di tag specifico per gli elementi di borsa.
I problemi sono identificare esattamente l'ambito di ciascun elemento, conciliare la presenza di elementi definiti in uno di più vocabolari, e soprattutto conciliare la presenza di elementi definiti con lo stesso nome in più vocabolari diversi. I namespace in XML si propongono per risolvere questi problemi. Il namespace è solo un modo per differenziare nomi dello stesso documento.
Ogni nome di elemento o attributo del documento XML è preceduto da un prefisso che ne specifica l'ambito. Il prefisso è separato da il carattere ':' dal nome dell'elemento o dell'attributo.
L'attributo predefinito "xmlns" serve per introdurre i prefissi usati dai namespace del documento. Il valore dell'attributo è un URI che non ha nessun valore dichiarativo, ma solo informativo. Si usa un URI perché si sa già che è unico su Internet.
Poiché ogni namespace userà un prefisso diverso, è possibile capire quali elementi appartengono all'uno e all'altro, e di evitare qualunque problema di collisione.

### 2.1.2.c - IL DTD (DOCUMENT TIPE DEFINITION):

Un documento XML può inoltre contenere una descrizione della sua grammatica, in modo che un'applicazione sia in grado di validarne la struttura.
Il DTD (Document Type Definition) (3) è infatti un file esterno che contiene la definizione precisa di ogni elemento e attributo che potrebbe essere presente in un documento XML, e per questo esprime vincoli meno stretti di quelli descritti in un sistema relazionale.

## 2.1.2.c1 - DICHIARAZIONE DI ELEMENTI:

Nel DTD gli elementi sono definiti nel seguente modo:

```
[7] Element_definition ::= "<"!ELEMENT element-name "("
     element-content | child-element-name ("+" | "*"| "?")? )+ ")>"
```

dove

```
[8] element-content ::= EMPTY | #CDATA | #PCDATA | #ANY
```

EMPTY dichiara un elemento vuoto

#CDATA significa che l'elemento contiene caratteri non analizzati da un parser

#PCDATA significa che l'elemento contiene caratteri che saranno analizzati da un parser

#ANY dichiara un elemento dal contenuto qualsiasi

## 2.1.2.c2 - DICHIARAZIONE DI ATTRIBUTI:

```
[8] Attribute_definition ::= "<" !ATTLIST element-name
     attribute-name attribute-type default-value ">"
```

```
[9] attribute-type ::= CDATA | ID | IDREF | xml:
```

CDATA indica che il valore è un dato character
ID indica che il valore è un identificatore unico
IDREF indica che il valore è un id di un altro elemento
xml: indica che il valore è predefinito
(ci sono altri attribute-type, ma qui sono stati riportati i principali)

```
[10] default-value ::= #REQUIRED | #IMPIED | #DEFAULT value |
     #FIXED value
```

#REQUIRED indica che l'attributo è necessario
#IMPIED indica che l'attributo non è necessario
#DEFAULT value indica che l'attributo ha il valore di default "value"
#FIXED value indica che l'attributo ha il valore fissato "value"

## Esempio 2: DTD

Questo DTD descrive i dati presenti in un documento che rispecchia la tabella "Person" vista in precedenza nell'esempio di tabella di modello di dati relazionale.

```
<!ELEMENT persons (person* )>
<!ELEMENT person  (name, tel,(email?), (address?) )>
<!ELEMENT name (#PCDATA )>
<!ELEMENT tel (#PCDATA )>
<!ELEMENT email (#PCDATA )>
<!ELEMENT address (#PCDATA )>
```

## Esempio 3: dati che rispettano il DTD dell'Esempio 2

In un documento che si può ottenere dal DTD dell'esempio 2 l'elemento <person> deve avere obbligatoriamente i due sottoelementi <name> e <tel>, mentre i sottoelementi <email> e <address> sono opzionali. Di seguito sono rappresentate alcune possibili strutture degli elementi <person> ottenibili dal DTD dell'esempio 2.

```
<persons>
  <person>
      <name> Giulia Gialli </name>
      <tel>059 754328</tel>
      <email>gialli.giu@tin.it</email>
      <address>via Verdi 92</address>
 </person>

<persons>
  <person>
      <name> Mario Rossi </name>
      <tel>059 121314</tel>
      <email>rossi.mario@tin.it</email>
  </person>

  <person>
      <name> Ugo Verdi </name>
      <tel>059 244296</tel>
  </person>
 <person>

      <name> Paola Bianchi </name>
      <tel>059 222214</tel>
      <email>p.bianchi@tin.it</email>]
      <address>via Roma 22</address>
  </person>

  <person>
      <name> Michele Neri </name>
      <tel>059 454554</tel>
      <address>via Appia 823</address>
  </person>
</persons>
```

La sintassi dell’XML è più rigida rispetto a quella dell'HTML, fondamentalmente per una questione di prestazioni, ed una sintassi chiara aumenta la leggibilità del documento.

Per evidenziare meglio le differenze fra il modello semi-strutturato di XML e quello relazionale, di seguito è riportato un esempio che riprende quello precedente, in cui veniva mostrata la tabella “PERSON”, tradotto in XML (con opportune modifiche): gli elementi sono indicati racchiusi fra il simbolo “ < ” e il simbolo “ > ”.

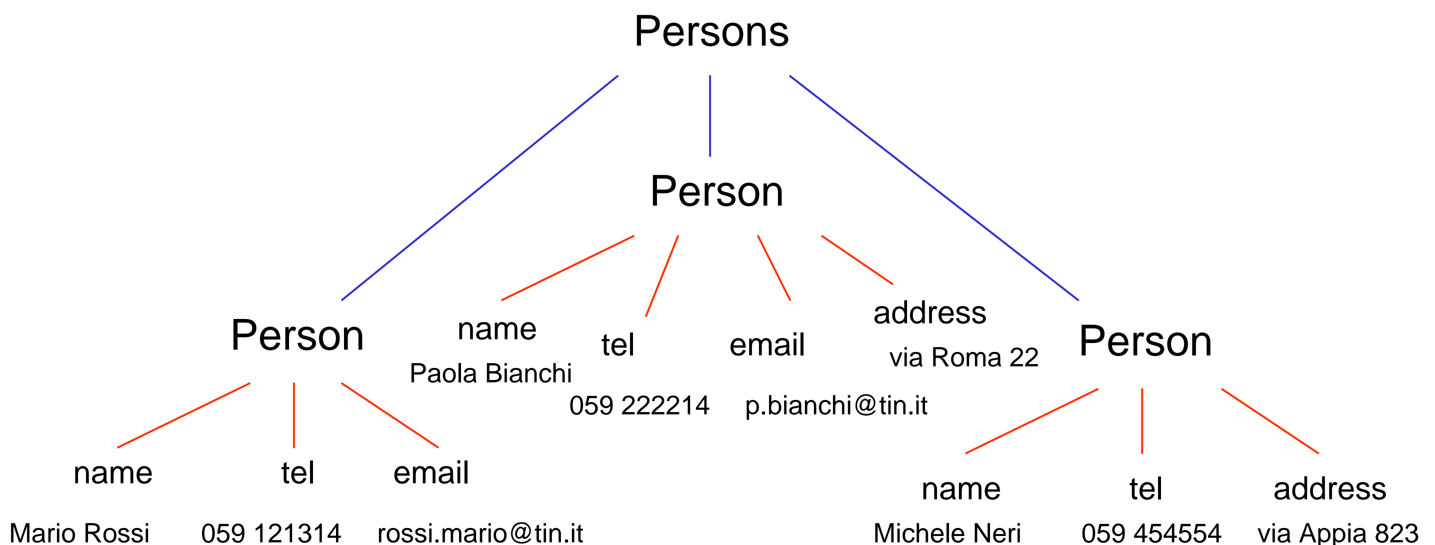
## Esempio 4: formato XML

```
<persons>
  <person>
      <name> Mario Rossi </name>
      <tel>059 121314</tel>
      <email>rossi.mario@tin.it</email>
  </person>
  <person>
      <name> Paola Bianchi </name>
      <tel>059 222214</tel>
      <email>p.bianchi@tin.it</email>]
      <address>via Roma 22</address>
  </person>
  <person>
      <name> Michele Neri </name>
      <tel>059 454554</tel>
      <address>via Appia 823</address>
  </person>
</persons>
```

Nel documento riportato sopra l’elemento root `<persons>` racchiude tre elementi figli `<person>`, i quali a loro volta contengono alcuni sottoelementi uguali e altri diversi, che per di più sono in numero variabile: infatti in XML i dati non sono più riconducibili a rigide tabelle con colonne fisse, come accade nel modello relazionale, ma a più complessi ed articolati alberi.

Per meglio comprendere questo, di sotto è riportato lo schema gerarchico del testo XML appena presentato:

Esempio 5: rappresentazione gerarchica di un documento XML




Un documento in formato XML o un database di tipo XML necessitano quindi di un apposito strumento per essere interrogati, proprio per la loro particolare struttura (infatti l'uso di SQL per le ricerche in dati XML non è una soluzione praticabile perché per sua natura SQL non è costruito per le ricerche di tipo gerarchico); nasce cosi' il linguaggio di interrogazione XQuery.

## 2.1.3 - VANTAGGI DI XQUERY:

Uno dei grandi vantaggi di XQuery è che questo linguaggio di interrogazione può essere utilizzato per unire dati provenienti da molteplici risorse contenute in locazioni differenti, e in questo è più potente di SQL, legato alla rigida struttura di tabelle e relazioni.
Invece che restituire un risultato di query a righe e colonne, le espressioni XQuery ritornano un documenti XML che contengono nodi che corrispondono ai criteri della query.

XQuery inoltre ha funzioni di aggregazione equivalenti a SQL (sum(), avg(), min() e max()), ma ha anche funzioni relative a documenti, come document(), empty(), distinct() e substring(). Le specifiche di funzioni e operatori di XQuery 1.0 e XPath 2.0 sono più di 200. La grande flessibilità di XQuery inoltre permette di costruire funzioni definite dall'utente.

## 2.1.4 - SVANTAGGI DI XQUERY:

Ovviamente il fatto di ricorrere a dato o database relazionali oppure di tipo XML ha pro e contro: un vantaggio dell’utilizzo di SQL rispetto ad XQuery è sicuramente dato dal fatto che SQL è meno prolisso (come si può vedere a colpo d’occhio dagli esempi di query nei paragrafi seguenti). Inoltre il modello gerarchico può essere meno efficiente; in particolare, i dati in formato gerarchico assorbono molte più risorse rispetto ad altri modelli di database, e le query possono essere perciò più lente e complesse da definire.

# 3 - I LINGUAGGI SQL 92 E XQUERY 1.0 A CONFRONTO:

## 3.1 - PREMESSA AI DUE LINGUAGGI:

### 3.1.1 - COSTRUTTORE DI ELEMENTI IN XQUERY 1.0:

In query XQuery si possono trovare costruttori di elementi, con la funzione di generare un nuovo elemento XML con una propria identità di nodo, eventualmente con elementi figli: tutti gli attributi e i nodi discendenti del nuovo elemento nodo sono anch'essi nodi con la propria identità, anche se sono copie di nodi preesistenti.
Se il nome, gli attributi e il contenuto degli elementi sono costanti, il costruttore viene chiamato costruttore di elementi "diretto"; i nomi di elementi non qualificati utilizzati in un costruttore diretto sono implicitamente qualificati dal namespace di default per i nomi di elementi.
In una query XQuery l'utilizzo dei costruttori di elementi si rivela utile poiché consente di generare il risultato come un insieme di oggetti che hanno ordine e caratteristiche volute.

Il costruttore è composto da due tag che rappresentano il nome dell'elemento, e che racchiudono gli eventuali sottoelementi dell'elemento da creare; il tag posto alla fine deve avere lo stesso nome di quello posto all'inizio, preceduto da " / ".

Sintassi Basic EBNF del costruttore di elementi:

```
[13] Constructor ::= ElementConstructor | XmlComment | XmlPI |
CdataSection | CompDocConstructor | CompElemConstructor |
CompAttrConstructor | CompNSConstructor | CompTextConstructor |
CompXmlPI | ComputedXmlComment

[14] ElementConstructor ::= "<" QName AttributeList ("/>" | (">"
ElementContent* "</" QName S? ">"))

[15] ElementContentChar ::= Char - [{}<&]

[16] ElementContent ::= ElementContentChar | "{{" | "}}" |
ElementConstructor | EnclosedExpr | CdataSection | CharRef |
PredefinedEntityRef | XmlComment | XmlPI

[17] AttributeList ::= (S (QName S? "=" S? AttributeValue)?

[18] AttributeValue ::= ('"' (EscapeQuot | QuotAttrValueContent)*
'"') | ("'" (EscapeApos | AposAttrValueContent)* "'")

[19] QuotAttrValueContent ::= QuotAttContentChar | CharRef | "{{"
| "}}" | EnclosedExpr | PredefinedEntityRef

[20] AposAttrValueContent ::= AposAttContentChar | CharRef | "{{"
| "}}" | EnclosedExpr | PredefinedEntityRef
```

```
[21] QuotAttContentChar ::= Char - ["{}<&]

[22] AposAttContentChar ::= Char - ['{}<&]

[23] EscapeQuot ::= '"' '"'

[24] EscapeApos ::= "''"

[25] EnclosedExpr ::= "{" Expr "}"
```

### Esempio 6: costruttore di elementi in XQuery 1.0

Generare un elemento che abbia nome dato da parametro, che contenga due elementi innestati, <description> e <price>:

```
<$tagname>
   <description> $d </description> ,
   <price> $p </price>
</$tagname>
```

## 3.1.1.a - ATTRIBUTI NEI COSTRUTTORI DI ELEMENTI:

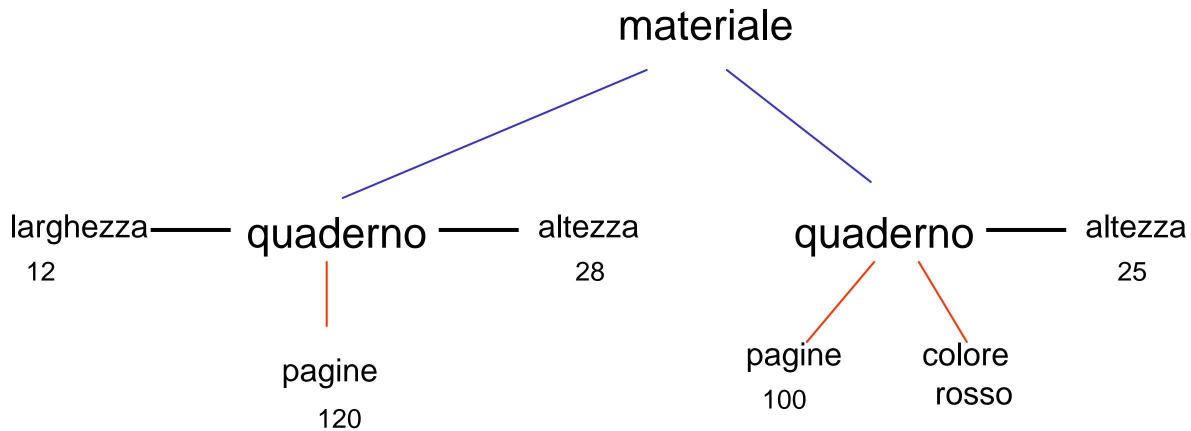
All’interno del tag iniziale di un costruttore di elementi si possono inserire uno o più attributi di quell’elemento, specificandone il nome e il valore.

### Esempio 7: attributi in un costruttore di elementi in XQuery 1.0

```
<materiale>
     <quaderno  altezza = “28” larghezza =”12”>
          <pagine> 120 </pagine>
     </quaderno>
     <quaderno  altezza = “25”>
          <pagine> 100 </pagine>
          <colore> rosso </colore>
     </quaderno>
</materiale>
```

Il nome dell’attributo (nell’esempio sopra sono nomi “altezza” e “larghezza”) è specificato da una costante, e il valore (nell’esempio sopra sono valori “28” e “12”) è rappresentato da una stringa racchiusa da virgolette o apici.
Naturalmente un attributo può contenere espressioni da valutare al momento della costruzione dell’elemento, racchiuse da parentesi graffe.
Ogni attributo che compare in un costruttore di elementi diretto crea un nuovo nodo attributo, il cui genitore è il nodo elemento creato (come si può vedere nel diagramma sottostante); tutti i nodi attributi generati da un costruttore di elementi devono avere nomi distinti.

Esempio 8: rappresentazione gerarchica del risultato di un costruttore di elementi con attributi




### 3.1.1.b - CONTENUTO DI UN COSTRUTTORE DI ELEMENTI:

Il contenuto di un costruttore di elementi è ciò che è compreso fra il tag di inizio del costruttore e il tag di fine.
Il contenuto può essere testo letterale, possono essere altri costruttori di elementi innestati oppure espressioni da valutare racchiuse da parentesi graffe.

### 3.1.1.c - SPAZI BIANCHI NEL CONTENUTO DI UN COSTRUTTORE DI ELEMENTI:

Nel contenuto di un costruttore di elementi diretto possono apparire spazi bianchi.

Esempio 9: spazi bianchi nel contenuto di un costruttore di elementi

```
<a> {"abc"} </a>
```

Se `xmlspace` è dichiarato `xmlspace = strip`, questo esempio è equivalente a

```
<a>abc</a>.
```

Se invece è dichiarato `xmlspace = preserve`, allora è equivalente a

```
<a> abc </a>.
```

È da notare che gli spazi bianchi generati dalla simbologia   sono sempre preservati, così come quelli racchiusi da virgolette.

### Esempio 10: spazi bianchi preservati in un costruttore di elementi

l'espressione

```
<a>{"  "}</a>
```

è equivalente a

```
<a>  </a.
```

## 3.1.2 - ESPRESSIONI DA VALUTARE IN XQUERY 1.0:

All'interno di un costruttore di elementi diretto potrei avere la necessità di distinguere parti di testo da valutare da parti di testo letterale; per questo scopo si usano le parentesi graffe ( " { " e " } "), che poste a racchiudere espressioni, le sostituiscono con il loro valore.

### Esempio 11: espressioni da valutare in XQuery 1.0

```
<bib>
 {
  for $b in document("http://www.bn.com/bib.xml")/bib/book
  where $b/publisher = "Addison-Wesley" and $b/@year > 1991
  return
    <book year="{ $b/@year }">
     { $b/title }
    </book>
 }
</bib>
```

Da questo esempio si nota che un costruttore di elementi si può utilizzare anche per ordinare gerarchicamente gli elementi che compongono il risultato; infatti l'output di questa query potrebbe essere:

```
<bib>
    <book year="1994">
        <title>TCP/IP Illustrated</title>
    </book>
    <book year="1992">
       <title>Advanced Programming in the Unix environment</title>
    </book>
</bib>
```

### 3.1.3 - CARATTERI CON SIGNIFICATI PARTICOLARI:

Una stringa potrebbe contenere caratteri che in altro contesto sono utilizzati con un particolare significato sintattico, come ad esempio accade per le parentesi graffe, che potrebbero essere utilizzate per valutare espressioni; in XQuery perciò esiste un modo alternativo per fare uso di questi particolari caratteri come normali caratteri in parti di testo letterale, senza generare confusione: nelle stringhe questi caratteri sono sostituiti con sequenze di caratteri che iniziano con “ & “ e terminano con “ ; “.

```
<           viene sostituito con &lt;
>           viene sostituito con &gt;
&           viene sostituito con &amp;
"           viene sostituito con &quot;
'           viene sostituito con &apos;
spazio bianco viene sostituito con &#x20;
```

Per quanto riguarda le parentesi invece:

```
{    viene sostituita con {{
}    viene sostituita con }}
```

### 3.1.4 - NOTAZIONI PARTICOLARI:

Il punto e virgola (“ ; ”) è un simbolo che in SQL può essere utilizzato per separare fra loro diverse richieste SQL nei sistemi di database che permettono che più di una espressione venga eseguita nella stessa chiamata al server, e alcuni tutorial SQL terminano ogni espressione col punto e virgola, ma non è obbligatorio.

Inoltre è importante sottolineare che per racchiudere stringhe in SQL si utilizzano gli apici singoli (anche se a volte sono accettate le virgolette), mentre in XQuery sono d’obbligo le virgolette.

### 3.1.5 - FORZARE L’ORDINE DI VALUTAZIONE DI ESPRESSIONI:

All’interno di una query, sia nel linguaggio SQL 92 che in XQuery 1.0 l’ordine di valutazione di espressioni può essere forzato con le parentesi tonde.
Ad esempio, l’espressione 2 + 4 * 5 dà come risultato 22, poiché l’operatore di moltiplicazione ( * ) ha la precedenza su quello di addizione ( + ); se invece scriviamo l’espressione nella forma (2 + 4)* 5, come risultato otteniamo 30, poiché il contenuto delle parentesi viene valutato assieme, e la successiva operazione viene effettuata dopo.

#### Sintassi Basic EBNF per XQuery 1.0:

```
[26]ParentesizedExpr ::="(" Expr? ")"
```

dove

```
[27] Expr ::= ExprSingle ("," ExprSingle)*
```

in cui

```
[28] ExprSingle ::= FLWORExpr
      | QuantifiedExpr
      | TypeswitchExpr
      | IfExpr
      | OrExpr
```

## 3.1.6 - ALIASES IN SQL 92:

In SQL 92 si possono associare alias a nomi di tabelle o colonne, per rendere più comprensibile una query, facendo seguire il nome originario dalla parola chiave AS e dal nome che si vuol dare, oppure semplicemente apponendo il nuovo nome di seguito al vecchio.
È da notare che l'alias ha importanza solo all'interno della query, non provoca nessun tipo di modifica nel risultato della query: se ad una colonna viene assegnato un alias, nella tabella risultato la colonna avrà sempre il suo nome originario: il suo scopo è semplicemente quello di semplificare nomi complessi che magari andrebbero ripetuti più volte, creando disagi a chi crea la query e a chi la legge.

L'utilizzo di alias non è previsto in XQuery.

### 3.1.6.a - COLUMN NAME ALIASES:

### Sintassi BNF:

```
[11] <derived column> ::= <value expression> [ <as clause> ]

[12] <as clause> ::= [ AS ] <column name>
```

### Esempio 12: alias di nomi di colonna

```
SELECT data_di_pagamento AS d
FROM Tabella_Pagamenti
```

oppure più semplicemente

```
SELECT data_di_pagamento d
FROM Tabella_Pagamenti
```

### 3.1.6.b - TABLE NAME ALIASES:

#### Esempio 13: alias di nomi di tabella

```
SELECT data_di_pagamento
FROM Tabella_Pagamenti AS T
```

## 3.2 - STRUTTURA BASE DI QUERY:

Per prima cosa è bene presentare la struttura base di query in entrambi i linguaggi.

### 3.2.1 - IN SQL 92:

#### Sintassi BNF:

```
[29] <query specification> ::=
    SELECT [ <set quantifier> ] <select list> <table expression>
```

dove

```
[30] <set quantifier> ::= DISTINCT | ALL


[31] <select list> ::=
      <asterisk>
    | <select sublist> [ { <comma> <select sublist> }... ]

[32] <table expression> ::=
    <from clause>
    [ <where clause> ]
    [ <group by clause> ]
    [ <having clause> ]
```

La query più semplice in SQL 92 è quindi del tipo:

```
SELECT colonna
FROM tabella
[WHERE] condizione
```

#### Esempio 14: query semplice in SQL 92

Selezionare il nome delle persone [di 22 anni].

```
SELECT nome
FROM Persona
```

```
[WHERE età = 22]
```

## 3.2.2 - IN XQUERY 1.0:

In XQuery invece la struttura base di una query è del tipo:

### Sintassi Basic EBNF:

```
[33] FLWORExpr ::= (ForClause | LetClause)+ WhereClause?
OrderByClause? "return" ExprSingle
```

Questo tipo di espressione viene definita FLWOR (dalle iniziali delle parole chiave for, let, where, order by e return), e verrà presentata nei particolari in seguito.

### Esempio 15: query semplice in XQuery 1.0

La stessa query di prima, in XQuery diventa:

```
for $p in document("Persone.xml")//persona
[where $p/età = 22]
return $p/nome
```

Per prima cosa è bene sottolineare che in SQL 92 è preferibile che le parole chiave (tipo select e from) siano in caratteri maiuscoli, mentre XQuery richiede che le parole chiave siano in minuscolo, pur supportando anche caratteri maiuscoli.

Il simbolo " * " viene utilizzato sia in XQuery 1.0 che in SQL 92 con la funzione di indicare tutti gli elementi (o tutti i campi) presenti in un documento (o in una tabella), qualunque sia il loro numero (compreso zero) e il loro nome.

Per indicare la tabella in cui ricercare gli elementi voluti in SQL si usa la clausola FROM, mentre in XQuery la stessa funzione è svolta la clausola in, che contiene il nome del documento in cui effettuare la ricerca, e data la particolare struttura dei documenti da analizzare, deve contenere anche una espressione di path con cui indicare in che posizione si trovano gli elementi voluti, che verranno riposti volta per volta in una variabile indicata da for.

SELECT è una clausola che va sempre posta all'inizio della query, e comprende indicazioni sul formato del risultato (in particolare indica gli attributi - o colonne – da restituire), mentre in XQuery si richiede la clausola return, che al contrario va sempre posta al termine della query.

Sia in SQL che in XQuery si possono infine (opzionalmente) imporre delle condizioni a cui il risultato prodotto dalla query dovrà attenersi, che vengono indicate in entrambi i linguaggi con la clausola "where".

Ora analizziamo più in dettaglio queste parole chiave.

## 3.3 - SQL 92: SELECT E FROM:

Il costrutto SELECT comprende sia la clausola SELECT che la clausola FROM, imprescindibili l'una dall'altra, ed è il più utilizzato dagli utenti di SQL (è stato stimato che ricorre circa nel novanta per cento dei casi d'uso). SELECT è la via principale per ricavare informazioni da un database: vi si ricorre infatti per selezionare in una tabella dati, che poi verranno riposti in una tabella-risultato. Questo tipo di costrutto, diversamente da altri costrutti di SQL 92, come INSERT, UPDATE e DELETE, non modifica in alcun modo il contenuto del database: semplicemente esegue query sui dati.

```
[34] <from clause> ::= FROM <table reference>
    [ { <comma> <table reference> }... ]

[35] <table reference> ::=
      <table name> [ [ AS ] <correlation name>
          [ <left paren> <derived column list> <right paren> ] ]
    | <derived table> [ AS ] <correlation name>
          [ <left paren> <derived column list> <right paren> ]
    | <joined table>
```

### 3.3.1 - SELEZIONE DI COLONNE (ATTRIBUTI):

Per selezionare una colonna (e tutte le righe – o tuple) di una tabella:

```
SELECT colonna1
FROM Tabella
```

Dove SELECT indica quali attributi selezionare e FROM indica da che tabella recuperare le tuple.

Per selezionare più colonne (e tutte le righe) di una tabella si indicano dopo SELECT i nomi di tutte le colonne desiderate, separati da virgola:

```
SELECT colonna1, colonna2
FROM Tabella
```

Per selezionare tutte le colonne (e tutte le righe) di una tabella si utilizza il simbolo " * " al posto dei nomi delle colonne:

```
SELECT *
FROM Tabella
```

Di seguito a FROM si possono avere più nomi di tabelle separate da virgola, ma questo tipo di query verranno analizzate nella sezione dei "join".

## 3.4 - IN E LE ESPRESSIONI DI PATH IN XQUERY 1.0:

Poiché , diversamente da SQL 92, XQuery effettua query su di un modello di dati di tipo semi-strutturato, e quindi ad albero, necessita di strumenti per la navigazione attraverso i rami del grafo, ovvero per esprimere espressioni di percorso (path), che ovviamente SQL non possiede.
Le espressioni di path in XQuery 1.0 fanno uso della sintassi di XPath 2.0 (2), standard del W3C, estendendola con nuovi operatori e predicati.

In XQuery il risultato di una espressione di path è la lista ordinata dei nodi richiesti, ognuno dei quali include i propri nodi figli, a modello dei grafi ad albero visti in precedenza: ogni nodo conserva la posizione gerarchica che aveva nel documento su cui viene effettuata la query.
Il risultato di una espressione di path può contenere valori duplicati (cioè nodi che abbiano lo stesso tipo e uguale contenuto), ma non può contenere nodi duplicati (ovvero nodi che abbiano la stessa identità di nodo).

Una espressione di path consiste in una serie di uno o più termini (steps) separati da “ / ” o da “ // ”. Ogni step rappresenta un movimento in una particolare direzione attraverso un documento, e ad ogni step può essere applicato uno (o più ) predicati per eliminare nodi che non soddisfino particolari condizioni. Il risultato di ogni step è una lista di nodi che serve da punto di partenza per lo step successivo.

Sintassi Basic EBNF di una path expression:

```
[36] PathExpr ::= ("/" RelativePathExpr?)
| ("//" RelativePathExpr) | RelativePathExpr

[37] RelativePathExpr ::= StepExpr (("/" | "//") StepExpr)*

[38] StepExpr  ::=  AxisStep | FilterStep

[39] AxisStep  ::=  (ForwardStep | ReverseStep) Predicates

[40] FilterStep     ::=  PrimaryExpr Predicates

[41] ForwardStep    ::=  (ForwardAxis NodeTest) | AbbrevForwardStep

[42] ReverseStep    ::=  (ReverseAxis NodeTest) | AbbrevReverseStep

[43] ForwardAxis    ::=  ("child" "::")
     | ("descendant" "::")
     | ("attribute" "::")
     | ("self" "::")
     | ("descendant-or-self" "::")
     | ("following-sibling" "::")
     | ("following" "::")

[44] ReverseAxis    ::=  "parent" "::"
     | "ancestor" "::"
     | "preceding-sibling" "::"
```

```
| "preceding" "::"
| "ancestor-or-self" "::"
```

Una espressione di path può avere inizio con una espressione che identifica un nodo specifico. Ad esempio:

```
document(string)
```

restituisce il nodo radice del documento nominato fra le parentesi tonde.
All'inizio di una espressione di path si possono anche avere " / " oppure " // "; questi simboli sono abbreviazione per step iniziali che sono implicitamente aggiunti all'inizio del path:

" / " è abbreviazione per lo step radice (root): il path ha inizio così al nodo radice dell'albero che contiene il nodo del contesto.

## Esempio 16: espressione di path (I)

L'espressione seguente seleziona tutti gli elementi <prezzo> di tutti gli elementi <cd> dell'elemento <catalogo> (2):

```
/catalogo/cd/prezzo
```

" // " ha l'effetto di selezionare tutti gli elementi col nome dato anche se essi si trovano a livelli differenti nell'albero XML.

## Esempio 17: espressione di path (II)

```
//cd
```

Seleziona tutti gli elementi <cd> presenti nel catalogo (2).

Vediamo ora il significato di altri simboli che possono essere contenuti in espressioni di path:

| | |
|---|---|
| . | Denota il nodo corrente. |
| .. | Denota il nodo genitore del nodo corrente. |
| / | Denota il nodo radice, o un separatore di step in un path. |
| // | Denota i discendenti del nodo corrente. |
| @ | Denota gli attributi del nodo corrente. |
| * | Denota tutti i nodi. |
| [ ] | Racchiudono un'espressione booleana che serve da predicato per lo step successivo. |
| [n] | Serve per selezionare da una lista di elementi l'elemento del numero indicato, quando un predicato è composto da numeri interi. |

### Esempio 18: espressione di path (III)

L'esempio seguente utilizza una espressione di path che consiste in tre steps; il primo step alloca il nodo radice del documento, il secondo il secondo elemento <capitoli> figlio dell'elemento root, e il terzo ritrova gli elementi <figura> che compaiono all'interno del <capitolo> , restituendo soltanto quelli che hanno una didascalia dal valore "Prato":

```
document("zoo.xml")/capitolo[2]//figura[didascalia = "Prato"]
```

### Esempio 19: espressione di path path (IV)

Quest'altro esempio mostra l'utilizzo di un predicato che indica un range di valori; vengono cercate tutte le figure nei capitoli dal 2 al 5 nel documento di nome "zoo.xml":

### Sintassi Basic EBNF

```
[45] RangeExpr ::= AdditiveExpr ( "to" AdditiveExpr )?
```

```
document("zoo.xml")/chapter[2 to 5]//figure
```

Come visto nello schema di una semplice query XQuery, ci si serve di questo tipo di espressioni dopo la clausola in, corrispondente (in unione con for) alla clausola FROM del linguaggio SQL.

## 3.5 - FOR E LE ESPRESSIONI FLWOR IN XQUERY 1.0:

In XQuery for appartiene ad una particolare categoria di espressioni, dette espressioni FLWOR(pronuncia "flower"), dalle iniziali di for, let, where, order by e return, che per XQuery hanno la medesima importanza che ha la clausola SELECT per il linguaggio SQL. Inoltre queste espressioni supportano le iterazioni, i join fra due o più documenti e le modifiche nella struttura dei dati, come vedremo in seguito.

```
[46] FLWORExpr ::= (ForClause | LetClause)+ WhereClause?
OrderByClause? "return" ExprSingle

[47] TypeDeclaration ::= "as" SequenceType

[48] PositionalVar ::= "at" "$" VarName

[49] WhereClause ::= "where" Expr

[50] OrderByClause ::= ("order" "by" | "stable" "order" "by")
OrderSpecList

[51] OrderSpecList ::= OrderSpec ("," OrderSpec)*

[52] OrderSpec ::= ExprSingle OrderModifier
```

```
[53] OrderModifier ::= ("ascending" | "descending")? (("empty"
"greatest") | ("empty" "least"))? ("collation"StringLiteral)?
```

### 3.5.1 - FOR:

```
[54] ForClause ::= "for" "$" VarName TypeDeclaration?
PositionalVar? "in" ExprSingle ("," "$" VarName TypeDeclaration?
PositionalVar? "in" ExprSingle)*
```

Per prima analizziamo le espressioni for che, come si è detto in precedenza, opera assieme ad in per sostituire la clausola FROM di SQL.
In XQuery 1.0 for itera ogni elemento selezionato mediante la clausola in e l'espressione di path, associandolo di volta in volta ad una variabile indicata con la notazione seguente:

```
for $variabile
```

L'esempio più semplice di una clausola for contiene una sola variabile, ma può anche contenere più variabili, ognuna delle quali deve essere seguita dalla propria espressione in `document()`e dalla propria espressione di path. In questo caso for itera ogni variabile sugli elementi che risultano dalla valutazione della sua espressione associata.

Il risultato contiene una tupla per ogni combinazione di valori del prodotto cartesiano della sequenza risultante dalla valutazione delle espressioni. L'ordine delle tuple è dato dall'ordine delle espressioni.

### Esempio 20: clausola "for" in XQuery 1.0

```
for $i in (1, 2),
      $j in (3, 4)
```

Lo stream di tuple generate è il seguente:

```
($i = 1, $j = 3)
($i = 1, $j = 4)
($i = 2, $j = 3)
($i = 2, $j = 4)
```

### 3.5.2 - LET

```
[55] LetClause ::= "let" "$" VarName TypeDeclaration? ":="
ExprSingle ("," "$" VarName TypeDeclaration? ":=" ExprSingle)*
```

La clausola let lega una variabile a un valore, tuttavia diversamente dal for, let associa ogni variabile al risultato della corrispondente espressione senza iterazione.

## Esempio 21: clausola "let" in XQuery 1.0

```
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.

Anche let come for può contenere più di una variabile, ognuna delle quali associata ad una espressione.
Sebbene sia for che let associno variabili, la maniera in cui queste variabili vengono legate ai risultati delle espressioni è piuttosto differente, come si può notare dal seguente esempio:

## Esempio 22: clausole "let" e "for" a confronto

```
let $s := (<uno/>, <due/>, <tre/>)
return <risultato>{$s}</risultato>
```

La variabile $s è associata al risultato dell'espressione (<uno/>, <due/>, <tre/>). Senza nessun for let genera una tupla che contiene ciò che viene associato a $s. La proposizione return è invocata per questa tupla, e genera il seguente risultato:

```
<risultato>
   <uno/>
   <due/>
   <tre/>
</risultato>
```

Nel prossimo esempio invece let è sostituito da for:

```
for $s in (<uno/>, <due/>, <tre/>)
return <risultato>{$s}</risultato>
```

In questo esempio la variabile $s viene iterata per tutta l'espressione, e viene prima associata a <uno/>, poi a <due/> e infine a <tre/>. Return viene poi applicato ad ognuna delle tuple, generando il seguente output:

```
<risultato>
   <uno/>
</risultato>
<risultato>
   <due/>
</risultato>
<risultato>
   <tre/>
</risultato>
```

Così come for, è da notare che anche let preserva l'ordine del documento.

### 3.5.3 - RETURN

L'espressione FLOWR più vicina a SELECT è , come abbiamo visto, return, la quale indica che tipo di risultato generare.
Il risultato può essere:

- Un valore: `F. Dürrenmatt`

- Un nodo: `<Autore>F. Dürrenmatt</Autore>`
(corrispondenti alla SELECT su una unica colonna ed un unica riga)

- Una lista ordinata di nodi:
```
<Autore>J.R.R. Tolkien</Autore>
<Autore>Umberto Eco</Autore>
<Autore>F. Dürrenmatt</Autore>
```
(corrispondente di una SELECT sulla colonna “autore”)

Può contenere costruttori di elementi, riferimenti a variabili definite nelle parti for e let e generiche espressioni annidate.

### 3.5.4 - WHERE:

In SQL 92 WHERE viene utilizzato in combinazione con SELECT per selezionare, oltre a particolari colonne di una tabella, anche particolari righe della stessa, ponendo alcune condizioni di selezione.

```
[56] <where clause> ::= WHERE <search condition>

[57] <search condition> ::=
      <boolean term>
    | <search condition> OR <boolean term>
```

In XQuery la clausola where ha la stessa funzione che ha in SQL: è una proposizione opzionale che filtra le tuple del prodotto cartesiano, generate da for e let, controllando se queste tuple soddisfano la condizione che where contiene (cioè se il valore booleano effettivo dell'espressione è “true”), lasciando ciò che rimane per il risultato e scartando il resto.
L'espressione contenuta in where è valutata una volta per ognuna delle tuple.

Le condizioni nella clausola where possono
contenere diversi predicati connessi da “and”, “or”, “not”.

Con where si possono utilizzare i seguenti operatori, che hanno lo stesso significato sia in SQL 92 che in XQuery 1.0:

| Operatore | Significato |
|---|---|
| = | Uguale |
| <> | Diverso (In alcune versioni di SQL e in XQuery si usa "!=") |
| > | Maggiore |
| < | Minore |
| >= | Maggiore o Uguale |
| <= | Minore o Uguale |
| AND | Concatena due espressioni di where, entrambe da verificare |
| NOT | Negazione |
| OR | Per trovare un subset di valori |

In SQL ho inoltre:

| Operatore | Significato |
|---|---|
| BETWEEN…AND... | Indica un range di valori |
| LIKE | Campione di ricerca |
| IS NULL | Per identificare valori nulli |

In XQuery invece posso anche avere:

| Operatore | Significato |
|---|---|
| (some\|every) $var in... satisfies | Espressioni quantificate |
| contains | Contiene |
| exists | Esiste |
| >> | Segue |
| << | Precede |
| empty | Per identificare valori nulli |
| eq | Uguaglianza (per singoli valori) |
| ne | Disuguaglianza (per singoli valori) |
| lt | Minore (per singoli valori) |
| le | Minore o uguale (per singoli valori) |
| gt | Maggiore (per singoli valori) |
| ge | Maggiore o uguale (per singoli valori) |
| is | Valuta se due espressioni indicano lo stesso nodo |
| is not | Valuta se due espressioni indicano nodi differenti |

Vediamo ora alcuni esempi di clausole where in SQL 92 e in XQuery 1.0, mentre successivamente verranno riprese nel dettaglio le parole chiave sopraelencate.

## Esempio 23: clausola "where" in SQL 92 e in XQuery 1.0

Questa query restituisce tutti i libri con casa editrice Bompiani che sono disponibili (disponibilità = "S", non disponibilità = "N").

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

La stessa query in SQL diventa:

```
SELECT nome_libro
FROM Libri
WHERE editore = 'Bompiani' AND disponibilità = 'S'
```

Da ricordare è l'obbligo di virgolette in XQuery e di apici in SQL per racchiudere le stringhe con cui confrontare le tuple.

In SQL al posto del simbolo di uguaglianza "=" si può mettere la parola chiave LIKE, mentre una sintassi alternativa in XQuery è del tipo:

```
where contains(elemento | attributo, "stringa" | valore_numerico)
```

## Esempio 24: alternative al simbolo "=" nella clausola "where" in SQL 92 e in XQuery 1.0

Selezionare il numero degli elementi con descrizione "Gear".

Versione SQL:

```
SELECT pno
FROM Parts
WHERE description LIKE 'Gear'
```

Versione XQuery:

```
for $p in document ("p.xml")//p_tuple
where contains ($p/description, "Gear")
return $p/pno
```

### 3.5.4.a - LIKE IN SQL 92:

Come accennato poco sopra, LIKE viene utilizzato in SQL 92 per specificare una ricerca per campioni in una colonna.

## Sintassi BNF:

```
[58] <like predicate> ::=
    <match value> [ NOT ] LIKE <pattern>
      [ ESCAPE <escape character> ]

[59] <match value> ::= <character value expression>

[60] <pattern> ::= <character value expression>

[61] <escape character> ::= <character value expression>
```

```
SELECT colonna
FROM tabella
WHERE colonna LIKE modello.
```

In SQL 92 si può utilizzare il simbolo “%” per definire wildcards, ovvero lettere mancanti in una stringa (in XQuery non esiste un simbolo che abbia la stessa funzione).

### Esempio 25: esempio di simbolo “%” come wildcard in SQL 92 (I)

La seguente espressione SQL restituisce le persone il cui nome inizia con la lettera “O”:

```
SELECT *
FROM Persone
WHERE Nome LIKE 'O%'
```

### Esempio 26: esempio di simbolo “%” come wildcard in SQL 92 (II)

L’esempio seguente restituisce le persone il cui nome termina con la lettera “a”:

```
SELECT *
FROM Persone
WHERE Nome LIKE ‘%a’
```

Infine per ricercare le persone il cui nome contiene la stringa “la”:

### Esempio 27: esempio di simbolo “%” come wildcard in SQL 92 (III)

```
SELECT *
FROM Persone
WHERE Nome LIKE '%la%'
```

## 3.5.4.b - AND E OR:

And e or congiungono due o più condizioni di un where; l’operatore and aggiunge una tupla al risultato solo se tutte le condizioni sono vere, or se almeno una è verificata.
And e or possono inoltre essere combinati fra loro, sia in XQuery 1.0 che in SQL 92, come nell’esempio in SQL:

### Esempio 28: operatori “and” e “or”

Selezionare le persone di nome “Ann” oppure “Stephen” e di cognome “Smith”.

```
SELECT *
FROM Persons
WHERE (FirstN='Ann' OR FirstN='Stephen') AND LastN='Smith'
```

## 3.5.4.c - BETWEEN … AND IN SQL 92:

Gli operatori BETWEEN  e AND selezionano un range di dati compresi fra due valori indicati, che possono essere numeri, caratteri testuali o date.

Sintassi BNF:

```
[62] <between predicate> ::=
    <row value constructor> [ NOT ] BETWEEN
      <row value constructor> AND <row value constructor>

[63] <row value constructor> ::=
       <row value constructor element>
    | <left paren> <row value constructor list> <right paren>
    | <row subquery>

[64] <row value constructor element> ::=
      <value expression>
    | <null specification>
    | <default specification>

[65] <value expression> ::=
      <numeric value expression>
    | <string value expression>
    | <datetime value expression>
    | <interval value expression>

[66] <row value constructor list> ::=
    <row value constructor element>
        [ { <comma> <row value constructor element> }... ]
```

## Esempio 29: “between…and” in SQL 92

Per selezionare i dipendenti con stipendio compreso fra 1000 e 1500 euro:

```
SELECT Cognome, Nome, Stipendio
FROM Dipendenti
WHERE Stipendio BETWEEN 1000 AND 1500
```

Nota: gli operatori BETWEEN … AND sono trattati con differenze a seconda del tipo di database SQL: alcuni database infatti escludono dal risultato i dati corrispondenti ai valori che indicano il range (valore1 e valore2), altri invece li includono, altri ancora includono solo il primo (valore1) escludendo il secondo (valore2). Per questo è essenziale conoscere il tipo di database su cui si sta lavorando.

### 3.5.4.d - EMPTY E IS NULL:

Nei costrutti where potrei avere la necessità di controllare se una tupla ha un valore nullo; in XQuery si utilizza la funzione empty(), mentre in SQL svolge il medesimo compito la parola chiave IS NULL.

Sintassi BNF:

```
[67] <null predicate> ::= <row value constructor>
    IS [ NOT ] NULL
```

dove

```
<row value constructor>
```

è stato definito nella EBNF [61].

## Esempio 30: “empty” in XQuery 1.0 e “is null” in SQL 92

Elencare i numeri e la descrizione di ordini per cui manca la data in cui è stato effettuato il pagamento:

in XQuery 1.0:

```
for $i in doc("orders.xml")//order
where empty($i/paid_date)
return
     <result>
          {$i/order_num}
          {$i/description}
     </result>
```

in SQL 92:

```
SELECT order_num, description
FROM Orders
WHERE paid_date IS NULL
```

### 3.5.4.e - SOME | EVERY…IN…SATISFIED IN XQUERY 1.0:

Altri due costrutti molo utili in XQuery che non hanno corrispondenza nel linguaggio SQL 92 sono some-in-satisfies e every-in-satisfied, che permettono di verificare determinate proprietà per gli elementi contenuti in una lista. Il valore di queste espressioni quantificate è sempre `true` o `false`.

## Sintassi Basic EBNF:

```
[68] QuantifiedExpr ::= (("some" "$") | ("every" "$"))VarName
TypeDeclaration? "in" ExprSingle ("," "$" VarName TypeDeclaration?
"in" ExprSingle)* "satisfies" ExprSingle
```

Una espressione quantificata ha inizio con un quantificatore, ovvero una fra le parole chiave some e every, seguita da una o più clausole in e dalla parola chiave satisfies e da una espressione di test.
Ogni clausola in associa una variabile ad una espressione che restituisce una sequenza di valori.

Se il quantificatore è some, l'espressione quantificata è `true` se almeno una valutazione delle espressioni ha il valore booleano `true`, altrimenti l'espressione quantificata è `false`. Se la clausola in genera zero tuple il valore dell'espressione quantificata è `false`.

### Esempio 31: utilizzo di "some-in-satisfies" in XQuery 1.0

```
for $l in document("arch_libri.xml")//libro
where some $t in $l/titolo satisfies (contains($t,"XML") and
contains($t,"tutorial"))
return
  <risultato>
    { $l/titolo }
  </risultato>
```

Se il quantificatore è every, l'espressione quantificata è `true` se tutte le valutazioni delle espressioni hanno il valore booleano `true`, altrimenti l'espressione quantificata è `false`. Se la clausola in genera zero tuple il valore dell'espressione quantificata è `true`.

### Esempio 32: utilizzo di "every-in-satisfies" in XQuery 1.0

```
for $l in document("arch_libri.xml")//libro
where every $t in $l/titolo satisfies contains($t,"XML")
return
  <risultato>
    { $l/titolo }
  </risultato>
```

La prima query restituisce come risultato il titolo di tutti i libri nei cui titoli compare contemporaneamente sia la stringa 'XML' che la stringa 'tutorial'; mentre la seconda query restituisce i titoli dei libri che contengono la stringa 'XML' nel loro titolo.

Anche in SQL esistono espressioni quantificate e di esprimono con la seguente sintassi:

```
[69] <quantified comparison predicate> ::=
    <row value constructor> <comp op> <quantifier> <table
subquery>


[70] <comp op> ::=
      <equals operator>
    | <not equals operator>
    | <less than operator>
    | <greater than operator>
    | <less than or equals operator>
    | <greater than or equals operator>

[71] <quantifier> ::= <all> | <some>

[72] <all> ::= ALL
```

```
[73] <some> ::= SOME | ANY
```

### 3.5.4.f - GLI OPERATORI DI SEQUENZA “ >> ” E “ << ” IN XQUERY:

In XQuery si utilizzano gli operatori “ << ” e “ >> ” per comparare nodi basandosi sull’ordine in cui compaiono nel documento preso in analisi.
Una comparazione con l’operatore “ << ” ritorna vero se il nodo del primo operando precede il nodo indicato dal secondo operando nell’ordine del documento, altrimenti ritorna falso; allo stesso modo l’operatore “ >> ” ritorna vero se il nodo del primo operando segue nell’ordine del documento il secondo, altrimenti falso.
Non esiste corrispondente in SQL.

#### Esempio 33: operatore di sequenza “>>” in XQuery 1.0

Selezionare gli oggetti che sono stati elencati nei primi due paragrafi del documento “doc.xml” dopo il secondo capitolo.

```
let $i2 := (document("doc.xml")//capitolo)[2]
for $a in (document("doc.xml")//paragr)[. >> $i2][position()<=2]
return $a//oggetti
```

## 3.6 - ORDER BY

### 3.6.1 – ORDER BY IN SQL 92:

La parola chiave ORDER BY viene aggiunta ad un costrutto SELECT per ottenere i dati del risultato in uno specifico ordine.

L’ordine indicato può essere ascendente (ASC) o discendente (DESC).

#### Sintassi BNF:

```
[74] <order by clause> ::=
    ORDER BY <sort specification list>
```

dove

```
[75] <sort specification list> ::=
    <sort specification> [ { <comma> <sort specification> }... ]

[76] <sort specification> ::=
    <sort key> [ <collate clause> ] [ <ordering specification> ]
```

```
[77] <sort key> ::=
       <column name>
     | <unsigned integer>

[78] <ordering specification> ::= ASC | DESC
```

### Esempio 34: clausola “order by” in SQL 92

Selezionare codice, prezzo, numero e nome di articoli presenti in magazzino, in ordine (crescente) di codice.

```
SELECT codice, prezzo, pno, nome
FROM Magazzino
ORDER BY codice
```

Si possono inoltre ordinare due o più attributi contemporaneamente, creando un ordinamento innestato, separando i nomi delle colonne con la virgola come nell’esempio, in cui gli articoli di un magazzino vengono ordinati per prima cosa in base al codice e poi in base al prezzo:

### Esempio 35: clausola “order by” in SQL 92 su più campi

Selezionare codice, prezzo, numero e nome di articoli presenti in magazzino, in ordine decrescente di codice e di prezzo.

```
SELECT codice, prezzo, pno, nome
FROM Magazzino
ORDER BY codice, prezzo DESC
```

Per ordinare prima in base al prezzo e poi in base al codice è sufficiente invertire i campi `codice` e `prezzo` nell’espressione ORDER BY:

```
SELECT codice, prezzo, pno, nome
FROM Magazzino
ORDER BY prezzo DESC, codice
```

Come inoltre si vede dalla sintassi BNF la chiave di sort, può essere oltre al nome di una colonna anche un numero identificativo della colonna (in base all’ordine in cui i nomi delle colonne compaiono nella clausola SELECT).

### Esempio 36: chiave di raggruppamento della clausola “order by” in SQL 92 con numero identificativo

Riprendendo l’esempio precedente:

```
SELECT codice, prezzo, pno, nome
FROM Magazzino
ORDER BY 1, 2 DESC
```

### 3.6.2 - ORDER BY IN XQUERY 1.0:

Anche in XQuery 1.0 è prevista la clausola order by con la funzione di ordinare il risultato delle espressioni FLOWR, che altrimenti sarebbe ordinato in base alla sequenza del risultato delle espressioni del for.
Le specifiche di ordinamento contenute in order by sono valutate per ogni tupla, e l'ordine relativo delle tuple è determinato comparandone i valori indicati dalle specifiche da sinistra a destra fino a che non si incontrano valori differenti.

Al posto di ASC oppure DESC del linguaggio SQL, in XQuery si usa indicare ascending o descending.

#### Esempio 37: clausola "order by" in XQuery 1.0

Riprendendo lo stesso esempio visto per ORDER BY in SQL 92:

```
for $a in document("Magazzino.xml")//articolo
order by $a/codice
return
	<risultato>
		{$a/codice}
		{$a/prezzo}
		{$a/pno}
		{$a/nome}
	</risultato>
```

Naturalmente il risultato può essere ordinato come in SQL in base a più criteri; il risultato della query successiva è infatti lo stesso della query SQL vista in precedenza (nel caso in cui vi siano due articoli con lo stesso codice e il medesimo prezzo la parola chiave stable indica che viene preservato l'ordine di input).

#### Esempio 38: clausola "order by" in XQuery 1.0 su più campi

```
for $a in document("Magazzino.xml")//articolo
order by $a/codice , $a/prezzo descending
return
	<risultato>
		{$a/codice}
		{$a/prezzo}
		{$a/pno}
		{$a/nome}
	</risultato>
```

Order by è l'unico mezzo previsto in XQuery per ordinare un documento in ordine differente da prima, perciò ogni query che produce un risultato con ordinamento differente dal documento sorgente deve contenere una espressione FLOWR, anche se l'iterazione non sarebbe necessaria.

### Esempio 39: confronto fra lista ottenuta con e senza la clausola “order by” in XQuery 1.0

Una lista di libri con prezzo inferiore a 100 si potrebbe ottenere con la semplice espressione:

```
$libri//libro[prezzo < 100]
```

Ma se la lista risultante dovesse essere in ordine alfabetico di titolo, la query dovrebbe essere espressa nel seguente modo:

```
for $l in $libri//libro[prezzo < 100]
order by $l/titolo
return $l
```

La proposizione order by viene anche utilizzata per ordinare il risultato di espressioni FLOWR anche se la chiave di ordinamento non è contenuta nel risultato dell’espressione; per esempio, l’espressione seguente restituisce i nomi dei dipendenti in ordine discendente di stipendio, senza però restituire i salari.

### Esempio 40: clausola “order by” in XQuery 1.0 con chiave di ordinamento non implicita nel risultato

```
for $e in $dipendenti
order by $e/stipendio
return $e/nome
```

## 3.7 - DISTINCT E DISTINCT-VALUES:

In SQL 92 la parola chiave DISTINCT aggiunta a SELECT viene utilizzata per restituire valori che siano uno diverso dall’altro, senza ripetizioni.
In XQuery 1.0 DISTINCT viene sostituita dalla funzione distinct-values, che svolge la medesima funzione.

In precedenza, nel parlare della clausola SELECT avevamo già fornito la sintassi BNF della clausola DISTINCT:

### Sintassi BNF:

```
[79] SELECT [ <set quantifier> ] <select list> <table expression>
```

### dove

```
[80] <set quantifier> ::= DISTINCT | ALL
```

## 3.8 - UNION:

L'operatore di unione è rappresentato in SQL dalla parola chiave UNION, che combina due query in un'unica query composta.
L'operatore UNION si può usare tra due o più costrutti SELECT per unirli: seleziona infatti tutte le righe risultanti dalle query, cancella i duplicati e restituisce come risultato ciò che rimane. Poiché il risultato di query diverse deve essere unito, ogni query deve riguardare lo stesso numero di colonne, e inoltre le colonne corrispondenti che sono selezionate da ogni tabella devono contenere tipi di dati compatibili (le colonne di tipo Character devono avere la stessa dimensione), oltre che ammettere tutte caratteri non nulli o non ammetterli.
UNION non può però essere utilizzato all'interno di sottoquery.

L'operatore union in XQuery 1.0 svolge la medesima funzione di UNION.

### Sintassi Basic EBNF:

```
[81] UnionExpr ::= IntersectExceptExpr ( ("union" | "|") IntersectExceptExpr )*
```

dove

```
IntersectExceptExpr
```

sarà approfondita in seguito.

## 3.9 - SOTTOQUERY IN SELECT:

Nel costrutto SELECT si possono avere sottoquery, ovvero altri costrutti SELECT annidati.
L'esempio sottostante mostra un caso d'uso di sottoquery: il risultato che si vuole ottenere è la spesa totale di spedizione da una tabella Ordini per ogni cliente della tabella Cliente.

### Esempio 41: costrutti “select” annidati in SQL 92

```
SELECT Cliente.num_cliente,
     (SELECT SUM(spese_spedizione)
     FROM Ordini
     WHERE Cliente.num_cliente = Ordini.num_cliente) AS totale
FROM Cliente
```

Lo stesso risultato si sarebbe ottenuto mediante un join fra le due tabelle, come si vedrà in seguito.

### 3.9.1 - SOTTOQUERY CON L'OPERATORE DI SET [NOT] IN:

La parola chiane IN viene utilizzata per ricercare particolari elementi in un insieme di elementi (equivale alla parola chiave INTERSECTION, che in genere non si utilizza), mentre NOT IN serve per ricercare particolari elementi che non appartengano ad una determinata collezione di elementi(equivale alla parola chiave EXCEPT).

#### Sintassi BNF:

```
[82] <in predicate> ::=
    <row value constructor>
      [ NOT ] IN <in predicate value>

[83] <in predicate value> ::=
      <table subquery>
    | <left paren> <in value list> <right paren>

[84] <in value list> ::=
    <value expression> { <comma> <value expression> }...
```

#### Esempio 42: parole chiave "not in" in SQL 92

Selezionare i numeri e la compagnia dei clienti che non hanno effettuato nessun ordine(6).

```
SELECT customer_num, company
FROM Customer c
WHERE customer_num NOT IN (  SELECT customer_num
                              FROM Orders o
                              WHERE c.customer_num = o.customer_num
                          )
```

### 3.9.2 SOTTOQUERY CON INTERSECT ED EXCEPT IN XQUERY 1.0:

In XQuery la medesima funzione delle parole chiave IN e NOT IN di SQL 92 viene svolta da intersect ed except :

#### Sintassi Basic EBNF:

```
[85] IntersectExceptExpr ::= ValueExpr ( ("intersect" | "except")
ValueExpr )*

[86] ValueExpr ::= ValidateExpr | PathExpr
```

### 3.9.3 - [NOT] EXISTS IN SQL 92:

EXISTS è un quantificatore esistenziale, che testa l'esistenza (o meno, con l'aggiunta di NOT) di almeno un valore nel risultato della sottoquery.

### Sintassi BNF:

```
[87] <exists predicate> ::= NOT EXIST | EXISTS <table subquery>

[88] <table subquery> ::= <subquery>

[89] <subquery> ::= <left paren> <query expression> <right paren>
```

### Esempio 43: quantificatore esistenziale "exists" in SQL 92

Selezionare nome e lead time di un articolo se presente nella tabella Stock con descrizione contenente la parola "shoe"(6).

```
SELECT manu_name, lead_time
FROM Manufact m
WHERE EXISTS (SELECT *
              FROM Stock s
              WHERE description LIKE '*shoe*'
              AND m.manu_code = s.manu_code)
```

### 3.9.4 - EXISTS IN XQUERY 1.0:

Lo stesso tipo di test si può effettuare in XQuery 1.0 con exists.

### Esempio 44: quantificatore esistenziale "exist" in XQuery 1.0

Fare una lista che riporti il guadagno medio per ogni combinazione di lavoro svolto e di stato.

```
for $s in distinct-values(doc("census.xml")//state),
    $j in distinct-values(doc("census.xml")//job)
let $p := doc("census.xml")//person[state = $s and job = $j]
order by $s, $j
return
   if (exists($p)) then
      <group>
         <state> {$s} </state>
         <job> {$j} </job>
         <avgincome> {avg($p/income)} </avgincome>
      </group>
   else ()
```

## 3.10 - RAGGRUPPAMENTI IN SQL 92:

### 3.10.1 - GROUP BY:

La funzione GROUP BY è stata aggiunta ad SQL poiché funzioni aggreganti (come SUM) restituiscono un'aggregazione di tutti i valori delle colonne ogni volta che sono invocate, e senza GROUP BY sarebbe impossibile avere la somma di ogni singolo gruppo di valori di colonne.

GROUP BY combina righe simili, producendo una singola riga di risultato per ogni gruppo di righe che hanno lo stesso valore per tutte le colonne elencate nella lista di SELECT.

#### Sintassi BNF:

```
[90] <group by clause> ::=
    GROUP BY <grouping column reference list>

[91] <grouping column reference list> ::=
    <grouping column reference>
        [ { <comma> <grouping column reference> }... ]

[92] <grouping column reference> ::=
    <column reference> [ <collate clause> ]
```

#### Esempio 45: clausola "group by" in SQL 92

Selezionare il costo totale di ogni ordine.

```
SELECT num_ordine, SUM(costo)
FROM Ordini
GROUP BY num_ordine
```

### 3.10.2 - HAVING:

La parola chiave WHERE non potrebbe essere utilizzata con funzioni aggreganti (come SUM) senza la clausola HAVING.
HAVING setta condizioni sui raggruppamenti, e può essere utilizzata anche senza GROUP BY, così come si può far uso di GROUP BY senza HAVING.
L'effetto di HAVING sui raggruppamenti è simile a quello di WHERE sulle righe singole.

#### Sintassi BNF:

```
[93] <having clause> ::= HAVING <search condition>

[94] <search condition> ::=
      <boolean term>
    | <search condition> OR <boolean term>
```

Ogni condizione HAVING compara una colonna o un'espressione aggregata del gruppo con un'altra espressione aggregata del gruppo o con una costante.

Un esempio verrà fornito nel paragrafo successivo.

## 3.11 - RAGGRUPPAMENTI IN XQUERY 1.0:

In XQuery non esiste una parola chiave che abbia la stessa funzione che svolge GROUP BY in SQL.
Nonostante ciò si riescono a tradurre in XQuery query SQL che prevedano l'uso della clausola GROUP BY, e si possono applicare funzioni di aggregazione come "count()" o "avg()", mediante le parole chiave let e order by.

### Esempio 46: confronto fra raggruppamenti con "let" e "order by" in XQuery 1.0 e "group by" in SQL 92

La query seguente mostra i numeri e il prezzo medio degli articoli che hanno come minimo tre fornitori.

```
for $pn in distinct-values(document("catalog.xml")//partno)
let $i := document("catalog.xml")//item[partno = $pn]
where count($i) >= 3
order by $pn
return
   <well-supplied-item>
      <partno> {$pn} </partno>
      <avgprice> {avg($i/price)} </avgprice>
   </well-supplied-item>
```

La corrispondente query in SQL è:

```
SELECT pno, avg(price)
FROM Catalogs
GROUP BY pno
HAVING count(*) >=3
```

Si deve notare che $pn della clausola for rappresenta un singolo numero di articolo, mentre $i della clausola let rappresenta un set di elementi da utilizzare come argomento per le funzioni di aggregazione count() e avg().

Per effettuare raggruppamenti in base a più di un valore si utilizza lo stesso metodo:

### Esempio 47: raggruppamenti in base a più valori in XQuery 1.0

La seguente query XQuery 1.0 produce una lista in base ai guadagni medi per ogni combinazione di stato e professione(8):

```
for $s in distinct-values(document("census.xml")//state),
    $j in distinct-values(document("census.xml")//job)
let $p := document("census.xml")//person[state = $s and job = $j]
order by $s, $j
return
   if (exists($p)) then
      <group>
         <state> {$s} </state>
         <job> {$j} </job>
         <avgincome> {avg($p/income)} </avgincome>
      </group>
   else ()
```

L'espressione if-then-else (di cui si parlerà in seguito) previene la formazione di gruppi che non contengono dati (se per esempio in uno stato nessuno svolge una particolare professione).

## 3.12 - FIRST E "[ ]":

In un costrutto SELECT si può includere la proposizione FIRST n, per specificare che la query deve restituire le prime n righe che soddisfano le condizioni di SELECT.
Si può quindi utilizzare per indicare il numero massimo di righe che si vogliono ottenere, magari per testare una query che restituirebbe un numero molto grande di righe, oppure quando si vuole semplicemente conoscere il nome di tutte le colonne e i tipi di dati che esse contengono.

Questo tipo di espressione non è da confondere con l'espressione di XPath ripresa da XQuery e vista in precedenza, che permette invece di interrogare *l'ordine effettivo* con cui i dati si trovano all'interno di un documento, espressa mediante la simbologia " [ n ] ", dove " n " indica il numero d'ordine con cui compare l'elemento ricercato.
Infatti in SQL non c'è modo di tenere traccia dell'ordine in cui sono memorizzati i record all'interno delle tabelle: FIRST n non si riferisce alle prime n tuple in base all'ordine di una tabella, ma alle prime n tuple riportate dal database server. Per avere un risultato preciso nel costrutto SELECT deve essere contenuta la clausola ORDER BY, altrimenti è il database server che decide quale sia l'ordine delle tuple che compongono il risultato.

### Esempio 48: operatori "[" e "]" in XQuery 1.0

Selezionare gli elementi contenuti nel SECONDO libro del documento libri.xml.

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

Infine è da sottolineare che non si può ricorrere a FIRST quando il costrutto SELECT si trova in una sottoquery.

## 3.13 - ESPRESSIONI ARITMETICHE:

In SQL 92 una espressione aritmetica contiene almeno uno degli “operatori aritmetici” elencati nella seguente tabella e il risultato è di tipo numerico.

Sintassi BNF di espressioni aritmetiche in SQL 92:

```
[95] <numeric value expression> ::=
       <term>
     | <numeric value expression> <plus sign> <term>
     | <numeric value expression> <minus sign> <term>
      | <numeric value expression> <mult sign> <term>
      | <numeric value expression> <div sign> <term>

[96] <plus sign> ::= +

[97] <minus sign> ::= -

[98] <mult sign> ::= *

[99] <div sign> ::= /
```

La sintassi Basic EBNF di espressioni aritmetiche in XQuery 1.0 è la seguente:

```
[100] AdditiveExpr ::= MultiplicativeExpr ( ("+" | "-")
MultiplicativeExpr )*

[101] MultiplicativeExpr ::= UnaryExpr ( ("*" | "div" | "idiv" |
"mod") UnaryExpr )*

[102] UnaryExpr ::= ("-" | "+")* UnionExpr
```

L’operatore binario di sottrazione per essere considerato operatore aritmetico deve essere preceduto da spazio (es. a - b), altrimenti viene interpretato in modo diverso (esempio a-b viene interpretato come nome).

## 3.14 - ESPRESSIONI CONDIZIONALI (IF…THEN…ELSE) IN XQUERY 1.0:

Le espressioni condizionali in XQuery sono utili quando la struttura dell’informazione che deve essere restituita dipende da alcune condizioni; queste condizioni possono essere espresse mediante le parole chiave if, then e else. L’espressione if contiene la condizione, then indica cosa fare se la condizione viene soddisfatta, ed else cosa fare se invece la condizione non viene soddisfatta.

In SQL 92 non esiste un’espressione corrispondente.

Sintassi Basic EBNF:

```
[103] IfExpr ::= "if" "(" Expr ")" "then" ExprSingle "else"
ExprSingle
```

Esempio 49: espressione condizionale “if then else” in XQuery 1.0 (I)

Restituire fra due oggetti quello con minor prezzo unitario:

```
if ($oggetto1/prezzounitario < $oggetto2/prezzounitario)
 then $oggetto1
  else $oggetto2
```

Esempio 50: espressione condizionale “if then else” in XQuery 1.0 (II)

In questo esempio viene invece testata la presenza dell’attributo “scontato” (senza valutarne il valore):

```
if ($articolo/@scontato)
 then $articolo/ingrosso
  else $articolo/dettaglio
```

## 3.15 - JOIN IN SQL 92:

Può capitare di dover selezionare dati da due diverse tabelle per avere un risultato completo: per questo scopo l’SQL prevede diversi tipi di istruzioni join, a seconda dei dati che si desiderano ricavare dalle tabelle.

Le tabelle nei database si relazionano con le chiavi: una chiave primaria è una colonna della tabella che ha valori diversi l’uno dall’altro per ogni riga: il fatto che ogni riga abbia un valore differente permette di distinguerla da tutte le altre.

Un join si utilizza per selezionare dati da due tabelle diverse che abbiano una o più colonne in comune; in SQL il join si esprime nella clausola FROM, elencando le tabelle da unire separate da virgola.

Sintassi:

```
[104] <joined table> ::=
      <cross join>
     | <qualified join>
     | <left paren> <joined table> <right paren>

[105] <cross join> ::=
    <table reference> CROSS JOIN <table reference>

[106] <qualified join> ::=
    <table reference> [ NATURAL ] [ <join type> ] JOIN
```

```
      <table reference> [ <join specification> ]

[107] <join type> ::=
      INNER
    | <outer join type> [ OUTER ]
    | UNION

[108] <outer join type> ::=
      LEFT
    | RIGHT
    | FULL

[109] <join specification> ::=
      <join condition>
    | <named columns join>

[110] <join condition> ::= ON <search condition>
```

Se non viene posta nessuna condizione al join, si crea un prodotto cartesiano fra le due tabelle, ovvero si generano tutte le possibili combinazioni fra le righe delle tabelle; ovviamente questo non è di molta utilità.

### Esempio 51: prodotto cartesiano in SQL 92

Selezionare tutte le combinazioni di tuple delle tabelle Customer e State.

```
SELECT *
FROM Customer, State
```

Se la ricerca deve fornire un risultato più preciso, allora è necessario includere una clausola WHERE, in cui vanno espresse le condizioni in base a cui selezionare certe righe del risultato del prodotto cartesiano, e scartarne altre.

### Esempio 52: join con condizione in SQL 92

Selezionare chi ha ordinato un prodotto e che cosa ha ordinato.

```
SELECT Employees.Name, Orders.Product
FROM Employees, Orders
WHERE Employees.Employee_ID=Orders.Employee_ID
```

## 3.16 - JOIN IN XQUERY 1.0:

Così come in SQL, anche in XQuery si possono esprimere i join, molto importanti per combinare dati di diversa provenienza.
Il modo di costruire un join nei due linguaggi è abbastanza simile; infatti data la corrispondenza delle clausole FROM e in, per effettuare un join fra due documenti in XQuery 1.0 bisognerà indicare i documenti nella clausola in (e associarli con un for a due distinte variabili).

L'effetto di un join in XQuery 1.0 è il medesimo di quello di un join in SQL 92.

Sintassi di un join senza condizione (prodotto cartesiano): (non in sintassi Basic EBNF)

```
for variabile1 in documento1
     variabile2 in documento2
return risultato
```

# 3.17 - DIVERSI TIPI DI JOIN:

Naturalmente in SQL 92 esistono diversi tipi di join, e sono tutti associativi, cioè non importa l'ordine in cui i termini su cui si effettua il join vengono disposti nella clausola "where".

## 1.17.1 - INNER JOIN IN SQL 92:

Sintassi (non BNF):

```
SELECT colonna1, colonna2...
FROM Tabella1, Tabella2
WHERE condizione di uguaglianza
```

Un inner-join si basa quindi sull'uguaglianza fra valori, indicata dal simbolo di uguaglianza " = " che deve essere presente nella condizione ON del join oppure all'interno della clausola WHERE, come mostra l'esempio:

Esempio 53: inner join in SQL 92

Selezionare i manufatti e il loro numero di stock (qualora nel database il nome e il codice dei manufatti fossero nella tabella "Manufact" e il loro numero di stock fosse memorizzato nella tabella "Stock")(6).

```
SELECT manu_code, manu_name, stock_number
FROM Manufact [INNER] JOIN Stock
ON (Manufact.manu_code = Stock.manu_code)
```

La medesima query potrebbe essere espressa nel seguente modo per ragioni di semplicità:

Esempio 54: inner join semplificato in SQL 92

```
SELECT manu_code, manu_name, stock_number
FROM Manufact, Stock
WHERE Manufact.manu_code = Stock.manu_code
```

In questo esempio(6) vengono unite due tabelle (“Manufact” e “Stock”), e nel risultato vengono incluse solo le colonne “manu_code”, “manu_name” e “stock_number”, per le quali i valori di “manu_code” sono uguali. L’effetto di un inner-join è quindi quello di generare da due tabelle che abbiano una o più colonne in comune una unica tabella che contiene le tuple che nelle due tabelle hanno l’attributo (o gli attributi) specificati nella condizione del join.

### 1.17.2 - INNER JOIN IN XQUERY:

#### Sintassi (non in Basic EBNF):

```
for variabile1 in documento1
     variabile2 in documento2
where condizione di uguaglianza
return risultato
```

#### Esempio 55: inner join in XQuery 1.0

Traducendo in XQuery lo stesso esempio dell’inner-join in SQL:

```
for $m in document("Manufacts.xml")/Articles/Manufacts/Manufact,
     $s in document("Stocks.xml")/Stocks/Stock,
where $m/manu_code = $s/manu_code
return <answer>
     {$m/manu_code}
     {$m/manu_name}
     {$s/stock_number}
           </answer>
```

### 1.17.3 - SELF JOIN IN SQL 92:

Un join si può effettuare anche per unire una tabella a se stessa; questo può rivelarsi utile qualora ci fosse la necessità di comparare valori di una colonna con altri valori della stessa colonna.
Per creare un self join bisogna effettuare un inner join elencando due volte la stessa tabella nell’espressione FROM, e assegnarla ogni volta ad un alias differente; per il resto un self join si comporta come gli altri join.

#### Esempio 56: self join in SQL 92 (I)

Rilevare valori duplicati della colonna “rowid” (6).

```
SELECT x.rowid, x.customer_num
FROM Cust_calls x, Cust_calls y
WHERE x.customer_num = y.customer_num
```

```
AND x.rowid != y.rowid
```

La condizione

```
x.rowid != y.rowid
```

permette di distinguere due righe differenti.

Vediamo ora un esempio leggermente più complesso del precedente:

### Esempio 57: self join in SQL 92 (II)

Ricercare coppie di ordini per cui “ship_weight” differisce di un fattore di cinque o più e “ship_date” non è nulla, ordinando il risultato in base a “ship_date”(6).

```
SELECT x.order_num, x.ship_weight, x.ship_date,
          y.order_num, y.ship_weight, y.ship_date
FROM orders x, orders y
WHERE x.ship_weight >= 5 * y.ship_weight
AND x.ship_date IS NOT NULL
AND y.ship_date IS NOT NULL
ORDER BY x.ship_date
```

## 1.17.4 - SELF JOIN IN XQUERY 1.0:

In XQuery un self-join può essere espresso mediante assegnamento a due variabili differenti dello stesso path.

### Esempio 58: self join in XQuery 1.0

Per ottenere lo stesso risultato della query dell’esempio visto in SQL 92:

```
for $c1 in document("Customers.xml")/cust_calls,
     $c2 in document("Customer.xml")/cust_calls,
where $c1/customer_num = $c2/customer_num
     and $c1/rowid = $c2/rowid
return <answer>
     {$c1/rowid}
     {$c1/customer_num}
      </answer>
```

## 3.17.5 - (LEFT | RIGHT) OUTER JOIN IN SQL 92:

Mentre i tipi di join finora analizzati trattano le due tabelle allo stesso modo, un outer-join le tratta asimmetricamente: un outer-join infatti fa sì che una tabella sia dominante (tabella “preserved”) sull’altra.

Se si ha un LEFT- outer-join nel risultato vengono preservate tutte le tuple della prima tabella (nell'ordine di apparizione nella clausola FROM), anche se non hanno corrispondenza, in base alla condizione espressa, con le tuple della seconda; le tuple della prima tabella che non hanno corrispondenza verranno completate con valori NULL.
Nel caso di un RIGHT-outer-join invece vengono preservate tutte le tuple della seconda colonna.

Un outer join perciò deve avere una clausola SELECT, una clausola FROM e una clausola WHERE.

## Esempio 59: confronto fra inner join e outer join in SQL 92

Confrontiamo un inner-join con un outer-join per evidenziare la differenza fra i due (6):

Inner-join:

```
SELECT c.c_cod, c.city, s.s_cod, s.city
FROM Customer c, Suppliers s
WHERE c.city = s.city
```

Risultato atteso (da un eventuale database):

```
c_cod     c0101
city      New York
s_cod     s20
city      New York

c_cod     c2073
city      Los Angeles
s_cod     s16
city      Los Angeles
```

Con un Outer-join:

```
SELECT c.c_cod, c.city, s.s_cod, s.city
FROM Customer c OUTER Suppliers s
WHERE c.city = s.city
```

Risultato atteso:

```
c_cod     c0101
city      New York
s_cod     s20
city      New York

c_cod     c0275
city      New York
s_cod     s19
city      null

c_cod     c1009
```

```
city      Los Angeles
s_cod     s16
city      null

c_cod     c2073
city      Los Angeles
s_cod     s16
city      Los Angeles
```

Come si può vedere sono state incluse nel risultato anche i campi che non avevano corrispondenze (campi null).

### 3.17.6 - (LEFT | RIGHT) OUTER JOIN IN XQUERY 1.0:

Anche in XQuery è possibile esprimere l'outer-join; mentre però i sistemi relazionali in genere restituiscono valori nulli, una query XQuery può rappresentare i dati mancanti con un elemento vuoto o con l'assenza dell'elemento.
La seguente query è un esempio di outer-join in XQuery, che ritorna i nomi di tutti i fornitori (suppliers) in ordine alfabetico, includendo quelli che non forniscono nessun articolo. Internamente a ciascun elemento <supplier> sono elencate le descrizioni degli articoli, in ordine alfabetico.

### Esempio 60: outer join in XQuery 1.0

```
for $s in document("suppliers.xml")//supplier
order by $s/suppname
return
  <supplier>
     {
      $s/suppname,
      for $i in document("catalog.xml")//item[suppno = $s/suppno],
           $p in document("parts.xml")//part[partno = $i/pno]
        order by $p/description
        return $p/description
      }
   </supplier>
```

Come si può notare al primo colpo d'occhio, la rappresentazione di un outer join in XQuery 1.0 (9) è molto più complessa di quella in SQL 92.

### 3.17.7 - FULL OUTER JOIN IN SQL 92:

Nel caso di un full-join, nel risultato vengono incluse tutte le tuple delle due tabelle considerate, e le mancate corrispondenze vengono completate con valori NULL.

### 3.17.8 - FULL OUTER JOIN IN XQUERY 1.0:

In XQuery il risultato di un full-outer-join può essere rappresentato in alcuni modi diversi; nell'esempio sottostante sono riportati gli articoli innestati nei fornitori, seguiti da una lista di articoli che non hanno fornitori e una lista di fornitori che non riforniscono alcun pezzo.

#### Esempio 61: full outer join in XQuery 1.0

```
<master_list>
    (for $s in document("s.xml")//s_tuple
     return
        <supplier>
           $s/sname,
           for $sp in document("sp.xml")//sp_tuple
                   [sno = $s/sno],
               $p in document("p.xml")//p_tuple
                   [pno = $sp/pno]
           return
              <part>
                 $p/descrip,
                 $sp/price
              </part> sortby(descrip)
        </supplier> sortby(sname)
    )

 union

    (: parts that have no supplier :)

    <orphan_parts>
       for $p in document("p.xml")//p_tuple
       where empty(document("sp.xml")//sp_tuple
             [pno = $p/pno] )
       return $p/descrip sortby(.)
    </orphan_parts>
</master_list>
```

L'operatore union che in questa query(9) è utilizzato per combinare due liste ordinate, ritorna la prima lista con la seconda aggiunta in append.

### 3.17.9 - JOIN MULTIPLI IN SQL 92:

È possibile inoltre eseguire join su più di due tabelle, separandole con virgole nella clausola FROM.

#### Esempio 62: join multiplo in SQL 92

Ricercare tutti i nomi di fornitori e la descrizione delle parti che rifornisce:

```
SELECT sname, descrip
FROM S, P, SP
WHERE S.sno=SP.sno AND P.pno=SP.pno
```

### 3.17.10 - JOIN MULTIPLI IN XQUERY 1.0:

Anche in XQuery si possono effettuare join multipli, indicando questa volta i documenti di cui si vuole effettuare il join nella clausola for, ognuno preceduto da una diversa variabile a cui assegnare gli elementi indicati nel path che segue il nome del documento.

#### Esempio 63: join multiplo in XQuery 1.0

Lo stesso esempio visto in SQL 92 assume la forma:

```
for $sp in document("sp.xml")//sp_tuple,
  $p in document("p.xml")//p_tuple[pno=$sp/pno],
  $s in document("s.xml")//s_tuple[sno=$sp/sno]
return
  <sp_pair>
    $s/sname, $p/descrip
  </sp_pair>
```

## 3.18 - FUNZIONI IN SQL 92:

SQL 92 ha parecchie funzioni predefinite per il conteggio e il calcolo ma, al contrario di XQuery non prevede la possibilità per l'utente di definire proprie funzioni, sebbene altre forme di SQL (come ad esempio quella di Informix) dispongano di una sintassi apposita. Questa lacuna può però almeno in parte essere colmata mediante l'uso di "viste" grazie al comando CREATE VIEW, come vedremo successivamente.
Una funzione contenuta in una espressione viene valutata per ogni riga della query e tutte le espressioni con funzioni richiedono argomenti.

### 3.18.1 - TIPI DI FUNZIONI:

Ci sono diversi tipi e categorie di funzioni in SQL:
funzioni di aggregazione e funzioni scalari: le prime si applicano a gruppi di valori e restituiscono un singolo valore, le altre si applicano a valori singoli e restituiscono valori singoli.

## 3.18.2 - FUNZIONI DI AGGREGAZIONE :

AVG
COUNT
MAX
MIN
RANGE
STDEV
SUM
VARIANCE

Le funzioni di aggregazione restituiscono un unico valore per un set di righe di una query in genere risultanti da un'espressione WHERE di un costrutto SELECT; in assenza di WHERE le funzioni di aggregazione agiscono su tutti i valori delle righe ottenute da FROM.

AVG(colonna) ritorna il valore medio di una colonna

COUNT(colonna) ritorna il numero di righe (senza valori NULL) di una colonna

COUNT(*) ritorna il numero delle righe selezionate

COUNT =(DISTINCT colonna) ritorna il numero di risultati differenti

MAX(colonna) e MIN(colonna) restituiscono rispettivamente il valore massimo e il valore minimo di una colonna

SUM(colonna) ritorna la somma totale dei valori di una colonna.

## 3.18.3 - FUNZIONI SCALARI:

### 3.18.3.a FUNZIONI DI TEMPO:

DAY
MDY
MONTH
WEEKDAY
YEAR

Le funzioni di tempo possono essere usate sia nei costrutti SELECT, sia nelle espressioni WHERE. Queste funzioni restituiscono un valore che corrisponde alle espressioni o agli argomenti che sono usati per invocare la funzione.

Inoltre

CURRENT

restituisce un valore contenente data e orario corrente, ed

EXTEND

modifica la precisione con cui vengono restituiti i valori di DATE e DATETIME.

### 3.18.3.b - FUNZIONI DI CONVERSIONE DI DATE:

Le seguenti funzioni convertono date in caratteri e viceversa:

DATE
TO_CHAR
TO_DATE

### 3.18.3.c - FUNZIONI CHE AGISCONO SUI CARATTERI:

Queste funzioni convertono stringhe di caratteri minuscole in maiuscole, e viceversa:

LOWER
UPPER

### 3.18.4 - LE VISTE IN SQL 92:

Nel linguaggio SQL 92 viene data la possibilità di definire delle viste, cioè tabelle virtuali che permettono di semplificare query particolarmente complesse o permettono di effettuare query che richiedono l'applicazione successiva di funzioni aggregate, non permesse perché potrebbero generare effetti imprevedibili.
Per questo le viste sono in parte paragonabili a funzioni definibili dall'utente, sebbene non possano essere definite tali.
Quelli che in una funzione sarebbero i parametri in ingresso, nelle viste sono contenuti nelle tabelle presenti nel database, e sono ottenibili da una normale query con clausola FROM (ed eventualmente WHERE), mentre i parametri in uscita, ovvero il risultato prodotto dalla vista, sono generati dalla clausola SELECT.

## Sintassi BNF:

```
[111] <view definition> ::= CREATE VIEW <table name>
     [ <left paren> <view column list> <right paren> ]
     AS <query expression>
     [ WITH [ <levels clause> ] CHECK OPTION ]

[112] <view column list> ::= <column name list>

[113] <levels clause> ::=
    CASCADED | LOCAL
```

### Esempio 64: semplificazione di query mediante l’uso di una vista

Una query complessa del tipo:
ricercare il codice del reparto che ha effettuato il maggior numero di ordini

```
SELECT codice_reparto
FROM Ordini
GROUP BY codice_reparto
HAVING COUNT(*) >= ALL (SELECT COUNT(*)
                        FROM Ordini
                        GROUP BY codice_reparto)
```

può essere semplificata generando una vista (di cui ci si potrà servire anche in altre query!!!)

```
CREATE VIEW V1(codice_reparto, NOrdini)
     AS SELECT codice_reparto, COUNT(ordine)
     FROM Ordini
     GROUP BY codice_reparto
```

e utilizzandola nella query nel seguente modo:

```
SELECT codice_reparto
FROM Ordini
GROUP BY codice_reparto
WHERE  NOrdini = (SELECT MAX(NOrdini)
                   FROM V1)
```

## 3.19 - FUNZIONI IN XQUERY 1.0:

XQuery1.0 prevede una libreria di funzioni predefinite e, al contrario di SQL 92 permette inoltre all’utente di definire le proprie, garantendo maggiore flessibilità nell’uso del linguaggio e maggiore semplicità in caso di query particolarmente complesse. Una funzione può avere in ingresso zero o più parametri.

### Sintassi Basic EBNF di chiamata di funzione:

```
[114] FunctionCall ::= QName "(" (ExprSingle("," ExprSingle)*)? ")"
```

### 3.19.1 - FUNZIONI PREDEFINITE IN XQUERY 1.0:

### 3.19.1.a - FUNZIONI DI AGGREGAZIONE:

Le librerie di funzioni di XQuery comprendono tutte le funzioni di aggregazione di SQL (come avg, sum, count…) e altre utili funzioni, come ad esempio le funzioni già viste:

| | |
|---|---|
| `distinct-values` | che elimina i duplicati da un elenco. |
| `empty` | ritorna vero se e solo se ha per argomento un elenco vuoto. |

### 3.19.1.b - FUNZIONI DI TEMPO:

Come SQL 92 XQuery 1.0 prevede funzioni di tempo:

| | |
|---|---|
| `current-date` | restituisce la data corrente |
| `current-time` | restituisce l'orario corrente |
| `current-dateTime` | restituisce data e orario correnti |

### 3.19.1.c - FUNZIONI BOOLEANE:

| | |
|---|---|
| `boolean` | restituisce l'effettivo valore booleano di un elemento |
| `true` | rappresenta il valore booleano "true" |
| `false` | rappresenta il valore booleano "false" |

### 3.19.1.d - FUNZIONI DI ERRORE:

XQuery 1.0 prevede anche funzioni built-in di errore:

`error`

### 3.19.1.e - FUNZIONI SULLE STRINGHE:

| | |
|---|---|
| `string` | estrae il valore di stringa di un nodo |

### 3.19.1.f - ALTRE FUNZIONI:

| | |
|---|---|
| `unordered` | data una sequenza di elementi li restituisce senza un ordine preciso |
| `document` | data una stringa contenente un URI riferito ad un documento XML, restituisce un nodo documento che contiene la rappresentazione dati di quel documento |

| | |
|---|---|
| `collection` | restituisce i nodi trovati in una collezione identificata da una stringa contenente un URI |

### 3.19.2 - FUNZIONI DEFINIBILI DALL'UTENTE:

La definizione di una funzione deve specificare il nome della funzione e il nome dei suoi parametri; opzionalmente può specificare il tipo dei parametri e il tipo del risultato. Inoltre è prevista la definizione del corpo della funzione, racchiuso fra parentesi graffe; quando la funzione viene invocata, gli argomenti della funzione vengono assegnati ai parametri e il corpo della funzione viene eseguito per produrre il risultato.
Se non viene specificato il tipo di un parametro, questo accetta valori di ogni tipo, così come se non viene specificato il tipo del valore risultante, la funzione può restituire un risultato di qualunque tipo.

#### Sintassi Basic EBNF di dichiarazione di una funzione:

```
[115] FunctionDecl ::= "declare" "function" QName "(" ParamList?
(")" | (")" "as" SequenceType)) (EnclosedExpr | "external")

[116] ParamList ::= Param ("," Param)*

[117] Param ::= "$" VarName TypeDeclaration?

[118] TypeDeclaration ::= "as" SequenceType
```

L'esempio successivo definisce una funzione chiamata "highbid" che accetta un elemento nodo come parametro e restituisce un valore decimale; la funzione ha il compito di avere come parametro un articolo, di estrarre il suo numero, poi di trovare e restituire la maggiore offerta ("bid-amount") che sia mai stata registrata per il numero di quell'articolo.

#### Esempio 65: dichiarazione di funzione in XQuery 1.0 (I)

```
declare function highbid(element $item)
returns decimal
{
max(document("bids.xml")
//bid[itemno _ $item/itemno]/bid-amount)
}
highbid(document("items.xml")
```

Di seguito viene riportata una possibile chiamata di questa funzione su un articolo di numero 1234:

```
//item[itemno _ "1234"])
```

Un'ulteriore esempio di funzione definita dall'utente potrebbe essere il seguente:

## Esempio 66: dichiarazione di funzione in XQuery 1.0 (II)

Funzione che permette di testare (restituendo true o false) se un nodo preceda un altro nodo in un documento (senza essere un suo nodo genitore) (8).

```
declare function local:precedes($a as node(), $b as node())
   as boolean
   {
      $a << $b
        and
      empty($a//node() intersect $b)
   };
```

Nella definizione della funzione i tipi indicati per gli argomenti e per il risultato possono essere semplici, come ad esempio decimali, oppure complessi, come elementi e attributi.

XQuery 1.0 non consente un eccesso di funzioni definite dall'utente: due funzioni non possono infatti avere lo stesso nome e lo stesso numero di parametri in ingresso. Tuttavia alcune delle funzioni predefinite in XQuery sono sovraccaricate: per esempio la funzione sulle "string" può convertire un argomento di quasi qualunque tipo in una stringa.

In XQuery è possibile inoltre invocare una funzione su una lista di parametri i cui tipi non corrispondano esattamente ai tipi dichiarati nella funzione.
Per questo è stata stabilita una gerarchia di tipi primitivi; una parte di questa gerarchia ad esempio potrebbe essere:

```
integer -> decimal -> float -> double
```

ad indicare che una funzione con un parametro dichiarato float potrebbe essere invocata con un argomento intero, convertendo l'intero in un float.

Una funzione può essere invocata ricorsivamente.

## Esempio 67: dichiarazione di funzione ricorsiva in XQuery 1.0

Trovare il massimo livello di profondità del documento "partlist.xml"(8):

```
NAMESPACE xsd = "http://www.w3.org/2000/10/XMLSchema-datatypes"

function depth(ELEMENT $e) returns xsd:integer
{
   (: Un elemento vuoto ha profondità 1 :)
   (: Altrimenti, aggiungi 1 alla massima profondità del figlio :)
if empty($e/*) then 1
   else max(depth($e/*)) + 1
}

depth(document("partlist.xml"))
```

## 3.20 - COMMENTI IN XQUERY 1.0:

In XQuery si possono introdurre commenti nel testo della query, al fine di renderla maggiormente comprensibile. Un commento è una porzione di query che non viene valutata, perciò viene distinto dalle altre espressioni racchiudendolo con parentesi tonde e due punti, come mostrato di seguito:

```
(:  text  :)
```

## 3.21 - COMMENTI IN SQL 92:

In SQL 92 all'interno di una query i commenti invece si indicano preceduti da due trattini:

```
-- text
```

# 4 - QUADRO SINOTTICO RIASSUNTIVO:

Il quadro sottostante ha la funzione di riportare schematicamente il confronto fra le parole chiave del linguaggio SQL 92 e del linguaggio XQuery 1.0. Nei casi in cui non esiste corrispondenza di una parola chiave nell'altro linguaggio, nel quadro viene lasciata la casella vuota.

| SQL 92 | pag. | XQuery 1.0 | pag. |
|---|---|---|---|
| SELECT | 23 | return | 29 |
| FROM | 23 | in + for | 27 |
| | - | let | 27 |
| WHERE | 29 | where | 29 |
| =, <>, >, <, >=, <=, AND, NOT, OR | 30 | =, <>, >, <, >=, <=, and, not, or | 30 |
| BETWEEN…AND | 32 | | - |
| LIKE | 31 | contains | 31 |
| IS NULL | 33 | empty | 33 |
| EXISTS | 42 | exists | 42 |
| | - | some \| every … in … satisfies | 34 |
| | - | <<, >> | 36 |
| ORDER BY (ASC \| DESC) | 36 | order by (ascending \| descending) | 38 |
| DISTINCT | 39 | distinct values | 39 |
| UNION | 40 | union | 40 |
| IN \| NOT IN | 41 | intersect \| except | 41 |
| GROUP BY | 43 | let + order by | 44 |
| HAVING | 43 | | - |
| FIRST | 45 | | - |
| | - | if … then … else | 46 |
| AVG(), SUM(), COUNT() | 56 | avg(), sum(), count() | 59 |
| CURRENT() | 56 | current() | 59 |

# 5 - CONCLUSIONI:

Nei capitoli precedenti si è cercato di mettere a confronto due diversi linguaggi di interrogazione dati, l'SQL 92 e l'XQuery 1.0, cercando di ricavare il maggior numero di somiglianze e differenze fra essi, per poter fornire un aiuto a chi stesse cercando orientamento per scegliere l'uno o l'altro linguaggio, o per chi, avendo già esperienza in SQL 92 desiderasse cimentarsi con il nuovo XQuery.

Dal confronto è emerso che XQuery, considerato che è ancora alla sua prima versione, ha già dimostrato ottime potenzialità di poter competere con il linguaggio SQL nell'interrogazione di database e documenti; personalmente non ho riscontrato grandi difficoltà nel confrontare i due linguaggi di interrogazione, dal momento che essi presentano numerose affinità, sia nella scelta delle parole chiave (ad esempio "where", "exists" e "order by" sono parole chiave di notevole importanza, che hanno lo stesso nome e il medesimo significato in entrambi i linguaggi) che nella struttura generale delle query (ad eccezione delle clausole "SELECT" e "return" e di "FROM" e "in", che come visto in precedenza occupano posizioni differenti all'interno delle query).
L'unica nota negativa del linguaggio XQuery 1.0 che è emersa dal confronto con l'SQL è la sua maggiore prolissità; infatti riportando uno fra gli esempi di query messe a confronto in precedenza salta subito all'occhio come la query in SQL sia più sintetica di quella XQuery, pur generando il medesimo risultato:

Esempio in SQL 92:

```
SELECT pno, avg(price)
FROM Catalogs
GROUP BY pno
HAVING count(*) >=3
```

La traduzione in XQuery 1.0:

```
for $pn in distinct-values(document("catalog.xml")//partno)
let $i := document("catalog.xml")//item[partno = $pn]
where count($i) >= 3
order by $pn
return
   <well-supplied-item>
      <partno> {$pn} </partno>
      <avgprice> {avg($i/price)} </avgprice>
   </well-supplied-item>
```

A questo si può però aggiungere che la query in XQuery 1.0 potrebbe rivelarsi più leggibile, in quanto vengono indicati chiaramente il tipo di risultato generato e la sua struttura.

Poiché quindi SQL 92 e XQuery 1.0 mostrano parecchie somiglianze, ed SQL è considerato dai più un linguaggio piuttosto semplice ed efficace, si può certamente affermare che pur essendo una novità, XQuery 1.0 non rappresenterà un problema per chi ha già esperienza con l'SQL o per chi non ha mai fatto uso di linguaggi di interrogazione.
Inoltre XQuery garantisce la possibilità di interrogare dati provenienti da fonti differenti, e di superare la rigida struttura di tabelle e relazioni del modello di dati relazionale.

Molte case distributrici di software hanno già colto il potenziale di questo nuovo linguaggio di interrogazione; la Microsoft ha infatti già messo in programma di fornire assieme all'ultima versione di SQL Server (denominata Yukon) un supporto per XQuery, e sia IBM che Oracle hanno espresso l'intenzione di effettuare scelte analoghe.

Il possibile successo di XQuery 1.0 sarà principalmente riconducibile all'ampia diffusione che potrebbe avere il linguaggio XML in futuro, molto probabile dal momento che XML sembra dimostrare di avere la capacità di esprimere documenti eterogenei con sufficiente semplicità; inoltre il fatto di essere un linguaggio estensibile pone XML in netto vantaggio rispetto al linguaggio HTML, che diviene incapace di adattarsi alle nuove richieste di gestione di fonti multiple di dati.
La necessità di espandere le capacità di HTML ha infatti spinto i produttori di browser a introdurre nuovi tag nella sintassi, generando ulteriori problemi: una pagina HTML che sfrutta marcatori proprietari non può essere visualizzata correttamente se non con il browser adatto, con le ovvie conseguenze che seguono.
XML sembra quindi essere al giorno d'oggi lo strumento più idoneo allo scambio di dati e alla costruzione di pagine Web, spianando la strada ad XQuery; in quanto tecnologia del W3C, XML è gratuito, e non lega ad un singolo fornitore, per cui pur non essendo in ogni situazione la soluzione ottimale, è comunque consigliabile prenderlo in considerazione.
Per di più Microsoft SQL Server 2000 offre la possibilità mediante SQLXML di estrarre dati da un database SQL Server formattato come XML, utilizzando il protocollo HTTP.

# 6 - NOTAZIONE BNF E BASIC EBNF:

Nelle pagine precedenti si è fatto ricorso alla notazione BNF (Backus-Naur Form) introdotta da John Backus e Peter Naur, per indicare con precisione grammaticale la sintassi delle espressioni del linguaggio SQL, e alla Basic EBNF (Basic Extended BNF) per descrivere il linguaggio XQuery.
La notazione Basic EBNF (5) prevede che, ad eccezione di casi segnalati, gli spazi bianchi non abbiano significato. I simboli non terminali (che richiedono cioè ulteriori livelli di approfondimento) sono sottolineati e il testo letterale è racchiuso da virgolette ( " ).

## Simboli utilizzati:

, indicatore di sequenza
| indicatore di alternativa

## Indicatori di occorrenza:

* indica zero o più occorrenze (da 0 a n)
+ indica una o più occorrenze (da 1 a n)
? indica occorrenza opzionale (0 o 1)

Nella notazione BNF (10) invece i simboli non terminali sono racchiusi da " < " e " > ", mentre come indicatori di occorrenza opzionale si utilizzano le parentesi quadre ("[ " e " ]"); i simboli terminali sono invece indicati senza nessuna particolare notazione.

# BIBLIOGRAFIA:

(1) “Progetto di Basi di Dati Relazionali”, Beneventano, Bergamaschi, Vincini, Pitagora Ed., Bologna 2000

(2) “XML Path Language (XPath)”, James Clark, Steve De Rose, Nov. 16, 1999

http://www.w3.org/TR/xpath

(3) “XML e Dati Semistrutturati”, Francesco Guerra

(4) “XML and Databases”, Ronald Bourret, Jan. 2003

http://www.rpbourret.com/xml/XMLAndDatabases.htm

(5) “XQuery1.0: An XML Query Language”, Don Chamberlin, Mary F. Fernàndez, Daniela Florescu, Jonathan Robie, Jérome Siméon, Scott Boag, Aug. 22, 2003

http://www.w3.org/2003/08/DIFF-xquery

(6) “Informix – Guide to SQL: Tutorial, version 9.1”, Diana Chase, Brian Deutscher, Geeta Karmarkar, Jennifer Leland, March, 1997

http://elib.cs.berkeley.edu/admin/informix/doc/answers/pubs/pdf/3856.pdf

(7) “XML Query Requirements”, Don Chamberlin, Peter Fankhauser, Massimo Marchiori, Jonathan Robie, Feb. 15, 2001

http://www.w3.org/TR/xmlquery-req

(8) “XML Query Use Cases”, Don Chamberlin, Daniela Florescu, Peter Fankhauser, Massimo Marchiori, Jonathan Robie, Nov. 12, 2003

http://www.w3.org/TR/xquery-use-cases

(9) “Extensible Markup Language (XML) 1.0 (Second Edition) – W3C Recommendation”, Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, Oct. 6, 2000

http://www.w3.org/TR/REC-xml

(10) SQL92 (SQL-2) Syntax Reference, Aug. 27, 1992

http://www.contrib.andrew.cmu.edu/~shadow/sql/sql2bnf.aug92.txt