

Davide Mauri

SQL Server 2005 XML

Davide Mauri
16/07/2007

SQL Server 2005 introduce il pieno supporto per XML. Rispetto a quanto accadeva con la precedente versione di SQL Server, dove XML veniva gestito come semplice testo, e quindi si perdevano tutte le caratteristiche fondamentali dello stesso, come ad esempio il supporto per la validazione nei confronti di un schema o la possibilità di effettuare query XPath, con SQL Server 2005 il supporto per XML, invece, è totale e completo.

XML è e rimane XML, e questo accade grazie all'introduzione del nuovo tipo di dato *xml*.

Il tipo di dato xml può essere utilizzato ovunque: come tipo di dato di una variabile, come colonna, come parametro di una stored procedure oppure di una UDF.

Grazie a questo nuovo tipo di dato è ora possibile manipolare XML in modo nativo - utilizzando gli standard XPath ed XQuery, ma anche le estensioni XML-DML, sempre rimanendo all'interno di SQL Server.

Ma che relazione c'è tra XML ed un database relazionale? Le implicazioni dell'utilizzo di XML in un database relazionale, e le considerazioni relative al suo utilizzo nel mondo reale sono molte e di grande importanza, in quanto XML ed il mondo relazione fanno letteralmente a pugni ☺ tra di loro.

Per fare più chiarezza possibile, in quest'articolo affronteremo l'argomento XML in due modalità. Nella prima parte verrà definito in modo distaccato e puramente tecnico come avviene e cosa permette di fare l'integrazione tra XML e SQL Server. Questa prima parte ha l'obiettivo di avvicinare all'utilizzo di XML tutti coloro che ancora non l'hanno provato e – magari – fanno ancora abbondante uso lunghe stringhe comma-delimited (o qualche-carattere-strano-delimited) all'interno dei loro database per gestire situazioni apparentemente complesse, in cui il modello relazionale *sembra* andare stretto.

Nella seconda ed ultima parte, invece, valuteremo in modo più specifico – tenendo conto anche degli impatti architetturali – di quando e come usare XML all'interno di una soluzione, definendo con precisione i campi di applicazione di XML all'interno di un RDBMS.

XML Data Type

Il tipo di dato *xml* permette di memorizzare documenti fino a 2GB (ovviamente se avete documenti di XML di questa dimensione è *fortemente probabile* che ci sia qualche problema nella soluzione che state adottando...☺). SQL Server supporta sia documenti XML che frammenti XML e la memorizzazione viene sempre fatta come UTF-16 utilizzando un formato binario interno. All'atto pratico, però e per fortuna, XML è manipolabile come se fosse una stringa unicode:

```
declare @xmlData xml;
set @xmlData = N'<libri><libro titolo="Il quinto giorno" autore="Schätzing Frank"></libro></libri>';
```

La conversione da testo ad XML, infatti, è implicita. Ovviamente tale conversione per andare a buon fine prevede che l'XML sia almeno *well-formed*.

Nel caso si desiderasse utilizzare xml in una tabella e possibile farlo in questo modo:

```
create table dbo.TestXml
(
    id int identity not null primary key,
    product_code char(8) not null,
```

```
) product_data xml not null
```

mentre se invece serve utilizzarlo come parametro di una stored procedure (cosa molto utile!) lo si deve fare in questo modo:

```
create procedure dbo.stp_SaveOrder
@order xml
as
...
```

Typed & Untyped XML

Come ben sapete XML supporta l'utilizzo di XML Schema per la definizione della "grammatica" del documento XML stesso.

E' possibile usare XML Schema anche in SQL Server 2005, associando una variabile (o colonna o parametro) xml ad uno o più XML Schema. Un oggetto xml di questo tipo prende il nome di Typed XML.

Per farlo è necessario definire lo schema e salvarlo in una *xml schema collection*:

```
create xml schema collection dbo.simple_schema as
'<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="urn:person" xmlns:ps="urn:person"
elementFormDefault="qualified">
  <xs:complexType name="personType">
    <xs:attribute name="name" type="xs:string" />
    <xs:attribute name="surname" type="xs:string" />
  </xs:complexType>
  <xs:complexType name="peopleType">
    <xs:sequence>
      <xs:element name="person" type="ps:personType"/>
      <xs:element name="group" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="people" type="ps:peopleType">
  </xs:element>
</xs:schema>'
go
```

fatto ciò è possibile dichiarare un oggetto di tipo xml a cui viene associato lo schema appena creato:

```
declare @product xml(dbo.simple_schema)
```

a questo punto sarà possibile memorizzare un documento XML all'interno della variabile @product se e solo se aderente allo schema. Questi comandi (equivalenti, e che mostrano come poter specificare in modo esplicito o meno l'utilizzo del namespace associato allo schema da usare) quindi funzioneranno perfettamente:

```
declare @product xml(dbo.simple_schema)
set @product = '<Product
xmlns="urn:simple"><Name>Mouse</Name><Value>30</Value></Product>'
select @product
```

```
set @product = '<s:Product
xmlns:s="urn:simple"><s:Name>Mouse</s:Name><s:Value>30</s:Value></s:Product>'
select @product
```

mentre questo invece darà un errore:

```
declare @product xml(dbo.simple_schema)
set @product = '<Product
xmlns="urn:simple"><Name>Mouse</Name><Value>Trenta</Value></Product>'
select @product
```

da notare che l'errore è molto dettagliato e fornisce anche la query XQuery da utilizzare per identificare il nodo contenente l'errore:

```
Msg 6926, Level 16, State 1, Line 2
XML Validation: Invalid simple type value: 'Trenta'. Location:
/*:Product[1]/*:Value[1]
```

FOR XML

Una delle prime modalità di integrazione tra SQL Server e XML è la produzione di XML. Sin dalla versione 2000 era possibile restituire il risultato di una query non sottoforma di classico rowset, ma come documento o frammento XML. Per farlo è sufficiente utilizzare la clausola FOR XML all'interno del comando SELECT.

FOR XML ha diverse modalità di funzionamento, ognuna delle quali permette di avere un controllo più o meno completo della forma che l'XML di output deve assumere. La più semplice e la più immediata – e quella che però da meno possibilità di creare un XML che sia conforme ad uno schema imposto a priori – è la modalità AUTO:

```
select
    [ProductID],
    [Name],
    [ProductNumber]
from
    [SalesLT].[Product] as Product
for xml auto, root('Products')
```

che produce un documento XML di questo tipo:

```
<Products>
  <Product ProductID="680" Name="HL Road Frame - Black, 58" ProductNumber="FR-
R92B-58" />
  <Product ProductID="706" Name="HL Road Frame - Red, 58" ProductNumber="FR-
R92R-58" />
  [...]
</Products>
```

L'opzione aggiuntiva "root" permette la definizione dell'elemento root del documento XML. E' possibile ometterla ed in tal caso si otterrà un frammento XML, in quanto non sarà presente l'elemento radice.

Se è necessario rispettare un certo schema la modalità AUTO non è la modalità migliore da scegliere, in quanto non permette di definire cosa deve essere rappresentato come attributo e cosa come elemento. Con FOR XML PATH (introdotto in SQL Server 2005), invece questa possibilità è prevista:

```
select
    [ProductID] as "@ProductId",
    [Name],
    [ProductNumber]
from
    [SalesLT].[Product]
for xml path('Product'), root('Products')
```

ed il risultato è qualcosa come:

```
<Products>
  <Product ProductId="680">
    <Name>HL Road Frame - Black, 58</Name>
    <ProductNumber>FR-R92B-58</ProductNumber>
  </Product>
  <Product ProductId="706">
    <Name>HL Road Frame - Red, 58</Name>
    <ProductNumber>FR-R92R-58</ProductNumber>
  </Product>
  [...]
</Products>
```

In pratica, si assegnano alle colonne dei nomi che rappresentano, tramite una query XPath, la posizione e la forma dell'elemento XML che i valori delle stesse devono assumere.

In entrambi i casi – AUTO e PATH – se la query che vogliamo restituire sottoforma di XML contiene una (o più) clausole di JOIN, SQL Server creerà un XML i cui elementi sono relazionati in modo gerarchico così da definire anche nel documento la relazione di parentela presente tra le tabelle in join:

```
select
    Category.[Name],
    Product.[ProductID],
    Product.[Name],
    Product.[ProductNumber]
from
    [SalesLT].[ProductCategory] Category
inner join
    [SalesLT].[Product] Product on [Category].[ProductCategoryID] =
[Product].[ProductCategoryID]
for xml auto, root('Categories')
```

ed il risultato è:

```
<Categories>
  <Category Name="Mountain Bikes">
    <Product ProductID="771" Name="Mountain-100 Silver, 38" ProductNumber="BK-
M82S-38" />
    [...]
  </Category>
  <Category Name="Touring Bikes">
```

```

    <Product ProductID="978" Name="Touring-3000 Blue, 44" ProductNumber="BK-
T18U-44" />
    [...]
</Category>
[... ]
</Categories>

```

Oltre a queste modalità ve ne sono altre due – RAW ed EXPLICIT – di cui non tratteremo in questo articolo (altrimenti solo per la modalità EXPLICIT avremmo bisogno di diverse pagine...), e che cito semplicemente a titolo di completezza. La prima è la modalità “grezza” dove non è praticamente possibile specificare in alcun modo la forma dell’XML in output, mentre invece la seconda è una modalità dove è possibile definire in modo estremamente preciso (ma anche molto verboso e complesso) l’esatta forma che l’XML deve avere.

Una volta prodotto il documento XML voluto è possibile – ovviamente – inviarlo all’applicazione in modo che lo possa utilizzare come tale. Questo approccio mi è stato più volte utile per integrare in modo molto semplice e veloce soluzioni web basate su Flash (tipico esempio il menu di navigazione di un sito), ed in generale ovunque si presenti la necessità – lato applicativo – di gestire dati in forma gerarchica.

Per quanto riguarda, invece, l’utilizzo di FOR XML all’interno di SQL Server, è possibile assegnare il risultato della query ad un documento XML:

```

declare @xmlDemo xml

set @xmlDemo = (
    select
        Category.[Name],
        Product.[ProductID],
        Product.[Name],
        Product.[ProductNumber]
    from
        [SalesLT].[ProductCategory] Category
    inner join
        [SalesLT].[Product] Product on [Category].[ProductCategoryID] =
[Product].[ProductCategoryID]
    for xml auto, root('Categories')
)

select @xmlDemo

```

In questo modo possiamo manipolare il documento XML in modo ancora più potente utilizzando i metodi che vedremo nel paragrafo successivo.

XML Data Type Methods

Ogni oggetto di tipo xml supporta l’utilizzo di cinque metodi per la manipolazione dell’XML che esso contiene. I metodi in questione sono:

- exist()
- value()
- query()

- nodes()
- modify()

Tutti questi metodi accettano come parametro una query XQuery attraverso la quale si definiscono le clausole di ricerca all'interno del documento XML. L'utilizzo è molto semplice:

```
select @xmlDemo.exist('/Categories/Category[@Name eq "Mountain Bikes"]')
```

com'è possibile notare il metodo va invocato esattamente nello stesso modo in cui viene invocato il metodo di un oggetto in linguaggio di programmazioni tipo C# o VB.NET o Java, ed il parametro da passare (in questo caso) è una stringa contenente una XQuery. E' bene notare che XQuery è un'evoluzione di XPath e ne supporta appieno la sintassi, quindi è possibile utilizzare indifferentemente l'uno o l'altro metodo per effettuare query all'interno di XML.

NOTA:

XQuery è un linguaggio molto più complesso ed articolato di XPath e nel caso non conosciate perfettamente l'argomento vi invito a consultare i Book Online di SQL Server, cercando nell'indice la voce "XQuery, about XQuery" (se avete bisogno anche di informazioni anche su XPath cercate "XPath queries [SQLXML], about XPath queries"). Troverete tutta la documentazione necessaria per far vostre queste tecnologie.

Prima di passare all'analisi dei vari metodi associati al tipo xml, prepariamo e popoliamo una piccola tabella di esempio:

```
create xml schema collection dbo.person_schema as
'<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="urn:person" xmlns:ps="urn:person"
elementFormDefault="qualified">
  <xs:complexType name="personType">
    <xs:attribute name="name" type="xs:string" />
    <xs:attribute name="surname" type="xs:string" />
  </xs:complexType>
  <xs:complexType name="peopleType">
    <xs:sequence>
      <xs:element name="person" type="ps:personType" />
      <xs:element name="group" type="xs:string" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
  <xs:element name="people" type="ps:peopleType">
  </xs:element>
</xs:schema>'
go

create table dbo.xmltab
(
  id int primary key,
  person xml(dbo.person_schema)
)
go
```

```

-- insert some values
insert into dbo.xmltab (id, person)
values (1, '<people xmlns="urn:person"><person name="Eric"/></people>')

insert into dbo.xmltab (id, person)
values (2, '<people xmlns="urn:person"><person name="Bruce"/></people>')

insert into dbo.xmltab (id, person)
values (3, '<people xmlns="urn:person"><person name="James"/></people>')

insert into dbo.xmltab (id, person)
values (4, '<people xmlns="urn:person"><person name="Dave"/></people>')

insert into dbo.xmltab (id, person)
values (5, '<people xmlns="urn:person"><person name="Kay"/></people>')

insert into dbo.xmltab (id, person)
values (6, '<people xmlns="urn:person"><person name="Andy"/></people><people
xmlns="urn:person"><person name="Micheal"/></people>')
go

```

exist()

Il metodo exists permette di verificare l'esistenza di un elemento identificato dalla query XQuery passata come parametro. Usando la tabella appena creata, possiamo verificare se in una riga della stessa è presente un documento XML nel cui contenuto esiste un elemento "person" per cui l'attributo "name" è uguale a "Dave":

```

with xmlnamespaces(default 'urn:person')
select * from dbo.xmltab
where
    person.exist('//person[@name="Dave"]') = 1

```

Il risultato è un valore booleano che indica o meno l'esistenza dell'elemento cercato.

Attenzione che in XML tutto è case sensitive, quindi la clausola

```
@name="dave"
```

non avrebbe invece portato a nessun risultato.

Da notare la clausola "with xmlnamespaces" che permette di definire il namespace di default e/o i namespace ed i relativi prefissi utilizzabili nella query. La definizione dei namespace in uso è fondamentale quando si usa un xml tipizzato: senza la dichiarazione dei namespace utilizzati, infatti, avremmo un errore come questo

```

Msg 2260, Level 16, State 1, Line 1
XQuery [xmltab.person.exist()]: There is no element named 'person'

```

In quanto verrebbe effettuata la ricerca dell'elemento "person" non appartenente ad alcun namespace.

value()

Il metodo value() permette di estrarre valori da XML e trasformarli in valori scalari utilizzabili in modo usuale:

```
with xmlnamespaces(default 'urn:person')
select
    person.value('/people/person/@name', 'varchar(50)')
from
    dbo.xmltab
```

In pratica il primo parametro del metodo value() rappresenta la query XQuery tramite la quale SQL Server identifica i valori che ci interessano, ed il secondo parametro rappresenta invece il tipo di dato di SQL Server nel quale convertire il valore prelevato dal documento xml.

La query d'esempio però, se eseguita, vi restituirà un errore abbastanza criptico:

```
XQuery [dbo.xmltab.person.value()]: 'value()' requires a singleton (or empty sequence), found operand of type 'xs:string *'
```

L'errore sta ad indicarci che una XQuery come quella utilizzata non necessariamente restituisce uno ed un solo valore per ogni riga. Ed infatti, nella tabella di esempio, la riga con ID = 6 restituisce due nodi in quanto ci sono due elementi "person" all'interno del documento XML.

Ovviamente "due" mal si concilia con "uno ed uno solo" ☺. E' quindi necessario specificare che di ogni elemento vogliamo prendere solo il primo:

```
with xmlnamespaces(default 'urn:person')
select
    person.value('/people[1]/person[1]/@name', 'varchar(50)')
from
    dbo.xmltab
go
```

oppure, usando una forma più compatta, la query può essere anche scritta così:

```
with xmlnamespaces(default 'urn:person')
select
    person.value('(//people/person/@name)[1]', 'varchar(50)')
from
    dbo.xmltab
go
```

ora il risultato è corretto:

	(No column name)
1	Eric
2	Bruce
3	James
4	Dave
5	Kay
6	Andy

Come si può vedere abbiamo appena ottenuto un risultato la cui forma è quella canonica di un RDBMS, e quindi può essere tranquillamente utilizzato con ogni altra funzionalità che normalmente è applicabile sulle tabelle fisiche o virtuali (viste, subquery, cte, udf).

query()

Il metodo `query()` permette di estrarre dati da documenti xml producendo altro xml, eventualmente con una forma diversa:

```
with xmlnamespaces('urn:person' as ps)
select
    person.query('
        for $p in //ps:person
        return element singer {
            data($p/@name)
        }
    ')
from
    xmltab
```

in questo caso l'utilizzo delle funzionalità offerte da XQuery rendono piuttosto semplice e la definizione di un XML di destinazione. Il risultato della query è il seguente:

	(No column name)
1	<singer>Eric</singer>
2	<singer>Bruce</singer>
3	<singer>James</singer>
4	<singer>Dave</singer>
5	<singer>Kay</singer>
6	<singer>Andy</singer><singer>Micheal</singer>

nodes()

Questo potente metodo permette di trasformare nodi xml in righe di una tabella. La sua invocazione è quindi simile a quella di una Table-Valued User Defined Function. I dati che restituisce però, sono ancora frammenti XML e devono quindi essere gestiti con i metodi visti fino ad ora.

Per semplificare le cose, utilizziamo una variabile xml come esempio, nella quale mettiamo lo stesso contenuto della riga con ID = 6 della tabella di esempio:

```
declare @x xml(dbo.person_schema);
set @x = N'<people xmlns="urn:person"><person name="Andy"/></people><people
xmlns="urn:person"><person name="Micheal"/></people>';

with xmlnamespaces('urn:person' as ps)
select T.c.query('.') from @x.nodes('//ps:person') T(c)
```

com'è possibile notare il metodo nodes() viene invocato nella clausola *from* del comando *select*, ed è inoltre necessario dare un nome alla tabella risultate ("T" nell'esempio) ed anche un nome alla colonna della tabella stessa ("c" nell'esempio). Diventa così possibile utilizzare la colonna "c" della tabella "T" all'interno della select list. Dato che la colonna "c" è xml, è possibile utilizzare il metodo query() per vedere l'xml in essa contenuto.

Questo metodo diventa quindi *indispensabile*, quando si vogliono estrarre – nel nostro esempio – tutti i nomi delle persone memorizzate nei documenti xml memorizzati nella tabella.

Con la clausola query() questo non è stato possibile, infatti, se ritornate a guardare l'immagine con il risultato della query, vedrete che il risultato contiene solo sei nomi, e non sette come invece dovrebbero essere, visto che la riga con ID = 6 contiene un documento xml nella quale è presente più di un nome.

La clausola nodes() usata in congiunzione con la clausola CROSS APPLY permette di ottenere il risultato voluto:

```
with xmlnamespaces(default 'urn:person')
select
    T.c.value('@name', 'varchar(50)')
from
    dbo.xmltab x
cross apply
    x.person.nodes('//person') T(c)
```

ed ecco il risultato corretto:

	(No column name)
1	Eric
2	Bruce
3	James
4	Dave
5	Kay
6	Andy
7	Micheal

modify()

Questo metodo è l'unico che non può essere usato in una clausola SELECT ma deve essere utilizzato in un comando UPDATE (nel caso di tabelle) o in un comando SET (nel caso di variabili):

```
-- update value
with namespaces(default 'urn:person')
update
    dbo.xmltab
set
    person.modify('insert attribute surname {"Dickinson"} into
(/people[1]/person[1])')
where
    id = 2
```

Il motivo è legato allo scopo di questo metodo, ossia la modifica del contenuto del documento XML. Per farlo è necessario inoltre utilizzare il comandi DML di XQuery, e visto che l'argomento è abbastanza ampio, è stato creato un paragrafo ad hoc, che potete trovare subito al termine di questa riga. 😊

XQuery DML

La query XQuery utilizzata come parametro del metodo modify() può modificare l'xml sulla quale opera andando ad effettuare operazioni DML (Insert, Update e Delete) di qualsiasi nodo presente nel documento.

Per inserire un nuovo attributo è possibile usare la clausola *insert attribute*:

```
with namespaces(default 'urn:person')
update
    dbo.xmltab
set
    person.modify('insert attribute surname {"Dickinson"} into
(/people[1]/person[1])')
where
    id = 2
```

nel caso di elementi, è possibile anche specificare la posizione degli stessi (*before*, *after* o *into*) rispetto ad un altro elemento:

```

with xmlnamespaces('urn:person' as ps)
update
    dbo.xmltab
set
    person.modify('insert <ps:group>Unknown</ps:group> into (/ps:people[1])')
where
    id = 2
go

```

la modifica di un nodo avviene con il comando *replace value of*:

```

with xmlnamespaces(default 'urn:person')
update
    dbo.xmltab
set
    person.modify('replace value of (/people[1]/group[1]) with "Iron Maiden"')
where
    id = 2
go

```

e per finire, è possibile eliminare dei nodi con l'istruzione *delete*:

```

with xmlnamespaces(default 'urn:person')
update
    xmltab
set
    person.modify('delete (/people/group)[1]')
where
    id = 2

```

Come al solito, nel caso di argomenti così ampi, vi rimando ai Books Online di SQL Server, in particolare alla voce dell'indice "modify() method".

XML Indexes

XML, seppur supportato nativamente da SQL Server 2005, non è certo l'ambiente migliore nella quale un database relazionale si trova ad operare.

Ogni qualvolta viene eseguita una query su XML, il Query Optimizer di SQL Server 2005 deve decomporre i dati XML sulla quale dovrà andare ad operare in formato relazionale, in modo tale da poter utilizzare gli operatori relazionali di cui è a disposizione e per cui è stato progettato ed ottimizzato e quindi poter risolvere la query stessa.

Potete ben immaginare come questo compito possa essere molto dispendioso! Analizzando il piano di esecuzione di una query semplicissima

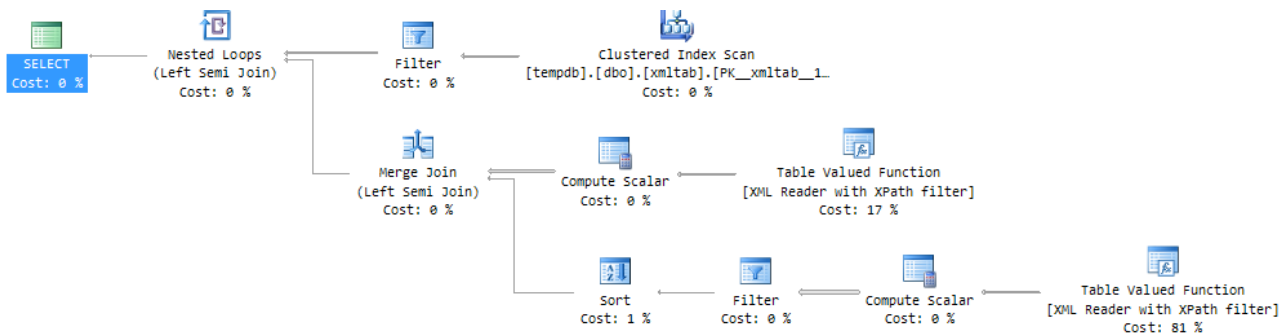
```

with xmlnamespaces(default 'urn:person')
select
    id
from
    dbo.xmltab
where

```

```
person.exist('/people/person[@name="Dave"]') = 1
```

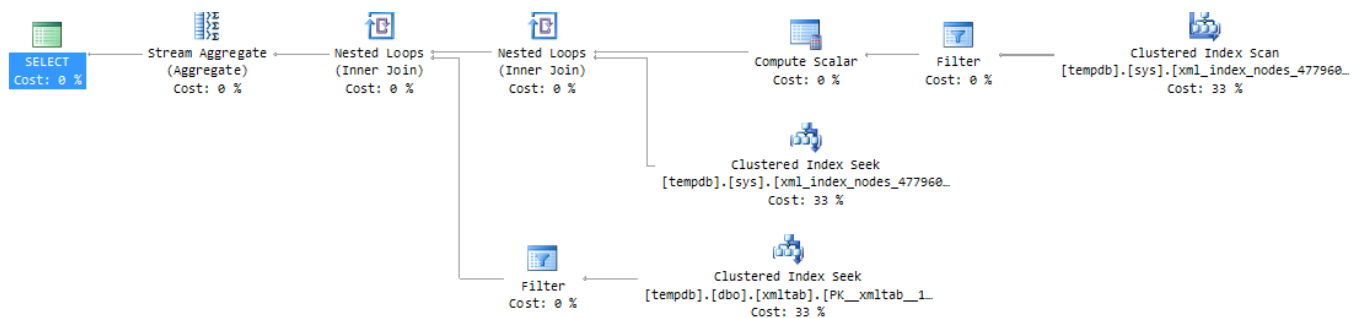
si ha infatti la conferma della cosa:



Con la creazione di un indice XML

```
create primary xml index PXI_xmltab on dbo.xmltab(person)
```

Le cose migliorano decisamente:



Il piano di esecuzione rimane complesso, ma se non altro ci sono dei performanti Index Seek che aiutano ad avere performance accettabili.

Oltre all'indice XML primario è possibile creare altri tipi di indici secondari, in modo tale da supportare meglio le query di ricerca all'interno di XML.

Una trattazione approfondita dell'argomento la potete trovare nei Books Online alla voce dell'indice "XML [SQL Server], indexes".

Quando usare XML

E' bene fare pochi giri di parole e chiarire subito che nel maggior parte dei casi, memorizzare un documento o un frammento di XML all'interno di un database non serve ed, anzi, è assolutamente controproducente. Diciamo pure che nel 99% dei casi l'utilizzo di xml come tipo di dato per la memorizzazione dei dati nelle colonne di una tabella è inutile, deleterio ed ingiustificato.

Se state pensando di poter fare una tabella tipo

```
create table dbo.TabellaErrata
(
    id int identity(1,1) not null primary key,
```

```
    tipo varchar(20) not null,  
    valore xml not null  
)
```

così da “risolvere” tutti i problemi che incontrate quando dovete aggiungere nuove colonne per gestire nuovi attributi...beh state sbagliando tutto. ☺

Ricordate che come regola generale non dovrete MAI e poi MAI (e poi ancora MAI) modificare lo schema delle vostre tabelle aggiungendo colonne al volo (o peggio ancora a run-time) per gestire informazioni aggiuntive non previste (a questo proposito vi invito a leggere l’articolo sulla modellazione “SQL Server 2005 Developer Guidelines - Parte 2”), pena un aumento considerevole della complessità del codice necessario per gestire il tutto, una enorme difficoltà nell’implementazione dell’integrità dei dati, e un notevole impatto negativo sulle performance.

L’utilizzo di XML sembrerebbe risolvere tutti e tre questi problemi....ma non è vero.

La complessità del codice aumenta ugualmente, richiedendo l’utilizzo di XQuery per la manipolazione dei dati, che è – come avrete potuto notare – sicuramente più complesso che semplice codice T-SQL. La complessità aumenta anche a livello di manutenzione, in quanto è necessario utilizzare indici XML per avere performance decenti. Performance che sono comunque notevolmente più basse rispetto a quelle ottenibili con i normali strumenti relazionali.

Per ciò che concerne l’integrità dei dati, in effetti, XML è molto ma molto più flessibile che una tabella relazionale. Ma questo è un problema! Immaginatevi di avere una colonna che deve contenere documenti XML rappresentanti degli ordini. Come potete essere sicuri che effettivamente tutte le righe abbiano per questa colonna un documento XML che sia effettivamente un ordine? La risposta sta nell’utilizzo di un XML Schema. Ma un XML Schema non rende forse più “rigida” la struttura di un documento XML, definendo esattamente cosa può contenere e come devono essere fatti i dati per esser considerati validi?

Ne esce quindi una considerazione: la “rigidità” dello schema di un database non è allora un male, ma anzi è una cosa voluta e desiderabile, tanto che anche XML, attraverso un XML Schema, cerca di farla propria. Il motivo è molto semplice: uno schema (identificando con questo termine in modo indifferente lo schema di un database o un XML Schema) garantisce una conformità dei dati contenuti, permettendo agli sviluppatori di poter scrivere codice molto più semplice, senza doversi preoccupare di dover validare ogni singolo dato con la quale devono lavorare. A voler fare un paragone è un po’ come la differenza che passa tra un linguaggio di programmazione tipizzato rispetto ad uno non tipizzato. Quest’ultimo sembra molto più produttivo all’inizio, in quanto meno rigido e rigoroso, ma si rivela disastroso nel giro di poco tempo, quando il codice si fa complesso, quando è necessario capire perché un algoritmo sembra funzionare correttamente, ma non dà il risultato voluto...

Come avrete capito la rigidità di una struttura (che non deve essere eccessiva, ovviamente) non è un difetto ma un pregio ricercato; una caratteristica che ci garantisce la stabilità e la prevedibilità del nostro sistema.

In che area si colloca, quindi, quell’1% dei casi dove invece l’utilizzo di XML è lecito e consigliabile?

XML è utilissimo in due principali casi.

Il primo caso è legato alla memorizzazione di un documento XML come oggetto “atomico”. La memorizzazione di documenti Word, Excel o altri documenti salvati in modo nativo in formato XML. Grazie

al supporto ad xml è sempre possibile effettuare ricerche in questi documenti, utilizzando il metodo exist() o values(). E' ovviamente consigliabile salvare le informazioni più utili per la ricerca in formato relazionale, così da non aver problemi di performance; attenzione in questo caso a mettere in piedi l'infrastruttura necessaria (stored procedure e/o trigger) per il mantenimento dell'integrità dei dati così duplicati.

Il secondo caso è invece legato a quelle situazioni in cui non è effettivamente possibile conoscere a priori la struttura dei dati. In questo caso, al posto di mettere in piedi tutta l'architettura necessaria per implementare un modello open-schema sul database, si può valutare l'utilizzo di XML come supporto per la memorizzazione d'informazioni destrutturate o parzialmente strutturate. Un tipico esempio possono essere le schede tecniche dei prodotti di un sito e-commerce, oppure delle informazioni personalizzate di un sistema CRM o Help-Desk. Un altro esempio di dove XML sarebbe stato ideale è la tabella dei profili del database di asp.net.

Tutto queste considerazioni valgono per quanto riguarda l'utilizzo di XML come colonna e quindi come supporto alla memorizzazione d'informazioni. L'utilizzo del tipo di dato xml come parametro di stored procedure o funzioni, invece, è molto più libero ed ha molte meno controindicazioni. Esso permette infatti il passaggio tra applicazioni e database di strutture di dati complesse, come ad esempio array o collection, in modo molto elegante ed di facile gestione (per un esempio pratico, fare riferimento al tip "Passare un array di dati ad una stored procedure").

Il supporto ad xml è un fattore fondamentale nei database della generazione corrente, in quanto è uno standard *de-facto*, ed è pertanto ottimo che SQL Server 2005 ne dia un supporto così ampio, così che, nel momento in cui verifica l'esigenza di farne uso, il supporto da parte del database server è completo e nativo, evitando agli sviluppatore ed i DBA l'inventarsi di soluzioni complesse ed arzigogolate. Attenzione solo a non abusarne, come tutte le cose deve essere utilizzato quando è necessario.

Conclusioni

In quest'articolo abbiamo visto come il supporto ad XML sia davvero esteso e completo. XML può essere un grande alleato se utilizzato in modo corretto e ponderato, ma può anche tramutarsi in un grosso problema se sfruttato in modo eccessivo e senza riguardo rispetto alle funzionalità di un RDBMS.

Utilizzatelo quindi senza paura ma senza eccessi, consci della grande flessibilità ma anche che l'alta complessità dei piani di esecuzione anche per semplicissime query ha un impatto pesante sulle prestazioni, e sicuramente le vostre soluzioni ne beneficeranno parecchio, sia intermini di manutenibilità, ma anche di performances e facilità di sviluppo.

Riferimenti:

Database di esempio AdventureWorksLT

<http://www.codeplex.com/MSFTDBProdSamples>

XML Support in SQL Server 2005:

<http://msdn2.microsoft.com/en-us/library/ms345117.aspx>

“SQL Server 2005 Developer Guidelines - Parte 2”

<http://www.microsoft.com/italy/msdn/risorsemsdn/sql/sql2.mspix>

“Passare un array di dati ad una stored procedure”

<http://www.microsoft.com/italy/msdn/risorsemsdn/community/tips/0612.mspix#E3B>

Davide Mauri è Mentor e Socio Fondatore di Solid Quality Learning Italia, società specializzata nella consulenza e nella formazione su .NET, SQL Server 2005 e tecnologie ad essi collegate.

Nel 2006 è stato insignito del premio MVP Award per SQL Server da Microsoft per le sue conoscenze e il suo supporto alla community di SQL Server.

Oltre alla consulenza si dedica alla formazione ed alla divulgazione sempre in ambito .NET e SQL Server 2005, erogando corsi e partecipando come speaker a conferenze nazionali ed internazionali.

Dal 2006 coordina tutte le attività di UGISS, il primo e più importante User Group Italiano di SQL Server, e dal 1° gennaio 2007 ha assunto la carica di presidente.

Per contattarlo:

www.ugiss.org

www.davidemauri.it

dmauri@solidq.com