Полуструктурированная информация

Базы данных являются основой всех современных информационных систем и до сих пор СУБД неплохо справлялись с этой функцией. Но с уничтожением информационных границ возникают новые виды информации, которые требуют обработки. СУБД работают с реляционной моделью, т.е., с данными, хорошо структурированными. В современный информационный оборот включается неструктурированная информация (большие тексты, мультимедийные файлы) и, что более интересно - полуструктурированная.

Информацию можно считать полуструктурированной, если она удовлетворяет хотя бы одному из следующих условий

информация имеет структуру, но эта структура различна в различных источниках; при интеграции информации из разных источников приведение их к единой структуре может оказаться дорогостоящим (случай, типичный для федеративных баз данных: одни и те же по смыслу данные о людях, например в разных источниках имеют разный формат);

структура информации неявна, то есть, не имеет достаточно четкого описания (например, текстовый документ/книга имеет разделы и подразделы, но описания этой структуры нет);

информация структурирована частично, в одном документе имеются как структурированные части, так и неструктурированные (типичный пример - HTML-страница);

структура информации может достаточно часто изменяться (причинами этого может быть. изменение спецификаций, например, внешних требований, необходимость интеграции с новыми источниками и т.д.).

Язык XML

Стандартом де-факто для представления полуструктурированной информации является язык XML. XML-документ представляет собой плоский текст, в котором выполнена разметка тегами. Очень похож на HTML, но если HTML говорит о том, как представлять информацию, то XML дает возможность сообщать о смысле информации.

Свойства XML, отличающие его от HTML приведены ниже, их общий смысл - более строгие синтаксические требования, чем в HTML:

Строгая иерархическая структура документа. Закрывающие теги обязательны (кроме случая тега с пустым телом).

Чувствительность к регистру

Значения аргументов обязательно берутся в кавычки или в апострофы

Обратите внимание: ограничителем символьной строки могут быть. на равных правах кавычки и апострофы, только открывающий и закрывающий ограничители должны быть одинаковы.

Существенно, что XML-документы могут довольно легко восприниматься человеком и легко обрабатываться программно. Пример XML-документа

<order id="7916">

<order\_date>2007-01-08</order\_date>

<ship\_date>2007-03-19</ship\_date>

<item>

<product id="10345"/>

<price>10.20</price>

<quantity>3</quantity>

<total>30.60<total>

</item>

<item>

<product id="10349"/>

<price>1.75</price>

<quantity>100</quantity>

<total>175.00<total>

</item>

<total>205.60<total>

</order>

Значения атрибутов выделены зеленым цветом, а тела тегов - синим.

Базы данных XML

Все расширяющееся применение языка XML породило программные продукты, поддерживающие базы данных XML. В приведенной ниже таблице даны сравнительные характеристики традиционных (реляционных) баз данных (баз данных SQL) и баз данных XML

|  |  |  |
| --- | --- | --- |
| Базы данных SQL | Базы данных XML | Комментарий |
| Реляционная структура (жесткая) | Иерархическая структура (гибкая) | Считаем, что реляционная модель данных, в основе которой лежат таблицы, нам знакома. В основе модели XML лежит дерево - иерархия. О модели данных XML мы еще расскажем позже. |
| Разделение данных и метаданных | Совмещение данных и метаданных | В реляционной базе данных метаданные (т.е., данные, описывающие данные) составляют отдельные от данных таблицы. В XML метаданные (теги) распределены по тексту, т.е., непосредственно сопутствуют самим данным. Объем метаданных в XML значительно больше. |
| Плотное заполнение данными | Сильно разреженные данные | У нас могут отсутствовать какие-то данных (например, мы можем не иметь информации о дате рождения сотрудника). В реляционной базе данных структура жесткая, соответствующее поле все равно предусматривается, в него заносится значение NULL. В XML отсутствующих данных просто нет. Это особенно сказывается, если данные сильно разрежены. |
| Записи не упорядочены, по определению | Порядок записей имеет значение | В реляционной базе данных, если мы хотим как-то упорядочить данные, нам нужно применять ORDER BY, которая не является реляционной операцией. В XML имеется естественная упорядоченность документа (1-я глава предшествует 2-й), в некоторых случаях она очень важна. |
| Язык запросов, нечувствительный к регистру | Язык запросов, чувствительный к регистру |  |

XQuery - стандарт W3C

Как видите, XML-документ может содержать очень много данных, поэтому нужно уметь как-то искать те или иные данные в нем. Для этого консорциумом W3C был создан язык запросов XQuery.

Составной частью XQuery является XPath - язык адресации в XML-документе. XPath с самого начала создавался не как самостоятельная спецификация, а как вспомогательная - для использования в спецификациях XSLT и XPointer. Так была создана версия XPath 1.0.

Разработчики XQuery с самого начала решили использовать язык XPath и в своей спецификации и несколько расширили его для своих нужд. Так возникла версия XPath 2.0.

Официальный статус стандартов W3C XQuery и XPath получили в 2007 году

XPath 1.0 - рекомендация 16.11.1999

XQuery - рекомендация 23.01.2007

XPath 2.0 - рекомендация 23.01.2007

XQ-DM - модель данных Xpath/XQuery

Модель данных XQuery реализует широко известный шаблон проектирования Composite, служащий для представления иерархических структур.

В XQ-DM каждое значение является упорядоченной последовательностью из нуля или более объектов. Объект может быть атомарным значением или узлом. Атомарное значение имеет тип, который является одним из атомарных типов, определенных XML Schema или производным по ограничению от такого типа. Узел имеет тип, который является одним из семи типов, определенных XPath. Узлы имеют идентификацию и определенный в пределах документа порядок следования - порядок документа.

Если документ хранится в базе данных XML, то он представляет из себя один или несколько экземпляров XQ-DM.

Подмножеством данной модели является модель данных XPath 1.0, XQ-DM расширяет модель XPath типизацией атомарных объектов в соответствии с типами XML Schema. XPath 2.0 уже включает в себя типизацию.

Корпорация Кинга - база данных XML

Для целей данной презентации мы перевели данные "Корпорации Кинга" в XML. Ниже графически представлена схема XML-документа, содержащего все денные "Корпорации Кинга".

Голубым цветом на схеме показаны элементы, желтым - атрибуты.

Красные значки показывают, сколько раз данный элемент/атрибут может встречаться:

? - 0 или 1 раз

\* - 0 или более раз

+ - 1 или более раз

если никакого значка нет - один и только один раз

Корпорация Кинга - реляционная база данных

На занятиях по организации баз данных мы используем реляционную базу данных "Корпорация Кинга" и формулируем SQL-запросы к ней.

XPath - язык для адресации частей XML-документа

Мы начнем с рассмотрения языка XPath, который является частью языка XQuery.

Вот основные свойства XPath:

XPath поддерживает числовые выражения, операции равенства и сравнения, и булевы выражения.

В результате обработки выражения получается объект, относящийся к одному из четырех основных типов:

набор узлов - неупорядоченный набор узлов без дубликатов

булево значение - true или false

число - число с плавающей точкой

строка - последовательность символов

Обработка выражений осуществляется, отталкиваясь от контекстного (текущего) узла

Главным выражением XPath является выражение адресации

Выражение адресации

Выражение адресации состоит из последовательности шагов, разделенных символом /, например:

/self::king\_corp/child::departments/

descendant::employee[position()=1]

Выражение адресации очень напоминает полное имя файла в файловой системе (это сделано сознательно). На каждом шагу мы движемся по дереву. Контекстный узел аналогичен текущему каталогу - это тот узел, в котором мы находимся на данном шаге адресации. Все выражения вычисляются относительно контекстного узла.

Но обратите внимание: в отличие от каталогов файловой системы, шаг адресации может выбрать не один, а множество (кортеж в терминах XQuery) узлов. Например, ели мы переходим от узла departments к узлу, department, то мы получим множество узлов.

Шаги адресации состоят из трех частей:

oсь, указывающая дерево взаимоотношений между текущим узлом контекста и узлами, отбираемыми на данном шаге адресации,

правило проверки узлов, указывающее тип и название узлов, отбираемых на данном шаге адресации,

нуль или более предикатов для дальнейшего отсева в наборе узлов, собранных на данном шаге адресации.

Оси адресации

В таблице показаны некоторые возможные оси адресации (XQuery использует не все оси адресации XPath)

 self::name  Сам контекстный узел  .

 child::name  Непосредственный потомок контекстного узла   name

 parent::name  Непосредственный предок контекстного узла   ..

 descendant::name  Все потомки контекстного узла  //name

 descendant-or-self::name   Контекстный узел и все его потомки

 ancestor::name  Все предки контекстного узла

 ancestor-or-self::name  Контекстный узел и все его предки

 attribute::name  Атрибут контекстного узла  @name

Запись осей адресации весьма громоздкая. Но на практике используется сокращенная запись, показанная в правом столбце таблицы. Сокращенную запись имеют самые популярные оси, вы увидите, что во всех наших примерах нам ни разу не пришлось воспользоваться полной записью осей.

  Операции XPath

Вот что можно делать в выражениях (и кое-что еще, что осталось за пределами презентации)

Числовые операции: + - \* div mod

Операции сравнения: = != > >= < <=

Логические операции: and or

Функции:

Числовые (в т.ч. агрегатные)

Строковые

Булевские

Даты и времени

Преобразование типов

Функции над множествами

Подробности мы увидим в примерах.

Прочитать весь документ

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

Просто читаем весь файл king\_corp.xml. С этого текста будут начинаться операторы во всех последующих примерах. Не обращайте пока на него внимания.

Выбрать фамилии и имена всех сотрудников

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

/king\_corp/departments/department

/employees/employee/name

В этом запросе задан полный маршрут от корня документа

В дальнейшем мы будем пользоваться такой записью:

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee/name

Здесь выбираются элементы employee, лежащие на любом уровне, и их потомки name. Результат будет тот же.

Выбрать фамилии и имена всех сотрудников

В предыдущем примере выбрались элементы вместе с их тегами. Чтобы избавится от тегов, применяется функция text():

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee/name/text()

В этом запросе задан полный маршрут от корня документа

Выбрать сумму зарплаты и комиссионных для каждого сотрудника

В том, что мы выбираем, можно использовать выражения и функции.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee/(salary+commission)

Обратите внимание на скобки в этом примере, они обязательны. Это связано с типизацией данных.

В этом примере выбрались только 10 строк, хотя сотрудников 33. Это связано с тем, что не у всех есть комиссионные. Как это исправить, мы рассмотрим позже.

Выбрать, какой процент от зарплаты составляют комиссионные каждого сотрудника

Oперация div означает деление, поскольку применять / нельзя (это разделитель в выражении маршрута). round - одна из функций

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee/round(commission div salary \*100)

Выбрать имена (только имена) всех сотрудников

Здесь и далее демонстрируются некоторые строковые функции.

Функция substring-before выделяет часть строки до первого вхождения заданной подстроки (в нашем случае - до пробела

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee/substring-before(name,' ')

Выбрать фамилии (только фамилии) всех сотрудников

Функция tokenize разбивает тело элемента на части, расположенные между выражением-разделителем.

На любом шаге могут применяться предикаты. Они берутся в квадратные скобки.

В нашем случае разделитель для tokenize - пробел, предикат задает выделение 3-го выделенного этой функцией слова.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee/tokenize(name," ")[3]

Выбрать фамилии и имена всех сотрудников, работающих в Нью-Йорке

Выбираются элементы отделов и оставляются только те, у которых значение вложенного элемента location - "NEW YORK". В оставшихся выбираются имена сотрудников.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//department[location="NEW YORK"]//employee/name/text()

Выбрать фамилии и имена всех сотрудников отдела 12

Здесь в предикате используется значение аргумента

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//department[@id = "12"]

//employee/name/text()

Выбрать номера отделов, в которых больше 3 сотрудников

Функция count возвращает количество вложенных элементов. Для преобразования аргумента в текст используется функция string.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//department/employees[count(employee)>3]

/../string(@id)

Выбрать фамилии и имена всех сотрудников, получающих комиссионные

Эффективным логическим значением непустого элемента является true. Так что, простое указание в предикате имени элемента или атрибута - это проверка его существования. В нашем случае выбираются элементы employee, в которых есть вложенные элемент commission.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//department//employee[commission]

/name/text()

Выбрать фамилии и имена всех сотрудников и их зарплаты

В предыдущем примере мы получили только имена, но, наверное, было бы интересно знать, какие же у них комиссионные. До сих пор мы всегда выбирали только одно значение (столбец). Как выбрать их несколько? Мы используем функцию concat, чтобы представить два значения, как одно.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//department//employee[commission]/

concat(name,' ',commission)

Выбрать те товары, которые связаны с теннисом

В XPath нет функции pos или indexOf. Но функция contains, а также substring-before, substring-after позволяют решать те же задачи.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//product[contains(description,'TENNIS')]

/description/text()

Выбрать продавцов, у которых комиссионные 1000 или больше

В этом примере у нас два предиката, которые применяются последовательно. Несколько предикатов на одном шаге реализуют логику И.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee[function='SALESPERSON'][commission>=1000]

/concat(name,' ',commission)

Тот же результат мы получим в запросе:

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee[function='SALESPERSON' and commission>=1000]

/concat(name,' ',commission)

Выбрать фамилии и имена всех сотрудников, работающих продавцами или имеющих зарплату <1000

Если предикаты объединяются логикой ИЛИ, нужно явно указывать логическую операцию.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee[function='SALESPERSON' or salary<1000]/

concat(name,' ',salary,' - ',function)

Выбрать тех сотрудников, которые работают не в Нью-Йорке

Логическое НЕ обеспечивается функцией, которая инвертирует эффективное логическое значение.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//department[not(location='NEW YORK')]//employee/name

ыбрать номера отделов, в которых нет сотрудников

Еще один пример с логическим НЕ.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//department[not(employees)]/string(@id)

Выбрать фамилии и имена всех сотрудников и номера их отделов

В этом запросе проблема состоит в том, что нам нужно выбрать вместе данные, находящиеся на разных уровнях иерархии: номер отдела является атрибутом элемента department, а имя сотрудника - содержится в элементе name, вложенном в элемент employee, вложенном в элемент employes, вложенном в элемент department.

Как видите из текста запроса, в выражениях и в аргументах функций можно применять выражения маршрута.

В данном случае мы применили "шаг вверх" - ../: мы спустились по дереву до сотрудника. На этом шаге мы имеем кортеж имен сотрудников. От каждого имени мы поднимаемся на 2 уровня вверх по дереву и выбираем номер отдела, в котором этот сотрудник работает.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//department//employee/concat(name,' - ',../../string(@id))

Выбрать фамилии и имена всех сотрудников, работающих клерками или в Далласе

Здесь та же проблема, но, кроме того, на разных уровнях иерархии находятся также те значения, которые нужно проверять. Как видите, выражение маршрута можно применять и в предикатах.

Обратите внимание: выражение маршрута, вложенное в функцию или в предикат, начинается от контекстного (текущего) узла.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee[function='CLERK' or ../../location='DALLAS']/

concat(name,' ',function,' ',../../location)

Выбрать имена и фамилии менеджеров всех отделов, находящихся в Нью-Йорке

В этом примере демонстрируется применение предикатов на разных шагах.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//department[location='NEW YORK']//employee[function='MANAGER']

/concat(name,' ',../../string(@id),' ',../../location)

Выбрать название отдела, в котором работает м-р Кинг

Иллюстрация того, что в предикатах возможны довольно сложные выражения.

В этом примере мы вышли на м-ра Кинга, а потом вернулись вверх по дереву, чтобы выбрать номер отдела.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

// employee[tokenize(name," ")[3]='KING']

/../../name/text()

Результат показывает 2 выбранных элемента, так как в фирме есть два м-ра Кинга, они работают в одном отделе.

А вот другая логика решения: Мы для каждого отдела проверяем, нет ли в нем м-ра Кинга.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//department[employees//tokenize(name," ")[last()]='KING']/name/text()

Для каждого сотрудника выбрать разность между зарплатой менеджера его отдела и его зарплатой

Предикаты могут применяться и в выражениях пути для операндов выражений.

Здесь мы спускаемся до сотрудника (он является контекстным узлом) и берем его имя, а затем возвращаемся вверх к отделу и спускаемся к менеджеру этого отдела. В выражении salary - salaryпервое salary является конечным шагом выражения маршрута, второе salary выбирается из контекстного узла.

Обратите внимание на пробелы с обеих сторон операции -. Они являются обязательными, так как дают возможность отличить арифметическую операцию "минус" от символа "дефис", который допуcкается в именах XML. Если бы пробелов не было, то salary-salary интерпретировалось бы как имя элемента.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee

/concat(name,' (',(../..//employee[function='MANAGER']/salary - salary),')')

Выбрать имя и сумму зарплаты и комиссионных для каждого сотрудника

Проблема состоит в том, что во многих элементах employee отсутствует вложенный элемент commission, эти сотрудники не получают комиссионных. Если мы будем выводить salary+commission, то для этих сотрудников не выведется ничего.

В XPath есть условное выражение if - then - else, которое мы здесь и применим.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee/concat(name,' - ',

if (commission) then (salary+commission)

else salary)

Или так:

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee/concat(name,' - ',

salary+(if (commission) then commission else 0))

Выбрать сотрудников, поступивших на работу после 1 января.1986г.

В XPath вoзможна работа с датой. Дата обязательно представляется в формате ISO (ДД-ММ-ГГГГ), который обеспечивает легкое сравнение дат.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee[hire\_date>'1986-01-01']/concat(name,' - ',hire\_date)

Сколько времени проработал в фирме каждый сотрудник?

В XPath возможна и арифметика даты. Операнды должны иметь тип xs:date, поэтому мы преобразуем к этому типу hire\_date. Функция current-date() возвращает текущую дату.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee/concat(name,' - ',current-date()- xs:date(hire\_date))

Но у вас, конечно, возник вопрос: а что же это мы получили? Что за P8278D и т.п.? Результатом вычитания дат является тип xs:dayTimeDuration, формат которого такой:

Значит, Кэрол Кларк проработала в фирме 8278 дней

Определить заказы, в которых задержка между заказом и поставкой превышала 2 недели

Еще один пример на арифметику даты. Здесь мы используем константу длительности в предикате. Поскольку константа задается как строковая, она явным образом приводится к типу xs:dayTimeDuration.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//customers

//order

[( xs:date(ship\_date) - xs:date(order\_date) )

> xs:dayTimeDuration('P14D')]

/concat(@id,' - ',

(xs:date(ship\_date) - xs:date(order\_date)))

Выбрать среднюю зарплату по отделам

Здесь мы применяем агрегатную функцию (avg()). Мы спускаемся до отдела и считаем среднее значение для элементов salary, вложенных в данный отдел.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//department/concat(@id,' - ',avg(employees//salary))

Хотим предостеречь читателей от довольно типичной ошибке, связанной с применением агрегатных функций. Агрегатная функция применима на том уровне, где мы имеем множествовложенных значений, по которым она и считается. В следующем, неправильном решении мы уже вышли на уровень отдельного сотрудника, и будет вычисляться средняя зарплата одного сотрудника.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//department//employee/concat(../../@id,' - ',avg(salary))

Выбрать максимальную зарплату по фирме

Опять применение агрегатной функции.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//departments/max(//salary)

Кто в фирме получает максимальную зарплату?

Еще раз о том, что агрегатная функция должна применяться на должном уровне. Здесь выходим на уровень сотрудника (контекстный узел - элемент employee) и выбираем его зарплату. Затем мы возвращаемся на 3 уровня выше (контекстным узлом становится элемент departments - все отделы) и в этом узле вычисляем максимальную зарплату.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//departments/department/employees

/employee[salary=max(../../..//salary)]/concat(name,' - ',salary)

Кто получает максимальную зарплату в каждом отделе?

Запрос очень похож на предыдущий. Но здесь, выйдя на уровень сотрудника (контекстный узел - элемент employee) и выбрав его зарплату, мы возвращаемся только на один уровень выше (контекстным узлом становится элемент employees - все сотрудники одного отдела), и максимальная зарплата вычисляется в этом узле.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee[salary=max(..//salary)]

/concat(../../@id,' - ',name,' - ',salary)

Для отдела 20 выбралось два значения: там два сотрудника получают одинаково высокую зарплату.

Для каждого клерка выбрать разность между его зарплатой и средней зарплатой клерков

Еще один не самый простой запрос с применением агрегатной функции. Выходим на уровень employee и выбираем зарплату, затем возвращаемся на уровень deparnments и определяем среднюю зарплату для всех клерков.

Запрос:

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee[function='CLERK']/

concat(name,' - ',

(salary - ../../../avg(//employee[function='CLERK']/salary)))

Этот же запрос можно записать и так:

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee[function='CLERK']

/concat(name,' - ',

(salary -

doc('file:///c:/KING/kingx/king\_corp.xml')//departments

/avg(//employee[function='CLERK']/salary)))

То есть, вложенные выражения могут начинать свой путь не от контекстного узла, а от корня документа

Замечание: нам не удалось средствами XPath сделать запрос: Для каждого сотрудника выбрать разность между его зарплатой и средней сотрудников, занимающих ту же должность . Попробуйте сами это сделать и объясните, почему это не получается.

Что XQuery добавляет к XPath?

На предыдущей странице мы нарочно сделали замечание о запросе, который мы не смогли сформулировать в XPath, а то у читателя могло бы сложиться впечатление, что при помощи XPath можно выбрать все, что угодно.

Мы увидели, что при помощи XPath можно сделать очень многое. Но не все.

XPath не определяет, откуда берутся данные, в которых мы ищем. Поскольку это не самостоятельная спецификация, источник данных определяется во внешней по отношению XPath спецификацией - XPointer или XSLT. В XQuery источник данных определяется функцией ввода. Одну из этих функций - doc - мы уже видели и применяли. Она используется в том случае, если в источнике содержится правильное дерево (с единственным корневым элементом). Вторая функция ввода - collection - для случая "леса" (отсутствует единый корень). Аргументом обеих функций является URL источника, в нашем случае - локальная файловая система.

В XPath мы выбирали данные, но сформировать из этих данных что-то структурированное, например, другой XML-документ, нам бы не удалось.

В XPath нам не удалось бы соединить естественным образом данные, расположенные на разных ветвях. Запрос Выбрать для каждого покупателя имя продавца, который его обслуживает у нас не получился Попробуйте - может быть, он получится у Вас?

Некоторые свойства XQuery приближают его к языкам программирования. Кое в чем это облегчает то, что возможно и в XPath, кое в чем создает новые возможности. Например, переменные позволяют сохранить результат вычисления выражения и использовать его многократно.

Замечание: С языками программирования XQuery роднит то, что, в отличие от SQL, XQuery - процедурный язык. SQL-запрос описывает не процедуру выполнения, а то, что должно получиться, в РСУБД SQL-engine сама определяет оптимальный порядок выполнения запроса. Операции же запроса XQuery выполняются именно в том порядке, в каком они записаны.

Выражение FLOWR

Главной частью XQuery является выражение FLOWR. Это аббревиатура из первых букв операторов, которые могут входить в это выражение:

for

let

order by

where

return

Произносится как "flower".

Оператор for

связывает одну или более переменных с выражениями, создавая поток кортежей. В потоке каждый кортеж связывает данную переменную с одним из значений, получаемых в результате вычисления выражения.

Пример:

Выбрать имена и фамилии всех сотрудников

Запрос:

for $a in doc('file:///c:/KING/kingx/king\_corp.xml')//employee/name

return data($a)

Во-первых, в данном примере мы впервые встречаемся с переменной. Переменная в XQuery предваряется значком $.

Во-вторых, с некоторой степенью приближения можно трактовать for как оператор цикла. Выбирается множество узлов. В примере - выбираются все элементы employee, лежащие на любом уровне, а в них - элементы name. Поток кортежей и составляет это множество элементов name. Переменная $aпоследовательно (в цикле) принимает все значения из потока.

В третьих, данном примере мы встретились также с функцией data(), извлекающей данные из результата выражения. В данном случае мы могли бы применить и известную нам функцию text().

Оператор let

связывает переменные с полным результатом вычисления выражения, добавляя эти связи к кортежам, полученным оператором for, или создавая единственный кортеж (при отсутствии оператора for).

Пример:

Выбрать имена и должности всех сотрудников

Запрос:

for $a in doc('file:///c:/KING/kingx/king\_corp.xml') //employee/name

let $b:=$a/../function

return ($a, $b)

Это оператор присваивания. В данном случае для каждого значения $a вычисляется выражение, выбирающее для данного сотрудника его должность, и результат этого вычисления сохраняется в переменной $b.

Как мы увидим далее, let может употребляться и вне for.

Оператор order by

сортирует поток кортежей.

Пример:

Выбрать список сотрудников, упорядоченный по убыванию зарплаты

Запрос:

for $a in doc('file:///c:/KING/kingx/king\_corp.xml')//employee

order by xs:decimal(data($a/salary)) descending

return ($a/name,$a/salary)

Чтобы больше нагрузить пример, мы показали, что в order by может быть сложное выражение и задали обратную сортировку. Если нет descending , будет прямая сортировка

Обратите внимание: результат выражения мы преобразовали в xs:decimal. Если бы мы этого не сделали, он интерпретировался бы как строка, и зарплата 750 считалась бы большей, чем 5000.

То выражение, по которому происходит упорядочивание, не обязательно входит в выбираемое значение (в SQL это не так)

Оператор where

оставляет в потоке только те кортежи, которые удовлетворяют условию, являющемуся параметром данного оператора.

Пример:

Выбрать всех клерков

Запрос:

for $a in doc('file:///c:/KING/kingx/king\_corp.xml')//employee

where $a/function='CLERK'

return ($a/name/text(), $a/function/text())

Оператор return

создает результат выражения FLWOR для данного кортежа.

Пример:

Выбрать имена всех клерков

Запрос:

for $a in doc('file:///c:/KING/kingx/king\_corp.xml')//employee

where $a/function='CLERK'

return <clerk> { data($a/name) } </clerk>

Именно здесь, как правило, конструируется результат. В данном случае мы создаем новый элемент. Про конструирование результата мы еще поговорим ниже. Пока отметим, что фигурные скобки применяются для конструирования элемента, круглые (в предыдущих примерах) - для конструирования последовательности.

Для каждого покупателя выбрать продавца, который его обслуживает

Запрос:

for $a in doc('file:///c:/KING/kingx/king\_corp.xml')//employee

for $b in doc('file:///c:/KING/kingx/king\_corp.xml')//customers/customer

where $a/customer/@id = $b/@id

return concat($b/name,' - ',$a/name)

Пример демонстрирует нечто вроде "естественного соединения" - мы соединяем данные, лежащие на разных ветвях нашего дерева (это могли бы быть данные и из разных источников). Но говорить о "естественном соединении" мы можем только с кавычками, так как внешние ключи в базе данных XML не заданы.

Два оператора for создают два разных контекста.

Это может рассматриваться как вложенный цикл.

Вывести список штатов, в которых находятся покупатели, обслуживаемые отделами, расположенными в городе NEW YORK.

Еще пример на "естественное соединение".

Запрос:

for $a in doc('file:///c:/KING/kingx/king\_corp.xml')//

department[location='NEW YORK']//employee/@id

for $b in doc('file:///c:/KING/kingx/king\_corp.xml')//customer

where $b/salesperson=$a

return $b/state/text()

Для каждого сотрудника выбрать его непосредственного начальника

Запрос:

for $a in doc('file:///c:/KING/kingx/king\_corp.xml')//employee

for $b in doc('file:///c:/KING/kingx/king\_corp.xml')//employee

where $b/@id = $a/manager\_id

return concat($a/name,' - ',$b/name)

Это как бы "автосоединение" - два разных контекста в одной ветви.

В результат не попал Francis A. King, у которого начальника нет: он - президент фирмы

Определить те товары, которых сотрудник JONES может купить на свою зарплату больше 1000 штук

Запрос:

for $a in doc('file:///c:/KING/kingx/king\_corp.xml')

//employee[tokenize(name,' ')[3]='JONES']/salary

for $b in doc('file:///c:/KING/kingx/king\_corp.xml')

//product/price[not(@end)]

where xs:decimal($b/min)\*1000<="" pre="">

Это как бы соединение по произвольному условию.

Выбрать тех сотрудников, которые имеют подчиненных

Это почти то же самое, что выбрать для каждого сотрудника его начальника.

Запрос:

for $a in doc('file:///c:/KING/kingx/king\_corp.xml')//employee

for $b in doc('file:///c:/KING/kingx/king\_corp.xml')//employee

where $b/@id = $a/manager\_id

return $b/name/text()

Но запрос выдал нам повторяющиеся имена, так как выдается начальник для каждого сотрудника, а несколько сотрудников имеют одного и того же начальника. Как избавиться от дубликатов?

Выбрать тех сотрудников, которые имеют подчиненных

В этом запросе мы избавляемся от дубликатов

Запрос:

for $s in {

for $a in doc('file:///c:/KING/kingx/king\_corp.xml') //employee

for $b in doc('file:///c:/KING/kingx/king\_corp.xml') //employee

where $b/@id = $a/manager\_id

return $b/name

}

return distinct-values($s/name)

В предыдущем примере применение distinct-values() ничего бы нам не дало, так как returnвозвращает элементы кортежа по одному, а не последовательность.

Здесь же мы формируем элемент a, в который вкладываются элементы name, а потом выбираем неповторяющиеся значения.

Выбрать тех сотрудников, которые имеют подчиненных

Другой вариант того же - использование оператора let для упрощения оператора for. Поток кортежей сначала формируется и сохраняется оператором let в переменной $x, которая потом используется для for.

Запрос:

let $x:={

for $a in

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee

for $b in

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee

where $b/@id = $a/manager\_id

return $b/name

}

for $a in $x

return distinct-values($a/name)

Для каждого клерка выбрать разность между его зарплатой и средней зарплатой клерков

В XPath этот запрос у нас не получился.

Здесь же мы определяем среднюю зарплату клерков и сохраняем ее в переменной $a. Затем перебираем клерков и из зарплаты каждого вычитаем $a.

Запрос:

let $a :=

doc('file:///c:/KING/kingx/king\_corp.xml')

//departments

/avg(//employee[function='CLERK']/salary)

for $b in

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee[function='CLERK']

return ( $b/name/text(), $b/salary - $a)

Выбрать название товара, которого в 1990 г. было продано максимальное количество экземпляров

Не самый простой запрос

Запрос:

1 let $x:= <z> {

2 for $y in

3 <y>{for $b in

4 doc('file:///c:/KING/kingx/king\_corp.xml')//customers/

5 distinct-values(//product\_id)

6 return <x> {

7 <id> {$b} </id>,

8 for $a in

9 doc('file:///c:/KING/kingx/king\_corp.xml')//customers//

10 order[substring-before(order\_date,'-')='1990']/item

11 where $a/product\_id=$b

12 return $a/quantity

13 } </x>

13 } </y>

14 /x

15 order by $y/sum(quantity)

16 return ($y/id,$y/sum(quantity))

17 }</z>

18 let $z:=$x/id[last()]

19 for $a in doc('file:///c:/KING/kingx/king\_corp.xml')//products/product

20 where $a/@id=$z

21 return $a/description/text()

Попробуем отследить выполнение этого запроса

Самый внутренний оператор for с переменной цикла $a (8-12) без оператора where (11) отбирает те заказы, которые были сделаны в 1990 и выбирает из них количества.

Оператор for с переменной цикла $b (2-5) выбирает все коды товаров.

В возвращаемом им значении оператор конструирует элемент x (6-13), в который вложен элемент id, тело которого - id товара, и количество проданного товара, выбранное вложенным оператором for (8-12). За счет where (11) количество "естественным образом" соединено с соответствующим id товара.

Все элементы x, сформированные на предыдущем шаге, вкладываются в элемент y (3,13).

Этот элемент y является источником для оператора for с переменной цикла $y (2-16). Этот оператор выбирает каждый вложенный элемент x (14) и вычисляет для него сумму значений quantity. Он возвращает id товара и суммарное количество для него (16), упорядоченные по возрастанию суммарного количества (15).

Все, что сформировал предыдущий оператор for, вкладывается в элемент z и запоминается в переменной $x (1,17).

Из переменной $x извлекается последний id (т.е., id товара с максимальным количеством) и запоминается в переменной $z (18).

Наконец, последний оператор for (19-21) выбирает название того товара, id которого совпадает с запомненным в переменной $z.

Выбрать тех сотрудников, которые НЕ имеют подчиненных

Запрос:

let $x:=<x>{

for $a in doc('file:///c:/KING/kingx/king\_corp.xml')//employee

for $b in doc('file:///c:/KING/kingx/king\_corp.xml')//employee

where $b/@id = $a/manager\_id

return $b/name

} </x>

for $c in doc('file:///c:/KING/kingx/king\_corp.xml')//employee

where every $h in $x/name

satisfies $h!=$c/name

return $c/name/text()

Здесь мы применяем один из, так называемых, кванторов every ... satisfies .... Полагаем, что семантика квантора понятна из текста.

Оператором let создается список сотрудников, которые имеют подчиненных и запоминается в переменной $x. Затем перебирается (for) список сотрудников и выбираются те, в котором имя не совпадает ни с одним элементом в списке в переменной $x

Выбрать коды тех отделов, в которых есть КАКИЕ-НИБУДЬ из должностей, имеющихся в отделе 23

Запрос:

let $x:= <x>{

for $a in

doc('file:///c:/KING/kingx/king\_corp.xml')//department[@id='23']

return $a

}</x>

for $c in

doc('file:///c:/KING/kingx/king\_corp.xml')//department[@id!='23']

where some $h in $x//function

satisfies; $h=$c//function

return $c/data(@id)

Еще квантор.

Оператором let создается список должностей отдела 23 и запоминается в переменной $x. Затем перебираются (for) отделы (кроме 23) и выбираются те, у которых какая-нибудь должность входит в список в переменной $x

Выбрать коды тех отделов, в которых есть ВСЕ должности, имеющиеся в отделе 23

Запрос:

let $x:= <x>{

for $a in

doc('file:///c:/KING/kingx/king\_corp.xml')//department[@id='23']

return $a

}</x>

for $c in doc('file:///c:/KING/kingx/king\_corp.xml')//department[@id!='23']

where every $h in $x//function satisfies $h=$c//function

return $c/data(@id)

Оператором let создается список должностей отдела 23 и запоминается в переменной $x. Затем перебираются (for) отделы (кроме 23) и выбираются те, для которых какая-все должности списка в переменной $x входят в список должностей данного отдела.

Выбрать коды тех отделов, в которых ТОЛЬКО должности, имеющиеся в отделе 23

Запрос:

let $x:= <x>{

for $a in

doc('file:///c:/KING/kingx/king\_corp.xml')//department[@id='23']

return $a

}</x>

for $c in

doc('file:///c:/KING/kingx/king\_corp.xml')//department[@id!='23']

where (count($c//function)<=count($x//function))

and (every $h in $c//function satisfies $h=$x//function)

return $c/data(@id)

Оператором let создается список должностей отдела 23 и запоминается в переменной $x. Затем перебираются (for) отделы (кроме 23) и выбираются те, у которых все должности входят в список в переменной $x и общее количество должностей не больше, чем в $x.

Выбрать коды тех отделов, в которых ТОЧНО ТОТ ЖЕ набор должностей, что и в отделе 23

Запрос:

let $x:= <x>{

for $a in

doc('file:///c:/KING/kingx/king\_corp.xml')//department[@id='23']

return $a

}</x>

for $c in

doc('file:///c:/KING/kingx/king\_corp.xml')//department[@id!='23']

where (every $h in $x//function satisfies $h=$c//function)

and (every $h in $c//function satisfies $h=$x//function)

return $c/data(@id)

Оператором let создается список должностей отдела 23 и запоминается в переменной $x. Затем перебираются (for) отделы (кроме 23) и выбираются те, у которых все должности входят в список в переменной $x и для которых все должности из $x входят в список должностей данного отдела.

Выбрать однофамильцев

Запрос:

let $x:=<x> {

for $a in

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee

for $b in

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee

where ($b/tokenize(name,' ')[3] =

$a/tokenize(name,' ')[3])

and ($b/name!=$a/name)

order by $b/tokenize(name,' ')[3]

return $b/name

} </x>

return distinct-values($x/name)

Два оператора for создаются два списка имен-фамилий - $a и $b. Из второго списка ($b) выбираются те люди, для которых в первом списке ($a) есть пара с совпадающей фамилией и не совпадающим именем.

distinct-values() здесь употребляется потому, что если есть более двух однофамильцев, то каждый из них выберется несколько раз.

Выбрать имена и зарплаты 10 самых высокооплачиваемых сотрудников фирмы (без учета комиссионных)

Запрос:

let $x:=<x>{

for $a in

doc('file:///c:/KING/kingx/king\_corp.xml')

//employee

order by xs:decimal($a/salary) descending

return <y>{$a/name, $a/salary}</y>

}</x>

for $b in $x/y[position()<=10]

return concat($b/name,' - ',$b/salary)

Этот запрос представлял для нас некоторую трудность в SQL. Мы не могли решить эту задачу иначе, чем коррелированным запросом.

Здесь же мы просто формируем список, упорядоченный по зарплатам и выбираем из него первые 10 позиций.

Но обратите внимание: если в фирме есть еще кто-то, получающий 2750, то этот сотрудник не попал в список. Правильно ли это? (В SQL этот сотрудник был бы выбран 11-ым.)

Определить, на какую сумму подорожал товар 'DUNK HOOP' с того момента, когда он был выставлен на продажу

Запрос:

for $b in

<x>{

for $a in

doc('file:///c:/KING/kingx/king\_corp.xml')

//product[description='DUNK HOOP']/price/list

order by $a/../string(@start)

return $a

}</x>

return ($b/list[last()] - $b/list[1])

Опять-таки, в SQL решение было довольно громоздким.

Здесь же мы просто упорядочиваем цены по времени установления и выбираем разность между последней и первой ценами.

Прямой конструктор

Мы уже встречались в наших примерах с конструкторами - записью, которая конструирует XML-элемент на выходе выражения.

Прямой конструктор задает разметку выходного элемента явным образом.

Пример:

<employee id="2265">

<name>Edsger W. Dejkstra</name>

<salary>4500.00</salary>

<commission>1000</commission>

<hire\_date>1967-01-10</hire\_date>

<function>RESEARCH WORKER</function>

<manager\_id>7250</manager\_id>

</employee>

Но атрибуты и тела элементов в прямом конструкторе, так же как и сами элементы, могут формироваться выражениями FROWR.

Составить отчет по работе отделов продаж, в котором указать:

id отдела

город

список покупателей

общую сумму продаж

Запрос:

1 <report> {

2 let $x:=<x>{

3 for $a in doc('file:///c:/KING/kingx/king\_corp.xml')

4 //department

5 where $a/name='SALES'

6 return $a

7 }</x>

8 for $a in $x//department

9 return <department>{$a/@id} {

10 $a/location,

11 <customers> {

12 for $b in doc('file:///c:/KING/kingx/king\_corp.xml')

13 //customers/customer

14 for $c in $a//employee

15 where $c/data(@id)=$b/data(salesperson)

16 return $b/name

17 } </customers>,

18 <total> {

19 let $y:=<y>{

20 for $b in doc('file:///c:/KING/kingx/king\_corp.xml')

21 //customers/customer

22 for $c in $a//employee

23 where $c/data(@id)=$b/data(salesperson)

24 return <t>{sum($b/order/total)}</t>

25 }</y>

26 for $d in $y return sum($d//t)

27 }</total>

28 } </department>

29 }</report>

Выполнение запроса:

Все, что мы сконструируем, будет вложено в корневой элемент report (1, 29).

Формируем и сохраняем в переменной $x элемент x, в который вложены элементы department всех отделов, имеющих название SALES.

Для каждого вложенного элемента department формируем выходной элемент department(9 - 28), в который вкладываются:

элемент location из исходного department (10);

элемент customers (11,17), в который вкладываются:

элементы name (12-16) - имена покупателей, для которых значение в элементе salesperson равно значению атрибута id какого-то из элементов employee, вложенных в данный department

элемент total (11,17), для вычисления которого:

формируется и сохраняется в переменной $y (19-25) элемент y, в который вложены элементы t, содержащие суммарный объем закупок по каждому покупателю;

вычисляется общая сумма по всем покупателям (26)

Вычисляемый конструктор

В вычисляемом конструкторе используются ключевые слова для создания элементов и атрибутов. Значения элементов и атрибутов задаются либо константами, либо выражениями, в том числе, это могут быть выражения FLOWR.

Пример:

element employee {

attribute id {"2265"},

element name {"Edsger W. Dejkstra"},

element salary {"4500.00"},

element commission {"1000"},

element hire\_date {"1967-01-10"},

element function {"RESEARCH WORKER"},

element manager\_id {"7250"}

}

Составить отчет по работе отделов продаж, в котором указать:

id отдела

город

список покупателей

общую сумму продаж

Это та же задача, что и в предыдущем примере, но на сей раз мы сформируем результат в виде HTML-страницыю

Запрос:

1 element html{

2 element b{"Department report"},

3 element table{

4 attribute border{"1"},

5 let $x:={

6 for $a in doc('file:///c:/KING/kingx/king\_corp.xml')//department

7 where $a/name='SALES'

8 return $a }

9 for $a in $x//department

10 return (

11 element tr{

12 element td{attribute colspan {3},'Depatrment',$a/data(@id)}

13 },

14 element tr{

15 element td{ attribute valign{"top"},$a/location/text()},

16 element td{

17 for $b in doc('file:///c:/KING/kingx/king\_corp.xml')//customers/customer

18 for $c in $a//employee

19 where $c/data(@id)=$b/data(salesperson)

20 return ($b/name/text(),element br {})

21 },

22 element td{

23 attribute valign{"top"},

24 let $y:={

25 for $b in doc('file:///c:/KING/kingx/king\_corp.xml')//customers/customer

26 for $c in $a//employee

27 where $c/data(@id)=$b/data(salesperson)

28 return element t {sum($b/order/total)} }

29 for $d in $y return sum($d//t)

30 }

31 })

32 }

33 }

Выполнение запроса:

Корневым элементом формируемого отчета будет элемент html (1-33).

Первым в корневой элемент вкладывается элемент b (полужирный шрифт) с заголовком (2).

Далее формируется элемент table (3-32), в котором:

содержится атрибут border со значением 1 (4);

формируется в переменной $x элемент x, в который вложены элементы departmentвсех отделов, имеющих название SALES;

для каждого элемента department из переменной $x:

создается элемент tr, в который вкладывается элемент td с атрибутом colspan и номером отдела (11-13) - это строка заголовка отдела;

создается элемент tr (14-31), в который вкладываются:

элемент td с атрибутом valign и значением местоположения отдела (15);

элемент td, тело которого формируется (16-21) из названий покупателей, связанных с данным отделом через обслуживающих их продавцов, после каждого названия покупателя вставляется пустой элемент br;

элемент td с атрибутом valign, значение элемента вычисляется (22-30) как сумма продаж по отделу (как и в предыдущем примере.

ыбрать всех прямых начальников сотрудницы DIANE G. ADAMS

Запрос:

1 declare namespace mySpace = "http://khpi-iip/derev"

2 define function mySpace:getManager($a as element(employee)\*)

3 as element(employee)\* {

4 for $b in doc('file:///c:/KING/kingx/king\_corp.xml')//employee

5 where $b/@id = $a/manager\_id

6 return ($a, $b, mySpace:getManager($b))

7 }

8 for $a in doc('file:///c:/KING/kingx/king\_corp.xml')//employee[name='DIANE G. ADAMS']

9 return mySpace:getManager($a)/name/text()

Это рекурсивный запрос. Мы должны выбрать саму DIANE G. ADAMS, потом - ее начальника, потом - его начальника и т.д.

В SQL это не предусмотрено в стандарте SQL/96. Разные СУБД используют разные средства для решения этой задачи, стандарт SQL/2000 решает ее при помощи рекурсивных временных представлений.

Здесь мы используем рекурсивную функцию. Сначала разберем формальный синтаксис этого запроса:

Строка 1 - определение пространства имен и его префикса для функции. Пространство по умолчанию применяется для стандартных функций.

Строка 2 - заголовок функции с ее параметрами и их типами и с типом возвращаемого значения.

Строки 3-7 - тело функции.

Строки 8-9 - запрос, в котором вызывается функция.

Теперь - о семантике запроса.

Функция получает элемент сотрудника, выбирает его менеджера и возвращает полученного сотрудника, менеджера и тот результат, который вернула эта же функция при вызове ее с параметром - менеджером. Все это образует последовательность (берется в круглые скобки).

Вывести структуру подчиненности в фирме

Еще один рекурсивный запрос. Теперь мы движемся по иерархии подчиненности фирмы сверху вниз.

Запрос:

declare namespace mySpace = "http://khpi-iip/derev"

define function mySpace:getManager($a as element(employee))

as element(employee)\* {

for $b in doc('file:///c:/KING/kingx/king\_corp.xml')//employee

where $a/@id = $b/manager\_id

return element employee { $b/name, mySpace:getManager($b) }

}

for $a in doc('file:///c:/KING/kingx/king\_corp.xml')

//employee[function='PRESIDENT']

return element employee{ $a/name, mySpace:getManager($a) }

Функция возвращает элемент employee, в который вложено имя сотрудника и элементы employee для всех его подчиненных. В запросе мы выбираем президента фирмы и раскручиваем рекурсию от него.

Глава 4 из книги "W3C XML: XQuery от экспертов. Руководство по языку запросов"

Редактор Говард Кац   
Дон Чамберлин, Дениз Дрейпер, Мэри Фернандес, Майкл Кей, Джонатан Роби, Майкл Рис, Жером Симеон, Джим Тивай, Филип Уодлер 

Эта глава посвящена исследованию взаимоотношений между XSLT, XPath и XQuery. В ней объясняются причины, по которым нам выгоднее иметь три разных языка программирования, а не один, а также взаимосвязь этих языков между собой. Читатели, знакомые с XSLT, узнают в данной главе, в каких областях XQuery похож на XSLT, а в каких имеет принципиальные отличия. В ней также исследуются некоторые различия между преобразованием и запросом, которые влияют на архитектуру реализации – в частности, взаимоотношение со схемами и определениями типов и различия в способах оптимизации в этих двух языковых средах.

XSLT: краткое введение

XSLT – это язык для описания преобразований XML. Его операторы принимают один или более документов XML в качестве входных данных и выдают один или более XML документов в качестве результата. Этот язык разрабатывался в рамках W3C как часть более крупного проекта, связанного с представлением (или отображением) XML: отсюда происходит название «расширяемый язык преобразований стилей» (eXtensible Stylesheet Language – Transformations).

XSLT 1.0 был опубликован в качестве рекомендации W3C 16 ноября 1999 года. После этого в течение первых трех лет были разработаны не менее двадцати его реализаций, включая реализации, встроенные в два наиболее широко используемых веб-браузера – Internet Explorer и Netscape, а также множество продуктов с открытым кодом, один из которых – Saxon – был разработан автором этой главы. Большинство реализаций достигли превосходного уровня соответствия спецификации W3C, хотя наличие расширений языка, созданных разработчиками программного обеспечения, означает, что не всегда переносимость различных версий может быть достигнута настолько легко, как бы этого хотелось. Язык получил широкое распространение среди пользователей, несмотря на свою репутацию трудного для изучения и медленного в работе.

Вероятно, 80 процентов фактического использования XSLT сегодня приходится на преобразование XML в HTML. Этот процесс происходит следующим образом: полученный в результате документ рассматривается как правильно построенное дерево XML, затем выполняется преобразование с последующей стадией приведения в последовательную форму, которая переводит это дерево в конечный HTML. В других 10 процентах случаев использования XSLT выполняет функцию представления XML в других форматах отображения, таких как SVG, WML или PDF (с помощью другой части XSL – словаря форматирования объектов). Оставшиеся 10 процентов использования приходятся на приложения, преобразующие XML в XML, особенно это касается преобразования сообщений, посылаемых друг другу приложениями в интегрированной инфраструктуре предприятия либо внутри организации, либо за ее пределами. Хотя на сегодняшний день эта часть рынка невелика, она развивается наиболее быстро и обещает в будущем приносить самую большую прибыль.

Некоторые из ключевых характеристик языка XSLT приведены ниже:  Синтаксис на основе XML: программа преобразования XSLT (называемая по историческим причинам «таблицей стилей» – stylesheet) сама является XML документом. Эта особенность языка особенно полезна в том случае, когда большие части таблицы стилей содержат фиксированные или относительно фиксированные элементы и атрибуты XML, которые должны быть включены непосредственно в результат, потому что в этом случае мы можем использовать таблицу стилей в качестве шаблона для конечного документа. Другим полезным следствием такого подхода к дизайну языка является то, что мы можем использовать таблицы стилей XSLT в качестве источника или цели для дальнейших преобразований. Хотя на первый взгляд это кажется довольно странной идеей, но на практике в крупномасштабных приложениях таблицы стилей обычно создаются или адаптируются с использованием «метатаблиц стилей», которые сами написаны на XSLT.  Декларативная, функциональная модель программирования: основной парадигмой программирования XSLT является функциональное программирование. Таблицы стилей описывают преобразование исходного дерева в дерево результата. Дерево результата является функцией источника, а индивидуальные поддеревья результата – функциями исходной информации, из который они получены. Хотя таблица стилей содержит такие конструкции, как условия и повторения, которые известны из процедурного программирования, в самом языке определенный порядок выполнения действий не задается. В частности, нет никаких операторов присвоения и обновляемых переменных. Эта особенность, вероятно, послужила причиной, по которой XSLT получил репутацию трудного для изучения языка, поскольку программисты, привыкшие к языкам, подобным JavaScript, обнаружили, что программирование на XSLT может потребовать значительной реорганизации мышления. Это также объясняет сообщения о медленной работе приложений, потому что без четкого представления о том, какие действия фактически выполняет машина, очень легко написать чрезвычайно неэффективный код. (Для дальнейшего обсуждения см. раздел, посвященный оптимизации, далее в этой главе.)

Язык XSLT основан на правилах: таблица стилей XSLT выражена в виде собрания правил в традициях языков обработки текста, подобных awk и sed. Правила состоят из шаблона, которому должен соответствовать входной документ, и инструкций для создания узлов в дереве результата (шаблоне), когда достигну-то соответствие шаблону. Однако в отличие от правил, содержащихся в языках обработки текста, эти правила не применяются последовательно для каждой строки входного документа; вместо этого они выполняют обход вершин входного дерева. В наиболее простых преобразованиях каждое правило шаблона для родительского узла вызывает активацию правил для его дочерних узлов, что дает в результате рекурсивный обход «в глубину» исходного дерева. Но этот процесс находится полностью под контролем автора таблицы стилей, и обходить входное дерево можно любым способом на выбор автора. Преимущество такого подхода на основе правил состоит в том, что таблица стилей может очень гибко реагировать на изменения в деталях структуры входного документа. Это особенно хорошо при обработке рекурсивных структур, встречающихся в «документо-ориентированном» XML, которые часто имеют очень свободные правила для вложений тегов. Для преобразования «информационно-ориентированного» XML, где структуры более жесткие, этот стиль обработки имеет меньше преимуществ, и на практике в этом случае нет необходимости писать каждую таблицу стилей таким способом.

Преобразование дерева в дерево: входные данные и результат преобразования моделируются в виде деревьев, а не в виде последовательной формы XML. Создание исходного дерева (с использованием анализатора XML) и приведение в последовательную форму дерева конечного результата являются действиями, отличными от непосредственно преобразования, и во многих приложениях они фактически не выполняются. Например, обычным действием является создание конвейера преобразований, в котором результат одного из них используется в качестве входных данных для следующего, без промежуточного приведения в последовательную форму. Это означает, что непредвиденные детали исходного XML (например, различие между одинарными и двойными кавычками, в которые заключены атрибуты) не видимы для приложений и в общем случае не сохраняются во время преобразования. Иногда это может вызвать проблемы использования; например, преобразование будет всегда расширять значения атрибутов по умолчанию и ссылки на объекты, определенные в DTD, что неудобно, если конечный документ предназначен для дальнейшего редактирования пользователем.

Двухъязыковая модель: XSLT использует XPath в качестве подъязыка. Мы исследуем взаимоотношения между XSLT и XPath более подробно в следующем разделе. Грубо говоря, инструкции XSLT используются для создания узлов в дереве результата и управления последовательностью обработки. Выражения XPath используются для выбора данных из исходного дерева. Выражения XPath всегда вызываются из инструкций XSLT; не существует возможности (в XSLT 1.0) какого-либо вызова в обратном направлении. Это означает, что язык не является полностью композиционным в том смысле, что любое выражение может быть вложено внутри любого другого.

Сегмент кода XSLT в листинге 3.1 иллюстрирует эти особенности.

Листинг 3.1. Код, иллюстрирующий ключевые особенности XSLT

<xslt:template match="appendix">

<h2>

Appendix <xslt:number format="A">

<xslt:text>&nbsp;</xslt:text>

<a name="{@id}"/>

<xslt:value-of select="@title"/>

</h2>

<xslt:apply-templates/>

</xslt:template>

Этот пример показывает отдельное правило шаблона. Он очень прост: match="appendix" указывает на то, что шаблон соответствует элементам сименем appendix. Тело правила шаблона определяет узлы, которые должны быть записаны в дерево результата. Здесь xslt:number является инструкцией для создания последовательности чисел; xslt:text указывает текст, который должен быть записан в дерево результата; xslt:value-of вычисляет результат выражения XPath и записывает его в виде текста в дерево результата; и xslt:apply-templates выбирает дальнейшие узлы из исходного дерева (по умолчанию – это дочерние узлы текущего узла) и вызывает их обработку, применяя для каждого соответствующее правило шаблона. Элементы a и h2, которые не находятся в пространстве имен XSLT, копируются непосредственно в результат. Фигурные скобки в имени атрибута элемента a указывают шаблон значения атрибута: в эти скобки включено выражение XPath, вычисляющее строковое значение, которое будет вставлено в содержание атрибута. Эта конструкция используется по той причине, что ограничения синтаксиса XML лишают возможности включать инструкции внутри значения атрибута. Предположим, что входной документ выглядит так:

<appendix id="bibl" title="Bibliography">

<para>A reference</para>

</appendix>

Мы не показали правило шаблона, которое обрабатывает элементы para, но если предположить, что оно создает в результате элемент HTML p и копирует текстовое содержание, то результат применения этой таблицы стилей, вероятно, будет такой:

<h2>Appendix C&nbsp;<a name="bibl"/>Bibliography</h2>

<p>A reference</p>

XPath 1.0

XPath 1.0 был издан в качестве Рекомендации W3C в тот же день, что и XSLT 1.0: 16 ноября 1999 года. Из-за метода включения выражений XPath в таблицы стилей XSLT эти два языка тесно связаны между собой. Однако XPath был преднамеренно опубликован в качестве автономного документа, так как ожидалось, что он может быть использован во многих других контекстах помимо XSLT. Фактически причиной первоначального решения отделить XPath от проекта XSLT был тот факт, что XSLT и XPointer (формат гиперссылок, используемый спецификацией XLink для связи документов) развивались как различные языки, которые имели высокую степень совпадения функциональных возможностей. Поэтому все согласились, что будет лучше, если W3C определит отдельный базовый язык для адресации внутри XML документов.

Решение сделать XPath отдельным проектом было оправданно последующими событиями. Многие разработчики представили реализации XPath, которые являются или автономными, или поставляются вместе с реализацией, либо модели объекта документа (DOM – Document Object Model)[1](http://citforum.ru/internet/xml/xquery_xpath_xslt/" \l "b1), либо одной из других моделей на основе дерева XML, такой как JDOM[2](http://citforum.ru/internet/xml/xquery_xpath_xslt/" \l "b2). Подмножества XPath были адаптированы другими спецификациями в семействе XML, такими как XML Schema. И конечно, XPath теперь формирует основу XQuery.

Центральной конструкцией XPath, давшей языку его название, является выражение пути, которое для обращения к узлам в пределах представленного в виде дерева XML документа использует последовательность шагов, разделенных символом /. Синтаксис является производным от синтаксиса системы имен UNIX или URI, но это может ввести в заблуждение, потому что детальная семантика этого языка имеет намного больше возможностей. С точки зрения семантики, каждый шаг в выражении пути фактически имеет три части:

Ось, которая описывает взаимоотношения между узлами, по которым требуется совершить обход: например, она выбирает дочерние или родительские узлы для узла контекста, предков, потомков или узлы, имеющие общего родителя. Поскольку дочерняя ось используется наиболее часто, то она является осью по умолчанию, используемой в том случае, когда не вызывается никакая другая ось.

Проверка узла, устанавливающая ограничения на имена или виды узлов, которые должны быть выбраны: например, она могла бы выбрать все элементы или атрибуты с именем code.

Опционально один или более предикатов, устанавливающих дополнительные ограничения на последовательности узлов, которые должны быть выбраны. Эти ограничения могут зависеть от содержания узлов или от их положения в последовательности узлов. Предикаты могут также содержать последующие выражения пути, так что условие выбора узлов будет зависеть от дальнейшего сложного обхода по структуре дерева узлов.

Таким образом, выражение пути, подобное этому:

/book/\* [1] / @id

состоит из трех шагов. Первый шаг неявно использует дочернюю ось для выбора элементов с именем book; второй шаг выбирает первый дочерний элемент независимо от его имени; и третий использует ось атрибута (обозначенную символом @) для выбора узлов атрибута с именем id.

Подобно именам файлов и URI, выражения пути могут быть абсолютными или относительными. Относительные выражения пути выбирают узлы, начиная с отправной точки (узла контекста), которая в действительности является неявным параметром в выражении пути. Абсолютные выражения пути осуществляют выбор, начиная с корневого узла документа (хотя несколько неверно называть их «абсолютными», так как могут быть несколько документов, и выбор конкретного документа снова является неявным параметром).

Самым большим различием между выражениями пути XPath и именами файлов или URI, которые они напоминают, является то, что каждый шаг выбирает множество узлов, а не один узел. Каждый шаг применяется ко всем узлам, выбранным предыдущим шагом. Поэтому XPath и SQL имеют одну и ту же особенность, выражающуюся в том, что они всегда обрабатывают множества (узлов в случае XPath, кортежей в случае SQL), а не отдельные узлы.

Наряду с выражениями пути, которые выбирают узлы, в XPath 1.0 также имеется ряд операторов и функций для вычисления значений. Например, count(/book/chapter) возвращает число узлов, выбранных по выражению пути /book/chapter, в то время как substring(@desc, 1, 1) выбирает первый символ атрибута desc узла контекста. Эти операторы и функции используют только три типа данных в дополнение к множествам узлов, с которыми имеют дело выражения пути: строки, логические значения и числа. Вся числовая арифметика основывается на вещественных числах двойной точности. Когда действия применяются к значениям, имеющим неправильный тип, то имеют место неявные преобразования; например, использование строки в операции сложения не вызывает проблем, если строка содержит число. Этот аспект языка очень хорошо знаком программистам JavaScript, которые привыкли к использованию функций и операторов без особого учета типов данных.

Почему был необходим новый язык запросов?

С исторической точки зрения, исследования, которые привели к разработке XQuery, были начаты задолго до того, как были опубликованы XSLT 1.0 и XPath 1.0. Поначалу эти две рабочие группы мало контактировали между собой. В течение 1998 и 1999 годов XQL, один из предшественников XQuery, и новый язык XPath в некоторой степени влияли друг на друга, хотя это трудно проверить. Но ни одна из групп не находила, что другой язык соответствует тем требованиям, которым был адресован их проект – степень похожести двух языков стала очевидной позже.

Существуют два вида различий между XSLT и XQuery. Во-первых, к этим языкам предъявлялись различные требования, и поэтому конструкторские решения, подходящие для XSLT, необязательно были бы правильными для XQuery, и наоборот. Во-вторых, эти языки разрабатывались различными людьми из различных компаний с различными традициями программирования, с несовпадающими представлениями о том, каким должен быть хороший проект, и имеющих различный опыт относительно того, какие решения работают хорошо, а какие – нет.

Различные требования

Как мы уже видели, XSLT появился в результате работы над XSL (eXtensible Stylesheet Language – расширяемый язык стилей), чьей главной целью было представление (для восприятия человеком) информации, содержавшейся в XML документах. Хотя концепция преобразования имеет намного более широкое применение, и язык был разработан пригодным для выполнения широкого круга задач преобразования, моделирование XML осталось основным вариантом использования. Тот факт, что рабочая группа хотела сконцентрироваться на этом требовании, очевиден из утверждения в самом начале спецификации XSLT 1.0: «XSLT не разрабатывался в качестве многоцелевого языка преобразований. Напротив, он предназначен прежде всего для тех видов преобразований, которые требуются в том случае, когда XSLT используется как часть XSL» [XSLT, p. 1].

Я не был членом рабочей группы в то время, но я легко представил ее членов, договаривающихся об этом утверждении как о вопросе политики группы, и затем использующие его, чтобы отклонить включение в проект функциональных возможностей, которые считались находящимися вне этой сферы; например, включение расширенных математических операторов или операторов обработки текста. Но также легко вообразить, что некоторые члены группы в душе знали, что был необходим язык преобразований общего назначения, и были уверены, что XSLT должен быть способен выполнить эту задачу. Действительно, если не было людей, веривших в это, то трудно понять, почему было включено утверждение о политике языка, воспроизведенное выше.

Концепция языка преобразований подразумевает некоторые предположения относительно среды обработки. Преобразование по сути принимает один (или несколько) документ на входе и выдает один (или несколько) документ на выходе. Хотя документы обрабатываются в виде деревьев, обычно они поступают для анализа непосредственно из файлов прямо перед началом их преобразования. Документы не загружаются предварительно в базы данных, обеспечивающие специализированную индексацию или методы доступа. Исходный документ не модифицируется процессом преобразования, и он обычно помещается в основную память[3](http://citforum.ru/internet/xml/xquery_xpath_xslt/" \l "b3).

Тот факт, что рабочая группа сосредоточила усилия на преобразованиях, встречающихся во время моделирования документа, послужил причиной дальнейших предположений. Документо-ориентированный XML встречается чаще, чем информационно-ориентированный XML. Исходные документы могут быть, а могут не быть корректными согласно DTD. Таблицы стилей в основном писались бы для обработки разнообразных исходных документов с различной структурой. Обработка была бы чаще всего последовательной по своей природе: порядок элементов в дереве результата обычно был бы таким же, как и порядок соответствующих элементов в исходном документе. Язык должен, вероятно, быть не очень строгим при обработке ошибок: ошибки в таблице стилей должны позволить получить в результате как можно более полное отображение исходного документа вместо того, чтобы вызвать сообщение об ошибке во время выполнения, которое не означало бы ничего для конечного пользователя.

Предназначенная для языка роль также создала представление об ожидаемом пользователе, который стал бы писать преобразования. Этот пользователь, вероятно, самостоятельно разрабатывал бы как XML документы, так и шаблоны стилей с использованием общих инструментов редактирования XML, вследствие чего также была бы удобной возможность копирования и вставки частей кода XML в таблицы стилей. Эти таблицы обычно создавались бы как компилируемые по требованию автономные документы, доступные с помощью URL; иногда они могли бы быть вложены непосредственно в исходные документы.

Сценарий использования XQuery сильно отличался от приведенного выше. Как язык запросов баз данных, XQuery был предназначен для извлечения информации из больших собраний документов (или больших отдельных документов), которые обычно будут храниться на диске в базах данных с физическими структурами хранения, такими как индексы. Эти индексы позволяют пользователю осуществлять быстрое получение данных. Такие собрания документов часто могли создаваться централизованно, иметь однородную схему и ратифицироваться с помощью этой схемы перед загрузкой в базу данных. Действительно, некоторые разработчики программного обеспечения рассматривают использование XQuery главным образом для запросов представленных в виде XML обычных реляционных баз данных.

Такие различные сценарии использования этих двух языков приводят к раз-личным требованиям или, по крайней мере, к различию в акцентах среди предъявляемых требований. Документы чаще всего бывают информационно-ориентированными, а не документо-ориентированными, хотя язык запросов, как предполагалось, будет способен работать с обоими типами документов. Оптимизация запросов была бы важна для достижения приемлемой производительности, и эта оптимизация включала бы анализ запроса с помощью схемы целевой базы данных только для обнаружения доступных индексов. Поскольку документы часто являются информационно-ориентированными, то сохранение порядка было бы менее важным, а во многих случаях просто ненужным. Обработка ошибок, вероятно, должна быть строгой: если запрос некорректен, то было бы лучше выводить сообщение об ошибке как можно раньше, а не выполнять возможно больший запрос и давать в результате ответ на вопрос, который пользователь и не думал задавать.

Ожидаемый сценарий использования XQuery был бы подобен сценарию использования других языков запросов баз данных, подобных SQL. Иногда опытные пользователи могут использовать язык запросов непосредственно через терминал; но намного чаще запросы вложены в программы, написанные на базовых языках, таких как Java или C#, и возвращают свои результаты переменным базового языка для дальнейшей обработки приложением. Некоторые люди даже рассматривали XQuery в качестве вложенного в SQL подъязыка для поддержки запросов XML в пределах реляционных баз данных. Приведение результатов запроса в последовательную форму в виде XML документа могло бы быть одним из вариантов представления результатов, но ни в коем случае не единственным вариантом. Таким образом, несмотря на значительную схожесть применения языков XSLT и XQuery (они оба выбирают данные из входных XML документов и создают новые XML документы из этих данных), существуют принципиальные различия в основных сценариях использования, которые привели к некоторым главным различиям в заданных параметрах проектов этих языков.

Различные культуры

В начале этого раздела я описал две причины, из-за которых различаются языки XSLT и XQuery. Мы рассмотрели различия в технических требованиях для этих двух языков; теперь мы рассмотрим те отличия, которые являются следствием разных культур. Они не менее важны: так же как архитектор, проектирующий здания в Токио, должен принять во внимание тот факт, что образ жизни в этом городе отличается от такового в Лос-Анджелесе, так и разработчики компьютерного языка должны работать в рамках определенных традиций. Эти традиции устанавливают критерии, определяющие понятия «удачного» и «неудачного» проектов. Разработка программного обеспечения, так же как музыка или архитектура, является по существу творческой интеллектуальной деятельностью, и ее результат зависит во многом от опыта и творческих предпочтений людей, работающих над проектом, и группы поддержки, обеспечивающей обратную связь.

Проектировщики XSLT ранее работали по большей части с SGML (Standart Generalized Markup Language – Стандартный обобщенный язык разметки). Они были знакомы с обработкой документов, с абстракциями формальной модели, лежащей в основе языка SGML и его языка таблиц стилей DSSSL (Document Style Semantics and Specification Language – Язык семантики стиля и спецификации документа), который сам основан на функциональных языках программирования подобных Schema. Они понимали сложности алгоритмов нумерации страниц, заворачивания слов[4](http://citforum.ru/internet/xml/xquery_xpath_xslt/" \l "b4) и расстановки переносов, а также путей их варьирования, в зависимости от исходного языка текста и соответствующих типографских традиций. Но немногие из проектировщиков языка в прошлом имели опыт работы с технологиями баз данных. Они не были экспертами в методах оптимизации реляционной алгебры, а также не были знакомы с традициями написания отчетов баз данных или вычислениями, связанными с визуализацией данных.

В противоположность им все проектировщики XQuery вышли из мира баз данных. Некоторые из лидеров рабочей группы XQuery (включая нескольких авторов, представленных в этой книге) также сыграли значительную роль в развитии SQL и языков объектных баз данных, подобных OQL. Эти люди принесли с собой знания, полученные за тридцать лет прогресса технологий баз данных – прежде всего в разработке языков запросов и связанных стратегий оптимизации, вместе с постепенным развитием моделей данных, способных обращаться с более сложными структурами, чем традиционная «перфокартная» модель реляционных баз данных 1970-х годов. Однако немногие из этих людей в прошлом подвергались влиянию культуры языков SGML или XML с ее очень отличающимся представлением о структурных ограничениях, ратификации и виде структурных действий, требуемых для обращения с деревьями, возникающими с помощью разметки линейного текста.

Существует и другое заслуживающее упоминания различие в культуре, стоящей за этими двумя языками. Группа, которая разрабатывала XSLT 1.0, состояла из намного меньшего числа активных участников, чем группа XQuery, и в ней был человек, Джеймс Кларк (James Clark), который имел неофициальную роль главного архитектора, а остальные разработчики по существу действовали как подчиненная группа и рецензенты. Группа XQuery никогда не имела в своем составе человека, которого можно было бы назвать главным архитектором в том же смысле. В ней состояли (и все еще состоят) талантливые и высококвалифицированные люди, которые не всегда разделяют одни и те же взгляды. В результате такая группа менее предрасположена к ошибкам, которые могут быть сделаны одним человеком, но в то же время такая организация представляет значительные трудности для поддержания постоянства выбранных подходов для всего языка. Данный подход должен служить гарантией того, что различные решения в раз-личных областях будут сделаны на основе одинаковых критериев, но прежде всего он нужен для того, чтобы язык оставался компактным и простым. Другими словами, XQuery – это язык, разработанный сообществом, в то время как язык XSLT таковым не является.

Сближение концепций: XPath 2.0

В течение тех двенадцати месяцев после того как были опубликованы XSLT 1.0 и XPath 1.0, становилось все более и более ясным, что XQuery не может их игнорировать. Все основные производители программного обеспечения выпустили свои версии этих языков, которые были затребованы сообществом пользователей. В отличие от некоторых других организаций стандартизации, W3C предпочитает создавать свои спецификации последовательными: в W3C не любят наложения и дублирования между различными рекомендациями, которые предлагают несколько решений одной и той же проблемы. Итак, XQuery быстро был объединен с XPath, приняв выражения пути в качестве составной части языка, которая стала соответствовать спецификации XPath, хотя первоначально было много различий в деталях синтаксиса и семантики (так же, как множество пробелов, где синтаксис и семантика не были полностью определены).

Однако для руководства W3C внешнего подобия между языками было недостаточно. Если бы XQuery использовал тот же самый синтаксис, что и XPath, но с различными деталями семантики, то для пользователей в этом было бы мало пользы. На самом деле такая ситуация вызвала бы большую путаницу. Однако XQuery не мог просто включить XPath 1.0 без каких-либо изменений, потому что эти языки имели фундаментальные различия в модели данных и системе типов.

В это же время все больше завоевывала популярность XML Schema. XQuery и XML Schema всегда имели близкие отношения и взаимную зависимость того же самого вида, которую имели языки описания данных и языки для манипулирования данными с самых ранних дней появления технологий баз данных в середине 1960-х. В начале своего развития язык XQuery имел собственную систему типов, которая отделялась от системы типов XML Schema. Но вскоре стала ясна нелогичность такого подхода, и рабочая группа решила, что XQuery будет использовать в качестве своей системы типов XML Schema (несмотря на возражения теоретиков языка, например, Фила Уодлера (Phil Wadler), который приводил убедительные аргументы относительно технической некорректности описания XML Schema как системы типов).

На разработчиков XSLT также оказывалось давление, хотя и менее интенсивное, целью которого была реализация более близкой адаптации XSLT и XML Schema. Это движение породило довольно шумную оппозицию в некоторых кругах, особенно среди тех людей, чьим главным интересом была обработка документов. Но многие из крупных корпораций, все более обширно использующих XML, рассматривали XML Schema в качестве ключевого инструмента для управления интеграцией приложений в пределах и за пределами корпораций, а XSLT – как ключевой инструмент для развития этих приложений. Хотя первоначально никто не был уверен в том, что конкретно это будет означать, но некоторая степень интеграции XSLT (и поэтому XPath) с XML Schema была определена важной стратегической целью.

В итоге было решено, что рабочие группы XQuery и XSL будут сотрудничать при разработке проекта XPath 2.0. XPath 2.0 имел бы систему типов, подобную XML Schema, а XQuery должен стать надмножеством XPath 2.0. Было решено, что заседания, посвященные XSLT и XQuery, будут в дальнейшем проводиться одновременно, а также были организованы совместные заседания для разработки согласованности между XSLT и XQuery в других областях, таких как формализация общей модели данных и правила создания деревьев.

Создание языка XPath 2.0, как и предсказывали некоторые члены рабочей группы, было трудной задачей. Для представителей XSLT было естественным стремление бороться за обратную совместимость с XPath 1.0, в то время как представители XQuery утверждали, что в спецификации XPath 1.0 существовали аспекты, с которыми они не могут мириться. Постепенно были изобретены механизмы, удовлетворяющие обоим требованиям. Например, неявное преобразование типов, принятое в XPath 1.0, было сохранено в качестве «резервного режима», который поддерживался бы в XSLT, но не поддерживался бы в XQuery. В некоторых случаях были определены новые функции и операторы, которые совпадали по функциональным возможностям со старыми, но имели более «чистую» семантику. Часть семантики XPath 1.0, которая первоначально казалась недопустимой, (например, тот факт, что результаты выражений пути всегда находятся в порядке документа), в конце концов была признана приемлемой, особенно после того, как представители XQuery постепенно ознакомились с необычными свойствами структур разметки текста. В некоторых других областях, особенно там, где применение  XPath 1.0 привело к проблемам удобства и простоты использования, пришлось пойти на компромиссы, затрагивающие обратную совместимость, но в основном только в тех областях, где такими мерами был нанесен минимальный ущерб интересам пользователей. Было множество дебатов на тему того, какая часть языка XQuery должна быть включена в подмножество XPath, во время ко-торых некоторые выражали желание сохранить XPath настолько компактным, насколько это возможно, а другие приводили доводы в пользу включения в него любой функциональной возможности, которая могла бы быть полезной в контексте XSLT.

Результатом стал неизбежный компромисс. Язык XPath 2.0 существенно превосходит по масштабам XPath 1.0. В основном увеличение объема произошло из-за определения значительно расширенной библиотеки основных функций, что является хорошим способом для добавления новых возможностей без увеличения сложности языка. Кроме увеличения числа функций (и операторов, которые просто обеспечивают удобный синтаксис для лежащего в основе вызова функции), синтаксис языка был расширен примерно на 40 процентов, и на практике увеличение возможностей языка (в частности, обобщение выражений пути) было достигнуто частично путем устранения ограничений, представленных в XPath 1.0, что фактически сделало язык компактнее[5](http://citforum.ru/internet/xml/xquery_xpath_xslt/" \l "b5).

Сравнение XSLT и XQuery

Мы рассмотрели факторы, ставшие причиной различия языков XSLT и XQuery. В этом разделе мы в деталях исследуем сами различия между этими двумя языками. Везде, где возможно, я попробую объяснить, почему существуют эти различия, хотя во многих случаях единственным реальным объяснением будет то, что две различные команды разработчиков неизбежно придумают разные решения одной и той же проблемы.

Как мы уже видели, XSLT и XQuery имеют в качестве основы язык XPath. Поэтому при сравнении языков мы должны главным образом смотреть на их части, не содержащие XPath. Однако, несмотря на наличие единственного определения XPath, это определение предоставляет некоторую гибкость базовому языку, поэтому на практике некоторые различия в XSLT и XQuery наблюдаются даже на уровне XPath.

Стоит отметить, что существует значительное сходство между этими двумя языками:

Оба языка имеют средства для создания новых узлов в XML-дереве результата. Фактически языки включают две их разновидности: прямой XML-подобный синтаксис, в котором создаваемые элементы пишутся непосредственно в форме XML, и косвенный синтаксис, в котором имена элементов или атрибутов должны быть вычислены во время выполнения.

Оба языка позволяют пользователю самостоятельно определять функции, ко-торые можно вызывать, используя механизм вызова функций XPath. В случае XSLT это новый инструмент в XSLT 2.0 (в XSLT 1.0 допускались функции, моделируемые путем использования именованных шаблонов, но их нельзя было бы вызывать непосредственно из выражений XPath).

Оба языка обеспечивают структуры контроля для вложенных операций повторений или объединений: выражение FLWOR в XQuery и вложенные инструкции xslt:for-each в XSLT (в XSLT 2.0 это дополнено выражением for из XPath 2.0, который является подмножеством выражения FLWOR языка XQuery).  Оба языка допускают определение переменных. В обоих случаях переменные находятся в режиме «только чтение». Оба являются декларативными функциональными языками без назначения значений переменным.

Синтаксис на основе XML

Синтаксис XSLT основан на XML. Таблица стилей – это XML документ. Как мы уже видели ранее, для этого существуют две главных причины: легкость написания таблиц стилей, содержащих большие куски HTML, который будет скопиро-ван непосредственно в документ результата, и возможность использования XSLT для преобразования таблиц стилей.

Разработчики XQuery решили не идти этим путем (существует XML-представление запросов под названием XQueryX, но в действительности оно не является подходящим для использования человеком, поскольку дает представление в виде дерева грамматического разбора XQuery очень низкого уровня). Вместо этого XQuery имеет синтаксис, который имитирует XML там, где это необходимо, особенно при создании целевых элементов. В некоторых случаях этот синтаксис очень похож на XSLT; например, следующий отрывок мог быть написан на любом из этих языков:

<a href="{@href}">see below</a>

Различие состоит в том, что XSLT в действительности является XML (он обрабатывается анализатором XML и должен следовать всем правилам XML), тогда как XQuery просто имитирует XML. Это позволяет создавать вложенные выражения в XQuery таким способом, который невозможен в XSLT; например, в XQuery допустимо (хотя и не очень полезно) написать:

<item size="{ count((<b/>, <c/>, <d/>)) }"/>

что было бы невозможно в XSLT, потому что это не является корректным кодом XML (атрибуты не могут содержать элементы). XQuery таким образом имеет лучшую композиционность, чем XSLT, но это достигается определенной ценой: поскольку код XQuery не является чистым XML, то необходимо изучить новые правила обработки символов пустого пространства и символьные ссылки, а стандартные инструменты XML (такие как редакторы) не могут использоваться для манипулирования текстом запроса. Для такого выбора имеется серьезное основание: вероятно, XQuery будет часто использоваться в качестве вложенного языка с языками программирования, такими как Java и C#, или даже SQL, где инструменты XML в любом случае не были особо полезны.

Семантика создания элементов

За внешней похожестью инструментов для создания элементов и атрибутов вдереве результата до недавнего времени скрывались существенные различия в семантике модели обработки. Однако в последнем рабочем проекте XSLT 2.0 был приближен к модели XQuery.

В XSLT 1.0 необходимо, чтобы дерево целиком было построено до того, как клюбой его части можно было обратиться с использованием выражения XPath. В XSLT 1.0 даже невозможно было получить доступ к построенному дереву, известному под названием фрагмент дерева результата, не используя функцию расширения node-set(), которая, несмотря на широкую распространенность, фактически не является частью стандарта. Это ограничение исчезло в XSLT 2.0, но до проекта мая 2003 года действовало другое ограничение, согласно которому XPath мог осуществлять операции только с деревьями, представляющими документ целиком (то есть дерево с узлом документа в качестве корня). Это было естественным следствием двухъязыковой модели, посредством которой инструкции XSLT могут вызывать выражения XPath, но не наоборот. Временное дерево всегда создается посредством объявления переменной, а к узлам в пределах дерева можно обратиться только с помощью выражения XPath, которое ссылается на эту переменную. Правила видимости переменных гарантируют, что дерево всегда будет полностью построено, прежде чем будет сделана ссылка на любой из его узлов. Это иллюстрируется листингом 3.2.

Листинг 3.2. Создание временного дерева в XSLT

<xslt:variable name="tree">

<calendar>

<month number="1" length="31"/>

<month number="2" length="{if ($leap-year)

then 29 else 28}"/>

<month number="3" length="31"/>

<month number="4" length="30"/>

. . .

<month number="12" length="31"/>

</calendar>

</xslt:variable>

. . .

<xslt:value-of select="sum($tree/calendar/month/@length)"/>

. . .

Тот факт, что к деревьям нельзя получить доступ до завершения их создания, означает то, что можно описать создание дерева в XSLT с использованием нисходящей модели, в которой родительские узлы должны быть созданы перед своими дочерними узлами. Конечно, в функциональном языке фактический порядок выполнения не определен, так что это было бы просто способом описания эффекта языка и не обязательно описанием фактической работы реализации. Хотя на практике большинство XSLT 1.0 процессоров, вероятно, следовали этой модели весьма точно.

В отличие от этого конструктор элемента XQuery является простым выражением и может использоваться в любом месте, где могут встречаться другие виды выражений. Например, вполне правильно написать так:

sum(

<calendar>

<month number="1" length="31"/>

<month number="2" length="if ($leap-year)

then 29 else 28"/>

<month number="3" length="31"/>

<month number="4" length="30"/>

. . .

<month number="12" length="31"/>

</calendar> / month / @length )

Хотя отличие этого примера от предыдущего выглядит весьма незначительным, оно имеет большое значение для детальной семантики модели. XQuery рассматривает выражение, создающее узел атрибута или элемента, таким же образом, как илюбое другое выражение. Это означает, что семантика описана восходящим способом. Так же как в выражении x \*(y-1) сначала выполняется вычитание, а затем умножение, так и в выражении <a>{$x+1}</a>, сначала выполняется сложение, затем создается текстовый узел, содержащий результат, а потом создается узел элемента в качестве родительского для этого текстового узла.

Это означает, что в XQuery возможно манипулировать частично построенными деревьями (деревьями, которые не имеют узела документа в качестве своего корневого узла). Например, выражение пути может ссылаться на узел атрибута, который не связан ни с одним элементом. Атрибут может быть присоединен к элементу «позже», как показано ниже:

let $att1 := attribute code { "23" },

$att2 := attribute desc { "24" } return

if ($condition)

then <a> {$att1} </a>

else <a> {$att2} </a>

Конечно, это также означает, что узел атрибута, представленный $att1 или $att2, можно добавить к нескольким различным элементам. Поскольку атрибут не может на практике иметь два родительских узла, то семантика требует создания идентичной копии каждый раз, когда атрибут присоединен к элементу. В формальной модели такое копирование осуществляется по всему дереву: каждый раз, когда дочерний узел добавляется к родительскому узлу, дочерний узел копируется. На практике процессоры XQuery обычно избегают этого копирования и в большинстве случаев будут использовать реализацию такой же нисходящей обработки, как и процессоры XSLT 1.0. Но для увеличения общности языка формальная модель в XQuery полностью отличается от модели XSLT.

Так как XQuery копирует поддеревья неявно, тогда, когда они добавляются к новому родительскому узлу, то не требуется явной инструкции для осуществления копирования. В отличие от этого XSLT 1.0 сначала создает дерево целиком, а затем позволяет, если требуется, копировать это дерево явно с использованием инструкции xslt:copy-of. В самом последнем проекте XSLT 2.0 модель обработки была изменена и стала очень близкой к модели XQuery. Создание дерева описано восходящим способом, и узлы доступны до того, как они будут присоединены к родительскому узлу. Большинство пользователей не заметит изменений, а различие затрагивает многие незначительные детали, такие как способ, которым новые узлы ратифицируются с помощью типов схемы, и способ работы пространства имен. Это также означает, что необходима некоторая осторожность при использовании абсолютных выражений пути. Например, если корнем дерева является элемент a с дочерним узлом b, то для того чтобы выбрать элементы b, следует написать /b, а не /a/b, как можно было бы ожидать. Синтаксис /a/b (который является сокращением для root(.)/child::a/child::b) работает только там, где элемент с именем a является дочерним корневого узла.

Функции и именованные шаблоны

И XQuery, и XSLT 2.0 предоставляют возможность определять функции. Опять же, на первый взгляд эти инструменты не очень отличаются. В XQuery мы можем изменить порядок последовательности с помощью рекурсивной функции:

define function reverse ($seq as item()\*)

as item()\* {if (count($seq) < 2)

then $seq

else (reverse(subsequence($seq, 2)), $seq[1])

}

В XSLT 2.0 мы можем написать ту же самую функцию следующим образом:

<xslt:function name="reverse" as="item()\*">

<xslt:param name="seq" as="item()\*"/>

<xslt:sequence as="item()\*" select="

if (count($seq) &lt; 2)

then $seq

else (reverse(subsequence($seq, 2)), $seq[1])"/>

</xslt:function>

Небольшое различие. Однако тот факт, что XSLT является двухъязыковой системой, снова создает различия. Поскольку XSLT имеет один язык для создания узлов в дереве результата и другой язык (XPath) для выбора узлов из исходного дерева, то он фактически имеет два механизма для определения того, что логически является функциями: xslt:function для функций, которые могут вызываться из выражений XPath и возвращают значения; xslt:template для подпрограмм, которые можно вызывать на уровне XSLT и которые записывают узлы в дерево результата. XQuery имеет только один язык, поэтому его выражения более композиционные, а это означает, что они принимаются единственным механизмом определения функций.

Выражения FLWOR

Подобно тому, как ядро XPath составляет выражения пути, в основе языка XQuery находится выражение FLWOR. Выражения FLWOR выполняют ту же самую роль в XQuery, как и выражение SELECT в SQL. Действительно, выражение SELECT языка SQL и его преемники в различных постреляционных языках сильно повлияли на данную конструкцию языка XQuery.

Тем не менее, выражение FLWOR не содержит ни одной конструкции, которая не могла бы выполнить перевод запроса в термины XSLT. Оператор for выполняет перевод непосредственно в xslt:foreach, оператор let в xslt:vari-able, оператор where в xslt:if и order by в xslt:sort.

Выражения FLWOR в общем случае воспринимаются в декларативных терминах как основанные на действиях реляционного исчисления, таких как декартово произведение, выбор и проекция. Напротив, аналогичные конструкции XSLT часто воспринимаются в процедурных терминах: xslt:foreach считается аналогом цикла в процедурном языке программирования. Но если смотреть сквозь формальности языка, используемого при объяснении семантики, то можно увидеть, что на практике имеется очень небольшое различие в функциональных возможностях этих двух конструкций.

Вероятно, является верным утверждение о том, что большинство процессоров XSLT фактически выполняют инструкции xslt:foreach с использованием процедурного подхода, очень похожего на формальную модель: в Saxon это происходит именно так. Там, где две инструкции xslt:foreach являются вложенными, они, вероятно, будут реализованы посредством использования вложенного цикла. В выражениях FLWOR реализация, вероятнее всего, использует набор методов оптимизации, разработанных для реляционных баз данных, которые могут послужить причиной значительной перестановки порядка выполнения.

Но все это не является различиями, вызванными семантикой языка. Эти различия возникают потому, что XQuery используется в другой среде: его сферой является поиск информации в больших постоянных базах данных. В этой среде эффективность запроса зависит от обнаружения заранее созданных индексов, которые дают быстрый доступ к определенным наборам, встречающимся в запросе. Поэтому суть оптимизации XQuery заключается в перезаписи запроса таким образом, чтобы достичь оптимального использования индексов. Процессоры XSLT в общем случае не имеют такой роскоши: предварительно созданных индексов не существует. Доступный диапазон путей доступа намного более ограничен; следовательно, возможности увеличения производительности путем переписывания запроса также ограниченны.

Различие в подходах также возникает из-за того, что XSLT сосредоточен на обработке документо-ориентированного XML, тогда как XQuery делает акцент на обработку информационно-ориентированного XML. При обработке документов важен порядок как исходных документов, так и документа результата, и они часто совпадают. Поэтому обычно лучшей стратегией в этом случае является последовательная обработка. При обработке данных порядок (конечно, в реляционной традиции) часто не имеет значения и стратегии, изменяющие порядок выполнения, могут быть в данном случае высокоэффективными.

Правила шаблонов

Одним из немногих действительно значительных различий между XSLT и XQuery является использование в XSLT правил шаблонов: подход, использующий управляемые событиями объекты, в котором описание способа обработки индивидуальных элементов в исходном дереве отделено от любых предположений о контексте, в котором появляются эти элементы. XQuery не имеет соответствующего инструмента, хотя можно доказать, что этот факт не уменьшает выразительную способность языка: любая инструкция xslt:apply-templates может в принципе быть переведена в условное выражение, которое выполняет прямые вызовы явного шаблона в зависимости от свойств выбранного узла.

Возможности, предоставляемые правилами шаблонов, заключают в себе не дополнительную функциональность, а модульность и потенциал для изменения. Шаблоны позволяют таблицам стилей делать меньше предположений о структуре исходного документа, и поэтому они более гибки к изменениям в этой структуре. Это, конечно, очень важно для обработки документо-ориентированного XML и намного менее важно для информационно-ориентированного XML.

Сами возможности подхода проектирования управляемых событиями объектов, принятого для правил шаблонов, также является причиной его сложности: при этом усложняется оптимизация. Несмотря на то, что процессоры XSLT используют любые хитрости для ограничения числа шаблонов, с помощью которых проверяется каждый узел, и в будущем могут использовать информацию схемы для более строгого ограничения этого выбора, оптимизация таблицы стилей в целом является трудной задачей, в отличие от оптимизации индивидуальных правил шаблонов, потому что имеется очень мало статической информации о порядке вызова правил. Процессоры XSLT могут позволить осуществлять проверку соответствия правилам шаблонов, потому что самым обычным действием таблицы стилей является обработка каждого узла только один раз в последовательном порядке. XQuery не может позволить себе иметь это свойство, потому что сама осуществимость выполнения запроса в течение разумного отрезка времени зависит от статического анализа запроса для разработки стратегии выполнения, в которой совершается обход минимально возможного числа узлов. В требованиях, предъявляемых к языкам, как и в прикладных областях, для которых они предназначены, существуют принципиальные отличия, что ведет к подлинным различиям в философии проектов.

Оси

XQuery не имеет полного набора осей, используемых в выражениях пути XPath: это единственная часть XPath, которая не включена непосредственно в XQuery. XQuery не имеет осей following и preceding, following-sib-ling и preceding-sibling, а также оси ancestor. Почему было принято такое решение? Мне сложно объяснить этот факт, потому что я приводил доводы против такого решения, но аргументы обеих сторон были разумными, и я попробую воздать им должное.

Подход, используемый в XPath для выбора узлов в дереве, определяется способом, который некоторые называют навигационным. Я предпочитаю называть его функциональным, потому что слово навигационный подразумевает процедурные алгоритмы, но, по сути, означает то, что поиск узлов ведется на основании их взаимоотношений с другими узлами: три шага вверх, затем два влево, затем один шаг вниз. Такой подход традиционно является трудным для оптимизации. В базах данных, следующих реляционной традиции, подход состоит в выборе объектов на основании их свойств, а не описания маршрута, с помощью которого можно достичь этих объектов. Это заставляет некоторых людей, придерживающихся реляционной традиции, относиться с большим подозрением к выражениям пути и беспокоиться о сложности их оптимизации.

Некоторые разработчики реализаций XQuery также используют инструменты, существующие в базах данных, в которых структуры хранения и индексации хорошо настроены для реляционных данных и реляционных запросов. Адаптация этих инструментов к иерархическим структурам данных и рекурсивным выражениям пути – нелегкая задача, и запросы, которые хорошо вписываются в реляционный шаблон, вероятно, чаще будут выполняться в этом случае намного эффективнее, чем остальные запросы.

Любое выражение, включающее оси, отсутствующие в XQuery, может быть переписано таким образом, чтобы можно было избежать использования этих осей. Возьмем, например, запрос «Найти элемент figure, сразу за которым следует другой элемент figure без любого промежуточного текста или элемента». В XPath мы написали бы следующее:

//figure[following-sibling::node()[1][self::figure]]

XQuery не имеет оси following-sibling, поэтому как же мы сможем выразить этот запрос? Одним из возможных способов будет такой:

for $p in //\*

for $f at $i in $p/figure

where $p/node()[$i+1][self::figure]

return $f

Хотя данный способ кажется очень запутанным тем, кто знаком с XPath, мне сказали, что этот способ выглядит очень естественно для людей, думающих в терминах SQL. Является ли этот способ более легким для оптимизации, чем версия запроса XPath? Только не для процессора, оперирующего моделью дерева, выполненной в терминах полносвязной объектной модели либо в памяти, либо в постоянном хранилище (обычно в собственной базе данных XML, подобной [Software AG’s Tamino](http://www.softwareag.com/tamino)). Но для разработчиков, которые перевели дерево в структуру кортежей, представленную в табличном хранилище, – кто знает?

Возможно все сводится к различным представлениям относительно важности таких запросов. В документо-ориентированном XML важен порядок, и объекты вероятнее всего, будут идентифицированы по их взаимоотношению с другими объектами, а не по их свойствам. В информационно-ориентированном XML объекты, скорее всего, будут идентифицированы по их содержанию. Так что я сильно подозреваю, что реальной основной причиной принятия этого решения был естественный уклон к информационно-ориентированному XML, присутствующий в группе XQuery.

Строгость типа

В XSLT 1.0 и XPath 1.0 система типов была очень слабой в том смысле, что было определено очень мало типов, а большинство операций принимали аргументы любого типа. Как и в языках подготовки сценариев, таких как JavaScript, если вы использовали неправильный вид объекта, то система приложит все усилия, чтобы преобразовать его к требуемому типу. Поэтому вы могли складывать строки, выполнять конкатенацию чисел, могли сравнивать множество узлов с числом и получать при этом результат. Не всегда это был тот результат, которого вы, возможно, ожидали, но он был. Очень немногие действия вызвали бы когда-либо ошибку во время выполнения.

На это была своя причина. Главной целью XSLT был перевод XML в форматы представления, подобные HTML или PDF. Первоначально ожидалось, что этот процесс обычно будет выполняться на стороне клиента, в браузере. И менее всего вы хотите, чтобы при предоставлении документа браузером появилось сообщение об «ошибке в таблице стилей». Если таблица стилей в сумме представляет собой набор чисел и обнаруживается, что одно из полей в исходном документе не является числовым, то на экране должно появиться хоть что-нибудь, даже если это только звездочки, а не пустой экран.

Другой причиной для принятия такой системы типов было понимание того, что в мире XML все данные являются в конечном счете текстом. XML главным образом является (хотя мы часто забываем об этом) способом разметки текста и обозначения его структуры. Другие типы данных, такие как целые числа и логические значения, существуют только потому, что мы решаем выделить их из массы текста: они являются абстракциями, тогда как текст реален. С таким взглядом на мир будет очень естественным представление о том, что любые действия с данными должны неявно преобразовать любые передаваемые им входные данные в типы данных, для обработки которых эти операции предназначены.

В противоположность этому, разработчики XQuery всегда были твердо уверены в том, что в ядре языка должна находиться развитая система типов. Имеется множество веских причин, по которым языки программирования в общем случае и языки запросов в частности имеют тенденцию включать в себя строгие системы типов. С момента появления баз данных существовала идея отделения описания данных, или схемы от манипуляций данными, или языка запроса. Описание данных отражает общее понимание данных, которое разделяется сообществом пользователей, даже при том, что каждый пользователь создает различные запросы. Это описание определяет типы объектов, которые могут находиться в базе данных, и оно также естественным образом формирует систему типов языка запросов, потому что запрос, который не выражен в терминах этих типов объектов, не имеет смысла.

Процессор запросов использует информацию о типах, полученную из схемы базы данных, двумя основными способами. Первый вариант использования заключается в обнаружении ошибок. Полезно обнаруживать ошибки как можно раньше, и, конечно, желательнее всего обнаружить ошибку, а не возвращать неправильные данные. Каждый пользователь SQL получал ответ на запрос, который фактически не является ответом на тот запрос, который он собирался выполнить. Такие случаи невозможно полностью предотвратить, но хорошая система типов способна заранее обнаружить многие из наиболее грубых ошибок. Вторым вариантом использования информации о типах является оптимизация. Чем больше информации имеет система во время компиляции, тем успешнее она может разработать эффективный план выполнения запроса, и информация о типах объектов, обрабатываемых запросом, является ключевой частью картины.

Как это часто происходит, взгляды людей, работающих в группе XSL, касающиеся контроля типов, значительно эволюционировали со времен XSLT 1.0. Отчасти это произошло потому, что XSLT оказался востребованным в очень широком диапазоне клиентских задач, не связанных с вебпросмотром, а отчасти из-за появления XML Schema, ставшей мощным средством влияния на принятие XML крупными компаниями, даже при том, что она яростно отвергалась поклонниками SGML. Взгляды, присущие разработчикам XQuery, также претерпели изменения, в результате чего с течением времени все более признавалась законной потребность обработки полуструктурированных данных. Этот процесс представлял собой постепенное сближение позиций двух групп, и обе стороны все больше признавали факт существования широкого спектра требований, предъявляемых к технологиям обработки данных. Но до сих пор XSLT в большей степени, чем XQuery, склоняется к «свободному контролю типов», так как, во-первых, все еще существует большая потребность обработки документов, для которых не определена схема, а во-вторых, проект языка с его правилами шаблонов на основе подхода разработки управляемых событиями объектов сильно затрудняет эффективное использование статической информации о типах как для поиска ошибок, так и для оптимизации. Таким образом, можно сказать, что XQuery сделал первые шаги к более свободному контролю типов, в то время как XSLT сделал пробные шаги в противоположном направлении.

Диапазон представлений о принципах построения языков в обеих рабочих группах гарантировал, что обсуждение системы типов составит большую часть времени и затраченных усилий при разработке этих двух языков. Многие пользователи могут не заметить этого, так как и в XSLT, и в XQuery основное число пользователей может благополучно игнорировать большинство сложностей системы типов. Например, в обоих языках имеются тщательно разработанные инструменты для ратификации выходных документов с помощью схемы. Некоторые дизайнеры XQuery всегда стремились сделать возможным вывод сообщений об ошибках во время компиляции в том случае, если запрос не способен создать результат, который соответствует требуемой схеме, или даже разработать еще более строгое ограничение – сделать возможным появление сообщения об ошибках, если запрос способен создать результат, не соответствующий требуемой схеме. Однако я подозреваю, что многие пользователи будут просто создавать нетипизированный XML, и если, им потребуется ратифицировать его, они сделают ратификацию отдельным процессом, как только запрос или преобразование будут завершены.

Методы оптимизации

За три года, прошедших с тех пор, как XSLT стал рекомендацией W3C, появилось множество реализаций этого языка, у которых было некоторое время для совершенствования. Время от времени публиковались сравнения производительности этих реализаций, и, хотя эти сравнения имеют те же самые проблемы, как и все подобные эталонные тесты, тем не менее, это побуждало производителей улучшать качество своего программного обеспечения. Разработчики также поддерживали обратную связь со своими пользователями, которые указывали на проблемы производительности. Все программы были значительно улучшены со времени своего первого появления.

Поэтому разумно ожидать, что производительность обработки данных в XSLT является к настоящему времени действительно широко изучена, и что уроки, полученные производителями XSLT, будут доступны для ознакомления разработчикам XQuery, что даст им определенное преимущество. Тем не менее, к сожалению, было опубликовано очень малое количество материалов, посвященных методам, используемым в реализациях XSLT. Некоторые из программных продуктов поставляются пользователям с открытым исходным кодом, поэтому можно изучить методы, использующиеся в них, читая исходный код, но это сильно отличается от чтения опубликованных статей, написанных самими разработчиками. Кроме всего прочего, исходный код не говорит вам о том, какие методы были испробованы, но не удались в реализации, или какие идеи все еще присутствуют в продукте даже при том, что на практике они мало влияют на результат.

Замечания, приведенные здесь, возникли главным образом из моего опыта работы с [Saxon](http://saxon.sf.net/). Идеи, используемые в Saxon, подвергались влиянию идей, используемых в [xt](http://www.blnz.com/xt/index.html) и [Xalan](http://xml.apache.org/). Одной из интересных особенностей мира XSLT является тот факт, что в нем существует большое число продуктов с открытым исходным кодом, каждый из которых конкурирует с другими, но все они не имеют возможности скрыть свои секреты. Конечно, существуют другие продукты, среди которых следует отметить [MSXML](http://www.microsoft.com/xml) фирмы Microsoft, которые не обнародуют секреты своих проектов, так что весьма возможно, что список, приведенный здесь, не включает некоторые важные идеи.

Причина, побудившая меня написать этот раздел, следующая. Я думаю, что, вероятно, между сложностями оптимизации XSLT и оптимизацией XQuery будет много общего. Эти два языка имеют множество различий, к которым я снова обращусь в конце раздела; но я считаю, что подобие языков является важным свойством, и многое из опыта работы с XSLT также окажется уместным для XQuery.

На что тратится время?

Обсуждение данного вопроса полезно начать с анализа того, на что процессор XSLT тратит свое время. Конечно, это зависит от особенностей обсуждаемой проблемы. В диаграммах на рисунках 3.1 и 3.2 показан анализ распределения времени для двух конкретных задач преобразования. На первом рисунке представлена работа, заключающаяся в представлении XML-версии спецификации XSLT в виде текста HTML; а процесс, показанный на другом рисунке, использует ту же самую таблицу стилей, но применяет ее к документу, намного меньшему по объему. Результаты, приведенные на диаграммах, были получены при использовании Saxon 6.5.2.

На рисунке 3.1 показано распределение времени, потраченного на четыре основных действия при применении таблицы стилей к большому документу: компилирование таблицы стилей, создание дерева исходного документа, выполнение самого преобразования и приведение в последовательную форму дерева результата в виде HTML. В этом случае объем таблицы стилей составляет около 1350 строк, разделенных на три модуля; исходный документ имеет размер 600 Кб, а документ результата – 860 Кб. Полное время преобразования на моей машине составило приблизительно 8,3 секунды. В Saxon три стадии выполняются последовательно: компилирование, создание дерева и преобразование/приведение в последовательную форму. Распределение времени между преобразованием и приведением в последовательную форму рассчитано путем сравнения времени выполнения процессов без приведения в последовательную форму HTML (только с передачей результата написанному пользователем обработчику содержания, который ничего с ним не делает).

Этот вид преобразования обычно выполняется один раз. Может быть получена только небольшая выгода с использованием предварительной компиляции таблицы стилей и амортизации затрат времени на компиляцию при многократных преобразованиях различных источников документов, так как экономия времени, которая может быть достигнута этим способом, составила бы в лучшем случае 15 процентов. Общее количество затраченного времени (8,3 секунды) также включает в себя большие затраты на «инициализацию» виртуальной машины Java – загрузку и инициализацию всех требуемых классов и достижение устойчивого состояния JIT (Just-In-Time – системы оперативной поставки узлов) компилятора Java. Реальная таблица стилей выполняет четыре или пять проходов по исходному документу (один для отображения основного текста, дополнительные проходы для создания оглавления, глоссария и различных приложений), а также много раз получает доступ по ключу для создания развернутых перекрестных связей в форме гиперссылок. Поэтому поразительно то, что затраты времени на анализ исходного документа и построение структуры его дерева почти столь же велики, как и затраты на само преобразование.

Приведение в последовательную форму: 670 мс

Компиляция:

Для контраста на рисунке 3.2 показано преобразование, которое более типично для преобразования XML в HTML «по требованию», выполняющееся на веб-сайтах, где данные хранятся в XML и отображаются каждый раз, когда их содержание затребовано пользователем. В этом примере используется та же самая таблица стилей; единственное отличие в том, что исходный документ намного меньше по объему – на этот раз всего 8 Кб. Это уменьшает полное время преобразования до 1,6 секунды, из которых 79 процентов времени тратится на компиляцию таблицы стилей. Очевидным следствием этого факта является необходимость выполнения компиляции таблицы стилей только один раз с последующим многократным ее применением при таком сценарии использования.

Другой интересный факт заключается в том, что, хотя таблица стилей является той же самой, как и в примере, показанном на рисунке 3.1, отношение времени преобразования времени, затраченному на анализ и создание дерева, в этом случае намного выше. Вероятным объяснением этого факта будет то, что стадия создания дерева ответственна за запрос всей памяти, используемой для преобразования, и затраты на выделение требуемой памяти увеличиваются более чем линейно в зависимости от размера исходного документа, так как на это влияет страничная организация памяти или ее фрагментации. Также может иметь место тот факт, что стадия преобразования включает в себя затраты времени на запуск, которая не зависит от размера документа.

Приведение в последовательную форму:  50 мс

Какие последствия все это имеет для XQuery? Многие системы XQuery будут работать с базами данных, в которых данные предварительно загружены в структуру базы. Структура базы данных выполняет ту же роль, как и расположенное в памяти дерево, используемое процессорами XSLT, но эта структура создается только однажды, когда документ загружается или заменяется в базе данных, а не при выполнении каждого запроса. При этом не подразумевается, что время, затраченное на создание дерева, может игнорироваться – наоборот, время, расходуемое на загрузку и обновление данных, всегда было критическим фактором для баз данных, поддерживающих эффективные иерархические пути доступа, и этот факт можно легко забыть после двадцати лет доминирования реляционных баз данных с их плоской структурой хранения. Системы XQuery должны пойти на точно такой же компромисс, как и процессоры XSLT, между временем, затрачиваемым на создание и индексирование дерева и скоростью путей доступа, обеспечиваемых деревом, как только оно было создано. Во многих случаях им придется принимать более трудные решения, потому что процессор XSLT может создать дерево со знанием того, какое преобразование будет иметь место, тогда как деревья, созданные процессором XQuery, должны будут поддерживать множество различных видов запросов.

Наблюдения, касающиеся времени компиляции, также относятся к процессору запросов. Хотя запросы объемом 1000 строк вероятно будут менее обычными, чем таблицы стилей объемом 1000 строк, тем не менее будут существовать сложные запросы, и сложность их компиляции, вероятно, будет пропорционально больше, чем для таблиц стилей XSLT из-за критической важности нахождения оптимального пути доступа к данным, когда происходит обработка источников данных объемом в несколько гигабайт, а не просто несколько килобайт. Но как и в случае с XSLT, будут иметь место одноразовые запросы, для которых не будет никакой выгоды от многократного использования скомпилированного запроса, и будут запросы, где такое использование существенно.

Рассмотрев фактическое время выполнения запроса или преобразования в некоторой перспективе, мы теперь обратимся к различным методам оптимизации, которые могут быть использованы для увеличения производительности.

Внутренняя эффективность кода

В книгах, посвященных реляционным базам данных, вы не найдете того, что известно каждому конструктору систем баз данных: наиболее важным способом получения хорошей производительности системы является написание быстро выполняемого кода. Большинство запросов в системах баз данных и большинство преобразований XSLT являются весьма простыми, и самый опытный оптимизатор в мире может сделать очень немногое для улучшения производительности простого запроса. Если xt обрабатывает ту же самую таблицу стилей в пять раз быстрее, чем Xalan (что иногда так и бывает), то это происходит не из-за более качественной оптимизации, а в основном по причине того, что Джеймс Кларк пишет очень эффективный код.

С эффективностью кода также связана разработка эффективных структур данных. Дизайн структуры исходного дерева является критическим для производительности XSLT. Важен компромисс между временем выполнения и объемом (пользователи хотят преобразовывать документы даже такого размера, как 100 Мб), так же как и компромисс между временем, затраченным на создание дерева, и временем, затраченным на навигацию внутри него. Упрощенное представление дерева, использующее один объект Java для каждого узла в дереве, было бы совершенно неадекватным в обоих случаях. Saxon использует адаптивный подход, в котором многие из путей доступа (например, указатели от узлов на их предшествующие сестринские узлы) создаются только тогда, когда эти пути доступа действительно используются.

Но эта книга не о том, как написать хороший код или спроектировать эффективные структуры данных в Java или любом другом языке программирования. Поэтому, рассмотрев этот вопрос, я далее перейду к другим факторам, более специфичным для обработки XSLT и XQuery.

Конвейерная обработка и отложенное вычисление

Часто кажется, что мир функциональных языков программирования, таких как Schema и Haskell, и мир систем управления базами данных имеют мало общего. Но теория в обеих областях имеет нечто общее: идею конвейерной обработки. Терминология может различаться, но сама идея остается одной и той же.

В реляционной модели запрос SQL может быть переведен в выражения реляционной алгебры с использованием таких операций, как ограничение, проекция, объединение, сортировка и слияние. Все эти операции работают в режиме последовательной обработки наборов записей: каждое действие принимает на входе множества кортежей и производит множества (или иногда последовательности) кортежей в качестве своего результата. Фактическое размещение в памяти всех этих множеств кортежей было бы очень ресурсоемким методом. Динамическое распределение памяти является ресурсоемким процессом, а память – ограниченным ресурсом: если использовать больший объем памяти для промежуточных результатов, то меньший ее объем будет доступен для других целей, таких как кэширование. Те же самые факторы применимы к функциональным языкам про-граммирования, основанным на манипуляциях со списками, и одинаково справедливы для XPath, семантика которого определена в терминах множеств узлов. Размещение этих наборов в памяти является очень дорогостоящим процессом.

Технология, используемая для избежания выделения памяти под промежуточные результаты, называется конвейерной обработкой. Способ действия конвейерной обработки заключается в том, что любая операция, например, ограничение, реализована таким образом, чтобы на запрос другой операции возвращался только один объект, и в свою очередь один объект запрашивался бы от зависимых операторов. Дерево операторов, которое составляет выражение SQL или XPath, таким образом представлено во время выполнения последовательностью так называемых итераторов, каждый из которых поддерживает операцию get-next, возвращающую следующий объект в потоке. В реляционной системе объекты являются кортежами, и конвейер описан в терминах движения потока кортежей, поставляемых одним узлом в дереве выражения своему родительскому узлу в дереве выражения.

Не все операции XPath могут быть включены в конвейер. Очевидным примером является функция last(): чтобы определить значение функции last() в выражении типа $x[last() idiv 2] (которое возвращает объект в середине последовательности), вы должны знать, сколько объектов возвращает текущая операция, и единственный путь определения этого количества состоит в том, чтобы прочитать их все. Это нарушает конвейер и поэтому является затратным по использованию ресурсов действием. На практике существуют две стратегии реализации функции last(). Первая заключается в вычислении содержащегося выражения (в данном случае – $x) один раз и сохранении результата в памяти. Вторая заключается в том, что содержащееся выражение вычисляется дважды: первый раз для подсчета узлов и второй раз для передачи их следующему оператору в конвейере. Иногда лучше ис-пользовать первый вариант, иногда – второй: это является тем решением, при приня-тии которого имеет значение качество оптимизатора.

Другим обычным действием, нарушающим конвейер, является сортировка. Одна из технологий, часто используемых оптимизаторами, заключается в перезаписи выражения таким образом, чтобы сортировка выполнялась в последнюю очередь. Это часто может уменьшить число объектов, которые будут отсортированы, и может устранить необходимость многократных сортировок. Например, для выражениий XPath, записанных в форме a/b/c, семантика языка гарантирует, что результаты будут находиться в порядке документа. Но нет никакой необходимости сортировать промежуточные результаты выражения a/b (или b/c, в зависимости от стратегии вычисления). Полное выражение пути может быть вычислено в одном конвейере и отсортировано в самый последний момент.

Выражение пути часто может быть вычислено вообще без выполннения сортировки. Процессоры XSLT, подобные Saxon, имеют по крайней мере три метода, позволяющие избежать сортировки:

Выполнение статического анализа выражения пути для определения того, что результаты «естественным образом» отсортированы: то есть при вычислении с использованием «очевидной» стратегии вычисления результаты будут автоматически находиться в порядке документа. Правила для этой технологии являются весьма тонкими; не всегда очевидно, например, что в то время как оба выражения //b и a/b/c являются естественным путем отсортированными, выражение //b/c таковым не является.

Вычисление инвертированных осей в прямом порядке, чтобы сделать выражение естественным образом отсортированным, когда иначе оно таковым бы не являлось.

Обнаружение контекста, в котором используется выражение пути для того, чтобы осознать, что хотя семантика языка требует сортировать результаты, в некоторых контекстах это никак не влияет на результат. Очевидными примерами являются выражения, в которых выражение пути используется в качестве аргумента таких функций, как count(), sum() или boolean(). В XQuery такое положение также возникает в том случае, когда выражение пути используется в пределах выражения FLWOR, которое имеет оператор ORDER BY для определения порядка результатов. Однако здесь имеется и другая тонкость: для некоторых функций, таких как count(), требуется удаление дубликатов узлов из результата, тогда как для других, таких как boolean(), это не требуется.

Хотя устранение сортировки полезно хотя бы потому, что сортировка является ресурсоемким процессом, основной пользой от ее устранения является улучшение конвейерной обработки.

С конвейерной обработкой тесно связана другая технология, известная из литературы по функциональному программированию: отложенное вычисление. Принципом отложенного вычисления является избежание выполнения действий до тех пор, пока их результаты не станут действительно необходимы. Такой подход имеет следующие преимущества. Во-первых, вам не требуется память для хранения результатов. Во-вторых, вы можете обнаружить, что результаты могут вообще не потребоваться на практике.

В листинге 3.3 приведен простой пример работы отложенного вычисления в XSLT.

Листинг 3.3. Пример отложенного вычисления в XSLT

<xslt:param name="account-nr"/>

<xslt:template match="/">

<xslt:variable name="transactions"

select="/\*/transaction[account=$account-nr]"/>

<xslt:choose>

<xslt:when test="starts-with($account-nr, 'Z’)">

<xslt:value-of select="closing-balance"/>

</xslt:when>

<xslt:otherwise>

<xslt:value-of select="opening-balance +

sum($transactions/value)"/>

</xslt:otherwise>

</xslt:choose>

</xslt:template>

Инструкция xslt:variable потенциально является трудоемкой для вычисления: она включает выбор всех транзакций для данного счета, что, вероятно, означает необходимость просмотра всего исходного документа. А теперь посмотрите, как используется результат. В одной ветви условия xslt:choose переменная вообще не используется. В другой ветви она используется только в качестве аргумента функции sum(). Это означает, что путем задержки вычисления переменной до тех пор, пока она фактически не будет использована, мы могли бы вообще избежать вычисления ее значения; и даже если нам потребуется такое вычисление, мы можем включить его в конвейер таким образом, чтобы не было необходимости хранить в памяти список элементов transaction, и мы, конечно, можем избежать сортировки результатов.

Хитроумной частью отложенного вычисления является тот факт, что значение выражения XPath зависит от контекста, в котором оно появляется (текущий узел, значения других переменных, пространства имен, которые находятся в области видимости), поэтому значимые части контекста должны быть сохранены, так же как и само выражение. Это означает, что важной ролью оптимизатора XPath является определение зависимостей выражения – частей контекста, от которых оно зависит и которые должны быть сохранены при выполнении отложенного вычисления.

Конвейерная обработка и отложенное вычисление, вероятно, будут столь же важными для процессора XQuery, как и для процессоров XSLT и XPath.

Перезапись выражений

В теории реляционных баз данных термин оптимизация используется почти как синоним перезаписи запроса. Работа оптимизатора заключается в следующем: берется дерево, представляющее запрос в том виде, в котором он создан синтаксическим анализатором языка из первоначального текста запроса, и затем оно перестраивается в другое (но эквивалентное) дерево, анализ которого может быть осуществлен более эффективно. Для осуществления этого процесса используются многие стандарты методов, такие как приближение ограничений к листьям дерева. Выбор различных операторов для реализации функциональной возможности реляционного слияния был основной темой для научных журналов в течение почти тридцати лет.

Другим мощным средством, используемым оптимизатором XPath, также является перезапись выражений. Существуют два метода ее реализации. Первый вариант заключается в перезаписи выражения в терминах другого допустимого выражения XPath. Например, выражение

a/b/c | a/b/d

может быть переписано в таком виде:

a/b / (c|d)

(что является допустимым выражением в XPath 2.0, но не действительно в XPath 1.0).

Другим примером такой перезаписи будет выражение count($x) > 10, которое может быть переписано в виде exists($x [11]). Последнее выражение, вероятно, будет более эффективным, потому что (благодаря конвейерной обработке и отложенному вычислению) объекты, следующие за одиннадцатым по счету, вероятно, никогда не будут прочитаны. Перезаписи, которые удаляют подвыражения из цикла, можно также считать включенными в эту категорию. Например, выражение

items[value > $customer/credit-limit]

может быть переписано следующим образом:

let $x := $customer/credit-limit

return items[value > $x]

И опять этот метод полагается на анализ зависимости, то есть на знании того, что подвыражение $customer/credit-limit не зависит от объекта контекста или положения контекста, которые сделали бы значение различным для различных объектов.

Второй вариант этого метода заключается в перезаписи выражения в терминах внутренних операторов, которые недоступны в формальном языке. Одним из самых выдающихся примеров является выражение

$x[position() != last()]

которое может быть переписано как

$x[hasNext()]

где hasNext() – это внутренняя функция, которая проверяет, имеются ли еще какие-либо объекты в конвейере. Красота этой перезаписи заключается в том, что в ней избегается обрыв конвейера. Когда в исходном выражении прочитывается первый объект, его положение (1) нужно сравнить с числом узлов в конвейере (возможно, тысячами), а это означает, что узлы должны быть подсчитаны. В переписанном выражении каждый узел просто проверяет, является ли он последним, для чего требуется упреждающее чтение только одного объекта.

В традиции реляционных баз данных наиболее значительными (т. е. наиболее выгодными) являются перезаписи, связанные с оптимизацией слияний, встречающихся в выражении SELECT языка SQL. Поэтому весьма вероятно, что производители программного обеспечения на основе XQuery затратят достаточно усилий на оптимизацию слияний, имеющихся в выражениях FLWOR, которые в XQuery являются эквивалентом выражения SELECT из SQL. Существует определенный риск, что эти усилия могут оказаться напрасными из-за способа, которым пользователи решат написать свои запросы. В реляционной модели все взаимоотношения моделируются с использованием первичных ключей и внешних ключей, и объединение по эквивалентности первичных и внешних ключей встречаются в запросах повсеместно. В отличие от этого в иерархическом мире XML многие взаимоотношения моделируются посредством отношений содержания: для представления порядка обработки порядковые строки не содержат порядковые номера, вместо этого они представлены в виде подэлементов. Поэтому слияние, которое является неизбежным в запросе SQL, заменяется в формулировке XQuery выражением пути.

Вместо

SELECT customer.name, order.date, order-line.value   
FROM customer, order, order-line   
WHERE customer.id = order.customer-id AND order-line.order-no =   
order.order-no

мы, вероятно, увидим

for $c in /customers/customer,   
$o in $c/order, $ol in $o/order-line   
return $c/name, $ol/date, $o/value

В настоящий момент некоторые системы запросов, вероятно, неявно используют табличное хранение и выполнят этот запрос в виде слияния на основе значений даже при том, что не задается явных ключей слияния в запросе пользователя. Однако процессоры, использующие представление данных на основе структуры дерева, обработают запрос тем же самым способом, как и процессор XSLT – посредством обхода узлов дерева в глубину.

Конечно, оптимизация слияний играет важную роль в XQuery, особенно в тех случаях, когда на различных уровнях иерархии доступно множество индексов, что предоставляет широкий выбор путей доступа для выполнения одного и того же запроса. Слияния, относящиеся к данным, находящимся в разных документах, также могут быть очень важны. Но я подозреваю, что операция слияния будет намного менее важной по сравнению с ее значением в реляционных базах данных, и что оптимизация иерархических путей доступа – фактически выражений XPath – будет по крайней мере настолько же, а возможно, и более значимой.

Насколько мне известно, в процессорах XSLT до сих пор уделялось мало внимания оптимизации слияний. Я считаю, что для этого имеется несколько причин. Дизайн языка (с его вложенными инструкциями xslt:for-each и xslt:apply-templates) не делает запросы, содержащие слияния, легкими для обнаружения. В то же время тот факт, что дерево создается заново для каждого преобразования, означает отсутствие широкого выбора путей доступа. Но главная причина, как я подозреваю, заключается в том, что таблицы стилей, которые были изучены разработчиками XSLT для того, чтобы найти возможности для их оптимизации, выполняют очень небольшое число слияний. Их пути доступа в основ-ном следуют иерархическим отношениям, свойственным модели XML. Разработчики XQuery вступают в игру вооруженные огромным множеством инструментов для оптимизации, которые оказались полезными в мире реляционных данных. Только время покажет, насколько эффективными будут эти инструменты в мире XML.

Использование информации о типе

Как мы видели, XSLT 1.0 и XPath 1.0 работают без использования схемы для исходного или конечного документов. Авторам таблиц стилей нужно знать, какие ограничения накладываются на структуру этих документов, но им не нужно указывать системе, каковы эти ограничения.

Это положение изменилось в XQuery (и, конечно, в XSLT 2.0 и XPath 2.0), благодаря способности импортировать схемы и делать утверждения о типах выражений и аргументов функций: XQuery является намного более строго типизированным языком, чем его предшественники. Или, во всяком случае, он имеет в этом отношении хороший потенциал, особенно если пользователи решат использовать в своих интересах эти возможности.

Один из аргументов в пользу более строгого контроля типов заключается в том, что знание типов сделает возможной намного более мощную оптимизацию. Имеется хорошее теоретическое обоснование этого представления, хотя достигнуть результатов на практике нелегко. Вот простой пример. Весьма обычной конструкцией в таблицах стилей является использование выражений типа //item, чьим значением будет множество всех элементов item в документе. Вычисление этого выражения пути в большинстве процессоров XSLT является очень ресурсоемким процессом, потому что он включает в себя полный просмотр исходного дерева. Со знанием структуры схемы становится возможным ограничение поиска до тех ветвей дерева, в которых элементы item действительно могут появиться.

Стоит ли это делать в XSLT – спорный вопрос. Имеются некоторые практические проблемы, потому что в настоящее время невозможно получить статическую информацию о том, сооответствует ли определенной схеме документ, в котором будет производиться поиск с использованием выражения //item. Также является спорным и тот факт, что другие подходы к оптимизации этого выражения могут быть более действенными. Например, информация о том, какие типы элементов находятся в какой ветви дерева, доступна не только из схемы, – она может также быть собрана (с большей точностью) во время анализа исходного документа. В XQuery этот вид оптимизации на основе схемы намного более важен по той причине, что одна из самых главных задач оптимизатора запроса состоит в том, чтобы идентифицировать пути доступа, которые используют преимущества предварительно созданных индексов.

Процессор XSLT или XQuery, выполняющий статический анализ типов выражений, может принять множество решений на этапе компиляции, которые иначе были бы приняты во времени выполнения. Saxon делает это даже с помощью слабо типизированного языка XPath 1.0. Например, во время компиляции обычно можно выяснить, является ли предикат в выражении фильтра, таком как $x[FIL-TER], логическим фильтром или числовым индексом. Конечно, наличие самой схемы во время компиляции увеличит возможности принятия ранних решений о пути доступа. Однако многие из этих видов оптимизации приносят только не-большую пользу. Например, по сравнению с полным временем выполнения запроса, принятое при компиляции решение выполнять арифметические действия над целыми числами, а не над числами с плавающей точкой, не является большой победой.

Поэтому, на мой взгляд, все еще рано судить о важности оптимизации на основе схемы. Мы должны подождать реального опыта использования программ, прежде чем мы узнаем настоящие ответы. Экстраполирование опыта, полученного при работе с реляционными данными, не обязательно поможет в этой задаче. Мое собственное предположение заключается в том, что строгий контроль типов окажется менее полезным на практике, чем надеются некоторые из его защитников, в основном по той причине, что многие пользователи выберут легкий путь и не станут объявлять ожидаемые типы переменных, параметров функций и созданных узлов, если они могут обойтись без этого.

Заключение

В этой главе мы рассмотрели взаимоотношение между XQuery и XSLT (и их общим подмножеством XPath) со многих точек зрения. Мы начали с анализа сходства и различия этих двух языков и причин возникновения этих различий. Мы увидели, что, хотя функциональные возможности языков во многом перекрываются, существуют значительные различия в требованиях, в соответствии с которыми были разработаны эти два языка. Мы можем отнести многие из различий между этими проектами к тому факту, что языки были предназначены для использования в различных средах. Однако другие различия отражают разные культуры мира документов с его историей SGML и мира баз данных с его традициями, основанными на SQL. Возможно, наиболее значительное различие заключается в том, что XSLT имеет больше возможностей при обработке свободноструктурированных данных, в то время как XQuery наиболее силен в обработке данных, которые являются очень регулярными. Это, в свою очередь, говорит о различных акцентах использования схем и систем типов в этих двух сообществах – хотя, как мы видели, здесь дизайн языков сближается. При этом XSLT содержит хорошие инструменты для работы со строго типизированными данными, а XQuery – с полуструктурированной информацией.

Затем мы рассмотрели наиболее эффективные механизмы оптимизации в XSLT с целью понять степень применимости к XQuery тех уроков, которые были получены за три года опыта работы с XSLT. Это особенно справедливо для специфических методов, таких как конвейерная обработка, отложенное вычисление, перезапись выражений и избежание сортировки в выражениях пути. Однако среды, использующие XQuery, предоставят своим оптимизаторам дополнительные трудности и дополнительные возможности, потому что в общем случае они будут работать с базами данных, имеющими набор индексированных путей доступа. При работе с базой данных ключевая задача оптимизатора заключается в идентификации индексов, которые позволяют вычислить запрос без выполнения последовательного поиска. Разработчики XQuery для решения этой задачи используют большой опыт из мира реляционных данных. Но мы увидели в этом некоторый риск, заключающийся в том, что акцент, сделанный на оптимизацию слияний в XML, по сравнению с реляционными данными может не оправдать себя. Это очень зависит от приложений, для которых будут использоваться базы данных XML, тип которых в настоящее время трудно предсказать.

Тот факт, что XQuery и XSLT имеют общую модель данных и общий подъязык в виде XPath, является главным достижением W3C, учитывая разнообразный опыт предыдущей деятельности этих двух рабочих групп. Я полагаю, что это также очень пригодится пользователям – так как два рассмотренных нами языка имеют очень много общего, уменьшится время обучения и появится возможность использовать их в дополнительных ролях; например, будет возможно очень легко представить результат XQuery с использованием таблицы стилей XSLT. И я подозреваю – только время покажет, прав ли я, – что многим разработчикам удастся использовать одни и те же средства выполнения для реализации и того и другого языка, что, очевидно, пойдет на пользу обоим сообществам пользователей[6](http://citforum.ru/internet/xml/xquery_xpath_xslt/" \l "b6).

Примечание

[1](http://citforum.ru/internet/xml/xquery_xpath_xslt/" \l "t1) DOM: Модель объекта документа, см. http://www.w3.org/TR/DOM

[2](http://citforum.ru/internet/xml/xquery_xpath_xslt/" \l "t2) JDOM: Вариант DOM, разработанный специально для Java. См. http://www.jdom.org/

[3](http://citforum.ru/internet/xml/xquery_xpath_xslt/" \l "t3) На практике это предположение не всегда верно, но XSLT по существу предназначен для тех случаев, где оно верно. Не существует широко используемой реализации XSLT, которая избегает необходимости создания исходного дерева в памяти.

[4](http://citforum.ru/internet/xml/xquery_xpath_xslt/" \l "t4) Процедура переноса неумещающегося слова на следующую строку при редактировании. – Примеч. научн. ред.

[5](http://citforum.ru/internet/xml/xquery_xpath_xslt/" \l "t5) Число правил в грамматике XPath выросло с 39 до 71, но только 16 из новых правил вводят основные синтаксические конструкции. Остальные просто расширяют диапазон доступных двоичных операторов. Однако число функций выросло с 28 до более чем 100.

[6](http://citforum.ru/internet/xml/xquery_xpath_xslt/" \l "t6) С того момента как я написал эту главу, я выполнил свое собственное предсказание. В последней версии Saxon это реализовано.