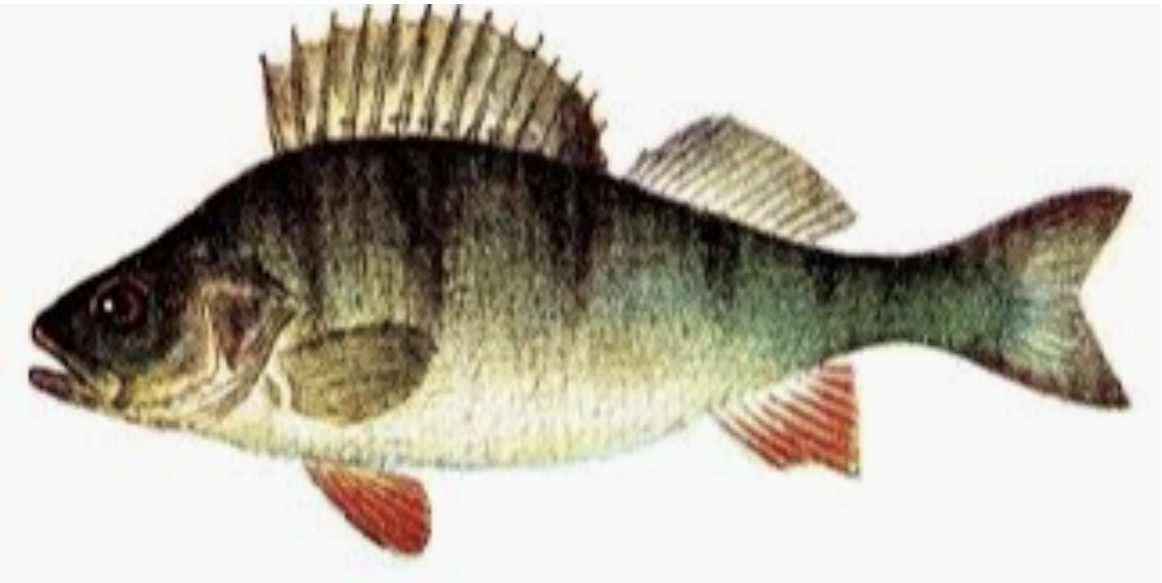
Запросы и обновления с помощью SPARQL 1.1

2-е ИЗДАНИЕ

Изучаем

SPARQL



Автор: ***Боб Дюшарм***

Перевод и редактирование: ***Феликс Филиппов***

O'REILLY

Пекин • Кембридж • Фарнхем • Кёльн • Севастополь • Токио

Оглавление

Предисловие

Все больше и больше людей использует язык запросов SPARQL для извлечения информации из растущей коллекции государственных и частных данных. Будь это данные являющиеся частью семантического веб-проекта или интеграция двух баз данных на разных платформах, SPARQL делает доступ к ним значительно легче. По словам директора W3C и изобретателя веб Тима Бернерс-Ли: "Попытка использовать семантический веб без SPARQL, аналогична попытке использовать реляционную базу данных (БД) без SQL".

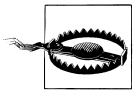
SPARQL был разработан не для запроса реляционных данных, а для работы с данными соответствующими модели RDF. Форматы данных на основе RDF еще не достигли статуса подобного базам данных XML и реляционным БД, но все большее число ИТ-специалистов обнаруживают, что инструменты, которые используют эту модель позволяет представить различные наборы данных (в том числе, как мы увидим, и реляционные БД) с общим, стандартизированным интерфейсом. Доступ к этим данным не требуют изучения новых API, потому что программные продукты с открытым исходным кодом и для коммерческого применения (в том числе Oracle и DB2 компании IBM) доступны с поддержкой SPARQL, что позволяет воспользоваться преимуществами этих источников данных. Благодаря этому SPARQL дал людям доступ к широкому кругу государственных данных и обеспечил более легкую интеграцию данных для многих предприятий.

Основная цель книги быстро научить комфортному использованию SPARQL для извлечения и обновления данных. После того, как вы сделаете это, вы сможете воспользоваться широким выбором инструментов и библиотек приложений, которые используют SPARQL для извлечения, обновления и интеграции огромного количества RDF-доступных данных.

Следующие значки предупредят вас о деталях, которым стоит уделить немного больше внимания:

Нечто важное, что может быть легко пропущено.

Совет, который может сделать ваши запросы более эффективными.



Предупреждение об общей проблеме или легкой ловушке.

Эта книга включает в себя примеры кода, которые вы можете использовать в ваших программах и документации. Вам не нужно обращаться к нам за разрешением, если вы не воспроизводите значительную часть кода. Например, написание программы, которая использует несколько фрагментов кода из этой книги не требует разрешения. Включение значительного количества примеров кода из этой книги в документацию вашего продукта требует разрешения.

Мы ценим, но не требуем ссылки, она, как правило, включает в себя название, автора, издателя и ISBN. Например: "Изучение SPARQL, 2-е издание, Боб Дюшарм (O'Reilly). Copyright 2013 O'Reilly Media, 978-1-449-37143-2 ".

Если вы считаете, что ваше использование примеров кода выходит за справедливое использование или разрешение, данное выше, не стесняйтесь обращаться к нам в permissions@oreilly.com.

Глава 1. Некоторые данные и некоторые запросы

Глава 2 предоставит основную информацию о RDF, семантической сети и о том, где применяется SPARQL. Но, прежде чем перейти к этому, давайте начнем с небольшого практического опыта написания и выполнения запросов SPARQL, чтобы основная часть не выглядела слишком теоретической.

Сначала разберем, что такое **SPARQL**? Это название - рекурсивный акроним для **S**PARQL **P**rotocol **A**nd **R**DF **Q**uery **L**anguage, который описывается набором спецификаций из W3C.

Как можно сказать по части названия "RQL", SPARQL предназначен для запроса RDF, но вы не ограничены запросами данных, хранящихся в одном из форматов RDF. Коммерческие и открытые для доступа источники реляционных данных, представленных в формате XML, JSON, электронных таблиц и других форматах, таких как RDF, так же подходят для формирования SPARQL запросов к ним. Можно также использовать комбинации этих источников, что является одним из самых мощных аспектов комбинации SPARQL / RDF.

"Protocol" часть названия SPARQL, которая ссылается на правила о том, как программа-клиент и сервер обработки SPARQL обмениваются запросами и результатами. Эти правила указаны в отдельном документе по спецификации запроса и в большей степени предназначены для разработчиков процессоров SPARQL.

1.1. Данные для запроса

Глава 2 опишет подробно RDF и все действия, которые выполняются с RDF, но главное заключается в следующем: RDF не формат данных, а модель данных с выбором синтаксиса для хранения файлов данных. В этой модели данных, вы выражаете факты с помощью утверждений, состоящих из трех частей, известных как триплеты (или тройки). Каждый триплет, как маленькое предложение, которое описывает факт. Эти три части триплета называются субъект (подлежащее), предикат (сказуемое) и объект (определение). Можно представлять триплет следующим образом: идентификатор некоторого ресурса, название свойства и значение свойства:

Субъект (идентификатор ресурса)	Предикат (название свойства)	Объект (значение свойства)
richard	homeTel	(229) 276-5135
cindy	email	cindym@gmail.com

Файл ex002.ttl ниже, включает некоторые триплеты записанные с использованием формата Turtle RDF (формат Turtle и другие форматы будут описаны в главе 2). Этот файл хранит данные адресной книги, используя триплеты. RDF не имеет никаких проблем с назначением нескольких значений для данного свойства в данном ресурсе, как вы можете видеть в этом файле, который показывает, что Craig имеет два электронных адреса:

```
# filename: ex002.ttl
@prefix ab: <http://learningsparql.com/ns/addressbook#> .

ab:richard      ab:homeTel      "(229) 276-5135" .
ab:richard      ab:email        "richard49@hotmail.com" .

ab:cindy        ab:homeTel      "(245) 646-5488" .
ab:cindy        ab:email        "cindym@gmail.com" .
```

```
ab:craig        ab:homeTel      "(194) 966-1505" .
ab:craig        ab:email        "craigellis@yahoo.com" .
ab:craig        ab:email        "c.ellis@usairwaysgroup.com" .
```

Как обычные предложения, Turtle (и SPARQL) триплеты обычно заканчиваются точкой. Пробелы необязательны и добавлены только для того, чтобы сделать данные проще для чтения.

Комментарии в данных Turtle и запросах SPARQL начинаются с символа решетка (#). Каждый файл запросов и образец данных в этой книге начинается с комментария, содержащего имя файла.

Первой непустой строкой из приведенных выше данных, после комментария об имени файла, также является триплет. Он говорит нам о том, что префикс "ab" будет использоваться вместо URI http://learningsparql.com/ns/addressbook. Субъект и предикат в RDF триплетах должны принадлежать к определенному пространству имен для того, чтобы избежать путаницы между подобными именами, если мы будем объединять эти данные с другими данными, поэтому мы представляем их с помощью URI. Префикс избавляет от написания длинных полных URI много раз.

URI – это унифицированный идентификатор ресурса. URL (унифицированный локатор ресурса), также известный как веб-адрес, один из видов URI. Локатор помогает найти что-то, типа веб-страницы, а идентификатор идентифицировать что-либо. Так, например, уникальным идентификатором для Ричарда в адресной книге представленной выше, является http://learningsparql.com/ns/addressbook#richard. URI может выглядеть как URL, и по этому адресу на самом деле может быть веб-страница, но может и не быть; его основное назначение заключается в обеспечении уникального имени чего-либо.

1.2. Запрос данных

SPARQL запрос, как правило, говорит: "Я хочу кусочки информации из подмножества данных, которые удовлетворяют этим условиям." Вы описываете условия в виде триплетов, которые похожи на RDF тройки, но могут включать в себя дополнительно переменные. Наши первые запросы будут иметь простые образцы триплетов, и потом мы будем строить более сложные запросы.

Следующий файл ex003.rq включает наш первый SPARQL запрос, который мы будем адресовать к данными из файла ex002.ttl.

Запрос состоит из одного триплета, выделенного жирным шрифтом, указывающего подмножество данных, которые мы хотим извлечь. Этот триплет состоит из субъекта ab:craig, предиката ab:email и переменной ?craigEmail на позиции объекта.

Переменная похожа на джокера. В дополнение к тому, что она указывает поисковой машине что ее устроит триплет удовлетворяющей шаблону с любым значением в этой позиции, все значения, которые будут запомнены в переменной ?craigEmail, мы можем использовать в другом месте в запросе:

```
# filename: ex003.rq
PREFIX ab: http://learningsparql.com/ns/addressbook#
SELECT ?craigEmail
WHERE
{ ab:craig ab:email ?craigEmail . }
```

Данный запрос запрашивает любой ab:email, связанный с ресурсом ab:craig. На простом языке, это означает поиск любых электронных адресов Крейга.

В наборе триплетов данных или триплетов запросов, точка после последнего не является обязательной, таким образом в шаблоне триплета последнего примера она не нужна. Тем не менее ставить ее – хорошая привычка, хотя бы потому, что потом проще добавлять новые шаблоны. В примерах этой книги, иногда будут записываться одиночные триплеты без точки на конце.

Как показано на рисунке 1-1, в запросе SPARQL, ключевое слово WHERE указывает «извлечь эти данные из набора данных», а SELECT «выбрать для извлекаемых данных указанные имена».

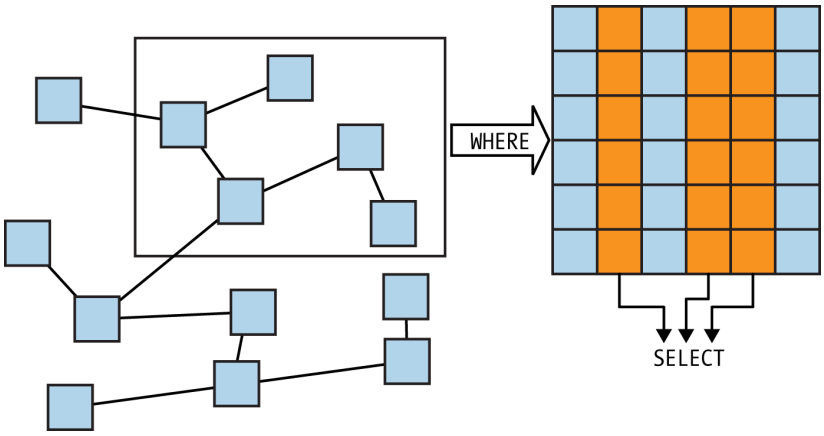


Рисунок 1-1. WHERE специфицирует извлекаемые данные; SELECT выбирает данные для отображения

Какую информацию запрос приведенный выше извлечет из троек данных? Все, что соответствует (удовлетворяет) его единственному триплету. Все, что будет получено в качестве результата запроса присваивается переменной ?CraigEmail.

Как и в любым языке программирования или языке запросов, имя переменной должно давать подсказку о назначении переменной. Вместо имени ?CraigEmail, можно было выбрать имя ?Zxzwzyx, что сделало бы запрос более трудным для его понимания человеком.

Разнообразие процессоров SPARQL доступно для выполнения запросов в отношении локальных и удаленных данных. (Вы услышите термины SPARQL процессор и SPARQL движок, но они имеют в виду одно и то же - программу, которая может применить SPARQL запрос к набору данных и выдать результат). Для запросов к файлу данных на вашем жестком диске, можно использовать свободно распространяемую Java-программу ARQ. ARQ является частью Apache Jena framework, поэтому ее можно загрузить с домашней страницы по адресу http://jena.apache.org/documentation/query и скачать бинарный файл, имя которого Apache-jena- * .zip. Разархивирование создаст подкаталог с именем, похожим на имя файла ZIP; это ваш домашний каталог Jena. Пользователи Windows, найдут там arq.bat и sparql.bat скрипты в bat подкаталоге, а пользователи с системами Linux - arq и sparql shell сценарии в подкаталоге bin. (Первый из каждой пары позволяет использовать расширения ARQ).

Либо на Windows, либо на Linux, вы можете запустить ex003.rq запрос к данным ex002.ttl с помощью следующей команды:

```
arq --data ex002.ttl --query ex003.rq
```

Формат вывода по умолчанию - ARQ показывает имя каждой выбранной переменной в верхней строке и линиями вокруг значений каждой переменной, с использованием дефиса, знаков равно и вертикальная линия:

```
-------------------------------------------
| craigEmail                              |
===========================================
| "c.ellis@usairwaysgroup.com" |
| "craigellis@yahoo.com"          |
-------------------------------------------
```

Следующий пересмотр ex003.rq запроса использует полные URI, для записи субъекта и предиката в триплете запроса, вместо того чтобы использовать префикс. По сути, это тот же запрос, и получает тот же ответ от ARQ:

```
# filename: ex006.rq
SELECT ?craigEmail
WHERE
{
  <http://learningsparql.com/ns/addressbook#craig>
  <http://learningsparql.com/ns/addressbook#email>
  ?craigEmail .
}
```

Различия между этим запросом и первым демонстрируют две вещи:

• Вы не должны использовать префиксы в запросе, но они могут сделать запрос более компактным и легким для чтения, чем тот, который использует полные URI. Когда вы используете полный URI, заключите его в скобки, чтобы показать, процессору что это URI.

• Пробелы не влияют на синтаксис SPARQL. Новый запрос использует возврат каретки, для разделения трех частей триплета и все равно работает отлично.

Команда ARQ приведенная выше указывает набор данных для запроса в командной строке. SPARQL запрос с ключевым словом FROM позволяет вам указать набор данных для запроса как часть самого запроса. Если вы опустите параметр --data ex002.ttl показанный в командной строки ARQ, и используете следующий запрос, вы получите тот же результат:

```
# filename: ex007.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?craigEmail FROM <ex002.ttl>
WHERE
{ ab:craig ab:email ?craigEmail . }
```

Угловые скобки вокруг "ex002.ttl" говорят процессору SPARQL использовать его в качестве URI. Но, поскольку это просто имя файла, а не полный URI, ARQ предполагает, что это файл находящийся в той же папке, что и сам запрос.

В запросах, которые мы видели до сих пор переменная была в позиции объекта триплета (третья позиция), но вы можете поместить ее в любой из трех позиций. Например, скажем, кто-то позвонил мне по телефону (229) 276-5135, и я ничего не ответил. Я хочу знать, кто пытался позвонить мне, для этого я создаю следующий запрос для набора данных адресной книги, поставив переменную в субъектной позиции, вместо позиции объекта:

```
# filename: ex008.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?person
WHERE
{ ?person ab:homeTel "(229) 276-5135" . }
```

При запуске этого запроса в ARQ к данным ex002.ttl адресной книги, мы получим следующий результат:

```
----------------------
| person           |
==============
| ab:richard       |
----------------------
```

Шаблоны триплетов в запросах часто имеют более чем одну переменную. Например, можно бы перечислить все, что в адресной книге о Синди с помощью следующего запроса, который имеет переменную ?PropertyName на позиции предиката и переменную ?PropertyValue на позиции объекта:

```
# filename: ex010.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?propertyName ?propertyValue
 WHERE
{ ab:cindy ?propertyName ?propertyValue . }
```

Пункт SELECT запроса спрашивает значения ?propertyName и ?propertyValue переменных, и ARQ показывает их в виде таблицы со столбцами для каждого:

```
-----------------------------------------------------
| propertyName |   propertyValue            |
==================================
| ab:email          |   "cindym@gmail.com"  |
| ab:homeTel     |   "(245) 646-5488"        |
-----------------------------------------------------
```

1.3. Более реалистичные данные и запросы из нескольких триплетов

В большинстве данных RDF, субъекты троек не должны выступать в роли названий понятных для человеческого глаза, как например в ex002.ttl ab:richard и ab:cindy имена ресурсов. Они, скорее всего, должны играть роль идентификаторов, аналогично значениям уникального ID поля таблицы реляционной базы данных. Вместо того чтобы хранить чье-то имя, как часть URI субъекта, как наш первый набор данных, более типично для RDF иметь для субъектов значения, которые составляют не удобочитаемый смысл, а выполняют роль уникальных идентификаторов. В этом случае firstName и lastName будут храниться с использованием отдельных троек, как значения homeTel и email.

Другая нереалистичная деталь ex002.ttl состоит в том, что идентификаторы ресурсов, такие как ab:richard и имена свойств, такие как ab:homeTel, берутся из одного и того же пространства имен http://learningsparql.com/ns/addressbook#, которое представляет префикс ab. Словарь имен свойств, как правило, имеет свое собственное пространство имен, чтобы упростить его использование его с другими наборами данных.

Если пересмотреть данный пример, чтобы использовать реалистичные идентификаторы ресурсов, чтобы сохранить имена и фамилии в качестве значений свойств и поставить значения данных в собственном пространстве имен отдельно от http://learningsparql.com/ns/data#, мы получим следующий набор данных:

```
# filename: ex012.ttl
@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d: <http://learningsparql.com/ns/data#> .

d:i0432 ab:firstName    "Richard" .
d:i0432 ab:lastName     "Mutt" .
d:i0432 ab:homeTel      "(229) 276-5135" .
d:i0432 ab:email        "richard49@hotmail.com" .

d:i9771 ab:firstName    "Cindy" .
d:i9771 ab:lastName     "Marshall" .
d:i9771 ab:homeTel      "(245) 646-5488" .
d:i9771 ab:email        "cindym@gmail.com" .

d:i8301 ab:firstName    "Craig" .
d:i8301 ab:lastName     "Ellis" .
d:i8301 ab:email        "craigellis@yahoo.com" .
d:i8301 ab:email        "c.ellis@usairwaysgroup.com" .
```

Запрос, для нахождения адреса электронной почты Крейга будет выглядеть так:

```
# filename: ex013.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?craigEmail
WHERE
{
 ?person ab:firstName "Craig" .
 ?person ab:email ?craigEmail .
}
```

Давайте предположим, что процессор SPARQL посмотрел все строки адресной книги и нашел совпадение в {ab:i8301 ab:firstName "Craig"}. Это будет связывать ab: i8301 с переменной ?person, так как обе находятся в субъектной позиции и остальные части триплетов совпадают.

Для запросов типа ex013.rq, которые имеют более одного триплета, когда процессор находит одно совпадение он переходит к другим триплетам, чтобы найти другие совпадения, с учетом предыдущего. Так в нашем случае выясняется, что две тройки в ex012.ttl имеющие в качестве субъекта d:i8301, а в качестве предиката ab:email возвратят два значения для переменной ?craigEmail:

```
-----------------------------------------------
| craigEmail                                  |
===============================================
| "c.ellis@usairwaysgroup.com"                |
| "craigellis@yahoo.com"                      |
-----------------------------------------------
```

Множество триплетов записанных между фигурными скобками в запросе SPARQL называется графовым шаблоном. Граф в данном случае является техническим термином для набора RDF троек. В то время как существуют утилиты набор RDF троек в «графовую картинку», это не относится к графике в визуальном смысле. В «графовой картинке» RDF, узлы представляют субъект или объект ресурсов, и предикаты соединения между этими узлами.

В ex013.rq запросе использована переменная ?person в двух разных триплетах, чтобы найти связанные тройки в данных. В более сложных запросах, это техника с использованием переменной для связывания различных троек данных становится все более распространенной. Когда вы перейдете на составление запросов к данным, которые поступают из нескольких источников, вы увидите, что эта способность, находить связи между тройками из различных источников является одним из лучших особенностей SPARQL.

Если бы ваша адресная книга была включала не одного абонента по имени Крейг, а вы определенно хотели бы найти адреса электронной почты Крейг Эллис, вы бы просто добавить еще один триплет к шаблону:

```
# filename: ex015.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?craigEmail
WHERE
{
 ?person ab:firstName "Craig" .
 ?person ab:lastName "Ellis" .
 ?person ab:email ?craigEmail .
}
```

Результат этого запроса аналогичен предыдущему.

Давайте предположим, что телефон показал, что кто-то звонит с номера телефона "(229) 276-5135" и используем запрос ex008.rq, который использовали прежде, но на этот раз к данным из ex012.ttl. Результат покажет субъект тройки у который предикат ab:homeTel и объект "(229) 276-5135":

```
-----------------------------------------------------------------------
|  person                                                             |
=======================================================================
|  <http://learningsparql.com/ns/data#i0432>                          |
-----------------------------------------------------------------------
```

Если я действительно хочу знать, кто позвонил мне, "http://learningsparql.com /ns/data#i0432" не очень подходящий ответ. То, что я хочу, это имя и фамилия звонившего с этого номера телефона, следующий запрос запрашивает именно это:

```
# filename: ex017.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?first ?last
WHERE
{
 ?person ab:homeTel "(229) 276-5135" .
 ?person ab:firstName ?first .
 ?person ab:lastName ?last .
}
```

ARQ вернет в этом случае то, что необходимо:

```
-----------------------------------
| first       | last          |
=======================
| "Richard"  | "Mutt"      |
-----------------------------------
```

Пересмотрим наш запрос, чтобы найти все сведения о Синди в данных ex012.ttl: мы запросим все предикаты и объекты (хранимые в переменных?propertyName и ?propertyValue), связанных с субъектом, который имеет ab:firstName “Cindy” и ab:lastName “Marshall”:

```
# filename: ex019.rq
PREFIX a: <http://learningsparql.com/ns/addressbook#>
SELECT ?propertyName ?propertyValue
WHERE
{
?person  a:firstName     "Cindy" .
?person  a:lastName      "Marshall" .
?person  ?propertyName ?propertyValue .
}
```

Обратите внимание, что значения ab:firstName и ab:lastName из файла ex012.ttl появляются в колонке PropertyValue. Другими словами, они получили связь с переменной PropertyValue, также как ab:email и ab:homeTel:

```
-------------------------------------------------------
| propertyName  | propertyValue              |
=====================================
| a:email           | cindym@gmail.com       |
| a:homeTel       | "(245) 646-5488"          |
| a:lastName      | "Marshall"                 |
| a:firstName     | "Cindy"                    |
-------------------------------------------------------
```

1.4. Поиск подстрок

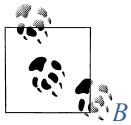
Что делать, если вы хотите, проверить часть данных, но даже не знаете, какой субъект или предикат может включать ее? Следующий запрос имеет только один триплет и все три части переменные, так что он будет соответствовать каждой тройки в наборе данных. Однако, в результате не будут присутствовать все тройки, потому что сюда добавлен FILTER, который инструктирует процессор запросов пройти только вдоль троек, которые отвечают определенному условию. В этом фильтре, условие задается с помощью регулярных выражений функцией regex(), которая проверяет соответствие строк, определенному в ней шаблону. (Мы узнаем больше о фильтрах в главе 3 и о регулярных выражениях в главе 5.) Этот вызов функции regex() проверяет, имеет ли объект подстроку "yahoo" в любом месте в нем:

```
# filename: ex021.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT *
WHERE
{
?s ?p ?o .
FILTER (regex(?o, "yahoo","i"))
}
```

Это общее соглашение SPARQL использовать имя переменной?s для субъекта, ?p для предиката, а ?o для объекта.

Процессор запросов найдет единственный триплет, имеющий "yahoo" в значении объекта:

```
-------------------------------------------------------------------------------------------------------
| s                                                | p          | o                                 |
=======================================================================================================
| <http://learningsparql.com/ns/data#i8301>  |  ab:email |  "craigellis@yahoo.com" |
-------------------------------------------------------------------------------------------------------
```



В этом запросе использована звездочка вместо перечень конкретных переменных в SELECT. Это просто сокращенный способ сказать "выбрать все переменные, использованы в этом запросе."

1.5. Что может пойти не так?

Давайте изменим запрос ex015.rq, запросив кроме адреса электронной почты Крейга Эллиса также его домашний телефон. (Если вы просмотрите данные ex012.ttl, вы увидите, что именно у Крейг его нет.)

```
# filename: ex023.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?craigEmail ?homeTel
WHERE
{
?person ab:firstName "Craig" .
?person ab:lastName "Ellis" .
```

```
?person ab:email ?craigEmail .
?person ab:homeTel ?homeTel .
}
```

Когда ARQ применит этот запрос к данным ex012.ttl, результат будет включать заголовки для переменных, но данные будут отсутствовать:

```
-----------------------------------
| craigEmail   | homeTel   |
===================================
-----------------------------------
```

Почему? Запрос спросил процессор SPARQL об адресе электронной почты и номере телефона Крэйга Эллис, т.е. поставил задачу найти ресурс, который удовлетворяет всем четырем условиям, перечисленным в шаблоне графа. Но такого ресурса не нашлось, поэтому процессор SPARQL не возвращает никаких данных.

В главе 3 мы узнаем, о дополнительном ключевом слове OPTIONAL, которое позволяет сделать запросы как "Покажите мне значение?CraigEmail и, если есть значение ?homeTel его также."

1.6. Запрос источника данных общего пользования

Запрос данных на вашем жестком диске полезен, но реальное удовольствие от SPARQL начинается, когда вы запрашиваете публичные источники данных. При этом Вы не нуждаетесь в специальном программном обеспечении, потому что эти сборники данных часто доступны публично через (endpoint) конечную точку SPARQL - веб-сервис, который принимает запросы SPARQL.

Самый популярный конечной точкой является DBpedia - это данные, которые вы часто видите на правой стороне страниц Википедии. Как и многие конечные точки SPARQL, DBpedia включает в себя веб-форму, где вы можете ввести запрос, а затем изучить его результаты. DBpedia использует программу под названием SNORQL для принятия запросов и возврата ответов на веб-странице. Если вы обратитесь к браузеру по адресу http://dbpedia.org/snorql/, вы увидите форму, где можете ввести запрос и выбрать желаемый формат представления результатов, как показано на рисунке 1-2. Для наших экспериментов мы будем использовать "Обзор", для задания формата результата.

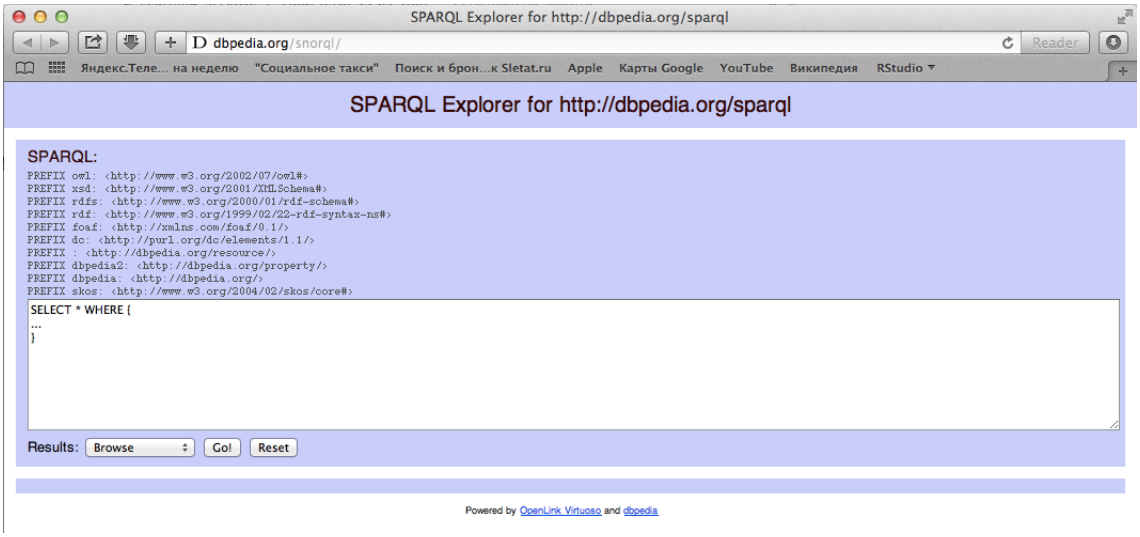


Рисунок 1-2. SNORQL веб-форма DBpedia

Попробуем из DBpedia получить список альбомов, производимых продюсером хип-хоп Тимбэлэндом и музыкантах, которые сделали эти альбомы. Если в Википедии есть страница на тему «*Some_Topic*» по адресу http://en.wikipedia.org/wiki/*Some_Topic*, то соответствующий URI (http://dbpedia.org/resource/Some_Topic) DBpedia как правило, представляет этот ресурс. Так, найдя страницу Википедии http://en.wikipedia.org/wiki/Timbaland , открываем в браузере http://dbpedia.org/resource/Timbaland где находятся интересующие нас данные. (Браузер фактически перенаправил наш последний вызов на http://dbpedia.org/page/Timbaland, потому что, когда браузер запрашивает информацию DBpedia перенаправляет его в HTML версию данных.) Этот URI представляет Тимбэлэнда аналогично тому, как http://learningsparql.com/ns/data#i8301 (или короче, префикс d:i8301) представляет Крейга Эллис в ex012.ttl.

В верхней части SNORQL веб-формы на рисунке 1-2 видно, что ресурсу http://dbpedia.org/resource/ назначен префикс ":", поэтому на продюсера Тимбэлэнда можно ссылаться в запросах как :Timbaland.

Префикс пространства имен может быть просто двоеточием. Это популярно для пространств имен, которые часто используются в конкретном документе, что делает документ легче для понимания человеком.

Producer и musicalArtist свойства, которые мы планируем использовать в своем запросе взяты из пространства имен http://dbpedia.org/ontology/, которое не объявлено на SNORQL веб-форме с которой мы будем вводить запрос, поэтому включим его декларацию в самом запросе:

```
# filename: ex025.rq
PREFIX d: <http://dbpedia.org/ontology/>
SELECT ?artist ?album
WHERE
{
 ?album d:producer :Timbaland .
?album d:musicalArtist ?artist .
}
```

Этот запрос вытаскивает информацию об альбомах, производимых Тимбэлэндом и исполнителей перечисленных в этих альбомах, записывая значения в переменные ?album и ?artist. После замены запроса представленного по умолчанию на веб-странице SNORQL, на наш запрос и нажав кнопку «Go!», мы получим результаты, как показано на рисунке 1-3.

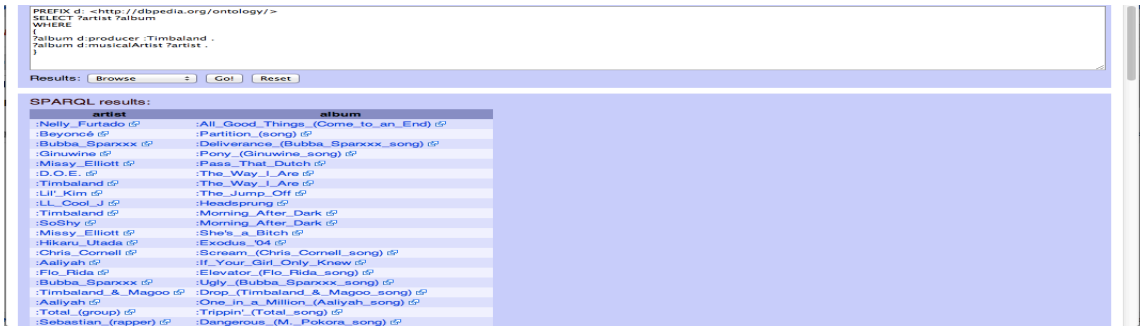


Рисунок 1-3. SNORQL веб-форма с результатами запроса

Прокрутка справа показывает, что этот список результатов только начало гораздо более длинного списка, хотя он может быть и не полным поскольку Википедия поддерживается добровольцами, и хотя есть некоторые усилия по обеспечению качества они являются незначительными сравнительно с масштабами обрабатываемых данных.

Также обратите внимание, что результаты не дают нам настоящие имена альбомов или исполнителей, а имена смешанные с пунктуацией и различными кодами. Помните, как :Timbaland в нашем запросе был аббревиатурой полного URI, представляющего производителя. Такие имена как :Cry_Me_a_River_%28Justin_Timberlake_song%29 и :Bj%C3%B6rk в результатах запроса являются также аббревеатурой URI. Эти исполнители и песни имеют свои собственные страницы Википедии и связанные с ними данные, которые включают в себя более читаемые версии имен, которые мы можем попросить в запросе. Мы узнаем о rdfs:label свойстве, значение которого часто хранит эти более читаемые данные, в главах 2 и 3.

1.7. Резюме

В этой главе мы узнали:

• Что такое SPARQL
• Основы RDF
• Значение и роль URI
• Части простого запроса SPARQL
• Как выполнить запрос SPARQL с ARQ
• Как переменная в нескольких тройках может объединять данные в разных триплетах
• Что может привести к запросу, который ничего не возвращает
• Что такое конечные точки SPARQL и как запросить одну из самых популярных – Dbpedia

В последующих главах описывается, как создать более сложные запросы, как изменить данные, как создавать приложения вокруг ваших запросов, о потенциальной роли логического вывода и корнях этой технологии в семантическом веб мире. Но если вы можете выполнить запросы как показано в этой главе , вы уже готовы заставить SPARQL работать для вас.

Глава 2. Семантический веб, RDF и связанные данные

Язык запросов SPARQL используется для данных, которые соответствуют определенной модели, но семантический веб не о языке запросов или о модели - он о данных. Ошеломляющее количество данных, которое становится доступным в семантической сети благодаря новым видам приложений реализованным на зрелом стандарте семантического веба. SPARQL является лучшим способом, чтобы получить эти данные и поместить их для работы в ваших приложениях.

Гибкость модели данных RDF означает, что она используется все больше и больше с проектами, которые не имеют ничего общего с семантическим вебом, кроме использования технологий на базе этих стандартов, поэтому вы часто будете видеть ссылки на "семантические веб-технологии".

2.1. Что же такое "Семантический веб"?

Поскольку ажиотаж вокруг семантической сети растет, некоторые производители используют эту фразу, чтобы продавать товары сильно связанные с идеями, лежащими в основе семантической сети, а другие используют его, чтобы продавать товары с более слабыми связями. Это может ввести в заблуждение людей, которые пытаются понять суть семантического веба.

Неплохо определить семантический веб как *набор стандартов и лучших практик для обмена данными и семантикой этих данных через веб для использования в приложениях*. Давайте разберем это определение по отдельным фразам и тогда оно станет более понятным.

Набор стандартов

Перед тем, как Тим Бернерс-Ли изобрел Всемирную паутину, были доступны более мощные гипертекстовые систем, но он построил ее вокруг простых спецификаций, которые он опубликовал в общедоступных стандартах. Это сделало возможным для людей реализовывать свои системы самостоятельно (то есть, писать свои собственные веб-серверы, веб-браузеры, и особенно веб-страницы) и его система росла, чтобы стать крупнейшей гипертекстовой системой когда-либо. Бернерс-Ли основал консорциум W3C для контроля этих стандартов, и семантический веб также построен на стандартах W3C: модель RDF данных, язык запросов SPARQL, и RDF Schema и OWL стандарты для хранения словарей и онтологий. Продукт или проект может иметь дело с семантикой, но если он не использует эти стандарты, он не может подключиться и быть частью семантической паутины. Больше, чем 1985 гипертекстовых систем может ссылаться на страницу на WWW, без использования HTML или HTTP стандартов. (Есть те, кто не согласен на этой последней точки).

Лучших практик для обмена данными ... через веб для использования в приложениях

Оригинальный веб Бернерса-Ли был разработан, чтобы доставлять человеко-читаемые документы. Если вы хотите лететь из одного аэропорта в другой в следующее воскресенье во второй половине дня, вы можете зайти на веб-сайт авиакомпании, заполнить форму запроса, а затем прочитать результаты запроса с экрана. Есть программы

для сравнения сайтов авиакомпаний, которые получают веб-страницы из нескольких сайтов авиакомпаний и извлекают необходимую информацию, в процессе, известном как "screen scraping (выскабливание экрана)", прежде чем использовать данные на своих веб-страницах. Прежде чем писать такую программу, разработчик должен анализировать HTML структуру сайта каждой авиакомпании чтобы определить, где осуществлять «выскабливание экрана» для получения именно тех данных, которые необходимы. Если авиакомпания переделывает свой веб-сайт, разработчик должен обновить свое «выскабливание экрана» для учета этих различий после переделки.

Бернерс-Ли придумал идею связанных данных в виде набора лучших практик для обмена данными через веб-инфраструктуры, так что приложения могут более легко извлекать данные из общедоступных сайтов без необходимости «выскабливания экрана», например, чтобы позволить получить информацию о рейсах нескольких веб-сайтов авиакомпаний в общей, машиночитаемой форме. Эти лучшие практики рекомендуют использование URI, чтобы назвать вещи и использование стандартов, таких как RDF и SPARQL. Они обеспечивают превосходное руководство по созданию инфраструктуры для семантической сети.

и семантикой этих данных

"Семантика" часто определяется как "смысла слов". Принципы связанных данных и соответствующие стандарты облегчают совместное использование данных, а использование URI, может обеспечить немного семантики, предоставляя контекст понятий. Например, даже если я не знаю, к чему относится понятие "sh98003588#concept ", я могу видеть из URI http://id.loc.gov/authorities/sh98003588#concept, что оно происходит от Библиотеки Конгресса США. Хранение полного смысла слов, так что компьютеры могли "понять" эти значения означает требовать слишком многого от современных компьютеров, но W3C язык Web-онтологий (также известный как OWL), уже позволяет сохранять особо ценные значения, так что мы можем получить больше из наших данных. Например, когда мы знаем, что термин "супруг" является симметричным (то есть, что если A это супруга B, то B является супругом A) или, что "продать" является противоположностью "купить", мы знаем больше о ресурсах, которые имеют эти свойства и отношения между этими ресурсами.

Давайте посмотрим на эти компоненты семантической сети более подробно.

2.2. URL, URI, IRI и пространства имен

Когда Бернерс-Ли изобрел веб, наряду с написанием первого веб-сервера и браузера, он разработал спецификации для трех вещей, так что все серверы и браузеры могут работать вместе:

• Способ представления структуры документа, так что браузер будет знать, какие части документа являются пунктами списка, какие - заголовками, какие - ссылками и так далее. Эта спецификация языка гипертекстовой разметки, или HTML.

• Способ для программ, таких как веб-браузеры и серверы, общаться друг с другом. Протокол передачи гипертекста, или HTTP, состоит из нескольких коротких команд и трехзначных кодов, которые, по существу позволяют клиентской программе, например, веб-браузеру говорить такие вещи, как "Эй www.learningsparql.com сервер, отправь мне файл index.html из каталога ресурсов!" Они также позволяют серверу сказать" OK, отправляю!" или "К сожалению, я не знаю, о том, что такое ресурс". Мы узнаем больше о HTTP в разделе "SPARQL и HTTP ".

• Компактный способ для клиента, чтобы указать, какой ресурс он хочет, например, имя файла, каталог, в котором он хранится, и сервер, который имеет эту файловую систему. Вы могли бы назвать это веб-адрес или указатель ресурса. Бернерс-Ли называет

комбинацию сервер-каталог-ресурс, которую клиент посылает с использованием конкретного протокола Интернет (например, http://www.learningsparql.com/resources/index.html) - Uniform Resource Locator (унифицированный локатор ресурса), или URL.

Когда у вас есть доменное имя типа learningsparql.com или redcross.org, вы контролируете имена структуры и файлов каталога, используемые для хранения там ресурсов. Эта возможность владельца домена управлять схемой именования (аналогично тому, как имена пакетов Java строятся на доменных именах) привело разработчиков использовать эти имена для ресурсов, которые не обязательно веб-адреса. Например, знакомый словарь Friend of a Friend (FOAF) использует http://xmlns.com/foaf/0.1/Person для представления концепции человека, но если запустить этот адрес на браузере, он просто перенаправит вызов на страницу спецификации http://xmlns.com/foaf/spec/.

Это смутило многих людей, потому что они полагали, что все, что начинается с "http://" - это адрес веб-страницы, которую можно просмотреть в браузере. Эта путаница привела двух инженеров из Массачусетского технологического института и фирмы Xerox написать спецификацию Universal Resource Names (универсальное имя ресурса) (URN). URN может принять форму urn:isbn:006251587X представлять ту или иную книгу или urn:schemas-microsoft-com:office:office для описания структуры файлов Microsoft Office.

Термин универсальный идентификатор ресурса был разработан, чтобы охватить как URL-адреса так и URN. Это означает, что URL является URI. Из-за того, что вряд ли кто-то использует URN, большинство URI являются URL-адресами поэтому люди иногда используют термины взаимозаменяемы. URI по-прежнему очень часто относятся к веб-адресам, как URL. Довольно типично для обозначения чего-то вроде http://xmlns.com/foaf/0.1/person использовать вместо URI, потому что это просто идентификатор, даже если это начинается с "http: //".

Как будто не было достаточно названия вариаций для URL-адресов, Internet Engineering Task Force выпустила спецификацию концепции многоязычных идентификаторов ресурсов. IRI являются URI, которые допускают более широкий диапазон символов, которые будут использоваться для того, чтобы вместить другие системы письма. Например, IRI может иметь китайские или кириллические символы, а URI не может. В общем же, IRI означает то же самое, что и URI. Спецификация языка запросов SPARQL относится к IRI, когда он говорит об именовании ресурсов (или о специальных функциях, которые работают с этими именами ресурсов), а не к URI или URL, потому что IRI является наиболее всеобъемлющим термином.

URI помогли решить еще одну проблему. По мере того, как язык разметки XML становился более популярным, разработчики XML стали совмещать коллекции элементов из разных областей, чтобы создать специализированные документы. Это привело к трудному вопросу: что делать, если два множества элементов для двух различных областей используют одно и то же имя для двух различных вещей? Например, если я хочу сказать, что *название* Тим Бернерс-Ли - это "директор" W3C и что *название* его книги 1999 года "Ткачество в Интернете," мне нужно как-то различать эти два смысла слова "*название*". Информатика использовала термин пространство имен в течение многих лет, чтобы обратиться к набору имен, используемых для конкретной цели. Консорциум W3C выпустил спецификацию, описывающую, как разработчики XML могут сказать, что определенные термины пришли из конкретного пространства имен. Таким образом, они могут различать смысл слова подобного термину "*название*".

Как мы обозначаем пространства имен и ссылаемся на них? Конечно с помощью URI. Например, название стандартного набора основных терминов метаданных Dublin Core (Дублинское ядро) имеет URI http://purl.org/dc/elements/1.1/. XML-документ часто включает в себя атрибут настройки xmlns:dc="http://purl.org/dc/elements/1.1/", чтобы указать, что префикс dc будет использоваться вместо URI http://purl.org/dc/elements/1.1/ в этом документе. Представьте себе, что процессор XML нашел следующий элемент в таком документе:

```
<dc:title> Ткачество в Интернете </dc:title>
```

Это будет означать, что "*название*" в пространстве имен Dublin Core – имеет смысл *название книги*.

Если в XML-документе также определен префикс пространства имен v с атрибутом xmlns:v="http://www.w3.org/2006/vcard/", процессор XML видя следующий элемент будет знать, что он имел в виду "*название*" в смысле "должность", потому что оно взято из словаря vCard, служащего для спецификации терминов относящихся к визитной карточке:

```
<v:title>Director</v:title>
```

Мы видели в главе 1, что утверждение RDF имеет три части. Мы также видели, что имена субъекта и предиката должны принадлежать конкретным пространствам имен, чтобы ни один человек или процесс не спутал эти имена с аналогичными, особенно если одни данные используются в сочетании с другими данными. Как и XML, RDF позволяет определить префикс для представления пространства имен URI, так что набор данных не окажется слишком «многосимвольным», но в отличие от XML, RDF позволяет использовать полный URI, а не префикс. После объявления, что префикс v: относится к http://www.w3.org/2006/vcard/ пространству имен, набор данных RDF можно сказать, что Бернерс-Ли имеет v:title «директор», но он также может сказать, что он имеет <http://www.w3.org/2006/vcard/title> «директор», используя весь URI вместо префикса.

RDF-синтаксис Turtle, N3 и SPARQL использует скобки <>, чтобы сообщить процессору, что это реальный URI, а не просто некоторые строка символов, которая начинается с "http://".

В любом месте в RDF и SPARQL, где вы можете использовать URI, вы можете использовать префикс, если префикс был должным образом объявлен. Мы ссылаемся на часть имени после двоеточия как на локальное имя - это имя берется из пространства имен определенного префиксом. Например, в dc:title локальное имя – title. название.

Подводя итог можно сказать, что люди иногда используют термины URI и URL взаимозаменяемо, но в RDF это URI, потому что мы хотим, чтобы определять ресурсы и информацию об этих ресурсах. URI обычно выглядит как URL, и может быть даже веб-страница по этому адресу, но не может и не быть. Основная задача URI заключается в предоставлении уникального имени ресурса или свойства, а не предоставлении вам веб-страницы. Технические обсуждения могут использовать термин IRI, но это вариация URI.

В SPARQL запросе, где WHERE описывает данные, которые надо вытащить из набора данных, однозначные идентификаторы имеют решающее значение для этого. В некоторых тройках RDF может не быть переменных, которые выбирает SELECT, чтобы показать в результатах, но они необходимы, чтобы определить перекрестные ссылки и взаимозависимости частей данных друг с другом, чтобы соединить их в результате. Это означает, что хорошее понимание роли URI, дает вам больший контроль над вашими запросами.

URI, которые идентифицируют ресурсы RDF, похожи на уникальный идентификатор поля таблиц реляционных баз данных, за исключением того, что они повсеместно уникальны, что позволяет связать данные из разных источников по всему миру, а не только данные из разных таблиц в одной базе данных.

2.3. Resource Description Framework (RDF)

В главе 1 мы узнали следующее об описания ресурсов:

• Это модель данных, в которой основной единицей информации является триплет.
• Триплет состоит из субъекта, предиката и объекта. Вы также можете рассматривать их как идентификатор ресурса, имя атрибута или свойства и значение атрибута или свойства.
• Чтобы избежать какой-либо двусмысленности в информации, изложенной в триплете, субъект и предикат триплета должны быть URI. (Мы также узнали, что мы можем использовать префикс пространства имен вместо URI.)

В этом разделе, мы узнаем больше о различных способах хранения и использования RDF и как субъекты и объекты могут быть больше, чем URI, представляющие конкретные ресурсы или простые строки.

2.3.1. Хранение RDF в файлах

Технический термин для сохранения RDF в виде строки байтов, которая может быть сохранена на диске - сериализация. Мы используем этот термин, а не "файл", потому что были операционные системы, которые не использовали термин "файл" для названной коллекции сохраненных данных, но на практике все RDF сериализации до сих пор были текстовыми файлами, которые использовали различные синтаксисы представления троек. (Хотя они могут быть файлами, хранящимися на дисках, они также могут быть получены динамически, как и многие HTML-страницы.) Формат сериализации, вы увидите наиболее часто, особенно в этой книге, называется Turtle.

Большинство средств работы с RDF может читать и писать все форматы, описанные в этом разделе, и включают в себя инструменты для преобразования между ними. Инструменты запроса, такие как ARQ, который позволяет запрашивать данные, хранящиеся в виде файлов на диске, имеет анализатор (parser) RDF, встроенный в механизм чтения данных, а затем передающий его в «исполнитель» (engine) запроса SPARQL. Это работа парсера беспокоиться о сериализации используемых данных, а не ваша. Вы, возможно, должны подсказать ARQ формат сериализации набора данных, но обычно эти процессоры узнают сами об этом из расширения файла расширения файла.

Этот раздел даст вам некоторое представление о некоторых вещей, которые вы возможно увидете в файле данных RDF. Нет необходимости, знать все подробности, но иногда это удобно знать, какие сериализации используются. Мы посмотрим на то, как разные форматы представляют следующие три факта:

• Книга с ISBN 006251587X имеет создателя Тим Бернерс-Ли.
• Книга с ISBN 006251587X имеет название "Ткачество Веб».
• Название Тим Бернерс-Ли имеет смысл «директор».

В примерах используется URI http://www.w3.org/People/Berners-Lee/card#i для представления Тима Бернерса-Ли, потому что этот URI, он сам использует, чтобы представить себя в FOAF файле. Для представления книги в примерах используется URN urn:ISBN:006251587X.

FOAF файл – это коллекции RDF фактов о людях, таких как, где они работают, где они

учились и кто их друзья. Первоначальная цель проекта FOAF состояла в том, чтобы обеспечить основу для распределенных данных социальных сетей на RDF-основе, но его словарный запас в настоящее время определяет такие основные факты о людях, которые используются гораздо шире, чем в FOAF файле.

Простейший формат называется N-тройка. (Это на самом деле подмножество другой сериализации под названием N3, к которой мы придем в ближайшее время.) В N-тройке, вы должны писать полный URI, внутри угловых скобок и строки внутри кавычек. Каждая тройка располагается на своей собственной строке с точкой на конце.

Например, мы можем представить три описанные выше факта, в виде N-троек:

```
# filename: ex028.nt

<urn:isbn:006251587X> <http://purl.org/dc/elements/1.1/creator>
<http://www.w3.org/People/Berners-Lee/card#i> .
<urn:isbn:006251587X> <http://purl.org/dc/elements/1.1/title> "Weaving the Web" .

<http://www.w3.org/People/Berners-Lee/card#i> <http://www.w3.org/2006/vcard/title> "Director" .
```

(Первая тройка здесь на самом деле не является законной N-тройкой троек, потому что нужно было вставить разрыв строки, чтобы поместить ее на этой странице, а файл ex028.nt включает пример, в котором каждая тройка целиком размещена на отдельной строке.)

Также как в таблицах SQL, порядок набора троек не имеет значения. Если вы перенесете третью тройку в примере (или в любом RDF примере сериализации в этом разделе), на место первой, набор информацией будет идентичным.

Простота N-троек делает этот формат популярным для обучения людей RDF, и некоторые анализаторы могут читать его более быстро, потому что они имеют меньше работы, но многословие формата делает его менее популярным, чем другие форматы.

Старейшая RDF сериализация, RDF/XML, была частью оригинальной спецификации RDF в 1999 году. Прежде чем мы рассмотрим некоторые примеры RDF/XML, имейте в виду, что мы покажем их так, что вы узнаете общие контуры RDF/XML-файла, когда увидите его.

Эти детали кое что для синтаксического анализатора RDF, а не для вас, не беспокойтесь о них, как только Turtle становится стандартом W3C, мы все меньше и меньше стали использовать RDF/XML. На момент написания книги, однако RDF/XML все еще стандартный формат сериализации RDF.

Вот три факта изложенные выше в RDF/XML:

```
<!-- Для XML и RDF/XML так вводится комментарий -->
<!-- filename: ex029.rdf -->

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/"
         xmlns:v="http://www.w3.org/2006/vcard/">

<rdf:Description rdf:about="urn:isbn:006251587X">
      <dc:title>Weaving the Web</dc:title>
      <dc:creator rdf:resource="http://www.w3.org/People/Berners-Lee/card#i"/>
</rdf:Description>
```

```
<rdf:Description rdf:about="http://www.w3.org/People/Berners-Lee/card#i">
        <v:title>Director</v:title>
</rdf:Description> </rdf:RDF>
```

Здесь следует отметить следующее:

• Элемент, содержащий все тройки должен быть элементом RDF из http://www.w3.org/1999/02/22-rdf-syntax-ns# пространства имен.
• Субъект каждой тройки определяется в атрибуте rdf:about элемента rdf:Description.
• Пример использует отдельные rdf:Description элементы для каждой тройки, в тройке об URN ресурса isbn:006251587X вставлены два дочерних элемента внутри rdf:Description - элемент dc:title и элемент dc:creator.
• Объекты dc:title и v:title, представлены в виде простого текста (или, в терминах XML, как PCDATA) между открывающим и закрывающим тегами. Чтобы показать, что dc:creator значение ресурса, а не строка, он находится в атрибуте rdf:resource элемента dc:creator.

Ниже показан другой способ, чтобы выразить ту же самую информацию в RDF/XML:

```
<!-- filename: ex030.rdf -->
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
            xmlns:dc="http://purl.org/dc/elements/1.1/"
             xmlns:v="http://www.w3.org/2006/vcard/">

<rdf:Description rdf:about="urn:isbn:006251587X" dc:title="Weaving the Web">
  <dc:creator>
     <rdf:Description rdf:about="http://www.w3.org/People/Berners-Lee/card#i">
      <v:title>Director</v:title>
     </rdf:Description>
    </dc:creator>
  </rdf:Description>
</rdf:RDF>
```

RDF/XML не стал популярным из-за потенциальной сложности написания и трудности его обработки, например, если часть информации, такая как dc:title значение, может появиться либо в качестве дочернего элемента или как значение атрибута элемента rdf:Description, то XSLT стили и другие инструменты для обработки этой информацию должны проверять много дополнительных вещей, чтобы обработать этот небольшой кусок информации.

Другой формат сериализации N3, который является сокращением от "Нотации 3." Это был личный проект Тима Бернерс-Ли, который он описал как "основной эквивалент RDF в синтаксисе XML,. Он сочетает в себе простоту N-троек с возможностью RDF/XML, чтобы сократить длинные URI он допускает префиксы. Три изложенные выше факта в N3:

```
# filename: ex031.n3
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix v: <http://www.w3.org/2006/vcard/> .

<http://www.w3.org/People/Berners-Lee/card#i> v:title "Director" .
```

```
<urn:isbn:006251587X> dc:creator <http://www.w3.org/People/Berners-Lee/card#i> ; dc:title
"Weaving the Web" .
```

Это выглядит очень похоже на N-тройки, допускается дополнительные пробелы для форматирования и вы можете использовать префиксы пространств имен, чтобы имена были короче. Здесь также необходимо объявить префиксы в первую очередь. Обратите внимание, что эти объявления также выражаются в виде троек с точками на конце.

Как RDF/XML, N3 предлагает некоторые упрощения для описания нескольких фактов о том же предмете. Приведенный выше пример использует идентификатор <urn: isbn:006251587X> только один раз, и после объекта < http://www.w3.org/People/Berners-Lee/card#i> первой тройки об этом книге, вы видите точку с запятой. Это означает, что далее следует еще один предикат для того же субъекта.

Конечный объект имеет в конце точку, чтобы показать, что предложение действительно закончено. Таким образом, вы могли бы сказать, что последние три строки ex031.n3 говорят нам ресурс <urn: isbn:006251587X> имеет значение dc:creator http://www.w3.org/People/Berners-Lee/card#i; кроме того, он имеет значение dc:title «Ткачество Web».

Запятая в N3 означает "следующий триплет имеет тот же субъект и предикат, что и предыдущий, но следующий новый объект". Например, следующий список из двух значений для dc:creator для книги с ISBN 0123735564:

```
#filename: ex032.n3
@prefix dc: <http://purl.org/dc/elements/1.1/> .
<urn:isbn:0123735564> dc:creator
<http://www.topquadrant.com/people/dallemang/foaf.rdf#me> ,
<http://www.cs.umd.edu/~hendler/2003/foaf.rdf#jhendler> .
```

N3 включает и другие интересные функции, такие как способность относиться к графу троек в качестве ресурса, так что вы могли бы сказать, "этот граф троек имеет dc:creator значение "Джейн Смит". Вы также можете определить правила, которые позволяют осуществить логический вывод новых троек, основанные на реальных или ложных условиях в существующем наборе троек, например, к выводу, что, если отец Бриджит Петр и отец Петра Генри, то дед Бриджит Генри. Логический вывод часто играет важную роль в семантических веб-приложениях.

N3 не стал стандартом, и никто не использовал эти дополнительные функции, потому что они вдохновили людей в W3C на другой проект, который и стал стандартом. Позже в этой книге, мы увидим, как обратиться к графам троек и как осуществлять логический вывод со SPARQL.

Если вы используете N3 без этих дополнительных функций, то вы уже используете наш следующий формат сериализации: Turtle. Нет необходимости показывать здесь примеры, потому что любое программное обеспечение, которое понимает Turtle поймет N3 примеры, которые не используют дополнительные функции. Несмотря на свое название Turtle-черепаха движется быстрый всех среди RDF форматов сериализации в гонке за популярность, и W3C планирует его стандартизировать.

Для остальной части этой книги, если не указано иное, все образцы данных будут показаны с использованием синтаксиса Turtle.

Еще более популярным способом хранения троек является стандарт W3C RDFa. Он позволяет хранить субъекты, предикаты и объекты в XML документе, который не был рассчитан на RDF, а также в HTML документе.

"a" в RDFa означает "атрибуты". RDFa определяет несколько новых атрибутов и специфицирует несколько существующих HTML в качестве места для хранения или идентификации субъектов, предикатов и объектов смешанных с данными в этом XML или HTML документе. Справочная роль RDFa в XML и HTML документах делает его отличным для метаданных о содержании контента в этих документах, и утилиты позволяют тянуть тройки из RDFa атрибутов в формате, который позволяет запрашивать их с помощью SPARQL.

Возможность RDFa для встраивания в HTML троек делает его отлично подходящим для совместного использования машиночитаемых данных на веб-страницах, так что автоматизированным процессам сбора данных не нужно делать «скрап экрана» этих страницах.

2.3.2. Хранение RDF в базах данных

Если вам нужно хранить очень большое количество троек, сохраняя их как Turtle или RDF/XML, то один большой текстовый файл не может быть лучшим вариантом, потому что среди систем, которые индексируют данные и решают, какие данные и когда необходимо загрузить в память, системы управления базами данных являются наиболее эффективными. Есть способы, чтобы хранить RDF в реляционных базах данных, таких как MySQL или Oracle, но лучший способ использовать менеджер баз данных оптимизированный для RDF троек. Мы называем его triplestore (тройкохранитель), и оба варианта triplestore - коммерческий и с открытым исходным кодом доступны.

При оценке triplestore, наряду с типичными вопросами менеджера баз данных, таких как размер, скорость, доступность платформы и стоимости, есть несколько вопросов связанных со SPARQL:

• Поддерживает ли он реальный SPARQL, или некоторые SPARQL-подобные запросы и обновления языка?

• Насколько легко дать triplestore SPARQL запрос и затем получить результат обратно, как интерактивно, так и программно?

• Может triplestore служить конечной точкой SPARQL?

• Поддерживает ли triplestore новейший стандарт SPARQL?

Важно

Возможности SPARQL 1.1 UPDATE позволяют весело играть с небольшими файлами данных, но вы увидите его реальную мощь, когда используете triplestore. Возможность добавлять, удалять и изменять данные важна для любой программы управления базами данных, и при использовании triplestore вы будете делать это с помощью признанного стандарта.

2.3.2. Ввод данных

До сих пор мы видели, что субъект и предикат триплета должен быть URI, и что его объект может быть URI или строкой. Объект на самом деле может быть больше, чем просто строкой, потому что вы можете присвоить ему определенный тип данных или тег, который идентифицирует язык текста, такой например, как канадский, французский или русский.

Техническим термином для этих не URI значений является - *литерал*. Введенный литерал имеет тип данных, назначенный для него. Как правило, тип выбирается из схем предлагаемых спецификацией «XML Часть 2», составленной W3C. Когда программа знает, что данное значение есть число или дата, то она понимает, какие математические действия или другую специализированную обработку она может выполнять с ним.

Различные сериализациии RDF имеют различные соглашения для определения типов данных. Ниже показаны несколько троек в формате Turtle с назначенными типами данных:

```
# filename: ex033.ttl
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix dm: <http://learningsparql.com/ns/demo#> .
d:item342 dm:shipped      "2011-02-14"^^<http://www.w3.org/2001/XMLSchema#date> .
d:item342 dm:quantity     "4"^^xsd:integer .
d:item342 dm:invoiced     "false"^^xsd:boolean .
d:item342 dm:costPerItem "3.50"^^xsd:decimal .
```

Как вы можете видеть, именем типа данных может быть полный URI или префикс имени. Также как префиксы используемые в других данных, xsd: префикс для типа данных также должен быть объявлен.

Когда вы опускаете кавычки у литерала в Turtle, процессор делает некоторые предположения о его типе, если значение является словом "true" или "false" или числом. Это означает, что SPARQL процессор будет интерпретировать эти литералы так же, как и в предыдущем примере:

```
# filename: ex034.ttl
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix dm: <http://learningsparql.com/ns/demo#> .
d:item342 dm:shipped      "2011-02-14"^^<http://www.w3.org/2001/XMLSchema#date> .
d:item342 dm:quantity      4 .
d:item342 dm:invoiced      false .
d:item342 dm:costPerItem 3.50 .
```

Назначение типов данных в RDF/XML довольно просто: указываем URI типа данных в атрибуте rdf:datatype. Ниже представлены те же данные, которые описаны выше в формате RDF/XML:

```
<!-- filename: ex035.rdf -->
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
           xmlns:dm="http://learningsparql.com/ns/demo#"
           xmlns:xsd="http://www.w3.org/2001/XMLSchema#">
<rdf:Description rdf:about="http://learningsparql.com/ns/demo#item342">
 <dm:shipped
  rdf:datatype="http://www.w3.org/2001/XMLSchema#date">2011-02-14</dm:shipped>
 <dm:quantity
  rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">4</dm:quantity>
 <dm:invoiced
  rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">false</dm:invoiced>
 <dm:costPerItem
  rdf:datatype="http://www.w3.org/2001/XMLSchema#decimal">3.50</dm:costPerItem>
</rdf:Description>
</rdf:RDF>
```

Потому что все файлы ex033.ttl, ex034.ttl и ex035.rdf хранят одинаковые тройки, SPARQL запрос к любому из этих файлов даст один и тот же ответ.

2.3.3. Делаем RDF более читабельным с помощью тегов и меток

Ранее мы видели, триплет, говорящий, что название должности Тима Бернерс-Ли в W3C является "Director", но для W3C сотрудников в их европейской штаб-квартире во Франции, это название будет "Directeur". У каждой RDF сериализации есть свой собственный способ прикрепить метку языка в строку текста, и мы увидим позже, как SPARQL позволяет сузить результаты запроса литералов, помеченных на определенном языке. Представлять название должности Бернерса-Ли в английском и французском языках, мы могли бы используя эти тройки:

```
# filename: ex036.ttl
@prefix v: <http://www.w3.org/2006/vcard/> .
<http://www.w3.org/People/Berners-Lee/card#i> v:title "Director"@en .
<http://www.w3.org/People/Berners-Lee/card#i> v:title "Directeur"@fr .
```

Двухбуквенные коды, такие как "en" для английского и "fr" для французского являются частью ISO 639 стандарта «Коды для представления названий языков." Вы можете расширить их с помощью дефиса и тега из стандарта ISO 3166-1 "Коды для представления названий стран и их подразделений:

```
# filename: ex037.ttl
@prefix : <http://www.learningsparql.com/ns/demo#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
:sideDish42 rdfs:label "french fries"@en-US .
:sideDish42 rdfs:label "chips"@en-GB .
:sideDish43 rdfs:label "chips"@en-US .
:sideDish43 rdfs:label "crisps"@en-GB .
```

Метка предикат из пространства имен RDF Schema (RDFS) всегда было одним из самых важных свойств RDF. Мы видели в главе 1, что субъект триплета редко передает много информации о себе (например :sideDish42 приведенный выше), потому что его задача - обеспечить уникальный идентификатор, а не описывать что-то. Эта работа предикатов и объектов, используемых с этим субъектом, чтобы описывать.

Из-за этого, в RDF хорошо назначать значения rdfs:label меток на ресурсы таким образом, чтобы человек мог легко понять, что они идентифицируют. Например, в файле FOAF Тим Бернерс-Ли использует URI http://www.w3.org/People/Berners-Lee/card#i для представления себя, но его FOAF файл также включает в себя следующий триплет:

```
# filename: ex038.ttl
<http://www.w3.org/People/Berners-Lee/card#i>
<http://www.w3.org/2000/01/rdf-schema#label> "Tim Berners-Lee" .
```

Использование нескольких значениий rdfs:label меток, каждая со своим собственным тегом языка, является обычной практикой. Коллекция RDF в DBpedia, извлекаемая из Википедии имеет 15 значений rdfs:label меток для ресурса http://dbpedia.org/resource/Switzerland. Следующая тройка, присваивают четыре метки к ресурсу; она использует запятые, чтобы показать, что четыре значения объекта, относятся к тому же субъекту и предикату:

```
# filename: ex039.ttl
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
<http://dbpedia.org/resource/Switzerland> rdfs:label "Switzerland"@en, "Suiza"@es,
"Sveitsi"@fi, "Suisse"@fr .
```

Когда приложения RDF получают информацию в ответ на запрос, то очень часто для этого используют значения rdfs:label меток, связанных с соответствующими ресурсами, если они доступны, а не загадочные URI.

RDF на основе стандарта W3C SKOS для определения словарей, таксономии и тезаурусов предлагает более специализированные версии rdfs:label, в том числе skos: prefLabel свойства для предпочтительных меток и skos:altLabel свойства для альтернативных меток. Единая концепция Продовольственной и сельскохозяйственной организации ООН в тезаурусе SKOS для сельского хозяйства, лесоводства, рыболовства, пищевой и смежных областей, возможно, даже больше значениий skos:prefLabel и десятки значений skos:altLabel для единой концепции, все с отдельными языковыми тегами, начиная от английского до фарси и тайского.

Наряду с rdfs:label, словарь RDF Schema предоставляет свойство RDFS:comment, которое, как правило, хранит более подробное описание ресурса. Используя это свойство, для описания как ресурс (или свойство) используется, можно сделать данные ресурса проще в использовании, так же, как добавление комментариев к исходному коду программы могут помочь людям понять, как работает программа, так что они могут более эффективно использовать ее исходный код.

2.3.4. Пустые узлы в RDF графах

В разделе "Более реалистичные данные и запросы из нескольких триплетов", мы узнали, что набор данных RDF известен как граф. Вы можете представить его визуально с узлами графа, представляющими субъекты и объекты ресурса из набора троек и линий, соединяющих эти узлы, представляющих предикаты. *Почти все узлы имеют имя*, состоящее из URI, которое представляет этот ресурс, или объектов литералов, строк или других типизированных значений.

Почти все узлы? Какую цель может безымянный узел служить? Давайте посмотрим на примере. Во-первых, у нас есть набор троек без каких-либо пустых узлов и теперь мы увидим, как пустой узел может помочь организовать их лучше.

Ниже приведены некоторые тройки, которые представляют записи в нашей адресной книге:

```
# filename: ex040.ttl
@prefix ab: <http://learningsparql.com/ns/addressbook#> .

ab:i0432 ab:firstName    "Richard" ;
         ab:lastName     "Mutt" ;
         ab:postalCode   "49345" ;
         ab:city         "Springfield" ;
         ab:homeTel      "(229) 276-5135" ;
         ab:streetAddress "32 Main St." ;
         ab:region       "Connecticut" ;
```

```
        ab:email        "richard49@hotmail.com" .
```

На рисунке 2-1 изображен граф этих данных. Каждый узел помечен значением субъекта или объекта, а дуги, соединяющие узлы помечены значениями предикатов.

Мы видели, что порядок троек не имеет значения в RDF, и информацию о почтовом адресе Ричарда трудно найти среди другой разбросанной информацией о нем. В терминах моделирования баз данных, запись в адресной книге ex040.ttl является плоской - просто список данных о Ричарде без структурирования их значений.

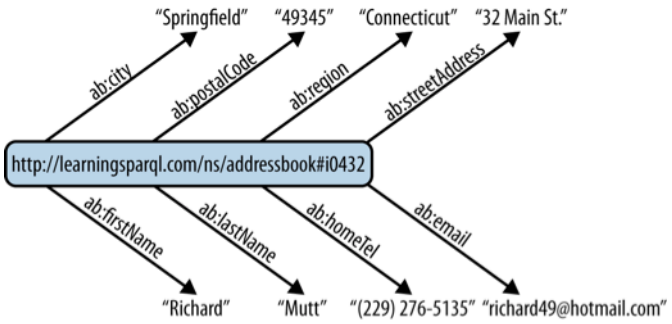


Рисунок 2-1. Граф ab:i0432 адресной книги

Следующая версия имеет новый предикат ab:address, но его значение имеет странный префикс пространства имен: подчеркивание. Это значение в свою очередь, имеет свои собственные значения, описывающие отдельные компоненты адреса Ричарда:

```
# filename: ex041.ttl
@prefix ab: <http://learningsparql.com/ns/addressbook#> .
ab:i0432 ab:firstName     "Richard" ;
         ab:lastName      "Mutt" ;
         ab:homeTel       "(229) 276-5135" ;
         ab:email         "richard49@hotmail.com" ;
         ab:address       _:b1 .

_:b1     ab:postalCode    "49345" ;
         ab:city          "Springfield" ;
         ab:streetAddress "32 Main St." ;
         ab:region        "Connecticut" .
```

Префикс с подчеркиванием означает, что это особый вид узла, известного как пустой узел (bnode – от blank node). Его единственная цель состоит в группировании вместе некоторых других значений. b1 используется как локальное имя для указания места выделенной группы когда необходимо обратиться к этой группе троек из других частей этого набора данных. RDF программное обеспечение, которое читает эти данные может игнорировать значение b1, но оно должно помнить, что Ричард (или, более технически, ресурс ab:i0432) имеет значение ab:address, которое указывает на четыре другие значения.

На рисунке 2-2 показан граф данных ex041.ttl. Узел, представляющий значение ab:address не имеет имени, потому что этот узел не имеет идентификатора - он пустой. Это показывает, что в то время как парсер RDF будет обращать внимание на названия всех других субъектов, предикатов и объектов в наборе данных ex041.ttl, b1 после _ ничего не значит для него, но парсер помнит, что этот узел связан с данными. b1 в ex041.ttl просто

временное местное название узла в этой версии данных. Это имя может не использоваться в новых версиях данных, но когда данные копируются некоторым приложением на основе RDF, то приложение отвечает за поддержание всех соединений в и из каждого пустого узла.

Turtle и SPARQL для представления пустого узла иногда используют пару квадратных скобок ([]) вместо префикса имени с подчеркиванием.

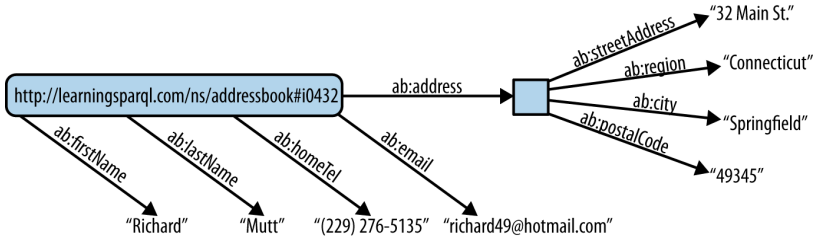


Рисунок 2-2. Использование пустого узла для группировки адресной информации

В примере _b1 пустой узел является объектом одного триплета и субъектом ряда других триплетов. Это является общим в RDF и SPARQL, потому что они используют пустые узлы для подключения ресурса. Например, если у меня есть данные адресной книги структурированные как образец ex041.ttl, я могу использовать запрос SPARQL, чтобы посмотреть улицу Ричарда запрашивая значение ab:StreetAddress из узла ab:address адресной книги, которая имеет ab:firstName "Richard" и ab:lastName "Mutt". Узел ab:address не имеет URI чтобы ссылаться на него, но запросу это и не нужно, потому что в нем можно указать, что он запрашивает значения адресов в графе данных с ab:firstName "Richard" и ab:lastName "Mutt".

2.3.5. Именованные графы

Именованные графы это еще один способ группировки троек вместе. Если вы хотите присвоить имя набору троек, то RDF позволяет приписать метаданные всему, что вы можете определить с URI. Таким образом вы можете приписать любые метаданные к этому набору троек. Например, вы могли бы сказать, что данный набор троек в triplestore пришел с определенного источника в определенное время, или что конкретный набор должны быть заменен другим набором.

Оригинальные спецификации RDF не описывают такой возможности, но поскольку в конечном итоге стало ясно, что это было бы ценно, то более поздние спецификации позволяют это, особенно спецификации SPARQL. Мы увидим в главах 3 и 6, как запросы именуют подмножества коллекции троек.

2.4. Использование и создание словарей: RDF Schema и OWL

Вы можете создать новые идентификаторы URI для всех ресурсов и свойств ваших троек, но когда вы используете существующие, легче подключить другие данные к вашим и это позволяет вам делать больше с данными. Мы уже использовали примеры свойств из нескольких существующих словарей, таких как FOAF и Dublin Core. Если вы хотите использовать существующие словари свойств, то необходимо знать как эти словари выглядят? Какой формат они используют?

Они, как правило, хранятся с использованием RDF Schema и OWL стандартов. Словари составленные с помощью одного из этих стандартов обеспечивает хорошие ссылки для того, кто пишет SPARQL запрос и интересуется какие свойства доступны в запрашиваемом наборе данных. В качестве бонуса, определения часто дополняются метаданными о заявленных терминах словаря, что упрощает его использование при

написании запросов.

Ранее при описании свойств rdfs:label и rdfs:comment, мы упомянули спецификацию W3C RDF Schema. Ее полное название "RDF словарь описания языка: RDF Schema," и она дает людям возможность описывать словари. Как вы уже может быть догадались, описываются эти словари с RDF, так что добраться до метаданных как и до самих данных можно с помощью SPARQL запросов.

Вот некоторые из троек из RDF Schema словарного описания лексики Dublin Core. Они описывают термин "создатель", который мы использовали, чтобы описать отношение Тима Бернерс-Ли к книге, представленной URI urn:isbn:006251587X:

```
# filename: ex042.ttl
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
dc:creator
        rdf:type rdf:Property ;
        # a rdf:Property ;
        rdfs:comment "An entity primarily responsible for making the resource."@en-US ;
        rdfs:label "Creator"@en-US .
```

Последние две тройки описывают свойство http://purl.org/dc/elements/1.1/creator с помощью rdfs:comment и rdfs:label, и они оба имеют языковые теги чтобы показать, что они написаны на американском английском. (Стандарт Dublin Core был разработан в Дублине, штат Огайо, а не в более известном в Ирландии Дублине.)

В первой тройке в ex042.ttl используется свойство, которое мы не видели прежде, для того, чтобы сказать, что dc:creator – это rdf:type. (Turtle, N3 и SPARQL предлагает нам использовать "a" вместо rdf:type. Свойство rdf:type указывает, что что-то определенного типа, как показано в закомментированный строке в ex042.ttl и строке над ней (обе строки означают одно и то же). На самом деле rdf:type говорит, что dc:creator это член класса RDF:Property.

RDF Schema это словарь, в котором тройки описывают факты (например, rdfs:label и rdfs:comment являются свойствами) так же, как в Dublin Core пример ex042.ttl объявляет, что dc:creator - свойство.

В дополнение к определению свойств, RDF Schema позволяет определять новые классы ресурсов. Например, ниже показано, как можно объявить классы ab:Musician и ab:MusicalInstrument для данных некоторой адресной книги:

```
# filename: ex043.ttl
@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
ab:Musician
        rdf:type rdfs:Class ;
        rdfs:label "Musician" ;
        rdfs:comment "Someone who plays a musical instrument" .
ab:MusicalInstrument
        a rdfs:Class ;
        rdfs:label "musical instrument" .
```

При объявлении класса можно описать гораздо больше метаданных – например, что является подклассом данного класса, но на примере приведенных выше метаданных, мы можем увидеть еще одну очень интересную особенность RDFS. Ниже объявлено

свойство ab:playsinstrument:

```
# filename: ex044.ttl
@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
ab:playsInstrument
        rdf:type rdf:Property ;
        rdfs:comment "Identifies the instrument that someone plays" ;
        rdfs:label "plays instrument" ;
        rdfs:domain ab:Musician ;
        rdfs:range ab:MusicalInstrument .
```

Свойство rdfs:domain означает, что, если использовать свойство ab:playsInstrument в триплете, то субъектом тройки является ab:Musician. Свойство rdfs:range означает, что объект такой тройки есть ab:musicalinstrument.

В традиционном объектно-ориентированном мышлении, если мы говорим "члены класса Musician имеют свойство playsInstrument," это означает, что член класса Musician должен иметь значение playsInstrument. RDFS и OWL подход следует противоположном направлении: последние два тройки ex044.ttl говорят нам, что если что-то имеет значение ab:playsinstrument, то это член класса ab:Musician и его значение ab:playsInstrument является членом класса ab:MusicalInstrument.

Скажем, добавим один триплет в конец ex040.ttl:

```
# filename: ex045.ttl
@prefix ab: <http://learningsparql.com/ns/addressbook#> .
ab:i0432 ab:firstName       "Richard" ;
        ab:lastName         "Mutt" ;
        ab:postalCode       "49345" ;
        ab:city             "Springfield" ;
        ab:homeTel          "(229) 276-5135" ;
        ab:streetAddress    "32 Main St." ;
        ab:region           "Connecticut" ;
        ab:email            "richard49@hotmail.com" ;
        ab:playsInstrument  ab:vacuumCleaner .
```

После добавления новой тройки в конце ex045.ttl, процессор SPARQL знает, что Ричард Матт (или, точнее, ресурс ab:i0432) теперь член класса ab:Musician, потому что ab:playsInstrument имеет домен ab:Musician. Также поскольку ab:playsinstrument имеет rdfs:range ab:MusicalInstrument, постольку ab:vacuumCleaner теперь является членом класса ab:MusicalInstrument, даже если он никогда не был им прежде.

Когда мы укажем в SPARQL запросе дать нам имена и фамилии всех музыкантов из вышеупомянутой адресной книги, он выдаст нам Ричарда Маттat, хотя данные не имеет триплета, который указывает, что он является членом этого класса. Если бы мы использовали объектно-ориентированную систему, то сначала объявили бы новый экземпляр класса Musician, а затем присвоили бы ему все данные о Ричарде. При использовании стандартов на основе RDF, добавление одного свойства в метаданные существующего ресурса ab:i0432, влечет за собой то, что ресурс становится членом класса, членом которого он не был раньше.

SPARQL обработчик RDFS данных, вероятно, будет обработчиком и OWL данных. Язык OWL опирается на RDFS, и не так много программного обеспечения, которое поддерживает RDFS и не

поддерживает, по меньшей мере некоторые из языков OWL.

Эта способность ресурсов RDF, становиться членами классов в зависимости от их значений данных сделала RDF технологии популярными в таких областях, как научно-исследовательская медицина и разведывательные органы. Исследователи могут постепенно накапливать данные возможно относящиеся к определенной структуре, а затем убедиться, что структура действительно там присутствует и установить какие ресурсы оказываются членами какого классы, и каковы их отношения.

RDFS позволяет определять классы как подклассы других классов, и (в отличие от объектно-ориентированных систем) свойства, как подсвойства других свойств, что расширяет возможности SPARQL в получении информации. Например, если бы мы сказали, что ab:Musician является подклассом foaf:Person, а затем запросили телефоны всех экземпляры foaf:Person в наборе данных, который включает данные из ex044.ttl и ex045.ttl , то в результате мы получили бы номер телефона Ричарда, потому что, будучи в классе музыкантов Ричард также в классе foaf:Person. Мы увидим еще примеры в главе 9.

Предлагаемый W3C, Web Ontology Language (Язык Web-онтологий), сокращенно OWL, опирается на RDFS и позволяет определять онтологии. Онтологии - это формальные определения словарей, которые позволяют определять сложные структуры, а также новые отношения между словарными терминами и между членами классов. Онтологии часто описывают очень конкретные областяи, таких как научные направления исследований, так что ученые из разных учреждений могут легко обмениваться данными.

Онтология определяемая OWL является еще одним набором троек. Сама по себе OWL онтология, объявив классы и свойства, которые понимает OWL-ориентированное программное обеспечение, позволяет делать логические выводы из данных.

Не определяя большую, сложную онтологию, многие разработчики RDF используют только несколько классов и свойств из OWL, чтобы добавить метаданные к их тройкам. Например, как вы думаете, сколько информации имеет следующий набор данных о ресурсе ab:i9771, Синди Маршалл?

```
# filename: ex046.ttl
@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
ab:i0432
        ab:firstName "Richard" ;
        ab:lastName "Mutt" ;
        ab:spouse ab:i9771 .
ab:i8301
        ab:firstName "Craig" ;
        ab:lastName "Ellis" ;
        ab:patient ab:i9771 .
ab:i9771
        ab:firstName "Cindy" ;
        ab:lastName "Marshall" .
ab:spouse
        rdf:type owl:SymmetricProperty ;
        rdfs:comment "Identifies someone's spouse" .
ab:patient
        rdf:type rdf:Property ;
```

```
        rdfs:comment "Identifies a doctor's patient" .
ab:doctor
        rdf:type rdf:Property ;
        rdfs:comment "Identifies a doctor treating the named resource" ;
        owl:inverseOf ab:patient .
```

Похоже, он включает только имя и фамилию, но OWL-ориентированный процессор увидит больше. Потому что свойство ab:spouse (супруг) является симметричным и ресурс ab:i0432 включает значение ab:i9771 (Синди), процессор OWL «поймет», что ресурс ab:i9771 имеет ресурс ab:i0432, как ее супруг. Он также «поймет», что если значение ab:patient является обратным значению ab:doctor и ресурс ab:i8301 имеет значение ab:patient ab:i9771, то ресурс ab:i9771 имеет значение ab:doctor. Теперь мы знаем, кто являются супругом и доктором Синди, хотя эти факты явно не входят в набор данных.

Если у вас много классов, свойств и отношений между ними, чтобы определять и изменять их, то вместо непосредственного использования синтаксиса RDF/XML или Turtle в ваших RDFS или OWL моделях, лучше использовать графический инструмент. Известным инструментом с открытым исходным кодом для этого является Protégé, разработанная в Университете Стэнфорда. Наиболее популярный коммерческий инструмент TopBraid Composer. (Бесплатная версия TopBraid Composer позволяет осуществлять моделирование, а также выполнять SPARQL запросы; коммерческие версии добавляют новые функции для разработки и развертывания приложений.) Эти инструменты значительно сократят время, затрачиваемое на разбор синтаксиса и при этом сохранят вашу работу, используя стандартный синтаксис.

OWL предлагает множество других способов определения свойств и отношений классов, так что процессор может выводить новую информацию из существующего набора. Этот пример показал, что вам не нужно много OWL, чтобы получить некоторые преимущества от него. Добавляя только немного метаданных (например, информацию о свойствах ab:spouse, ab:patient и ab:doctor в вышеприведенном примере) в небольшой набор данных (о Ричарде, Крейге, и Синди) мы получили больше из этого набора данных, чем мы первоначально вложили в него. Это один из самых больших выигрышей семантических веб-технологий.

Обновление OWL 2 к первоначальному стандарту OWL представило несколько профилей или подмножества OWL, которые специализируются на определенных видах приложений. Эти профили легче внедрять и использовать, чем пытаться взять на себя все OWL сразу. Если вы намереваетесь делать некоторые моделирования данных с OWL, посмотрите на OWL 2 RL, OWL 2 QL и OWL 2 EL в качестве возможных исходных точек для ваших нужд, особенно если ваше приложение нуждается в масштабировании, потому что эти профили разработаны для того, чтобы сделать проектирование проще для реализации крупномасштабных систем для конкретных областей.

Из всех W3C семантических веб-стандартов, OWL ключевой для добавления "семантики" в "семантической сети." Термин "семантика" иногда определяется как смысл слов, и те, кто сомневается в ценности технологий семантической сети мог бы поставить под сомнение жизнеспособность хранения всех значений слова машиночитаемым образом. Однако, как мы видели выше, нет необходимости хранить все смыслы слова, чтобы повысить ценность данного набора данных. Например, просто знание того, что "супруг" является симметричным термином, позволило выяснить личность супруга Синди, хотя этот факт не был частью набора данных. Мы узнаем больше о RDFS и OWL в главе 9.

2.5. Связанные данные

Идея связанных данных новее, чем идея семантической сети, но иногда легче думать о том, что семантическая сеть строилась на идеях связанных данных (Linked Data). Связанные данные не спецификация, а совокупность лучших практик для обеспечения инфраструктуры данных, что делает легче обмен данными по сети. Вы можете использовать семантические веб-технологии, такие как RDFS, OWL, SPARQL и для создания приложений вокруг связанных данных.

Тим Бернерс-Ли придумал эти четыре принципа связанных данных в 2006 году (полужирным отмечены его формулировки):

1. **Используйте URI, как имена для вещей**. URI, являются лучшим способом для однозначной идентификации вещи, и, следовательно, определения связи между вещами.
2. **Используйте HTTP URI, так чтобы люди могут посмотреть эти имена.** Вы, возможно, видели URI, которые начинаются с ftp:, mailto: или префиксов составленных определенным сообществом, но их использование снижает совместимость данных.
3. **Когда кто-то смотрит в URI, ресурс предоставляет полезную информацию, используя стандарты (RDF *, SPARQL).** В то время как URI может быть просто именем, а не адресом веб-страницы, этот принцип говорит, что вы можете также положить что-то там. Это может быть веб-страницей HTML, или чем-то еще, но оно должно использовать признанный стандарт. RDFS и OWL позволяют изложить перечень терминов и информацию об этих терминах и их взаимоотношениях в машиночитаемом виде, например, с помощью SPARQL запросов. Если URI, который идентифицирует ресурс приводит к RDF или OWL описанию ресурса, это большая помощь для приложений. (Звездочка в "RDF *" означает, что это относится и к RDF и RDFS в качестве стандартов.)
4. **Включайте ссылки на другие URI, так что они смогут обнаружить больше вещей.** Представьте себе, если ни одна из оригинальных HTML-страниц не имела бы элементов, связанных с другими страницами, это не было бы похоже на сеть. Выход за пределы этого HTML, соединяющей элемент, различные словари RDF предоставляющие другие свойства позволяют сказать: "эти данные (или этот элемент данных) имеет определенную связь с другим ресурсом в Интернете." Когда приложения могут следовать по ссылкам, они могут сделать интересное новые заключения.

В беседе в 2010 году на 2-ой Правительственный Экспо в Вашингтоне, округ Колумбия, Бернерс-Ли дал довольно нетехническое введение в Linked Data, в ней он предложил награждения правительств звездами за обмен данными в сети. Они получили бы, по крайней мере одну звезду просто за предоставление любого вида ресурса в сети доступном для всех. Им будет присвоено две звездочки за предоставление данных в машиночитаемом формате, (независимо от формата). Они заслужат три звезды за данные в сети, предоставленные в непатентованном формате, например CSV, а не таблиц Microsoft Excel. Предоставление данных в Linked Data формате, в котором понятия определены с помощью URLs, что облегчает реализовать перекрестные ссылки с другими данными, позволит заработать четыре звезды. Наконец, правительство получит полные пять звезд если данные подключены к другим данным, путем предоставления ссылки на связанные с ними данных, особенно ссылок с использованием URL-адресов.

Он вел этот разговор в то время, когда правительства США и Великобритании только начинали делать больше данных доступными в Интернете. Это было бы хорошо для приложений на SPARQL-основе, чтобы общественные данные были доступны в RDF, но это было бы слишком, просить об этом государственные органы с низкими бюджетами

и ограниченной технической экспертизой.

Термин "Linked Open Data" также становится популярным. Растет количество свободно доступных общественности данных - замечательная вещь.

2.6. Прошлое, настоящее и будущее SPARQL

RDF предоставляет большие возможности для моделирования и хранения данных, а также инфраструктура связанных данных предлагает огромное количество информации для анализа. Пока используется RDF, создаются библиотеки программирования, которые позволяют загружать тройки в структуры данных популярных языков программирования, на базе которых строятся приложения использующие эти данные. Как реляционные базы данных, так и XML миры показали, однако, что просто язык запросов, который не требует компиляции кода делает это гораздо проще для людей (в том числе разработчиков не вмешивающихся в технологии), чтобы быстро разработать приложение.

RDF стал стандартом в 1999 году. К 2004 году было разработано более десятка языков запросов в качестве коммерческих, академических и личных проектов. В том же году, W3C сформировал рабочую группу RDF-DAWG доступа к данным RDF. После анализа эксплуатационных требований, она выпустила первый проект спецификации языка SPARQL запросов в конце 2004 г. В начале 2008 года язык запросов, протокол и результаты запросов в формате XML стали рекомендацией или официальной спецификацией W3C.

К тому времени, реализации SPARQL уже существовали, и как только стандарт 1.0 стал официальным, появилось много вспомогательных для поддержки стандарта. Был добавлен Triplestores и больше автономные инструменты, такие как ARQ. Начали появляться конечные точки, принимающие SPARQL запросы и предоставляющие через Интернет результаты в различных форматах.

Раннее использование SPARQL привело к новым идеям о путях его улучшения. Устав RDF-DAWG истек в 2009 году и W3C создали новую Рабочую группу, чтобы заменить их: Рабочая группа SPARQL. Рабочая группа SPARQL выпустила свои первые, рабочие черновики спецификаций SPARQL 1.1 в конце 2009 года, и реализации новых возможностей начали появляться вскоре после этого. Они завершили свою работу (в том числе несколько новых спецификаций) в конце 2012 года, а после несколько шагов доработки в марте 2013 года спецификация SPARQL 1.1 стала рекомендацией W3C.

2.7. Характеристики SPARQL

Спецификации SPARQL 1.0 включают три документа:

- **SPARQL Query Language for RDF** охватывает синтаксис самих запросов. Главы 1 и 3 охватывают использование языка запросов, в том числе новых функций, которые добавляет SPARQL 1.1.
- **SPARQL Protocol for RDF** определяет, как программа должна передавать SPARQL запросы в SPARQL службы обработки запросов и как эти службы должны вернуть результаты. Эта спецификация больше касается SPARQL программных исполнителей, чем людей, пишущих запросы.
- **SPARQL Query Results XML Format** описывает простой формат XML для обработчиков запросов, используемый при возвращении результатов. Если вы отправляете запрос к процессору и задаете формат XML, вы можете использовать XSLT или другой инструмент для преобразования XML-результатов в то, что вы любите. Мы узнаем больше об этом в главе 8.

SPARQL 1.1 предпринял усилия пересмотреть эти три документа (без заметных изменений в XML-формате результатов запроса) и добавил восемь новых рекомендаций:

- The **Overview** - обзорный документ описывающий набор из 11 рекомендаций и роль каждого из них.
- The **Federated Query** - спецификация описывает, как один запрос может извлекать данные из нескольких источников. Это гораздо упрощает создание приложений, которые используют преимущества распределенных средах. В разделе "Федеративные запросы: поиск данных в нескольких наборах из одного запроса" главы 3 описывается, как это сделать.
- The **Update** - спецификация описывает основное различие между SPARQL 1.0 и 1.1, изменение SPARQL от просто языка запросов к чему-то, что может добавить данные в набор данных, заменить и удалить их. Глава 6 охватывает ключевые особенности этой спецификации.
- The **Service Description** - спецификация описывает, как программа-клиент может точно определить, какие характеристики движок SPARQL поддерживает.
- **Query Results JSON Format** - спецификация описывает JSON формат результатов запроса. Он описывается в главе 8.
- **Query Results CSV and TSV Formats** - спецификация описывает CSV и TSV форматы результатов запроса. Они также описываются в главе 8.
- The **Graph Store HTTP Protocol** - спецификация расширяет SPARQL протокол связи между клиентом и процессором SPARQL для работы с графами или наборами троек. Например, он обеспечивает HTTP способ сказать "вот граф, который надо добавить к набору данных" или "удалить граф http://my/fine/graph из набора данных." Это описано в разделе "SPARQL и HTTP" главы 10.
- The Entailment Regimes - спецификация описывает критерии для определения того, какую информацию процессор SPARQL должны принимать во внимание при выполнении следования. Что такое следование? Если A влечет B, и A истина, то B истина. Если A - сложный набор фактов, то может быть удобно иметь такие технологии, которые помогут вам выяснить, действительно ли B верно. Эта спецификация разъясняет, какие наборы информации, следует принять во внимание и когда при выполнении следования.

Вы можете найти спецификации SPARQL (и все другие стандарты и проекты W3C) на http://www.w3.org/TR/.

2.8. Резюме

В этой главе мы узнали:

- Что такое Семантический Веб
- Почему URI, являются основой семантической сети, его отношение к URL и IRIs, и роль пространства имен
- Как люди могут хранить RDF, и как они могут определить типы данных и языки строковых литералов в своих данных
- Что такое пустые узлы и именованные графы, какую гибкость они могут добавить к тому, как вам организовать и отслеживать данные
- Как RDFS и OWL позволяют определить свойства и классы, а также метаданные об этих свойствах и классах, чтобы получить больше информации
- Как используются связанные данные для обмена данными и построения семантических веб-приложений
- История SPARQL и технические характеристики, которые составляют стандарт SPARQL

Глава 3. SPARQL запросы: глубокое погружение

Глава 1 дала вам первые навыки написания и выполнения запросов SPARQL. В этой главе, мы будем разбираться в более мощных особенностях языка запросов SPARQL, вот аннотации некоторых разделов этой главы:

"Более читабельные результаты запросов"

URI, имеют важное значение для выявления и увязки вещей, но когда пришло время для отображения результатов запроса в приложении, пользователи хотят видеть информацию, которую они могут читать, не то, что выглядит как куча веб-адресов.

"Данные, которых там может не быть"

В главе 1, мы начали учиться тому, как запросить данные, которые соответствуют определенным шаблонам. Если вы умеете запросить данные, которые должны или наоборот не должны соответствовать определенным закономерностям, то это делает ваши запросы более гибкими, что особенно полезно при изучении данных, с которыми вы еще не знакомы.

"В поисках данных, которые не отвечают определенным условиям"

Большая часть SPARQL запросов служит для получения данных, которые соответствуют определенным шаблонам. Что делать, если вы хотите получить данные, которые не соответствует определенному шаблону, например, для его удаления?

" Дальнейший поиск данных"

SPARQL предлагает несколько простых способов, чтобы запросить набор троек и дополнительных троек, которые могут быть связаны с ними.

"Устранение избыточного вывода»

Если вы ищете тройки, которые соответствуют определённому шаблону, и исполнитель SPARQL запроса находит несколько одинаковых экземпляров некоторых значений, он покажет вам все из них, пока вы не скажете, что не надо показывать одинаковые результаты.

"Сочетание различных условий поиска»

SPARQL позволяет задать, в одном запросе, который соответствует определенным шаблонам и другие шаблоны, которые соответствуют другим триплетам. Вы можете также запросить данные, удовлетворяющие одному из двух наборов шаблонов.

"Фильтрация данных на основе условий"

Задание логических условий, которые могут включать регулярные выражения, позволяет еще больше уточнить ваш результат.

"Получение определенного количества результатов"

Если ваш запрос может получить больше результатов, чем вы хотите, вы можете ограничить количество возвращаемых значений.

"Запросы к поименованным графам"

Когда тройки в наборе данных организованы в поименованные графы, можно использовать их членство в одном или нескольких графах, как часть вашего условиям поиска.

"Запросы в запросах"

Подзапросы позволяют ставить запросы в запросах, так что вы можете разбить сложный запрос на более легко управляемые части.

"Объединение значений и присвоение значений переменным"

SPARQL 1.1 дает больший контроль над тем, как использовать переменные, чтобы ваши запросы и приложения имели еще большую гибкость для работы с полученными данными.

"Создание таблицы значений в запросах"

SPARQL 1.1 позволяет создавать таблицы значений для использования в качестве условий фильтрации.

"Сортировка и агрегирование"

Вы можете отсортировать результаты поиска, найти промежуточные, средние и другие совокупные значения для всей набора или для отсортированных групп данных.

"Запрос удаленной службы SPARQL"

Вы можете послать запрос к файлу данных, расположенных на вашем жестком диске, но вы также можете послать его к другим наборам данных по всему миру, которые доступны на вашем компьютере.

"Федеративные запросы"

Из одного запроса можно обратиться за данными из нескольких источников, как местных, так и удаленных, а затем поместить результаты вместе.

3.1. Более читабельные результаты запроса

В разделе "Более реалистичные данные и запросы из нескольких триплетов " в главе 1, мы видели запрос, который спрашивает, кто в данных адресной книги имеет телефонный номер (229) 276-5135:

```
# filename: ex008.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?person
 WHERE
{ ?person ab:homeTel "(229) 276-5135" . }
```

При запуске этого запроса к данным:

```
# filename: ex012.ttl
@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d: <http://learningsparql.com/ns/data#> .

d:i0432 ab:firstName   "Richard" .
d:i0432 ab:lastName    "Mutt" .
d:i0432 ab:homeTel     "(229) 276-5135" .
d:i0432 ab:email       "richard49@hotmail.com" .

d:i9771 ab:firstName   "Cindy" .
d:i9771 ab:lastName    "Marshall" .
d:i9771 ab:homeTel     "(245) 646-5488" .
d:i9771 ab:email       "cindym@gmail.com" .

d:i8301 ab:firstName   "Craig" .
d:i8301 ab:lastName    "Ellis" .
d:i8301 ab:email       "craigellis@yahoo.com" .
d:i8301 ab:email       "c.ellis@usairwaysgroup.com" .
```

он выдает результат:

```
-------------------------------------------------------------------------
|  person                                                               |
=========================================================================
|  <http://learningsparql.com/ns/data#i0432>                            |
-------------------------------------------------------------------------
```

Это не очень хороший ответ, но, запрашивая имя и фамилию человека с этим номером телефона:

```
# filename: ex017.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?first ?last
WHERE
{
 ?person ab:homeTel "(229) 276-5135" .
 ?person ab:firstName ?first .
 ?person ab:lastName ?last .
}
```

мы получали то, что необходимо:

```
-----------------------------------
|  first        |  last           |
=======================
| "Richard"  |  "Mutt"       |
-----------------------------------
```

Точка с запятой означает то же самое в SPARQL запросе, что это означает в Turtle: "Вот еще предикат и объект для данного субъекта". Используя эту аббревиатуру, построим следующий запрос, который будет работать точно так же, как в предыдущей :

```
# filename: ex047.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?first ?last
WHERE
{
 ?person ab:homeTel "(229) 276-5135" ;
          ab:firstName ?first ;
          ab:lastName ?last .
}
```

Многие (если не большинство) запросы SPARQL включают в себя несколько троек, которые ссылаются на один и тот же субъект. Точка с запятой для перечисления троек в том же субъекте является очень распространенным явлением.

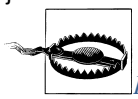
Запросы SPARQL часто выглядят так, что должны возвратить только удобочитаемые данные. Как например в адресной книге понятно, для чтения данных о человеке подойдут значения ab:firstName и ab:lastName, связанные с каждой записью. Различные наборы данных могут иметь различные свойства, связанные с их URI, как читаемых альтернатив, но одно свойство, в частности, не было так популярно в связи с удобночитаемостью, как.

Когда ваш запрос извлекает данные в виде URI, то хорошей идеей будет также извлекать значения rdfs:label, связанных с этими URI.

3.1.1. Использование меток, предусмотренных DBpedia

В примере главы 1, мы увидели запрос на получение списка альбомов, произведенных Timbaland и исполнителей. Результаты запроса фактически перечислили идентификаторы URI, которые представляли эти альбомы и исполнителей, и они были плохо читаемыми. Мы можем сделать запрос лучше, используя значения меток rdfs:label:

```
# filename: ex048.rq
 PREFIX d: <http://dbpedia.org/ontology/>
SELECT ?artistName ?albumName
WHERE
{
album d:producer :Timbaland .
?album d:musicalArtist ?artist .
?album rdfs:label ?albumName .
?artist rdfs:label ?artistName .
}
```

 *Как и несколько других запросов, которые мы уже видели, этот запрос использует префикс d:, но обратите внимание, что здесь он отсылает к другому пространства имен URI. Никогда не принимайте префикс как должное, что он обозначает конкретное URI; всегда проверяйте что он определяет. Кроме того, все префиксы должны быть объявлены в первую очередь; в приведенном выше запросе, кажется, что «:» перед Timbaland не объявлен, но на DBpedia форме, откуда этот запрос посылается, можно увидеть, что декларация «:» уже есть.*

Как выясняется, у каждого исполнителя и названия альбома имеется несколько значений rdfs:label с разными языковыми тегами, назначенными каждому значению, например "en" для английского и "de" (Deutsch) для немецкого языка. (Название альбома все еще отображается на английском языке, потому что он был выпущен под таким названием в этих странах). Когда мы вводим этот запрос в SNORQL DBpedia интерфейсе, он выдает нам каждую комбинацию из них, начиная с нескольких для Мисси Эллиот "Back in the day", как показано на рисунке 3-1.

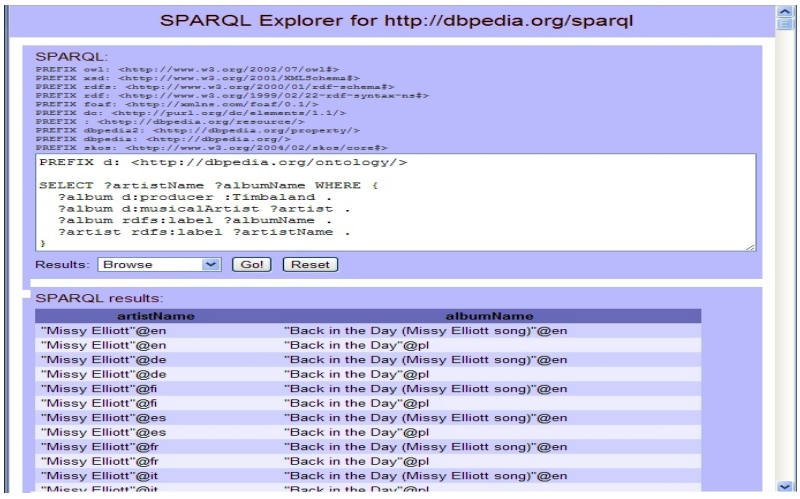


Рисунок 3-1. Результаты запроса об альбомах, производимых Timbaland

Ключевое слово FILTER позволит нам определить, что мы хотим получить значения меток и названия альбомов только на английском языке:

```
# filename: ex049.rq PREFIX d: <http://dbpedia.org/ontology/>
SELECT ?artistName ?albumName
WHERE
{
?album d:producer :Timbaland .
?album d:musicalArtist ?artist .
?album rdfs:label ?albumName .
?artist rdfs:label ?artistName .
FILTER ( lang(?artistName) = "en" )
FILTER ( lang(?albumName) = "en" )
}
```

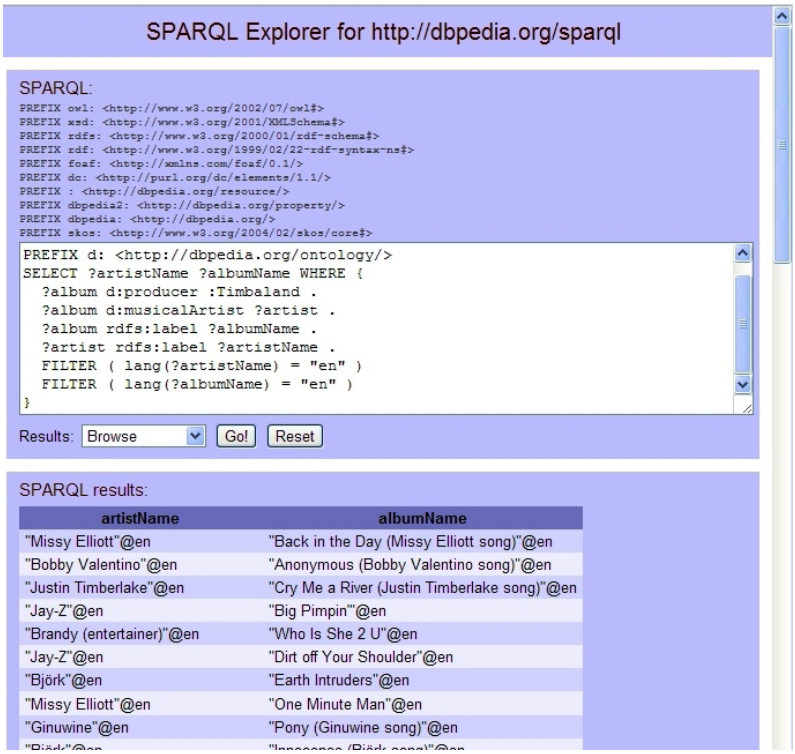


Рисунок 3-2. Альбомы, произведенные Timbaland на английском языке

3.1.2. Получение меток из схем и онтологий

RDF Schemas и OWL онтологии могут обеспечить все виды метаданных о терминах и отношениях, которые они определяют, а иногда простейшим и наиболее полезным набором информации является значение rdfs:label, связанное с терминами, определяемыми этими онтологиями.

Поскольку RDF Schemas и OWL онтологии сами являются коллекциями троек, и если набор данных имеет онтологию, связанную с ним, запрос информации из этого набора данных и онтологии вместе, может помочь вам получить больше информации за счет метаданных.

Ресурсы, используемые в качестве субъектов или объектов троек часто имеют метаданные, связанные с ними, но помните: предикаты тоже являются ресурсами, и часто также имеют ценные метаданные, связанные с ними в RDF Schema или OWL онтологии. Как и с любым другим ресурсом, значения rdfs:label являются одними из первых вещей, которые надо проверить.

Вот некоторые данные о Ричарде Матт представленные с помощью терминов FOAF словаря. Имена свойств не слишком загадочны, но не очень понятны:

```
# filename: ex050.ttl
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
<http://www.learningsparql.com/ns/demo#i93234>
foaf:nick "Dick" ;
foaf:givenname "Richard" ;
foaf:mbox "richard49@hotmail.com" ;
foaf:surname "Mutt" ;
foaf:workplaceHomepage <http://www.philamuseum.org/> ;
foaf:aimChatID "bridesbachelor" .
```

При запросе этих данных вместе с FOAF онтологией, мы можем запросить значения меток, ассоциированных со свойствами, что делает данные легче для чтения. ARQ может принимать несколько --data arguments, FOAF онтологиz хранится в файле под названием index.rdf, запрос ex052.rq выполняется к совокупности триплетов index.rdf и ex050.ttl:

```
arq --data ex050.ttl --data index.rdf --query ex052.rq
```

В представленном запросе ex052.rq первый шаблон триплета запрашивает все тройки, потому что все три части триплета переменные. Второй триплет связывает значения rdfs:label, связанных со значениями ?property первого триплета со значением переменной propertyLabel?. SELECT список просит для этих значений propertyLabel, а не URIs, которые представляют свойства и значения ?value:

```
#filename: ex052.rq
PREFIX foaf: http://xmlns.com/foaf/0.1/
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?propertyLabel ?value
WHERE
{
 ?s ?property ?value .
?property rdfs:label ?propertyLabel .
}
```

Обратите внимание, как переменная ?property находится в положении предикатов первого триплета и субъектной позиции второго. Ввод той же переменной в разных частях различных триплетов распространенный способ найти интересные связи между данными, о которых вы знаете и данными, которые вам хорошо неизвестны.

Результат выполнения этого запроса к комбинации данных о Ричард Матт, показанных выше и FOAF онтологии намного более читабельны, чем просто сами данные:

```
-------------------------------------------------------------------------------------------
| propertyLabel                | value                                                    |
===========================================================================================
| "personal mailbox"           | "richard49@hotmail.com"                                  |
| "nickname"                   | "Dick"                                                   |
| "Surname"                    | "Mutt"                                                   |
| "Given name"                 | "Richard"                                                |
| "workplace homepage"         | <http://www.philamuseum.org/>                            |
| "AIM chat ID"                | "bridesbachelor"                                         |
-------------------------------------------------------------------------------------------
```

rdfs:comment является еще одним популярным свойством, особенно в стандартах, которые используют RDFS или OWL термины, поэтому проверка значения ресурса rdfs:comment часто может помочь вам узнать больше.

Есть и другие виды меток помимо rdfs:label. W3C SKOS (Simple Knowledge Organization System) - простая система организации знаний - стандарт OWL онтологии, предназначенный для представления таксономии и тезаурусов. Его свойство skos:prefLabel предпочтительная метка для конкретного понятия, а свойство skos:altLabel альтернатива метке. Они обе объявлены в онтологии SKOS как подсвойства (то есть, более специализированные версии, чем rdfs:label).

Это характерно для части домена конкретных стандартов (таких как свойства skos:prefLabel и skos:altLabel) базироваться на чем-то из более общего стандарта, такого например, как rdfs:label. Подключение к обобщенному стандарта делает данные более удобными для программ, которые могут не знать о специализации версии.

3.2. Данные, которых может не быть

Давайте расширим нашу адресную книгу набором данных, добавив прозвище Ричарда и рабочий номер телефона для Крейг:

```
# filename: ex054.ttl
@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d: <http://learningsparql.com/ns/data#> .

d:i0432 ab:firstName   "Richard" .
d:i0432 ab:lastName    "Mutt" .
d:i0432 ab:homeTel     "(229) 276-5135" .
d:i0432 ab:nick        "Dick" .
d:i0432 ab:email       "richard49@hotmail.com" .

d:i9771 ab:firstName   "Cindy" .
d:i9771 ab:lastName    "Marshall" .
d:i9771 ab:homeTel     "(245) 646-5488" .
d:i9771 ab:email       "cindym@gmail.com" .

d:i8301 ab:firstName   "Craig" .
d:i8301 ab:lastName    "Ellis" .
d:i8301 ab:workTel     "(245) 315-5486" .
```

```
d:i8301 ab:email        "craigellis@yahoo.com" .
d:i8301 ab:email        "c.ellis@usairwaysgroup.com" .
```

Если выполнить запрос, который просит имена, фамилии и рабочие телефонные номера,

```
# filename: ex055.rq
 PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?first ?last ?workTel
WHERE
{
?s ab:firstName ?first ;
   ab:lastName ?last ;
   ab:workTel ?workTel .
}
```

то он даст эту информацию для Крейг, но больше ни для кого:

```
---------------------------------------- ---------------------------
| first              | last              | workTel             |
=============================================
| "Craig"            | "Ellis"           | "(245) 315-5486" |
---------------------------------------------------------------------
```

Это происходит потому, что тройки работают вместе как единое целое. Этот шаблон графа запрашивает кого-то, кто имеет значение ab:firstName, ab:lastName и ab:workTel одновременно, и Крейг единственный ресурс отвечающий всем этим условиям. Ввод триплета о рабочем телефоне в OPTIONAL шаблоне графа (помните, шаблон графа - это один или несколько триплетов внутри фигурных скобок) позволяет в запросе сказать, "покажи мне это значение, если оно есть":

```
# filename: ex057.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?first ?last ?workTel
WHERE
{
  ?s ab:firstName ?first ;
     ab:lastName ?last .
OPTIONAL
 { ?s ab:workTel ?workTel . }
}
```

Когда мы выполним этот запрос с теми же данными ex054.ttl, результат будет включать в себя имена и фамилии каждого и рабочий номер телефона Крейга:

```
---------------------------------------- ---------------------------
| first              | last              | workTel             |
=============================================
| "Craig"            | "Ellis"           | "(245) 315-5486" |
 | "Cindy"           | "Marshall"        |                     |
| "Richard"          | "Mutt"            |                     |
---------------------------------------------------------------------
```

Ричард имеет прозвище, которое хранится как значение ab:nick и никто больше не имеет прозвища. Что произойдет, если мы добавим ab:nick внутри OPTIONAL?

```
# filename: ex059.rq
 PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?first ?last ?workTel ?nick
WHERE
{
    ?s ab:firstName ?first ;
       ab:lastName ?last .
  OPTIONAL
  {
     ?s ab:workTel  ?workTel ;
        ab:nick       ?nick .
   }
}
```

Мы получаем имена и фамилии каждого, но ни прозвища ни рабочего телефона не получаем, хотя у Ричарда есть прозвище, а у Крейга рабочий телефон:

```
-------------------------------------------------------------------
| first            | last             | workTel  | nick   |
===================================================
| "Craig"          | "Ellis"          |          |        |
|"Cindy"           | "Marshall"       |          |        |
|"Richard"         | "Mutt"           |          |        |
-------------------------------------------------------------------
```

Поскольку OPTIONAL определяет граф требующий наличие как прозвища так и номера рабочего телефона одновременно, запрос не нашел подобного ресурса. Если мы сделаем OPTIONAL отдельно для прозвища и для номера рабочего телефона, то процессор запросов будет смотреть на них отдельно и мы получим желаемый результат:

```
# filename: ex061.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?first ?last ?workTel ?nick
WHERE
{
   ?s  ab:firstName  ?first ;
       ab:lastName   ?last .
OPTIONAL { ?s ab:workTel ?workTel . }
OPTIONAL { ?s ab:nick ?nick . }
}
```

```
-----------------------------------------------------------------------------------
| first            | last             | workTel                | nick      |
=====================================================
| "Craig"          | "Ellis"          | "(245) 315-5486"       |           |
|"Cindy"           | "Marshall"       |                        |           |
|"Richard"         | "Mutt"           |                        |  "Dick"   |
-----------------------------------------------------------------------------------
```

Процессор запросов пытается сопоставить триплеты из OPTIONAL графа шаблонов в том порядке, в котором они расположены, что позволяет нам выполнять некоторые трюки. Например, скажем, мы хотим, получить все имена и фамилии, но мы

предпочитаем использовать прозвище, если оно есть, в этом случае мы не хотим получить прозвище вместо имени. В нашем случае мы хотим получить "Дик", а не "Ричард" вместо имени г-на Mutt. Следующий запрос делает это: при первом анализе субъектов со значением ab:lastName проверяет, есть ли OPTIONAL значение ab:nick. Если он находит его, то связывает его с первой переменной ?first. Если нет, то будет связывать с ?first значение ab:firstName:

```
# filename: ex063.rq
 PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?first ?last
WHERE
{
   ?s ab:lastName ?last .
   OPTIONAL { ?s ab:nick ?first . }
   OPTIONAL { ?s ab:firstName ?first . }
}
```

Для двух других людей в адресной книге, ничего не происходит с этим первым OPTIONAL шаблоном, потому что они не имеют значения ab:nick, так что с переменной ?first связывается значение ab:firstName:

```
-------------------------------------------
| first            | last             |
===========================
| "Craig"          | "Ellis"          |
|"Cindy"           | "Marshall"       |
|"Dick"            | "Mutt"           |
-------------------------------------------
```

Ключевое слово OPTIONAL особенно полезно для изучения нового набора данных, с которым вы мало знакомы. Например, если вы видите определенные значения свойств определенных ресурсов, помните, что не все ресурсы этого типа могут иметь эти свойства. Ввод образцов триплетов внутри OPTIONAL позволяет извлекать значения, если они там, не мешая извлечению связанных данных, если эти значения не существуют.

3.3. Поиск данных, которые не отвечают определенным условиям

Теперь должно быть довольно ясно, как получить имя, фамилию и рабочий номер телефона каждого из адресной книги набора данных ex054.ttl, кто имеет рабочий номер телефона - просто запросите ab:firstName, ab:lastName и ab:workTel, как мы сделали это в запросе ex055.rq в разделе "Данные, которых там может не быть":

```
# filename: ex055.rq
 PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?first ?last ?workTel
WHERE
{
?s ab:firstName ?first ;
    ab:lastName ?last ;
    ab:workTel ?workTel .
}
```

Что делать, если вы хотите, чтобы получить всех, у кого нет рабочего номера телефона? Мы только что видели, как получить имена каждого и их рабочий телефон, если они его имеют. SPARQL 1.0 способ получить всех, у кого рабочий номер телефона отсутствует строит на этом. SPARQL 1.1 предоставляет два варианта, оба из которых являются простыми.

В первом примере, который демонстрирует единственный способ сделать это в SPARQL 1.0, мы запрашиваем именами и фамилиями каждого человека и номер рабочего телефона, если они есть, но вдобавок используем фильтр, который применяется к каждой извлекаемой тройки:

```
# filename: ex065.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?first ?last
WHERE
{
 ?s ab:firstName ?first ;
    ab:lastName ?last .
 OPTIONAL { ?s ab:workTel ?workNum . }
 FILTER (!bound(?workNum))
}
```

В разделе "Использование меток, предусмотренных DBpedia" мы использовали FILTER для получения содержимого меток только с конкретными языковыми тегами. Выше запрос использует логическую функцию bound(), чтобы решить, какие тройки пропустить. Эта функция возвращает истину, если переменной было присвоено значение и ложь, в противном случае.

Как и в нескольких других языках программирования, восклицательный знак означает оператор "не". Поэтому, ex065.rq запрос из ex054.ttl набора данных будет выбирать имена и фамилии всех, кто не имеют ab:workTel:

```
-------------------------------------------
| first             | last              |
===========================
| "Cindy"           | "Marshall"        |
| "Richard"         | "Mutt"            |
-------------------------------------------
```

Обе SPARQL 1.1 альтернативы !bound() являются более интуитивными. Первая FILTER NOT EXISTS является фильтром, который возвращает логическое верно, если указанный триплет не существует. Если запрос находит значения объектов ab:firstName и ab:lastName, то они попадут в результат только тогда, когда для соответствующего им субъекта не существует предиката ab:workTel :

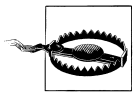
```
# filename: ex067.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?first ?last
WHERE
{
   ?s   ab:firstName   ?first ;
        ab:lastName    ?last .
  FILTER NOT EXISTS { ?s ab:workTel ?workNum }
}
```

Другой SPARQL 1.1 способ найти имена и фамилии людей, не имеющих

ab:workTel - использовать ключевое слово MINUS:

```
# filename: ex068.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?first ?last
WHERE
{
  ?s    ab:firstName    ?first ;
        ab:lastName     ?last .
  MINUS { ?s ab:workTel ?workNum }
}
```

Для наших целей, эти два SPARQL 1.1 подхода дают те же результаты, что и SPARQL 1.0 !bound() и в большинстве случаев они ведут себя одинаково.



Есть некоторые случаи, когда FILTER NOT EXISTS и MINUS могут возвращать разные результаты. Смотрите рекомендации SPARQL 1.1 в разделе «Отношения и различия между NOT EXISTS и MINUS" для уточнения этого вопроса.

3.4. Дальнейший поиск данных

Все данные, которые мы использовали до сих пор могли бы легко разместится в простой таблице. Это следующая версия наших данных добавляет новые данные. Если бы их требовалось хранить в реляционной базе данных, то потребовалось бы еще две таблицы: одна о читаемых курсах, а другая о тех, кто какие курсы посещал:

```
# filename: ex069.ttl
@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d: <http://learningsparql.com/ns/data#> .

# Люди
        d:i0432 ab:firstName    "Richard" .
        d:i0432 ab:lastName     "Mutt" .
        d:i0432 ab:homeTel      "(229) 276-5135" .
        d:i0432 ab:nick         "Dick" .
        d:i0432 ab:email        "richard49@hotmail.com" .

        d:i9771 ab:firstName    "Cindy" .
        d:i9771 ab:lastName     "Marshall" .
        d:i9771 ab:homeTel      "(245) 646-5488" .
        d:i9771 ab:email        "cindym@gmail.com" .

        d:i8301 ab:firstName    "Craig" .
        d:i8301 ab:lastName     "Ellis" .
        d:i8301 ab:workTel      "(245) 315-5486" .
        d:i8301 ab:email        "craigellis@yahoo.com" .
        d:i8301 ab:email        "c.ellis@usairwaysgroup.com" .

# Читаемые курсы
        d:course34 ab:courseTitle "Modeling Data with OWL" .
        d:course71 ab:courseTitle "Enhancing Websites with RDFa" .
        d:course59 ab:courseTitle "Using SPARQL with non-RDF Data" .
        d:course85 ab:courseTitle "Updating Data with SPARQL" .
```

 # Кто, какие курсы посещал

d:i8301 ab:takingCourse d:course59 .
d:i9771 ab:takingCourse d:course34 .
d:i0432 ab:takingCourse d:course85 .
d:i0432 ab:takingCourse d:course59 .
d:i9771 ab:takingCourse d:course59 .

В реляционной базе данных, каждой строке каждой таблицы должен быть присвоен уникальный идентификатор, чтобы можно было делать перекрестные ссылки между таблицами, чтобы например узнать, кто какие курсы посещал. Данные RDF построены аналогично: субъект каждой тройки является уникальным идентификатором для этого ресурса. SPARQL версия того, что разработчики реляционных баз данных называют join (присоединить) или сочетание данных из нескольких таблиц, реализует очень просто:

```
# filename: ex070.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?last ?first ?courseName
WHERE
{
  ?s   ab:firstName      ?first ;
       ab:lastName      ?last ;
       ab:takingCourse ?course .
?course   ab:courseTitle   ?courseName .
}
```

Этот запрос не использует новых SPARQL возможностей, которые мы раньше не видели. Один из методов, которые мы впервые увидели в ex048.rq, когда мы искали названия альбомов, производимых Timbaland, это использование той же переменной в положении объекта в одной тройки и субъектной позиции другой. В ex070.rq, когда процессор запросов ищет курс, который студент посещает, он ассоциирует курс по URI для переменной ?course, а затем ищет значение ab:courseTitle для этого URI и назначает его переменной ?courseName. Это очень распространенный способ связать различные наборы данных со SPARQL.

Объект из RDF троек может быть литералом или URI, и литерал может быть проще для чтения, но URI позволяет связать эти данные с другими данными. Последние несколько троек в ex069.ttl данных имеют свои идентификаторы URI, и запрос ex070.rq использует их для получения более читаемых значений ab:courseTitle для результата. При создании данных RDF, вы всегда можете дать URI значение rdfs:label (или что-то более специализированное, как ab:courseTitle, используемые выше) для использования в запросах, которые хотят более читабельного представления этого ресурса.

Как и в реляционной базе данных, информация о том, кто какие курсы посещал предоставляется на основе идентификаторов и запрос ex070.rq связывает эти идентификаторы с более читаемыми метками, показывая нам, что Синди и Ричард посещали по два курса, а Крейг только один:

```
-------------------------------------------------------------------------
| last          | first         | courseName                            |
=========================================================================
| "Ellis"       | "Craig"       | "Using SPARQL with non-RDF Data"      |
```

```
| "Marshall" | "Cindy"    | "Using SPARQL with non-RDF Data" |
| "Marshall" | "Cindy"    | "Modeling Data with OWL"         |
| "Mutt"     | "Richard" | "Using SPARQL with non-RDF Data" |
| "Mutt"     | "Richard" | "Updating Data with SPARQL"      |
-------------------------------------------------------------------------------
```

Если разместить ex069.ttl набор данных о людях, доступных курсах, и кто какие курсы посещал в трех различных файлах вместо одного, то что надо будет поменять в запросе ex070.rq? Абсолютно ничего. Если эти данные разместить в файлы с именами ex072.ttl, ex073.ttl и ex368.ttl, то командная строка называемая ARQ будет немного отличаться, потому что в ней надо будет указать три файла данных, но сам запрос не потребует никаких изменений вообще:

```
arq --query ex070.rq --data ex072.ttl --data ex073.ttl --data ex368.ttl
```

Данные запрашиваемые в этом примере являются небольшим искусственным примером различных наборов данных, которые нужно связать. Типичная реляционная база данных будет иметь две или более таблиц из сотен или даже тысяч строк, которые вы могли бы связать с помощью SQL-запроса, но эти таблицы должны быть определены вместе как часть одной и той же базе данных. Мощь RDF и SPARQL позволяет сделать это с наборами данных из разных мест, которые формировались разными людьми, благодаря наличию URI ресурсов одного набора данных, которые можно связать с URI ресурсов другого набора. Вот почему так ценно использовать существующие идентификаторы URI, чтобы представлять людей, места, вещи и отношения между данными, ибо это облегчает подключение одних данных к другим. (Если данные не выстраиваются красиво, можно использовать несколько вызовов функций так, чтобы преобразовать некоторые значения и выстроить их лучше.)

Еще один способ создать запрос SPARQL, который позволяет еще дальше анализировать пути отношений между данными, состоит в простом добавлении немного предикатной части в триплеты.

Важно

Пути отношений являются новыми для SPARQL 1.1.

Проще всего видеть мощь путей отношений на некоторых примерах. Для выборки данных представьте, что Ричард Матт опубликовал статью (назовем ее paper A) в престижном академическом журнале. Синди Маршалл позже опубликовал paper B, в которой ссылается на статью paper A Ричарда. Крейг Эллис также ссылается на статью paper A в своей классической работе paper C. Со временем, другие писали статьи со ссылками на Ричарда, Синди и Крейг, образуя рисунок цитирования, показаннный на рисунке 3-3.

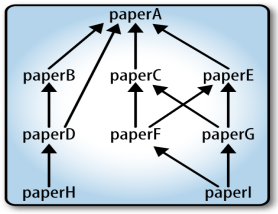


Рисунок 3-3. Взаимосвязь цитируемых статей из ex074.ttl данных

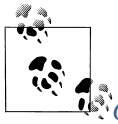
Ex074.ttl набор данных содержит данные о том какая статья цитируется в другой статье. Мы увидим, как пути отношений позволяют находить интересные вещи о моделях цитирования с помощью очень коротких запросов. Имена авторов опущены, для краткости:

```
# filename: ex074.ttl
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix c: <http://learningsparql.com/ns/citations#> .
@prefix : <http://learningsparql.com/ns/papers#> .

:paperA  dc:title "Paper A" .
:paperB  rdfs:label "Paper B" ;
         c:cites    :paperA .
:paperC  c:cites    :paperA .
:paperD  c:cites    :paperA , :paperB .
:paperE  c:cites    :paperA .
:paperF  c:cites    :paperC , :paperE .
:paperG c:cites    :paperC , :paperE .
:paperH c:cites    :paperD .
:paperI  c:cites    :paperF , :paperG .
```

Помните, что в Turtle и SPARQL, запятая означает "триплет имеет тот же субъект и предикат", так в ex074.ttl, первая запятая означает, что тройка после {:paperD c:cites :paperA} есть {:paperD c:cites :paperB}.

Для начала отметим, что название: paperA выражается как значение dc:title, а paperB как значение rdfs:label. Возможно, это результат слияния данных из двух источников, которые использовали разные стили для идентификации названий статей. Это не является проблемой, простой запрос может связать эти два различных обозначения с одной переменной:

```
# filename: ex075.rq
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX : http://learningsparql.com/ns/papers#

SELECT ?s ?title
WHERE { ?s (dc:title | rdfs:label) ?title . }
```



Скобки в ex075.rq запросе не влияют на поведение запроса, но делают его немного легче для восприятия.

Результат выполнения этого запроса показывает, как такой краткий запрос может сделать что-то, что потребовало бы более сложный запрос, с использованием ключевого слова UNION (о нем мы узнаем далее в разделе "Комбинирование различных условия поиска") в SPARQL 1.0. Это также демонстрирует хорошую стратегию для работы с данными, у которых имена свойств не выглядят так ясно, как хотелось бы.

```
-----------------------------
| s          | title        |
===========================
| :paperB | "Paper B" |
| :paperA | "Paper A" |
-----------------------------
```

С тем, что мы уже знаем о SPARQL достаточно легко запросить, какие статьи цитируют paperA. Следующий запрос делает это, и показывают нам, что такими статьями являются B, C, D и E:

```
# filename: ex077.rq
PREFIX : <http://learningsparql.com/ns/papers#>
PREFIX c: <http://learningsparql.com/ns/citations#>
SELECT ?s
WHERE { ?s c:cites :paperA . }
```

Использование нескольких символов, которые могут быть знакомы с синтаксиса регулярных выражений в языках программирования и утилитах, таких как Perl и Grep (например, символ | в запросе ex075.rq) позволяют говорить такие вещи, как "и продолжать искать далее." Простое добавление + к последнему запросу сообщает процессору запросов, чтобы искать статьи, которые ссылаются на paperA и статьи, которые ссылаются на те статьи и т.д. пока не достигнем статью со ссылкой на это дерево статей:

```
# filename: ex078.rq
PREFIX : <http://learningsparql.com/ns/papers#>
PREFIX c: <http://learningsparql.com/ns/citations#>
SELECT ?s
WHERE { ?s c:cites+ :paperA . }
```

Результат выполнения этого запроса показывает нам, какой влиятельной была статья paperA:

```
----------------
| s          |
==========
| :paperE  |
| :paperG  |
| :paperI  |
| :paperF  |
| :paperD  |
| :paperH  |
| :paperC  |
| :paperB  |
 ---------------
```

Как и в регулярных выражениях, плюс означает "один или больше". Можно использовать и звездочку, которая будет означать "ноль или более". Можно также быть гораздо более конкретным, используя путь отношений, чтобы запросить статьи, которые например находятся на расстоянии трех ссылок от цитируемой статьи paperA:

```
# filename: ex082.rq
PREFIX : <http://learningsparql.com/ns/papers#>
PREFIX c: <http://learningsparql.com/ns/citations#>
SELECT ?s
 WHERE { ?s c:cites/c:cites/c:cites :paperA . }
```

Это показывает, почему пути отношений называются путями: вы можете использовать их, чтобы выложить ряд шагов, разделенных косыми чертами, аналогично тому, как выражение XPath, XML или путь к каталогу в Linux или Windows.

Результат имеет повторы, потому что есть четыре способа, чтобы найти три ссылки c:cites образующие путь от статьи paperI к статье paperA:

```
---------------
| s          |
===========
| :paperI   |
| :paperI   |
| :paperI   |
| :paperI   |
| :paperH |
---------------
```

Обратные пути отношений позволяют перевернуть отношения, описанные предикатом тройки. Например, следующий запрос делает то же самое, что и запрос ex077.rq -перечисляет статьи, которые цитируют статью paperA:

```
# filename: ex083.rq
PREFIX : <http://learningsparql.com/ns/papers#>
PREFIX c: <http://learningsparql.com/ns/citations#>
SELECT ?s
WHERE {  :paperA   ^c:cites    ?s }
```

В отличие от ex077.rq, этот запрос ставит: paperA в субъектной позиции тройки, а переменную ?s в позиции объекта. Он по-прежнему ищет статьи, которые ссылаются на paperA, а не наоборот, потому что оператор ^ в начале c:cites сообщает процессору запросов, что мы хотим получить обратное отношение.

Пока оператор инверсии отношения позволил нам сделать что-то, что мы могли бы сделать и раньше, но с реквизитами, указанными в другом порядке. Можно также использовать SPARQL операторы на отдельных участках пути, и вот тогда оператор ^ позволяет сделать некоторые новые интересные вещи.

Например, следующий запрос запрашивает все статьи, которые ссылаются на статьи, цитируемые в статье paperF:

```
# filename: ex084.rq
PREFIX : <http://learningsparql.com/ns/papers#>
PREFIX c: <http://learningsparql.com/ns/citations#>
SELECT ?s
WHERE
{
  ?s   c:cites/^c:cites   :paperF .
  FILTER(?s != :paperF)
}
```

FILTER удаляет :paperF из результатов, потому что мы не должны говорит что :paperF является одной из статей, которая цитирует статьи, цитируемые в :PaperF. Ответ - статья paperG, становится понятен из схемы на рисунке 3-3.

Сеть статейных ссылок является одним из аспектов использования путей отношений. Когда речь идет о социальных сетях или компьютерных сетях или каких-либо других сетях, можно увидеть, что пути отношений могут дать некоторые интересные новые способы построения запросов определенной информации ко всем видам наборов данных в этих сетях.

3.5. Поиск в пустых узлах

Хотя пустые узлы не имеют постоянного имени, мы можем использовать их группируя вместе с другими значениями. Например, в следующем наборе данных, лицо, представленное префиксом имени ab:i0432 имеет значение адреса _: b1. Подчеркивание говорит нам, что часть после двоеточия является незначащим; важно то, что тот же ресурс имеет ab:PostalCode значение "49345", ab:city значение “Springfield”, ab:streetAddress значение “32 Main St.” и ab:region значение “Connecticut”. Другими словами, адрес Ричарда имеет следующие значения:

```
# filename: ex041.ttl
@prefix ab: <http://learningsparql.com/ns/addressbook#> .
ab:i0432 ab:firstName     "Richard" ;
         ab:lastName      "Mutt" ;
         ab:homeTel       "(229) 276-5135" ;
         ab:email         "richard49@hotmail.com" ;
         ab:address       _:b1 .

_:b1     ab:postalCode    "49345" ;
         ab:city          "Springfield" ;
         ab:streetAddress "32 Main St." ;
         ab:region        "Connecticut" .
```

Этот простой запрос спрашивает о значениях ab:address:

```
# filename: ex086.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?addressVal
WHERE { ?s  ab:address  ?addressVal }
```

Результат подчеркивает ключевую вещь, которую надо помнить о пустых узлах: что любое имя, присвоенное пустому узлу - временно, и что процессор беспокоиться о временном имени тех пор, пока помнит, какие тройки к нему относятся. В противном случае он формирует на выходе другое название (_: b0) в отличие от того, какое он имеет на входе (_: b1):

```
----------------------
| addressVal     |
==============
| _:b0           |
----------------------
```

В реальном запросе, который ссылается на пустые узлы, нужно использовать переменные для обращения к интересующим значениям размещенным в пустом узле. Эти переменные должны быть представлены в перечне оператора SELECT. Например:

```
# filename: ex088.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?firstName ?lastName ?streetAddress ?city ?region ?postalCode
WHERE
{ ?s      ab:firstName  ?firstName ;
          ab:lastName  ?lastName ;
          ab:address    ?address .

?address ab:postalCode ?postalCode ;
           ab:city ?city ;
           ab:streetAddress ?streetAddress ;
           ab:region ?region .
}
```

По сравнению с запросами, которые не ссылаются на пустые узлы, этот запрос не имеет ничего необычного, и дает идентичный результат при запуске с набором данных ex041.ttl:

```
-------------------------------------------------------------------------------------------------------
| firstName | lastName | streetAddress | city          | region          | postalCode |
=======================================================================
| "Richard"  | "Mutt"     | "32 Main St." | "Springfield" | "Connecticut" | "49345"     |
-------------------------------------------------------------------------------------------------------
```

Подводя итог, можно использовать переменную для ссылки на пустые узлы в данных RDF, аналогично использованию других узлов для нахождения связей между другими узлами.

3.6. Устранение избыточности

Как это происходит в SQL, ключевое слово DISTINCT позволяет указать процессору SPARQL "не показать совпадающие ответы". Например, следующий запрос, без ключевого слова DISTINCT, покажет предикат каждой тройки в наборе данных:

```
# filename: ex090.rq
SELECT ?p
WHERE
{ ?s ?p ?o . }
```

Вот результат при его запуске с набором данных ex069.ttl:

```
-----------------------------------------------------------------------------------
| p                                                                               |
=====================================================
| <http://learningsparql.com/ns/addressbook#courseTitle>       |
| <http://learningsparql.com/ns/addressbook#courseTitle>       |
| <http://learningsparql.com/ns/addressbook#courseTitle>       |
| <http://learningsparql.com/ns/addressbook#takingCourse>      |
| <http://learningsparql.com/ns/addressbook#email>             |
| <http://learningsparql.com/ns/addressbook#lastName>          |
| <http://learningsparql.com/ns/addressbook#firstName>         |
| <http://learningsparql.com/ns/addressbook#takingCourse>      |
| <http://learningsparql.com/ns/addressbook#takingCourse>      |
```

```
| <http://learningsparql.com/ns/addressbook#email>              |
| <http://learningsparql.com/ns/addressbook#lastName>           |
| <http://learningsparql.com/ns/addressbook#firstName>          |
| <http://learningsparql.com/ns/addressbook#takingCourse>       |
| <http://learningsparql.com/ns/addressbook#takingCourse>       |
| <http://learningsparql.com/ns/addressbook#email>              |
| <http://learningsparql.com/ns/addressbook#lastName>           |
| <http://learningsparql.com/ns/addressbook#firstName>          |
| <http://learningsparql.com/ns/addressbook#courseTitle>        |
-------------------------------------------------------------------------------
```

Это не очень полезный запрос, особенно для набора данных, в котором больше, чем несколько троек, потому что на выходе получаем как правило большое число беспорядочных данных. С ключевым словом DISTINCT, это очень полезный запрос, потому что он покажет, какие данные содержатся в исследуемом наборе данных:

```
# filename: ex092.rq
SELECT  DISTINCT ?p
WHERE
{ ?s ?p ?o . }
```

Запуск ex092.rq с тем же набором данных дает нам следующий результат:

```
-------------------------------------------------------------------------------
| p                                                                           |
===============================================================================
| <http://www.w3.org/2000/01/rdf-schema#label>                                |
| <http://learningsparql.com/ns/addressbook#takingCourse>                     |
| <http://learningsparql.com/ns/addressbook#email>                            |
| <http://learningsparql.com/ns/addressbook#lastName>                         |
| <http://learningsparql.com/ns/addressbook#firstName>                        |
-------------------------------------------------------------------------------
```

Результат этого запроса выглядит как небольшая упрощенная схема, потому что он показывает, какие данные отслеживаются в этом наборе данных. Это может помочь последующим запросам в изучении набора данных. Набор данных ex069.ttl не большой, значение ключевого слова DISTINCT становится яснее с набором данных большого размера.

Как можно запросить тот же набор данных о том, какой сотрудник какой курс посетил? Без ключевого слова DISTINCT, следующий запрос будет перечислить имя и фамилию сотрудников для каждой тройки с ab:takingCourse:

```
# filename: ex094.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT DISTINCT ?first ?last
WHERE
{ ?s   ab:takingCourse   ?class ;
       ab:firstName      ?first ;
       ab:lastName       ?last .
}
```

Две из этих троек имеют d:i9771 в качестве субъекта, а две имеют d:i0432, так что без ключевого слова DISTINCT в запросе, Синди Маршалл и Ричард Матт был бы в списке дважды. С ключевым словом DISTINCT, запрос выдает имя каждого человека, посетившего курс без каких-либо повторов:

```
-------------------------------
| first       | last          |
===================
| "Craig"     | "Ellis"       |
| "Cindy"     | "Marshall"    |
| "Richard"   | "Mutt"        |
-------------------------------
```

Ключевое слово DISTINCT не добавляет сложности в структуру запроса. Часто приходится писать запрос, даже не думая об этом ключевом слове и если в результате получаются повторяющиеся значения, то добавляется DISTINCT после ключевого слова SELECT, чтобы получить более удачные результаты.

3.7. Комбинирование различных условий поиска

Ключевое слово UNION позволяет указать несколько различных шаблонов графа, а затем запросить сочетание всех данных, которые соответствует любой из этих моделей. Сравните рисунок 3-4 с рисунком 1-1 в начале начале главы 1.

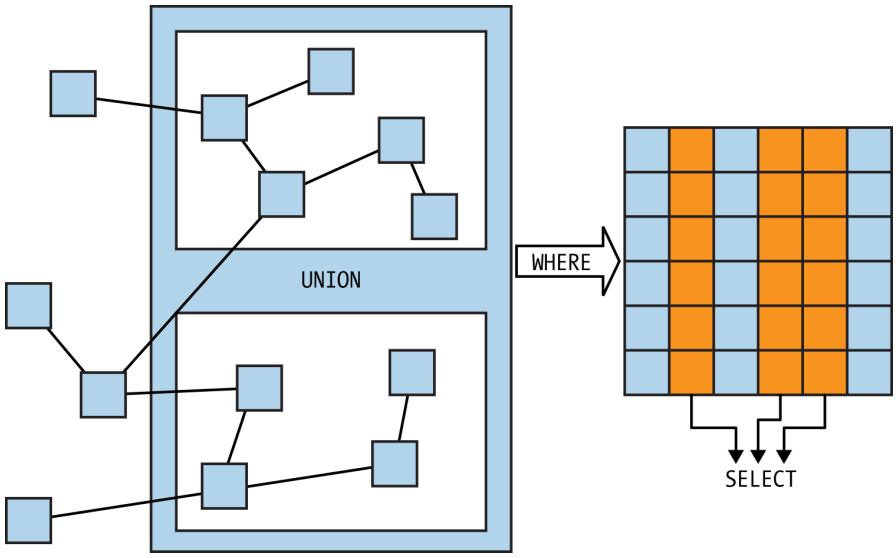


Рисунок 3-4. Ключевое слово UNION позволяет обрабатывать два набора данных

Например, для ex069.ttl данных, которые включают людей, курсы, и кто что посетил, мы могли бы объединить людей и курсы следующим запросом:

```
# filename: ex098.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX d: <http://learningsparql.com/ns/data#>
SELECT *
WHERE
```

```
{
{ ?person ab:firstName ?first ; ab:lastName ?last . }
UNION
{ ?course ab:courseTitle ?courseName . }
}
```

Результат имеет столбцы для каждой переменной, перечисленной в запросе:

```
-------------------------------------------------------------------------------------------------
| person    | first       | last          | course        | courseName                            |
|===============================================================================================|
| d:i8301   | "Craig"     | "Ellis"       |               |                                       |
| d:i9771   | "Cindy"     | "Marshall"    |               |                                       |
| d:i0432   | "Richard"   | "Mutt"        |               |                                       |
|           |             |               | d:course85    | "Updating Data with SPARQL"           |
|           |             |               | d:course59    | "Using SPARQL with non-RDF Data"      |
|           |             |               | d:course71    | "Enhancing Websites with RDFa"        |
|           |             |               | d:course34    | "Modeling Data with OWL"              |
-------------------------------------------------------------------------------------------------
```

Это не особенно полезный пример, но он показывает, как UNION может позволить в запросе связать два набора троек без указания какой-либо связи между множествами.

Почему ex098.rq запрос включает объявление d: префикса, который не используется? Как показывают результаты запроса, некоторые процессоры SPARQL используют префиксы в своих результатов, а не выписывают весь URI для каждого ресурса. Префиксы позволяют сделать данные проще для чтения, и как в этом случае, проще для размещения на странице.

Следующий пример немного более полезен, он использует UNION для того, чтобы получить два перекрывающихся набора данных. Представьте себе, что наша выборка хранит сведения о музыкальных инструментах, на которых играет каждый человек, и мы хотим извлечь имена, фамилии игроков и названия только духовых инструментов:

```
# filename: ex100.ttl
@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d: <http://learningsparql.com/ns/data#> .

d:i0432 ab:firstName    "Richard" ;
        ab:lastName     "Mutt" ;
        ab:instrument   "sax" ;
        ab:instrument   "clarinet" .

d:i9771 ab:firstName    "Cindy" ;
        ab:lastName     "Marshall" ;
        ab:instrument   "drums" .

d:i8301 ab:firstName    "Craig" ;
        ab:lastName     "Ellis" ;
        ab:instrument   "trumpet" .
```

Запрос ex101.rq имеет два графа шаблонов для извлечения имени, фамилии и названия инструмента для любых трубачей и той же информации для саксофонистов:

```
# filename: ex101.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?first ?last ?instrument
WHERE
{
  { ?person ab:firstName ?first ;
            ab:lastName ?last ;
            ab:instrument "trumpet" ;
            ab:instrument ?instrument .
  }
UNION
  { ?person ab:firstName ?first ;
            ab:lastName ?last ;
             ab:instrument "sax" ;
             ab:instrument ?instrument .
  }
}
```

Результаты показывают, кто играет на каких инструментах. Для каждого человека, который играет на саксофоне или трубе, в нем перечислены все инструменты на которых он играет, потому что это также соответствует последней тройке:

```
-----------------------------------------
| first          | last      | instrument |
=========================================
| "Craig"      | "Ellis" |  "trumpet" |
| "Richard" |  "Mutt" |  "clarinet" |
| "Richard" |  "Mutt" |  "sax"        |
-----------------------------------------
```

Этот запрос имеет некоторую ненужной избыточность. Мы можем исправить это с помощью ключевого слова UNION объединив мелкие детали графов и добавив их к совпадающей части обоих графов:

```
# filename: ex103.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?first ?last ?instrument
WHERE
{ ?person ab:firstName ?first ;
ab:lastName ?last ; ab:instrument ?instrument .
{ ?person ab:instrument "sax" . }
UNION
{ ?person ab:instrument "trumpet" . } }
```

Это пример используется ключевое слово UNION, чтобы объединить только два графика фразы, взятые, но вы можете связать, как многие, как вам нравится, повторяя ключевое слово перед каждым рисунком графа, что вы хотите, чтобы соединиться с другими.

3.8. Фильтрация данных в зависимости от условий

В главе 1 мы видели следующий запрос. Он имел самый гибкий шаблон из одного триплета, у которого во всех трех положениях переменные. Понятно, что ему будет соответствовать любая тройка из набора данных. Однако, этот запрос извлек только одну тройку, потому что выражение фильтра указывает, что мы хотели извлечь только тройки включающие строку "Yahoo" в объекте:

```
# filename: ex021.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT *
WHERE
{
  ?s ?p ?o .
  FILTER (regex(?o, "yahoo","i"))
}
```

FILTER имеет один аргумент. Здесь может быть любое выражение, которое возвращает логическое значение. Выражение выше - вызов функции regex(), о который мы узнаем в главе 5. В запросе ex021.rq проверяется, имеет ли объект ?o значение "yahoo" в качестве подстроки, при этом "i" (от “insensitive”) показывает нечувствительность к регистру.

Аргумент фильтра также может быть очень простым. Например, скажем есть некоторые данные отслеживания затрат и расположения нескольких продуктов:

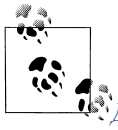
```
# filename: ex104.ttl
@prefix dm: <http://learningsparql.com/ns/demo#> .
@prefix d: <http://learningsparql.com/ns/data#> .

d:item432 dm:cost 8 ;
          dm:location <http://dbpedia.org/resource/Boston> .
d:item857 dm:cost 12 ;
          dm:location <http://dbpedia.org/resource/Montreal> .
d:item693 dm:cost 10 ;
          dm:location "Heidelberg" .
d:item126 dm:cost 5 ;
          dm:location <http://dbpedia.org/resource/Lisbon> .
```

Аргумент FILTER может не быть вызовом функции. Это может быть простое сравнение, которое также возвращает логическое значение:

```
# filename: ex105.rq
PREFIX dm: <http://learningsparql.com/ns/demo#>
SELECT ?s ?cost
WHERE
{
  ?s   dm:cost   ?cost .
  FILTER (?cost < 10)
}
```

Этот запрос выделяет продукты, которые стоят меньше, чем 10:

```
-------------------------------------------------------------------------
| s                                                          | cost |
=========================================================================
| <http://learningsparql.com/ns/data#item126>  | 5    |
| <http://learningsparql.com/ns/data#item432>  | 8    |
-------------------------------------------------------------------------
```



Два значения переменной ?s имеют полное URI, вместо того, чтобы использовать имена с префиксами, потому что процессор запросов не знает о префиксе d: во входных данных. Когда парсер RDF считывает входные данные, он отображает префиксы пространства имен соответствующих URI прежде, чем передать данные процессору запросов, и запрос не определить префикс для любого URI.

FILTER также полезно использовать для очистки данных. Известно, что объект тройки может быть строкой или URI. При этом, URI лучше потому, что вы можете использовать его, чтобы связать тройку с другими тройками. Данные о местоположении продукта, представленные выше имеют URI для большинства значений, но не все из них.

Следующий запрос выдает те тройки, объекты которых не URI. Функция isURI() возвращает логическое значение «истина», если ее аргумент является URI. Восклицательный знак действует как оператор "не", так что выражение !(isURI(?city)) будет возвращать логическое верно, если значение города не URI, а строка:

```
# filename: ex107.rq
PREFIX dm: <http://learningsparql.com/ns/demo#>
SELECT ?s ?city
WHERE
{
  ?s    dm:location    ?city .
  FILTER (!(isURI(?city)))
}
```

Результат показывает нам, где наши данные нуждаются в очистке:

```
-------------------------------------------------------------------------------------
| s                                                          | city                 |
 ====================================================================
| <http://learningsparql.com/ns/data#item693> | "Heidelberg" |
-------------------------------------------------------------------------------------
```

Как мы видели в разделе "Использование меток, предусмотренных DBpedia", фильтр также отлично подходит для получения значений на определенном языке из данных, которые хранят значения на разных языках.

Важно

Чем больше вы узнаете о функциях SPARQL, тем больше вы сможете сделать с помощью ключевого слова FILTER, поэтому новые функции, доступные в SPARQL 1.1 существенно расширят возможности использования фильтра.

Новые ключевое слово IN позволяет использовать нумерованный список, как часть запроса. Например, следующий запрос запрашивает данные, где значением местоположения является db:Montreal или db:Lisbon:

```
# filename: ex109.rq
PREFIX dm: <http://learningsparql.com/ns/demo#>
PREFIX db: <http://dbpedia.org/resource/>
SELECT ?s ?cost ?location
WHERE
{
  ?s   dm:location   ?location ;
       dm:cost       ?cost .
FILTER (?location IN (db:Montreal, db:Lisbon)) .
}
```

Этот список имеет только два значения внутри скобок после ключевого слова IN, но может иметь столько, сколько вам нравится. Вот результат работы ex109.rq с набором данных ex104.ttl:

```
-------------------------------------------------------------------------------------------
| s                                                    | cost | location       |
===========================================================
| <http://learningsparql.com/ns/data#item857> | 12   | db:Montreal |
| <http://learningsparql.com/ns/data#item126> | 5    | db:Lisbon     |
-------------------------------------------------------------------------------------------
```

Список может включать литералы в кавычках или любой другой тип данных. Например, вы могли бы запросить данные, используя значения стоимости:

```
# filename: ex111.rq
PREFIX dm: <http://learningsparql.com/ns/demo#>
PREFIX db: <http://dbpedia.org/resource/>
SELECT ?s ?cost ?location
WHERE
{
  ?s   dm:location   ?location ;
       dm:cost       ?cost .
  FILTER (?cost IN (8, 12, 10)) .
}
```

Вы также можете добавить ключевое слово NOT перед IN,

```
# filename: ex112.rq
PREFIX dm: <http://learningsparql.com/ns/demo#>
PREFIX db: <http://dbpedia.org/resource/>
SELECT ?s ?cost ?location
WHERE
{
  ?s   dm:location   ?location ;
       dm:cost       ?cost .
FILTER (?location NOT IN (db:Montreal, db:Lisbon)) .
}
```

и получить все данные, где ?location не в списке после ключевого слова IN:

```
-----------------------------------------------------------------------------------------
| s                                                    | cost | location        |
=========================================================================================
| <http://learningsparql.com/ns/data#item693> | 10   | "Heidelberg" |
| <http://learningsparql.com/ns/data#item432> | 8    | db:Boston      |
-----------------------------------------------------------------------------------------
```

3.9. Получение определенного количества результатов

Если вы послали следующий запрос DBpedia через форму запроса SNORQL, то вы получите слишком много данных в результате:

```
# filename: ex114.rq
SELECT ?label
WHERE
{ ?s   rdfs:label   ?label . }
```

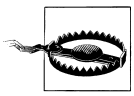
Это простенький запрос. Проблема не в том, что rdfs: префикс не объявлен в запросе, для запросов в DBpedia в форме SNORQL он уже предопределен. Однако, DBpedia имеет немало значений rdfs:label, точнее миллионы и этот запрос хочет все из них получить в результате. Скорее всего произойдет таймаут или вы получите ограниченное количество данных.

Вы можете настроить свой собственный предел на количество получаемых строк с помощью ключевого слова LIMIT. Рассмотрим следующий небольшой набор данных:

```
# filename: ex115.ttl
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

d:one   rdfs:label   "one" .
d:two   rdfs:label   "two" .
d:three rdfs:label   "three" .
d:four  rdfs:label   "four" .
d:five  rdfs:label   "five" .
d:six   rdfs:label   "six" .
```

Следующий запрос говорит процессору SPARQL, что независимо от того, сколько троек соответствует шаблону, мы хотим не более двух строк в результате:

```
# filename: ex116.rq
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?label
WHERE
{ ?s   rdfs:label   ?label . }
LIMIT 2
```



Обратите внимание, что ключевое слово LIMIT находится за пределами фигурных скобках, не внутри них.

Набор хранимых троек не упорядочен, поэтому этот запрос не обязательно извлекает первые две строки, которые вы видите на выборочных данных выше. Это могут

быть произвольные строки, например:

```
------------
| label  |
=======
| "six"   |
| "five" |
------------
```

Как мы увидим в "Сортировка, агрегирование, нахождение максимального и минимального ...", когда вы говорите процессору SPARQL сортировать данные, то LIMIT будет получать первые результаты отсортированных данных.

Ключевое слово OFFSET указывает процессору, что необходимо пропустить определенное количество результатов поиска прежде чем выбрать первые. Следующий запрос заставит его пропустить три из шести результатов:

```
# filename: ex118.rq
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?label
WHERE
{ ?s    rdfs:label    ?label . }
OFFSET 3
```

Это дает нам три результата:

```
--------------
| label     |
======= ==
| "three"  |
| "two"    |
| "one"    |
--------------
```

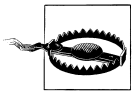
С помощью ключевого слова LIMIT, если вы не указываете процессору о сортировке результатов, то возвращаемые значения оказываются достаточно случайными. OFFSET фактически используется с LIMIT чтобы получать различные группы троек из большого собрания. Например, следующий запрос добавляет ключевое слово LIMIT к предыдущему запросу:

```
# filename: ex118.rq
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?label
WHERE
{ ?s    rdfs:label    ?label . }
OFFSET 3
LIMIT 1
```

При запуске запроса к тому же набору данных, мы получим только одно значение:

```
--------------
| label     |
```

```
======= ==
| "three"  |
--------------
```

 *При использовании LIMIT и OFFSET вместе, не следует ожидать согласованного результата, если вы не используете ORDER BY с ними.*

3.10. Запросы к поименованным графам

Набор данных может включать в себя наборы поименованных троек, чтобы упростить управление этими наборами. Например, множество троек пришло из конкретного источника в определенный день и должно быть заменено следующей партией из того же источника. Возможность обратиться к этому набору по имени делает это возможным. Мы узнаем больше о том, как делать такие замены в главе 6. Здесь мы узнаем, как использовать имена графа в запросах.

Во-первых, давайте рассмотрим то, что мы узнали в главе 1. Мы видели, что там вместо того, чтобы говорить процессору запросов, какие данные мы хотим запросить - отдельно от самого запроса (в команде ARQ) мы можем указать данные для запроса прямо в самом запросе с ключевым словом FROM. Используя это, ваш запрос может задать столько графов троек, сколько необходимо.

Например, скажем, у нас есть файл данных о новых студентах и его надо добавить в файл ex069.ttl, который мы использовали ранее. Этот файл включал данные о трех студентах, четырех курсах и о том кто какой курс прослушал. Новый файл содержит данные о еще двух студентах:

```
# filename: ex122.ttl
@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d: <http://learningsparql.com/ns/data#> .

d:i5433 ab:firstName    "Katherine" ;
        ab:lastName     "Duncan" ;
        ab:email        "katherine.duncan@elpaso.com" .

d:i2194 ab:firstName    "Bradley" ;
        ab:lastName     "Perry" ;
        ab:email        "bradley.perry@corning.com" .
```

Следующий запрос использует ключевое слово FROM, чтобы указать, что он хочет искать данные как в старом файле ex069.ttl так и в новом файле ex122.ttl:

```
# filename: ex123.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?email
FROM <ex069.ttl>
FROM <ex122.ttl>
WHERE
{ ?s   ab:email   ?email . }
```

Ex069.ttl набор данных имеет три адреса электронной почты и ex122.ttl имеет два, так что это запрос возвращает в общей сложности пять:

```
-----------------------------------------------
| email                                       |
 ==============================
| "bradley.perry@corning.com"      |
| "katherine.duncan@elpaso.com" |
| "c.ellis@usairwaysgroup.com"     |
| cindym@gmail.com                 |
| "richard49@hotmail.com"          |
-----------------------------------------------
```

Все наборы данных, которые вы укажете таким путем (а также указанные вне запроса в командной строке ARQ) суммируются, чтобы сформировать то, что называется граф по умолчанию. Это относится ко всем тройкам, доступным для запроса, которые не являются частью каких-либо поименованных графов. По умолчанию графы имеют единственный вид, который мы видели до сих пор в этой книге.

Если FROM это способ сказать "добавить тройки из следующего графа набора данных, который я собираюсь запросить", то FROM NAMED является способом сказать "добавить данные из этого конкретного графа, но не добавлять тройки из графа по умолчанию - когда я хочу данные из этого графа, я назову его по имени". Процессор SPARQL будет помнить о совместном использовании названных графов. Как мы увидим, мы даже можем запросить список пар графов.

Конечно, потому что мы говорим о RDF и SPARQL, именем поименованного графа является URI.

Возможность назначить эти имена есть важная особенность triplestores, потому что, поскольку вы можете добавлять, обновлять и удалять наборы данных, вы должны быть в состоянии идентифицировать наборы, которые вы хотите добавить, обновить и удалить. Если у вас есть файл RDF на диске в любом из популярных форматов сериализации RDF, там в настоящее время нет стандартного способа определить подмножество троек этого файла как принадлежность графа с определенном именем. Поэтому, чтобы продемонстрировать запросы к поименованным графам в этой главе мы воспользуемся конвенцией ARQ, где для локализации файлов в качестве имени каждого поименованного графа используется URI. В главе 6 мы увидим, стандартизованный SPARQL 1.1 способ создавать и удалять поименованные графы в triplestore, а также как добавлять и заменять в них тройки.

Ранее мы видели, что файл в ex122.ttl были данные о новых студентах. Вот ex125.ttl, который имеет данные о новых предлагаемых курсах:

```
# filename: ex125.ttl
@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d: <http://learningsparql.com/ns/data#> .

ab:course42     ab:courseTitle   "Combining Public and Private RDF Data" .
ab:course24     ab:courseTitle   "Using Named Graphs" .
```

В следующих примерах, имена поименованных графов являются относительными URI, поскольку когда ARQ видит <ex125.ttl>, он будет искать файл ex125.ttl в каталоге, где хранится файл запроса. Если хранить копию этого файла в каталоге примеров из www.learningsparql.com и ссылку <http://www.learningsparql.com/examples/ex125.ttl> как

имя поименованного графа в запросе, ARQ будет брать тройки из этого файла.

В разделе "Функции преобразования типов узлов" в главе 5 демонстрируется использование ключевого слова BASE, которое предоставляет больше контроля над тем, как процессор запросов разрешает относительные URI.

Следующий запрос загружает исходные данные ex069.ttl о студентах, курсах и о том кто какие курсы посещал в графе по умолчанию, а затем определяет файл ex125.ttl данных о новых курсах и файл ex122.ttl данных о новых студентах, как поименованные графы для ссылки в запросе:

```
# filename: ex126.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?lname ?courseName
 FROM <ex069.ttl>
 FROM NAMED <ex125.ttl>
 FROM NAMED <ex122.ttl>  #ненужный
WHERE
{
  { ?student   ab:lastName   ?lname }
   UNION
  { GRAPH <ex125.ttl> { ?course   ab:courseTitle   ?courseName } }
}
```

В дополнение к фразе FROM NAMED в запросе используется новое ключевое слово GRAPH, чтобы ссылаться на данные из поименованного графа. Вот результат выполнения этого запроса:

```
-------------------------------------------------------------------------
| lname       | courseName                                              |
=========================================================================
| "Ellis"     |                                                         |
|"Marshall" |                                                         |
|"Mutt"       |                                                         |
|               |"Using Named Graphs"                                   |
|               |"Combining Public and Private RDF Data"                |
-------------------------------------------------------------------------
```

Запрос использует союз двух наборов троек из указанных данных - троек из графа по умолчанию с ab:lastName как предикат и троек из ex125.ttl с ab:courseTitle как предикат, а потом выбирает значения ?LNAME и ?courseName из объединенного набора троек. Есть две важные вещи, которые следует отметить:

• Даже если файл ex069.ttl, включающий тройки графа по умолчанию имеет тройки соответствующие шаблону ?course ab:courseTitle ?courseName, они не попадут в результат, поскольку они должны браться только из поименованного графа <ex125.ttl>, а в графе по умолчанию они должны соответствовать шаблону ?student ab:lastName ?lname.

• Пусть поименованный граф ex122.ttl включает тройки, которые соответствуют шаблону ?student ab:last Name ?lname, но ни одна из них не появится на выходе, потому что его тройки не были добавлены в граф по умолчанию с помощью выражения FROM <ex122.ttl>. Выражение FROM NAMED <ex122.ttl> по сути говорит: "будем использовать

поименованный граф с именем <ex122.ttl>", но запрос не удосужился на самом деле использовать его, так как в запросе есть только GRAPH<ex125.ttl> часть , но нет GRAPH <ex122.ttl>. Вот почему комментарий описывает его как «ненужный». В запрос он просто добавлен, чтобы показать, что FROM NAMED не «поставляет» троек запросу, пока в нем реально не использовать поименованный граф.

В запросе ex126.rq, ключевое слово GRAPH указывает на поименованный граф, но мы можем использовать переменную, и пусть процессор SPARQL выбирает граф, который соответствуют шаблону. Следующий запрос проверяет все поименованные графы чтобы узнать о наличии в них троек, которые соответствуют шаблону ?s ?p ?o, и просит имена этих графов:

```
# filename: ex128.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?g
FROM NAMED <ex125.ttl>
FROM NAMED <ex122.ttl>
WHERE
{
  GRAPH ?g { ?s ?p ?o }
}
```

В список результата ex122.ttl входит шесть раз и дважды ex125.ttl, потому что процессор SPARQL помещает имя файла в результат для каждой найденной тройки, соответствующей образцу ?s ?p ?o:

```
------------------
| g              |
==================
| <ex122.ttl>    |
| <ex122.ttl>    |
| <ex122.ttl>    |
| <ex122.ttl>    |
| <ex122.ttl>    |
| <ex122.ttl>    |
| <ex125.ttl>    |
| <ex125.ttl>    |
------------------
```

Если добавить ключевое слово DISTINCT после SELECT в этом запросе, то каждый граф будет упомянут только один раз.

Запрос ex128.rq определяет только поименованные графы, без графов по умолчанию. Если бы имелся граф по умолчанию, ни одна из его троек не появилась бы в результате, потому что ключевое слово GRAPH означает, что тройки удовлетворяющие шаблону ищутся только в поименованных графах.

В более реалистичном примере использования переменной после ключевого слова GRAPH, мы можем использовать ее, чтобы сказать процессору SPARQL получить объединение всех значений ab:courseTitle графа по умолчанию, и из всех поименованных графов, фактически не указывая никаких имен графов после ключевого слова GRAPH:

```
# filename: ex130.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?courseName
FROM <ex069.ttl>
FROM NAMED <ex125.ttl>
FROM NAMED <ex122.ttl>
WHERE
{
  { ?course   ab:courseTitle ?  courseName }
  UNION
  {GRAPH ?g { ?course   ab:courseTitle ?  courseName } }
}
```

Этот запрос находит четыре ab:classTitle значения из ex069.ttl и два из ex125.ttl:

```
-----------------------------------------------------
| courseName                                        |
=====================================================
| "Updating Data with SPARQL"                       |
| "Using SPARQL with non-RDF Data"                  |
| "Enhancing Websites with RDFa"                    |
| "Modeling Data with OWL"                          |
| "Using Named Graphs"                              |
| "Combining Public and Private RDF Data"           |
 -----------------------------------------------------
```

Поскольку имена графа URI, вы можете использовать их в качестве либо субъекта, либо объекта троек, что делает вещи еще более интересным. Это означает, что имя графа может быть значением метаданных для чего-то. Например, вы можете сказать, что изображение в http://www.somepublisher.com/img/f43240.jpg имеет метаданные в triplestore репозитории метаданных владельца в поименованном графе http://www.somepublisher.com /ns/m43240 в тройке, подобной этой:

```
# filename: ex132.ttl
@prefix images: <http://www.somepublisher.com/img/> .
@prefix isi: <http://www.isi.edu/ikcap/Wingse/fileOntology.owl#> .

images:f43240.jpg    isi:hasMetadata    <http://www.somepublisher.com/ns/m43240> .
```

Поименованный граф http://www.somepublisher.com/ns/m43240 может иметь тройки, подобные этой:

```
# filename: ex133.ttl
@prefix images: <http://www.somepublisher.com/img/> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .

images:f43240.jpg    dc:creator "Richard Mutt" ;
                     dc:title "Fountain" ;
                     dc:format "jpeg" .
```

Чтобы найти подходящий предикат для этого примера, был взят словарный каталог Swoogle, в словаре http://www.isi.edu/ikcap/Wingse/fileOntology.owl нашлось свойство hasMetadata.

Названия графов могут также иметь свои собственные метаданные. Следующий набор данных включает в себя значения даты и создателя (date и creator в Dublin Core) для двух поименованных графов, используемых в приведенных выше примерах:

```
# filename: ex134.ttl
@prefix dc: <http://purl.org/dc/elements/1.1/> .
<ex125.ttl>     dc:date "2011-09-23" ;
                dc:creator "Richard Mutt" .
<ex122.ttl>     dc:date "2011-09-24" ;
                dc:creator "Richard Mutt" .
```

Следующий запрос запрашивает все адреса электронной почты из поименованного графа, который имеет значение dc:date "2011-09-24":

```
# filename: ex135.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?graph ?email
FROM <ex134.ttl>
FROM NAMED <ex125.ttl>
FROM NAMED <ex122.ttl>
WHERE
{
  ?graph dc:date "2011-09-24" .
  { GRAPH ?graph { ?s   ab:email   ?email } }
}
```

Ранее мы говорили, что запрос использует ключевое слово GRAPH для ссылки на данные из графа с определенным именем. Мы также говорили, что FROM NAMED идентифицирует поименованные графы, на которые можно ссылаться по ключевому слову GRAPH. Вы можете однако увидеть запросы SPARQL, в которых используется ключевое слово GRAPH для ссылки на поименованные графы, которые не были предварительно определены с помощью FROM NAMED. Как это может быть?

Некоторые процессоры SPARQL – в частности, те которые являются частью triplestore или SPARQL endpoints имеют предопределенные поменованные графы, которые не нуждаются, чтобы их определяли с помощью FROM NAMED, прежде чем ссылаться на них с помощью ключевого слова GRAPH.

Triplestore Сезам с открытым исходным кодом хранит графы по умолчанию и любые сопутствующие поименованные графы в своем хранилище. Если вы ссылаетесь на имя графа в этом хранилище с ключевым словом GRAPH, Сезам будет знать, о чем вы говорите, не требуя, чтобы этот граф быть определен сначала с помощью FROM NAMED. Другие triplestores и SPARQL endpoints могут делать это по-другому, потому что рекомендации W3C SPARQL не регламентируют того, что процессор SPARQL должен знать о внешних поименованных графах определенных с помощью FROM NAMED.

Вы можете спросить, о каком из файлов процессор SPARQL знает с помощью

простого запроса:

```
# filename: ex136.rq
SELECT DISTINCT ?g
WHERE
{
   GRAPH ?g {?s   ?p   ?o }
}
```

Иногда процессоры не дают никакого ответа на этот запрос, но все же знают о некоторых поименованных графах, упомянутых в других запросах. Так что не стоит слишком разочаровываться, если вы не видите результаты. Просто следует относиться к этому как еще одному хорошему общему запросу, который полезн при изучении нового процессора или источника данных SPARQL.

3.11. Запросы в запросах

Запросы внутри запросов известны как подзапросы. Эта возможность позволяет SPARQL преобразовать сложный запрос в более легко управляемые части, и это также позволяет объединить информацию из различных запросов в единый набор ответов.

Важно

Подзапросы являются новой возможностью в SPARQL 1.1.

Каждый подзапрос должен быть заключен в его собственный набор фигурных скобок. Следующий запрос, который вы можете проверить с файлом ex069.ttl данных, имеет один подзапрос для получения значений фамилий, а другой для получения значений название курса и ключевое слово SELECT в главном запросе извлекает значения для переменных ?lastName и ?courseName из подзапросов:

```
# filename: ex137.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?lastName ?courseName
WHERE
{
  {
   SELECT ?lastName
   WHERE { ?student ab:lastName ?lastName .
  }
  {
   SELECT ?courseName
   WHERE { ?course ab:courseTitle ?courseName . }
  }
}
```

Это довольно искусственный пример, потому что результат, который создает этот запрос (каждая возможная ?LastName - ?CourseName пара) не очень полезен. Мы увидим некоторые более продуктивные примеры подзапросов в следующем разделе.

3.12. Объединение значений и присвоение значений переменным

После того, как ваш запрос SPARQL получает значения из набора данных, он может использовать эти значения в выражениях, которые выполняют математические и функциональные вызовы. Это позволяет сделать ваши запросы еще больше эффективными для работы с данными.

Важно

Использование выражений для вычисления новых значений и переменных для их описания являются новой возможностью в SPARQL 1.1.

Для эксперимента с созданием выражений, мы будем использовать некоторые вымышленные данные отчета о расходах:

```
# filename: ex138.ttl
@prefix e: <http://learningsparql.com/ns/expenses#> .
@prefix d: <http://learningsparql.com/ns/data#> .

d:m40392        e:description "breakfast" ;
                e:date "2011-10-14T08:53" ;
                e:amount 6.53 .
d:m40393        e:description "lunch" ;
                e:date "2011-10-14T13:19" ;
                e:amount 11.13 .
d:m40394        e:description "dinner" ;
                e:date "2011-10-14T19:04" ;
                e:amount 28.30 .
```

Убрав кавычки от числовых значений мы указали процессору SPARQL рассматривать их как числа, а не строками. Глава 5 описывает это более подробно.

В следующем запросе WHERE просто определяет шаблон для значений переменных ?meal, ?description и ?amount, которому должны удовлетворять выбираемые тройки из набора данных ex138.ttl, а SELECT делает немного больше со значением переменной ?amount, чем просто отображать его:

```
# filename: ex139.rq
PREFIX e: <http://learningsparql.com/ns/expenses#>
SELECT ?description ?amount ((?amount * .2) AS ?tip)
       ((?amount + ?tip) AS ?total)
WHERE
{
  ?meal e:description ?description ;
  e:amount ?amount .
}
```

Будет легче понять, что делает этот запрос, если мы посмотрим сначала результат:

```
| description | amount | tip       | total     |
==============================
| "dinner"     |  28.30  |  5.660  |  33.960 |
| "lunch"      |  11.13  |  2.226   | 13.356 |
| "breakfast" |   6.53  |  1.306   |  7.836 |
-----------------------------------------------
```

В дополнение к отображению ?description и ?amount, пункт SELECT умножает стоимость ?amount на 0,2 и использует ключевое слово AS, чтобы сохранить результат в переменной ?tip, которую показывает, в качестве третьей колонки результатов поиска. Затем пункт SELECT суммирует значения стоимости ?amount и чаевых ?tip вместе и сохраняет результаты в переменной ?total (всего), отображая ее значения в четвертой колонке результатов запроса.

Как мы увидим в главе 5, SPARQL 1.0 поддерживает несколько функций для управления значениями данных, а SPARQL 1.1 добавляет еще много. Следующий запрос тех же данных ex138.ttl использует две функции манипуляции строками для описания расходов в виде трех первых заглавных букв от полных названий:

```
# filename: ex141.rq
PREFIX e: <http://learningsparql.com/ns/expenses#>
SELECT (UCASE(SUBSTR(?description,1,3))
as ?mealCode) ?amount
WHERE
{
  ?meal    e:description    ?description ;
           e:amount         ?amount .
}
```

Результат имеет один столбец для расчетной стоимости ?MealCode, а другой для значения ?amount:

```
-------------------------
| mealCode | amount |
================
| "DIN"      | 28.30  |
| "LUN"      | 11.13  |
| "BRE"      |  6.53  |
-------------------------
```

Выполнение сложных вычислений со многими переменными может сделать выражение в SELECT довольно длинным. Вот почему, когда мы формировали значение переменной ?total в ex139.rq, пришлось переехать на новую строку. Аналогично в запросе ex141.rq выражение в SELECT слишком громоздко. Следующий вариант этого запроса получает тот же результат с вычислением выражения в подзапросе. Это позволяет значительно упростить основной SELECT:

```
# filename: ex143.rq
PREFIX e: <http://learningsparql.com/ns/expenses#>
SELECT ?mealCode ?amount
WHERE
{
  {
  SELECT ?meal (UCASE(SUBSTR(?description,1,3)) as ?mealCode)
```

```
  WHERE { ?meal e:description ?description . }
  }
  {
   SELECT ?meal ?amount
   WHERE { ?meal e:amount ?amount . }
   }
}
```

Более распространенным способом перемещения выражениz из главного SELECT является использование ключевого слова BIND. Это наиболее краткая альтернатива поддержке стандарта SPARQL 1.1, потому что это позволяет присвоить значение выражения новой переменной в одной строке кода. Следующий запрос дает тот же результат, чтои в двух последних примерах:

```
# filename: ex144.rq
PREFIX e: <http://learningsparql.com/ns/expenses#>
SELECT ?mealCode ?amount
WHERE
{
  ?meal   e:description   ?description ;
          e:amount ?amount .
  BIND (UCASE(SUBSTR(?description,1,3)) as ?mealCode)
}
```

3.13. Создание таблицы значений в запросах

Ключевое слово VALUE позволяет вам создавать таблицы значений, предоставляя новые возможности при фильтрации результатов запроса.

Важно

Ключевое слово VALUE являются новой возможностью в SPARQL 1.1.

Следующий запрос игнорирует входные параметры и показывает, как можно создать таблицу значений. Этот пример заполняет таблицу префиксами имен и буквенными значениями, но вы можете использовать любые типы RDF значений:

```
# filename: ex492.rq
PREFIX dm: <http://learningsparql.com/ns/demo#>
SELECT *
WHERE { }
        VALUES (?color ?direction) {
        ( dm:red "north" )
        ( dm:blue "west" )
}
```

И вот результат:

```
| color     | direction |
==================
| dm:red   | "north"   |
| dm:blue | "west"    |
------------------------------
```

Этот результат не является особенно захватывающим, но он показывает, как просто создать двумерную таблицу в запросе SPARQL. Чтобы увидеть, какие значения можно добавить в запросы, мы будем использовать набор данных отчета о расходах, похожий на тот, который мы видели в предыдущем разделе, но на этот раз охватывающий три дня:

```
# filename: ex145.ttl
@prefix e: <http://learningsparql.com/ns/expenses#> .
@prefix d: <http://learningsparql.com/ns/data#> .
d:m40392        e:description "breakfast" ;
                e:date "2011-10-14" ;
                e:amount 6.53 .
d:m40393        e:description "lunch" ;
                e:date "2011-10-14" ;
                e:amount 11.13 .
d:m40394        e:description "dinner" ;
                e:date "2011-10-14" ;
                e:amount 28.30 .
d:m40395        e:description "breakfast" ;
                e:date "2011-10-15" ;
                e:amount 4.32 .
d:m40396        e:description "lunch" ;
                e:date "2011-10-15" ;
                e:amount 9.45 .
d:m40397        e:description "dinner" ;
                e:date "2011-10-15" ;
                e:amount 31.45 .
d:m40398        e:description "breakfast" ;
                e:date "2011-10-16" ;
                e:amount 6.65 .
d:m40399        e:description "lunch" ;
                e:date "2011-10-16" ;
                e:amount 10.00 .
d:m40400        e:description "dinner" ;
                e:date "2011-10-16" ;
                e:amount 25.05 .
```

Прежде чем мы используем ключевое слово VALUES, мы начнем с простого запроса, который запрашивает значения всех свойств набора данных без использования ключевого слова VALUES:

```
# filename: ex494.rq
PREFIX e: <http://learningsparql.com/ns/expenses#>
SELECT ?description ?date ?amount
WHERE
{
  ?meal   e:description   ?description ;
          e:date ?date ;
```

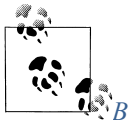
```
        e:amount ?amount .

}
```

При запуске этого запроса с приведенным выше набором данных, будут перечислены значения всех указанных в SELECT переменных:

```
-------------------------------------------------
| description | date           | amount |
 ================================
| "dinner"    |"2011-10-16" |25.05    |
| "lunch"     |"2011-10-16" |10.00    |
| "breakfast" |"2011-10-16" |  6.65   |
| "dinner"    |"2011-10-15" |31.45    |
| "lunch"     |"2011-10-15" |  9.45   |
| "breakfast" |"2011-10-15" |  4.32   |
| "dinner"    |"2011-10-14" |28.30    |
| "lunch"     |"2011-10-14" |11.13    |
| "breakfast" |"2011-10-14" | 6.53    |
-------------------------------------------------
```

Следующая версия запроса добавляет пункт VALUES говоря, что мы заинтересованы только в результатах, которые включают только "lunch" или "dinner" в описании значения ?description:

```
# filename: ex496.rq
PREFIX e: <http://learningsparql.com/ns/expenses#>
SELECT ?description ?date ?amount
WHERE
{
  ?meal   e:description   ?description ;
          e:date ?date ;
          e:amount ?amount .
  VALUES ?description { "lunch" "dinner" }
}
```



В этом случае создается одномерная структура данных VALUE, а не двумерная; это еще один шаг вверх от способности ключевого слова BIND, назначить только одно значение переменной одновременно.

С теми же данными расходов на еду, результат этого нового запроса похож на результат запроса приведенного выше только без "breakfast":

```
-------------------------------------------------
| description  | date           | amount |
=================================
| "lunch"      | "2011-10-16" | 10.00   |
| "lunch"      | "2011-10-15" | 9.45    |
| "lunch"      | "2011-10-14" | 11.13   |
|"dinner"      | "2011-10-16" |25.05    |
| "dinner"     | "2011-10-15" | 31.45   |
```

```
| "dinner"     | "2011-10-14" | 28.30    |
-----------------------------------------------------
```

Выражение VALUES этого запроса могло бы быть размещено после закрытия фигурной скобки и это не повлияло бы на результаты. Это не всегда будет так, когда мы будем использовать VALUES в GROUP BY и федеративных запросах.

Следующий запрос из тех же данных создает двумерную таблицу:

```
# filename: ex498.rq
PREFIX e: <http://learningsparql.com/ns/expenses#>
SELECT ?description ?date ?amount
WHERE
{
  ?meal    e:description ии?description ;
           e:date ?date ; e:amount ?amount .
  VALUES (?date ?description) {
           ("2011-10-15" "lunch")
           ("2011-10-16" "dinner")
   }
}
```

После получения всех данных о еде, этот запрос отбирает в результат только те данные, которые имеют либо значение ?date "2011-10-15" и ?description "lunch", либо значение ?data "2011-10-16 " и ?description "dinner":

```
-----------------------------------------------------
| description | date             | amount |
=================================
| "lunch"      | "2011-10-15"  |   9.45   |
| "dinner"     | "2011-10-16" | 25.05   |
-----------------------------------------------------
```

При использовании VALUES для создания таблицы данных, вы не должны присвваивать значение каждой позиции. Ключевое слово UNDEF действует в качестве джокера, принимая любое значение. В следующей вариации запрос просит все строки со значением "lunch" в качестве ?description, независимо от значения ?date, а также любые строки со значением ?date "2011-10-16", независимо от значения ?description:

```
# filename: ex500.rq
PREFIX e: <http://learningsparql.com/ns/expenses#>
SELECT ?description ?date ?amount
WHERE
{
   ?meal   e:description   ?description ;
           e:date          ?date ;
           e:amount        ?amount .
   VALUES (?date ?description) {
           (UNDEF "lunch")
           ("2011-10-16" UNDEF)
    }
}
```

Результат этого запроса имеет больше строк, чем в предыдущем запросе:

```
-------------------------------------------------
| description | date            | amount |
=================================
| "lunch"     | "2011-10-16" | 10.00  |
| "lunch"     | "2011-10-15" |  9.45  |
| "lunch"     | "2011-10-14" | 11.13  |
| "dinner"    | "2011-10-16" | 25.05  |
| "lunch"     | "2011-10-16" | 10.00  |
| "breakfast" | "2011-10-16" |  6.65  |
-------------------------------------------------
```

Когда вы работаете с большими объемами данных, и особенно с более сложными условиями фильтрации, ключевые слова VALUES обеспечивают дополнительный слой фильтрации результата, и могут дать вам больше возможностей управления вашими результатами поиска с очень небольшим дополнительным кодом в запросе.

3.14. Сортировка, агрегация, нахождения максимального и минимального ...

SPARQL позволяет сортировать данные и использовать встроенные функции, чтобы получить больше из анализируемых данных. Для экспериментов с этими функциями, мы будем использовать расширенную версию данных отчета о расходах, которые мы видели в предыдущем разделе:

```
# filename: ex145.ttl
@prefix e: <http://learningsparql.com/ns/expenses#> .
@prefix d: <http://learningsparql.com/ns/data#> .
d:m40392        e:description "breakfast" ;
                e:date "2011-10-14" ;
                e:amount 6.53 .
d:m40393        e:description "lunch" ;
                e:date "2011-10-14" ;
                e:amount 11.13 .
d:m40394        e:description "dinner" ;
                e:date "2011-10-14" ;
                e:amount 28.30 .
d:m40395        e:description "breakfast" ;
                e:date "2011-10-15" ;
                e:amount 4.32 .
d:m40396        e:description "lunch" ;
                e:date "2011-10-15" ;
                e:amount 9.45 .
d:m40397        e:description "dinner" ;
                e:date "2011-10-15" ;
                e:amount 31.45 .
d:m40398        e:description "breakfast" ;
                e:date "2011-10-16" ;
                e:amount 6.65 .
d:m40399        e:description "lunch" ;
                e:date "2011-10-16" ;
                e:amount 10.00 .
d:m40400        e:description "dinner" ;
                e:date "2011-10-16" ;
                e:amount 25.05 .
```

3.14.1. Сортировка данных

SPARQL использует ключевое слово ORDER BY, чтобы отсортировать данные. Это должно выглядеть знакомо пользователям SQL. Следующий запрос сортирует данные, используя значения, связанные с суммы переменной:

```
# filename: ex146.rq
PREFIX e: <http://learningsparql.com/ns/expenses#>
SELECT ?description ?date ?amount
WHERE
{
  ?meal  e:description ?description ;
         e:date ?date ;
         e:amount ?amount .

}
ORDER BY ?amount
```

Используя данные отчета о расходах приведенные выше, результат показывает данные о расходах, отсортированные от минимума к максимуму:

```
-------------------------------------------------
| description | date         | amount |
=================================
| "breakfast" | "2011-10-15" |  4.32  |
| "breakfast" | "2011-10-14" |  6.53  |
| "breakfast" | "2011-10-16" |  6.65  |
| "lunch"     | "2011-10-15" |  9.45  |
| "lunch"     | "2011-10-16" | 10.00  |
| "lunch"     | "2011-10-14" | 11.13  |
| "dinner"    | "2011-10-16" | 25.05  |
| "dinner"    | "2011-10-14" | 28.30  |
| "dinner"    | "2011-10-15" | 31.45  |
-------------------------------------------------
```

Для сортировки данных отчета о расходах в порядке убывания, сделайте ключ сортировки аргументом функции DESC():

```
# filename: ex148.rq
PREFIX e: <http://learningsparql.com/ns/expenses#>
SELECT ?description ?date ?amount
WHERE
{
   ?meal   e:description   ?description ;
           e:date          ?date ;
           e:amount        ?amount .
}
ORDER BY DESC(?amount)
```

Мы видели в разделе "Типы данных" что, когда объектом тройки является число, процессор обрабатывает его как целое или десятичное число, в зависимости от того, включает ли оно или нет десятичную точку. Когда ключевое слово SORT знает, что

это сортируются числа, то рассматривает их как значения, а не как строки.

Для сортировки по нескольким ключам, следует отделить ключевые имена значений пробелами. В следующем примере сортирует данные отчета о расходах в алфавитном порядке по описанию, а затем от значения суммы, от большего к меньшему:

```
# filename: ex149.rq
PREFIX e: <http://learningsparql.com/ns/expenses#>
SELECT ?description ?date ?amount
WHERE
{
   ?meal e:description ?description ;
         e:date ?date ;
         e:amount ?amount .
}
ORDER BY ?description DESC(?amount)
```

Вот результат, с описанием значений, перечисленных в алфавитном порядке, и суммами, отсортированных в порядке убывания в каждой категории пищи:

```
-----------------------------------------------------
| description | date             | amount |
=====================================
| "breakfast" | "2011-10-16" |   6.65     |
| "breakfast" | "2011-10-14" |   6.53     |
| "breakfast" | "2011-10-15" |   4.32     |
| "dinner"    | "2011-10-15" | 31.45      |
| "dinner"    | "2011-10-14" | 28.30      |
| "dinner"    | "2011-10-16" | 25.05      |
| "lunch"     | "2011-10-14" | 11.13      |
| "lunch"     | "2011-10-16" | 10.00      |
| "lunch"     | "2011-10-15" |   9.45     |
-----------------------------------------------------
```

3.14.2. Поиск минимального, максимального, количества, среднего значения ...

В SPARQL 1.0 способ найти наименьшее или наибольшее значение состоит в сортировке данных, а затем использовании ключевого слова LIMIT (описанного в разделе "Получение определенного количества результатов"), чтобы извлечь только первое значение. Например, запрос ex148.rq попросил данные отчета о расходах в порядке убывания, добавление LIMIT 1 в этом запросе указывает процессору SPARQL вернуть только первое из этих значений:

```
# filename: ex151.rq
PREFIX e: <http://learningsparql.com/ns/expenses#>
SELECT ?description ?date ?amount
 WHERE
{
   ?meal   e:description   ?description ;
           e:date          ?date ;
           e:amount        ?amount .
} ORDER BY DESC(?amount)
LIMIT 1
```

Выполнение запроса показывает нам самое дорогое блюдо:

```
-----------------------------------------------------
| description | date             | amount |
=================================
| "dinner"     | "2011-10-15" | 31.45   |
-----------------------------------------------------
```

Без функции убыванию DESC(), запрос отобразил бы данные для наименее дорогой еды, потому что с данными, отсортированных в порядке возрастания, она была бы в первую очередь.

Функция MAX () позволяет найти наибольшее значение с помощью гораздо более простого запроса.

Важно

MAX() и остальные функции описанные в этом разделе являются новой возможностью в SPARQL 1.1.

Вот простой пример:

```
# filename: ex153.rq
PREFIX e: http://learningsparql.com/ns/expenses#

SELECT (MAX(?amount) as ?maxAmount)
WHERE { ?meal   e:amount    ?amount . }
```

В пункте SELECT этот запрос сохранит максимальное значение переменной в переменной ?maxAmount, и это значение возвратится как результат:

```
--------------------
| maxAmount |
==============
| 31.45          |
--------------------
```

Чтобы найти описание ?description и дату ?date, связанные с максимальным значением ?amount с помощью функции MAX(), вам придется найти максимальное значение ?amount в подзапросе и отдельно запросить ?description и ?date, которые с ним связаны:

```
# filename: ex155.rq PREFIX e: <http://learningsparql.com/ns/expenses#>
SELECT ?description ?date ?maxAmount
WHERE
{
  {
  SELECT (MAX(?amount) as ?maxAmount
  WHERE { ?meal e:amount ?amount . }
  }
  {
```

```
    ?meal   e:description   ?description ;
            e:date          ?date ;
            e:amount  ?maxAmount.
  }
}
```

Используемый в SPARQL 1.0 трюк ORDER BY с LIMIT 1 на самом деле прост, но, как мы увидим, функция MAX() (и ее партнер функция MIN(), которая находит наименьшее значение) удобна в других ситуациях.

Подстановка других новых функций вместо MAX() в ex153.rq позволяет делать интересные вещи, которые не имеют SPARQL 1.0 эквивалента. Например, следующий запрос находит среднее значение стоимости всех блюд в данных отчета о расходах:

```
# filename: ex156.rq
PREFIX e: <http://learningsparql.com/ns/expenses#>
SELECT (AVG(?amount)  as  ?avgAmount)
WHERE { ?meal   e:amount   ?amount . }
```

ARQ вычисляет среднее с довольно большим числом знаков после запятой, такой точности не дают другие процессоры SPARQL:

```
-----------------------------------------------
| avgAmount                                   |
===============================
| 14.76444444444444444444444 |
-----------------------------------------------
```

С помощью функции SUM(), записанной в том же месте можно сложить значения, а функция COUNT() будет рассчитывать, количество найденных значений e:amount.

Другой интересной функцией является GROUP_CONCAT(), которая объединяет все значения, связанные с переменной, разделенных пробелом или разделителем, который вы укажете в необязательном втором аргументе. Следующий запрос вычислит все значения расходов в переменную ?amountList через запятую:

```
# filename: ex158.rq
PREFIX e: <http://learningsparql.com/ns/expenses#>
SELECT (GROUP_CONCAT(?amount; SEPARATOR = ",") AS ?amountList)
WHERE { ?meal   e:amount   ?amount . }
```

Результат (минус заголовок и пограничные знаки, которые ARQ добавляет для отображения в его выходной формат по умолчанию) можно легко импортировать в электронную таблицу:

```
-----------------------------------------------------------------------
| amountList                                                          |
=======================================================
| "25.05,10.00,6.65,31.45,9.45,4.32,28.30,11.13,6.53" |
-----------------------------------------------------------------------
```

3.14.3. Группировка значений данных и поиск средних значений внутри групп

Еще одной особенностью SPARQL, полученной в наследство от SQL является ключевое слово GROUP BY. Оно позволяет группировать вместе наборы данных для

выполнения агрегатных функций для каждой группы. Например, следующий запрос говорит процессору SPARQL, что надо группировать результаты вместе по значению ключа ?description и затем суммировать все ?amount для каждой группы:

```
# filename: ex160.rq
PREFIX e: <http://learningsparql.com/ns/expenses#>
SELECT ?description (SUM(?amount) AS ?mealTotal)
WHERE
{
  ?meal   e:description   ?description ;
          e:amount ?amount .
}
GROUP BY ?description
```

Выполнение этого запроса с данными ex145.ttl дает нам следующий результат:

```
-----------------------------------
| description | mealTotal |
=======================
| "dinner"    | 84.80     |
| "lunch"     | 30.58     |
| "breakfast" | 17.50     |
-----------------------------------
```

Подставляя AVG(), MIN(), MAX() или COUN() функции вместо SUM() в этот запрос мы получим среднее, минимум, максимум или число значений в каждой группе.

Следующий запрос демонстрирует хорошее использование COUNT() для исследования нового набора данных. Оно говорит нам о том, сколько раз был использован каждый предикат:

```
# filename: ex162.rq
SELECT ?p (COUNT(?p) AS ?pTotal)
WHERE
{ ?s   ?p   ?o . }
GROUP BY ?p
```

Обратите внимание, что ex162.rq даже не объявляет никакие пространства имен. Это запрос очень общего назначения.

Данные отчета о расходах имеет очень регулярный формат, с одинаковым количеством троек, описывающих каждый прием пищи, поэтому результат выполнения этого запроса для данных отчета о расходах не особенно интересен, но это показывает, что запрос делает свою работу должным образом:

```
-------------------------------------------------------------------------------
| p                                                                 | pTotal |
=====================================================
| <http://learningsparql.com/ns/expenses#date>          | 9       |
| <http://learningsparql.com/ns/expenses#description> | 9       |
| <http://learningsparql.com/ns/expenses#amount>      | 9       |
-------------------------------------------------------------------------------
```

Когда вам понадобиться изучить более неравномерные формы данных, этот вид запроса может рассказать вам много о нем.

Ключевое слово HAVING делает для агрегатных величин то же самое, что делает фильтр FILTER для отдельных значений: он определяет условие, что позволяет ограничить значения, которые вы хотите видеть в результатах поиска. Следующая версия запроса, который вычисляет расходоы на еду, условие HAVING указывает процессор SPARQL, что мы заинтересованы только в итогах, превышающих 20:

```
# filename: ex164.rq
PREFIX e: <http://learningsparql.com/ns/expenses#>
SELECT ?description (SUM(?amount) AS ?mealTotal)
WHERE
{
   ?meal   e:description   ?description ;
           e:amount        ?amount .
}
GROUP BY ?description
HAVING (SUM(?amount) > 20)
```

И вот, что мы получим:

```
-----------------------------------
| description | mealTotal |
=======================
| "dinner"     | 84.80       |
| "lunch"      | 30.58       |
 -----------------------------------
```

3.15. Запрос удаленной службы SPARQL

Мы видели, как ключевое FROM слово может определить набор данных для запроса, который может быть локальным или удаленным файлом. Например, следующий запрос запрашивает любые Dublin Core значения названий в FOAF файле Тима Бернерс-Ли, которые хранятся на сервере MIT:

```
# filename: ex166.rq
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?title
FROM <http://dig.csail.mit.edu/2008/webdav/timbl/foaf.rdf>
WHERE { ?s    dc:title    ?title .}
```

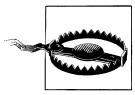
Ключевое слово SERVICE дает вам еще один способ запроса удаленных данных, но вместо того, чтобы указывать на файл RDF (или на службе доставки эквивалент файла RDF), вы указываете на конечную точку SPARQL. В то время как типичный SPARQL запрос или подзапрос извлекает данные из локального или удаленного файла, ключевое слово SERVICE позволяет сказать, "отправить этот запрос к указанной конечной точке SPARQL, так что она может выполнить запрос, а затем отправить обратно результат ". Это очень важное ключевое слово, потому что многие услуги SPARQL конечных точек доступны, и их данные могут добавить много вашим приложениям.

Важно

Ключевое слово SERVICE являются новой возможностью в SPARQL 1.1.

Ключевое слово SERVICE можете передать шаблон графа или весь запрос к конечной точке SPARQL. Ниже приведен пример отправки всего запрос:

```
# filename: ex167.rq
PREFIX cat: <http://dbpedia.org/resource/Category:>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?p ?o
WHERE
{
   SERVICE <http://DBpedia.org/sparql>
   { SELECT ?p ?o
     WHERE { <http://dbpedia.org/resource/Joseph_Hocking> ?p ?o . }
   }
}
```



ARQ ожидает указания некоторых данных запроса, либо в командной строке, либо с помощью ключевого слова FROM. Может показаться, что запрос ex167.rq указывает некоторые данные запроса, но на самом деле он только назвать конечную точку, чтобы отправить туда запрос. В любом случае, для запуска этого ARQ запроса из командной строки, вы должны назвать файл данных как аргумент --data, хотя этот запрос не будет ничего с данными из файла делать.

Запрос представленный выше довольно прост. Результатом его выполнения будут предикаты и объекты всех троек DBpedia, в которых субъектом является ресурс http://dbpedia.org/resource/Joseph_Hocking:

```
-------------------------------------------------------------------------------------|
| p              | o                                                                  |
=======================================================|
| owl:sameAs     | <http://rdf.freebase.com/ns/guid.9202a8c...>                      |
| rdfs:comment   | "Joseph Hocking (November 7, 1860–March ..."@en                   |
| skos:subject   | cat:Cornish_writers                                               |
| skos:subject   | cat:English_Methodist_clergy                                      |
| skos:subject   | cat:19th-century_Methodist_clergy                                 |
| skos:subject   | cat:People_from_St_Stephen-in-Brannel                             |
| skos:subject   | cat:1860_births                                                   |
| skos:subject   | cat:1937_deaths                                                   |
| skos:subject   | cat:English_novelists                                             |
| rdfs:label     | "Joseph Hocking"@en                                               |
| foaf:page      | <http://en.wikipedia.org/wiki/Joseph_Hocking>                     |
-------------------------------------------------------------------------------------
```

Этот результат не тонна данных, но только потому, что мы сознательно выбрали малоизвестного человека, чтобы спросить о нем. Мы также урезали данные в двух местах, где вы видите ... в первых двух строках таблицы, чтобы поместиться на странице; rdfs:comment - комментарий, характеризующий британского писателя и министра на самом деле занимает целый абзац.

Как правило, вместо использования SERVICE для запроса к удаленной службе, запросы просто отправляют шаблон графа, показывающий тройки, которые нужно извлечь, а затем они позволяют выбрать внешний SELECT (или CONSTRUCT или другой тип запроса) чтобы указать, в каких значениях они заинтересованы. Следующий запрос возвращает те же данные, что и предыдущий:

```
# filename: ex539.rq
PREFIX cat: <http://dbpedia.org/resource/Category:>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?p ?o
WHERE
{
  SERVICE <http://DBpedia.org/sparql>
  { <http://dbpedia.org/resource/Joseph_Hocking> ?p ?o . }
}
```

Следующий запрос посылает аналогичный запрос о том же британском писателе к немецкой коллекции метаданных в проекте Гутенберга - коллекции бесплатных электронных книг:

```
# filename: ex170.rq
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX gp: <http://wifo5-04.informatik.uni-mannheim.de/gutendata/resource/people/>
SELECT ?p ?o
WHERE
{
   SERVICE <http://wifo5-04.informatik.uni-mannheim.de/gutendata/sparql>
   { gp:Hocking_Joseph   ?p   ?o . }
 }
```

Вот результ:

```
-------------------------------------------------------------------------------------------------
|  p                                    |  o                                                    |
=================================================================================================
| rdfs:label                            | "Hocking, Joseph"                                     |
| <http://xmlns.com/foaf/0.1/name> | "Hocking, Joseph"                                     |
 | rdf:type                              | <http://xmlns.com/foaf/0.1/Person> |
-------------------------------------------------------------------------------------------------
```

Здесь не так много данных, но если вы измените этот запрос, и попросите субъекты и предикаты всех троек, где gp:Hocking_Joseph является объектом, то обнаружите, что

один из субъектов является URI, который представляет один из его романов. Наряду с gp: Hocking_Joseph в качестве значения создателя этого романа, вы увидите ссылку на сам роман и в других метаданных о нем. Эта способность получать доступ к произведениям автора становится еще более интересной, когда вы используете ту же технику с более известными авторами, чьи произведения также находятся в общественном достоянии - если вы интересуетесь литературой, это увлекательная сфера - облако связанных данных.

В этом облаке много источников данных, предлагающих интересные наборы троек для запроса, но помните, что удаленный сервис не должен быть очень далеко и он даже не должен хранить троек. Интерфейс D2RQ с открытым исходным кодом позволяет использовать SPARQL для запросов реляционных баз данных, и это довольно легко сделать - это означает, что вы можете делать свои собственные реляционные базы данных доступными для SPARQL запросов внешним пользователям за брандмауэром. Последний вариант делает технологии RDF более популярными для обеспечения более легкого доступа к данным в пределах организации, которые в противном случае будут скрыты, и ключевое слово SERVICE позволяет получить эти данные, указывая путь доступа.

3.16. Федеративные запросы: поиск в нескольких наборах данных с одним запросом

Ключевое слово SERVICE позволяет поместить подзапрос внутри другого запроса. Вы уже знаете все, что необходимо для ввода и выполнения запроса, который извлекает данные из нескольких наборов данных: просто создать подзапрос для каждого из них. (Смотри раздел "Запросы в запросах" для получения более подробной информации). Рассмотрим комбинацию двух примеров, которые продемонстрировали ключевое слово SERVICE в разделе "Запрос удаленной службы SPARQL", но изменим имена переменных, чтобы сделать их уникальным в пределах каждого подзапроса:

```
# filename: ex172.rq
PREFIX cat: <http://dbpedia.org/resource/Category:>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
 PREFIX gp: <http://wifo5-04.informatik.uni-mannheim.de/gutendata/resource/people/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>

SELECT ?dbpProperty ?dbpValue ?gutenProperty ?gutenValue
WHERE
{ SERVICE  <http://DBpedia.org/sparql>
   {
    <http://dbpedia.org/resource/Joseph_Hocking> ?dbpProperty ?dbpValue .
   }
SERVICE  <http://wifo5-04.informatik.uni-mannheim.de/gutendata/sparql>
   {
    gp:Hocking_Joseph ?gutenProperty ?gutenValue .
   }
}
```

Результаты, представленные на рисунке 3-5 (опять же, с обрезанными данными в местах ...) может быть не совсем то, что вы ожидали.

dbpProperty	dbpValue	gutenProperty	gutenValue
owl:sameAs	<http://rdf.freebase.com/ns/guid.9202a8c...>	rdfs:label	"Hocking, Joseph"
owl:sameAs	<http://rdf.freebase.com/ns/guid.9202a8c...>	foaf:name	"Hocking, Joseph"
owl:sameAs	<http://rdf.freebase.com/ns/guid.9202a8c...>	rdf:type	foaf:Person
rdfs:comment	"Joseph Hocking (November 7, 1860–March ..."@en	rdfs:label	"Hocking, Joseph"
rdfs:comment	"Joseph Hocking (November 7, 1860–March ..."@en	foaf:name	"Hocking, Joseph"
rdfs:comment	"Joseph Hocking (November 7, 1860–March ..."@en	rdf:type	foaf:Person
skos:subject	cat:Cornish_writers	rdfs:label	"Hocking, Joseph"
skos:subject	cat:Cornish_writers	foaf:name	"Hocking, Joseph"
skos:subject	cat:Cornish_writers	rdf:type	foaf:Person
skos:subject	cat:English_Methodist_clergy	rdfs:label	"Hocking, Joseph"
skos:subject	cat:English_Methodist_clergy	foaf:name	"Hocking, Joseph"
skos:subject	cat:English_Methodist_clergy	rdf:type	foaf:Person
skos:subject	cat:19th-century_Methodist_clergy	rdfs:label	"Hocking, Joseph"
skos:subject	cat:19th-century_Methodist_clergy	foaf:name	"Hocking, Joseph"
skos:subject	cat:19th-century_Methodist_clergy	rdf:type	foaf:Person
skos:subject	cat:People_from_St_Stephen-in-Brannel	rdfs:label	"Hocking, Joseph"
skos:subject	cat:People_from_St_Stephen-in-Brannel	foaf:name	"Hocking, Joseph"
skos:subject	cat:People_from_St_Stephen-in-Brannel	rdf:type	foaf:Person
skos:subject	cat:1860_births	rdfs:label	"Hocking, Joseph"
skos:subject	cat:1860_births	foaf:name	"Hocking, Joseph"
skos:subject	cat:1860_births	rdf:type	foaf:Person
skos:subject	cat:1937_deaths	rdfs:label	"Hocking, Joseph"
skos:subject	cat:1937_deaths	foaf:name	"Hocking, Joseph"
skos:subject	cat:1937_deaths	rdf:type	foaf:Person
skos:subject	cat:English_novelists	rdfs:label	"Hocking, Joseph"
skos:subject	cat:English_novelists	foaf:name	"Hocking, Joseph"
skos:subject	cat:English_novelists	rdf:type	foaf:Person
rdfs:label	"Joseph Hocking"@en	rdfs:label	"Hocking, Joseph"
rdfs:label	"Joseph Hocking"@en	foaf:name	"Hocking, Joseph"
rdfs:label	"Joseph Hocking"@en	rdf:type	foaf:Person
foaf:page	<http://en.wikipedia.org/wiki/Joseph_Hocking>	rdfs:label	"Hocking, Joseph"
foaf:page	<http://en.wikipedia.org/wiki/Joseph_Hocking>	foaf:name	"Hocking, Joseph"
foaf:page	<http://en.wikipedia.org/wiki/Joseph_Hocking>	rdf:type	foaf:Person

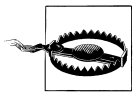
Рисунок 3-5. Результаты запроса ex172.rq

Помните, запросы использующие ключевое слово SERVICE служат для извлечения данных из конечных точек SPARQL.

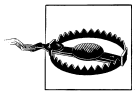
При запуске каждого из двух подзапросов предыдущего раздела мы получили 11 и 3 строки в результат. Сочетание этих запросов здесь дает нам 33 строки, так что вы можете догадаться, что происходит: результатом является кросс-продукт, или каждая комбинация из двух значений из каждой строки первого запроса с двумя значениями из каждой строки второго запроса.

Кросс-произведение этих двух наборов троек не особенно полезно, но мы увидим в главе 4, как несколько наборов данных из разных источников, могут быть более удачно связаны вместе для последующего повторного использования.

Федеративный запрос может иметь более двух подзапросов, но помните, что вы по сути вы говорите процессору запросов запросить один источник, затем другой, а потом, возможно, еще и еще, естественно это может занять значительное время.



Большинство известных общедоступных конечных точек SPARQL являются результатом добровольного труда, поэтому они не обязательно отвечает на все запросы 24 часа в сутки.



Не пишите федеративные запросы, которые проходят через целый ряд удаленных наборов данных, а затем ищут соответствующие значения для всего что он находят - в другом удаленном наборе данных, пока вы абсолютно не уверены, что два (или более) наборов данных имеют приемлемый размер, что делает такие запросы разумными. Добавление ключевых слов LIMIT и OFFSET для этих запросов может помочь вам оценить возможности использования конкретных конечных точек SPARQL.

3.17. Резюме

В этой главе мы узнали:

• Предикаты rdfs:label могут сделать результаты запроса более читабельными, показывая описание ресурсов, а не URI

• Ключевое слово OPTIONAL придает гибкость запросам для извлечения данных, которые могут быть и не быть в анализируемом наборе данных

• Как одна та же переменная в положении объекта в одной тройке и в положении субъекта в другой, может помочь вам найти связанные тройки

• Как пустым узлы позволяют группировать тройки вместе

• Ключевое слово DISTINCT, который позволяет устранить дублирующие данные в результатах запроса

• Ключевое слово UNION, которое позволяет указать несколько шаблонов графа, чтобы получить несколько различных наборов данных, а затем объединить их в результате запроса

• Ключевое слово FILTER, которые можно сократитьть количество результатов на основе логических условий; ключевое слово LIMIT, которое задать определенное количество результатов запроса; OFFSET, которое пропускает определенное количество результатов и как извлекать данные только от определенных поименованных графов

• Как использовать выражения для расчета новых значений и как ключевое слово BIND может упростить использование выражений

• Как ключевое слово VALUES позволяет создавать таблицы значений для использования в качестве условий фильтрации

• Как ORDER BY позволяет сортировать данные в порядке возрастания или убывания и, в сочетании с ключевым словом LIMIT, позволяет получить наибольшее и наименьшее значения; как MAX(), AVG(), SUM() и другие агрегатные функции позволяют преобразовывать ваши данные еще более; и как группировать данные по определенным ключевым значениям, чтобы найти совокупные значения в пределах группы

• Как создавать подзапросы

• Как назначить значения переменным

• Ключевое слово SERVICE, которое позволяет указать удаленную конечную точку SPARQL для запроса

• Как совместить несколько подзапросов, что каждый использовал ключевое слово SERVICE для проведения федеративных запросов

Глава 4. Копирование, создание, преобразование и поиск данных

В главе 3 описано множество способов получения троек из набора данных и отображения различных значений из этих троек. В этой главе мы узнаем, как можно сделать гораздо больше, чем просто отображать эти значения:

"Типы запросов: SELECT, DESCRIBE, ASK и CONSTRUCT"

Выбор нужных троек из набора данных на основе графого шаблона в значительной степени основная задача SPARQL, и вы уже знаете, несколько способов как сделать это. Кроме SELECT, есть еще три ключевых слова, которые вы можете использовать, чтобы указать, что вы хотите делать с извлеченными троек.

"Копирование данных"

Иногда вы просто хотите вытащить несколько троек из одной коллекции и сохранить в другой. Может быть, вы агрегируете данные по конкретной теме из различных источников, или, может быть, вы просто хотите, чтобы данные хранились локально, чтобы ваши приложения могли работать с этими данными более быстро и надежно.

"Создание новых данных"

Создание новых данных из существующих данных является одним из самых интересных аспектов SPARQL и RDF технологии.

"Преобразование данных"

Если ваше приложение ожидает данные, чтобы соответствовать определенной модели, и у вас есть данные, которые почти, но не совсем соответствует этой модели, то преобразовть их в тройки, которые соответствуют должным образом может быть легко. Если целевая модель является признанным стандартом, это дает вам новые возможности для интеграции ваших данных с другими данными и приложениями.

"Поиск плохих данных"

Если вы можете описать тип данных, которые вы не хотите видеть, вы можете найти их. При сборе данных из нескольких источников, эта возможность (и возможность конвертировать данные) может иметь неоценимое. Наряду с проверкой ограничений, таких, как использование соответствующих типов данных, эти методы могут также позволяет проверить набор данных на соответствие бизнес-правилам.

"Запрос описания ресурса"

Операция DESCRIBE позволяет запросить информацию о ресурсе, представленном конкретным URI.

Важно

Общие идеи, описанные в этой главе работают со SPARQL 1.0, а также 1.1, но несколько примеров с ключевым словом BIND и некоторыми функциями доступны только в SPARQL 1.1.

4.1. Типы запросов: SELECT, DESCRIBE, ASK и CONSTRUCT

Как и в SQL, самым популярным глаголом SPARQL является SELECT. Это позволяет запрашивать данные из коллекции, хотите ли вы один номер телефона или список имен, фамилий и номеров телефонов сотрудников, нанятых после 1 января и отсортированных по фамилии. Процессоры SPARQL, такие как ARQ, как правило, показывают результат запроса на выборку в виде таблицы строк, с колонкой для каждого выбранного имени переменной, и API-интерфейсы SPARQL загружают значения в подходящие для языка программирования структуры данных.

Кроме известной формы запроса SELECT в SPARQL есть еще три:

• CONSTRUCT возвращает тройки. Вы можете вытаскивать тройки непосредственно из источника данных, не меняя их, или вы можете вытащить значения, и использовать эти значения, чтобы создать новые тройки. Это позволяет копировать, создавать и конвертировать данные RDF, а также облегчает идентифицировать данные, которые не соответствуют определенным правилам бизнеса.

• ASK просит процессор запросов ответить, соответствует ли данный графовый шаблон наборе троек в конкретном наборе данных или нет, и процессор возвращает логическое истина или ложь. Это отличная возможность для выражения бизнес-правил об условиях, которые должны или не должны выполняться в ваших данных. Вы можете использовать наборы этих правил для автоматизации контроля качества обработки данных.

• DESCRIBE запрашивает тройки, которые описывают конкретный ресурс. Спецификация SPARQL оставляет ето для процессора - решить, что необходимо отправить обратно для описания ресурса. Это привело к несогласованности реализаций запросов DESCRIBE, так что это форма запроса не очень популярна, но это стоит изучить.

Большая часть этой главы охватывает широкий спектр применения формы запросов CONSTRUCT. Мы также увидим некоторые примеры того, как использовать ASK и DESCRIBE.

4.2. Копирование данных

Ключевое слово CONSTRUCT позволяет создавать тройки, путем копирования нужной информации из других наборов данных. Представьте, что мы хотим, чтобы запросить следующий набор данных из главы 1 для получения всей информации о Craig Ellis:

```
# filename: ex012.ttl
@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d: <http://learningsparql.com/ns/data#> .

d:i0432 ab:firstName    "Richard" .
d:i0432 ab:lastName     "Mutt" .
d:i0432 ab:homeTel      "(229) 276-5135" .
d:i0432 ab:email        "richard49@hotmail.com" .

d:i9771 ab:firstName    "Cindy" .
d:i9771 ab:lastName     "Marshall" .
d:i9771 ab:homeTel      "(245) 646-5488" .
d:i9771 ab:email        "cindym@gmail.com" .

d:i8301 ab:firstName    "Craig" .
d:i8301 ab:lastName     "Ellis" .
d:i8301 ab:email        "craigellis@yahoo.com" .
d:i8301 ab:email        "c.ellis@usairwaysgroup.com" .
```

Запрос с оператором SELECT будет простым. Мы хотим получить субъект, предикат и объект всех троек, в которых субъект с предикатом ab:firstName имеет значение “Craig” , а с ab:lastName значение Ellis:

```
# filename: ex174.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX d: <http://learningsparql.com/ns/data#>
SELECT ?person ?p ?o
WHERE
```

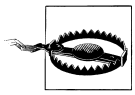
```
{
  ?person   ab:firstName   "Craig" ;
            ab:lastName    "Ellis" ;
            ?p             ?o .
}
```

Субъекты, предикаты и объекты сохранятся в ?person ?p и ?o переменных, и ARQ возвращает эти значения со столбцами для каждой переменной:

```
-------------------------------------------------------------------------
| person  | p             | o                                           |
=========================================================================
| d:i8301 | ab:email      | "c.ellis@usairwaysgroup.com"                |
| d:i8301 | ab:email      | "craigellis@yahoo.com"                      |
| d:i8301 | ab:lastName   | "Ellis"                                     |
| d:i8301 | ab:firstName  | "Craig"                                     |
-------------------------------------------------------------------------
```

CONSTRUCT версия того же самого запроса имеет тот же графовый шаблон после ключевого слова WHERE, но определяет создание тройки с каждым набором значений, связанных с тремя переменными:

```
# filename: ex176.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX d: <http://learningsparql.com/ns/data#>
CONSTRUCT { ?person ?p ?o . }
WHERE
{
   ?person   ab:firstName   "Craig" ;
             ab:lastName    "Ellis" ;
             ?p             ?o .
}
```



Набор моделей троек (только один в ex176.rq), которые описывают то, что требуется создать, сам по себе является графовым шаблоном, так что не забудьте заключить его в фигурные скобки.

Процессор запросов SPARQL возвращает данные на запрос CONSTRUCT как фактические тройки, а не как форматированный отчет с колонками для каждого имени переменной. Формат этих троек зависит от используемого процессора. ARQ возвращает их в формате Turtle, который должен быть знаком. Ниже представлен результа того, что ARQ возвращает после выполнения запроса ex176.rq к данным в ex012.ttl:

```
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix ab: <http://learningsparql.com/ns/addressbook#> .

d:i8301
        ab:firstName    "Craig" .
        ab:lastName     "Ellis" .
        ab:email        "craigellis@yahoo.com" .
        ab:email        "c.ellis@usairwaysgroup.com" .
```

Это может показаться не особенно интересно, но когда вы используете эту технику, чтобы собрать данные из одного или нескольких удаленных источников, она становится все более интересной. Ниже показаны вариации запроса ex172.rq, на этот раз примененного для получения троек о Джозефе Хокинге из двух конечных точек SPARQL:

```
# filename: ex178.rq
PREFIX cat: <http://dbpedia.org/resource/Category:>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX gp: <http://wifo5-04.informatik.uni-mannheim.de/gutendata/resource/people/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>

CONSTRUCT
{
  <http://dbpedia.org/resource/Joseph_Hocking>   ?dbpProperty   ?dbpValue .
  gp:Hocking_Joseph   ?gutenProperty   ?gutenValue .
 }
WHERE
{
  SERVICE <http://DBpedia.org/sparql>
  {
    <http://dbpedia.org/resource/Joseph_Hocking> ?dbpProperty ?dbpValue .
  }

  SERVICE <http://wifo5-04.informatik.uni-mannheim.de/gutendata/sparql>
  {
   gp:Hocking_Joseph ?gutenProperty ?gutenValue .
  }
}
```

Графовый шаблон CONSTRUCT в этом запросе имеет две трехместных компоненты (тройки). Он может иметь столько, сколько вам нравится.

Результат (с сокращенным "..." пунктом описания о Хокинге) включает троек о нем полученные из двух источников:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix cat: <http://dbpedia.org/resource/Category:> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> . <
@prefix owl: http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix gp: <http://wifo5-04.informatik.uni-mannheim.de/gutendata/resource/ people/> .

<http://dbpedia.org/resource/Joseph_Hocking>
rdfs:comment "Joseph Hocking (November 7, 1860–March 4, 1937) was ..."@en ;
rdfs:label "Joseph Hocking"@en ;
owl:sameAs <http://rdf.freebase.com/ns/guid.9202a8c04000641f800000000ab14b75> ;
skos:subject<http://dbpedia.org/resource/Category:People_from_St_Stephen-in- Brannel> ;
```

```
skos:subject <http://dbpedia.org/resource/Category:1860_births> ;
skos:subject <http://dbpedia.org/resource/Category:English_novelists> ;
skos:subject <http://dbpedia.org/resource/Category:Cornish_writers> ;
skos:subject <http://dbpedia.org/resource/Category:19th-century_Methodist_clergy> ;
skos:subject <http://dbpedia.org/resource/Category:1937_deaths> ;
skos:subject <http://dbpedia.org/resource/Category:English_Methodist_clergy> ;
foaf:page<http://en.wikipedia.org/wiki/Joseph_Hocking> .

gp:Hocking_Joseph
        rdf:type        foaf:Person ;
        rdfs:label      "Hocking, Joseph" ;
        foaf:name       "Hocking, Joseph" .
```

Вы также можете использовать ключевое слово GRAPH, чтобы запросить все тройки из конкретного поименованного графа:

```
# filename: ex180.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
CONSTRUCT { ?course    ab:courseTitle    ?courseName . }
FROM NAMED <ex125.ttl>
FROM NAMED <ex122.ttl>
WHERE
{
  GRAPH <ex125.ttl>  { ?course    ab:courseTitle    ?courseName }
}
```

Результат этого запроса по существу копирует данные из графа ex125.ttl, потому что все тройки там были с предикатами ab:courseTitle:

```
@prefix ab: <http://learningsparql.com/ns/addressbook#> .
ab:course24
        ab:courseTitle     "Using Named Graphs" .
ab:course42
        ab:courseTitle     "Combining Public and Private RDF Data" .
```

Это довольно искусственный пример, потому что не много смысла в том, чтобы указать два поименованных графа, а затем запросить все тройки из одного из них - особенно с утилитой командной строки ARQ, когда поименованный граф соответствует файлу существующему на диске и вы создаете копию того, что у вас уже есть. Однако, когда вы работаете с triplestores, которые хранят гораздо больше троек, чем вы когда-нибудь хранили в файле на жестком диске, вы лучше оцените способности захватывать все тройки из определенного поименованного графа.

В главе 3 мы видели, что ключевое слово FROM, за которым не следует ключевое слово NAMED, позволяет назвать набор данных для запроса прямо в запросе. Это также работает для запросов CONSTRUCT. Следующий запрос извлекает и выводит все тройки (22 на момент написания книги) из базы данных сообщества Freebase об Иосифе Хокинг:

```
# filename: ex182.rq
CONSTRUCT { ?s ?p ?o }
FROM <http://rdf.freebase.com/rdf/en.joseph_hocking>
WHERE
{ ?s   ?p   ?o }
```

Важно понять, что в конструкции запроса CONSTRUCT после ключевого слова WHERE можно использовать все методы, которые вы выучили в главах до этого, но, что после ключевого слова CONSTRUCT, вместо списка имен переменных, вы помещаете граф, показывающий шаблон тройки, которую вы хотите постить. В простейшем случае эти тройки прямые копии тех, которые извлекаются из исходного набора (или наборов) данных.

Если вы не помещаете графовый шаблон после CONSTRUCT, процессор SPARQL предполагает, что вы имели в виду то же самое описано в WHERE. Например, следующий запрос будет работать идентично предыдущему:

```
# filename: ex540.rq
CONSTRUCT { ?s ?p ?o }
FROM <http://rdf.freebase.com/rdf/en.joseph_hocking>
WHERE
{ ?s   ?p   ?o }
```

4.3. Создание новых данных

Выше запрос ex178.rq показал, что тройки, которые вы создаете в конструкции CONSTRUCT не должены полностью состоять из переменных. Если вы хотите, вы можете создать одну или несколько троек полностью из жестко-закодированных значений, с пустым графовым шаблоном после ключевого слова WHERE:

```
# filename: ex184.rq
PREFIX dc: <http://purl.org/dc/elements/1.1/>
CONSTRUCT
{
  <http://learningsparql.com/ns/data/book312>   dc:title   "Jabez Easterbrook" .
}
WHERE
{ }
```

Когда вы будете изменить и объединить значения, извлекаемые из набора данных, то увидите больше реальную мощь CONSTRUCT запросов. Например, при копировании данных в файл ex012.ttl, который включает номера телефонов, если вы можете быть уверены, что со второго по четвертый символ номер телефона является кодом города, то вы можете создать и заполнить новое свойство areaCode в запросе, как это представлено ниже:

```
# filename: ex185.rq PREFIX ab: <http://learningsparql.com/ns/addressbook#>
CONSTRUCT
{
   ?person ?p ?o ;
          ab:areaCode ?areaCode .
}
WHERE
{
   ?person   ab:homeTel   ?phone ;
             ?p           ?o .
   BIND (SUBSTR(?phone,2,3) as ?areaCode)
}
```

Тройной шаблон {?person ?p ?o} после ключевого слова WHERE вернул бы все тройки, в том числе значение ab:homeTel, даже если тройной модели {?person ab:homeTel ?phone} там не было. Предложение WHERE включает тройной шаблон ab:homeTel, чтобы позволить хранить значения телефонного номера в переменной ?phone, так чтобы директива BIND могла использовать ее для расчета кода города.

Результат выполнения этого запроса с данными ex012.ttl показывает все тройки связанные с двумя людьми из набора данных, которые имеют телефонные номера, и теперь они имеют новую тройку показывающую их код города:

```
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix ab: <http://learningsparql.com/ns/addressbook#> .

d:i9771
        ab:areaCode"245" ;
        ab:email "cindym@gmail.com" ;
        ab:firstName "Cindy" ;
        ab:homeTel"(245) 646-5488" ;
        ab:lastName"Marshall" .

d:i0432
        ab:areaCode"229" ;
        ab:email "richard49@hotmail.com" ;
        ab:firstName "Richard" ;
        ab:homeTel "(229) 276-5135" ;
        ab:lastName  "Mutt" .
```

Мы узнаем больше о функциях, таких как SUBSTR() в главе 5. По мере изучения возможностей запросов типа CONSTRUCT помните, что чем больше функций вы будете уметь использовать в своих запросах, тем больше видов данных вы можете создать.

Мы использовали функцию SUBSTR(), чтобы вычислить значения кода города, но вы можете не использовать вызовы функций, чтобы вывести новые данные из имеющихся данных. Очень распространенным способом в запросах SPARQL является поиск отношений между данными и затем использование условий в CONSTRUCT для создания новых троек, которые явно представляют эти отношения. Для нескольких примеров применения этого способа, мы будем использовать данные о родовых и родительских отношениях нескольких человек:

```
# filename: ex187.ttl
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix ab: <http://learningsparql.com/ns/addressbook#> .

d:jane  ab:hasParent    d:gene .
d:gene  ab:hasParent    d:pat;
        ab:gender       d:female .
d:joan  ab:hasParent    d:pat ;
        ab:gender       d:female .
```

```
d:pat    ab:gender       d:male .
d:mike   ab:hasParent    d:joan .
```

Наш первый запрос к этим данным - найти людей, которые имеют родителей, которые также имеют родителя мужского рода. Отсюда будет следовать выводит факт о наличии родителя родителя, являющегося дедушкой человека. Или, в терминах SPARQL, надо найти человека человека ?p с отношением ab:HasParent к кому-либо, чей идентификатор будет храниться в переменной ?parent, а затем найти кого-то, кто к ?parent, имеет отношение ab:hasParent с тем, кто имеет для ab:gender значение d:male. Если такой человек есть, то результатом будет тройка ?p ab:Grandfather ?g:

```
# filename: ex188.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX d: <http://learningsparql.com/ns/data#>
CONSTRUCT { ?p   ab:hasGrandfather   ?g . }
WHERE
{
  ?p         ab:hasParent    ?parent .
  ?parent    ab:hasParent    ?g .
  ?g         ab:gender       d:male .
}
```

Запрос создает две тройки о людях, имеющих отношение ab:grandParent к кому-то в наборе данных ex187.ttl:

```
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix ab: <http://learningsparql.com/ns/addressbook#> .

d:mike  ab:hasGrandfather       d:pat .
d:jane  ab:hasGrandfather       d:pat .
```

Другой запрос с теми же данными, создает тройки о том, кто кому является теткой:

```
# filename: ex190.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX d: <http://learningsparql.com/ns/data#>

CONSTRUCT
{ ?p ab:hasAunt ?aunt . }
WHERE
{
  ?p              ab:hasParent    ?parent .
  ?parent         ab:hasParent    ?g .
  ?aunt           ab:hasParent    ?g ;
                  ab:gender       d:female .
  FILTER (?parent != ?aunt)
}
```

Запрос не может просто спросить о сестрах чьих-то родителей, потому что в наборе данных нет никаких явных данных о сестрах, поэтому:

1. Запрос ищет бабушек и дедушек для ?p, как и прежде.

2. Он также ищет кого-то, отличного от родителя ?p (с той разницей, что здесь обеспечивается надежная фильтрация), кто имеет ту же самую прародительницу

(хранится в ?g) в виде родителя.

3. Если человек имеет для ab:gender значение d:female, то запрос выдает тройку о том, что у ?p есть тетя:

```
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix ab: <http://learningsparql.com/ns/addressbook#> .

d:mike  ab:hasAunt      d:gene .
d:jane  ab:hasAunt      d:joan .
```

Действительно ли эти запросы создают новую информацию? Разработчик реляционных БД быстро скажет что нет, что они на самом деле берут информацию, которая была неявной и делают ее явной. В реляционных БД, большая часть процесса, известного как нормализация предполагает поиск неявной информации в данных и хранение ее в явном виде, вместо ее динамического рассчита по мере необходимости, например, нахождение отношений дедушка тетя в последних двух запросах.

Реляционные БД представляют собой замкнутый мир с очень фиксированными границами. Данные, которые есть в конкретной БД – это данные только этой базы и об объединении двух реляционных БД для поиска новых отношений между строками таблиц из различных баз данных гораздо легче сказать, чем сделать. В приложениях, использующих технологию RDF, сочетание двух наборов данных осуществляется очень часто и простота агрегирования данных является одним из самых больших преимуществ RDF. Объединение данных, поиск шаблонов, а затем сохранение новых данных о чем-либо очень популярено во многих областях, которые используют эту технологию, например, в фармакологии и разведке.

В разделе "Повторное использование и создание словарей: RDF Schema и OWL" мы видели, как объявить ресурс, чтобы сделать его членом определенного класса, который может сказать людям больше о нем, потому что может иметь метаданные, связанные с этим классом. Мы узнаем больше об этом в главе 9, а сейчас, давайте посмотрим, как небольшая переделка этого последнего запроса может сделать его еще более явным - ресурс совпадающий с переменной есть тетя. Мы добавим тройку “говорящую” о том, что переменная ?aunt является членом этого определенного класса:

```
# filename: ex192.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

CONSTRUCT
{
  ?aunt   rdf:type      ab:Aunt .
  ?p      ab:hasAunt    ?aunt .
}
WHERE
{
  ?p            ab:hasParent    ?parent .
  ?parent       ab:hasParent    ?g .
  ?aunt         ab:hasParent    ?g ;
                ab:gender       d:female .

FILTER (?parent != ?aunt)
}
```

Выявление ресурсов в качестве членов классов является хорошей практикой, потому что это облегчает вывод информацию о ваших данных.

Объявление ресурса членом класса, который не был заявлен это не ошибка, но в этом не много смысла. Тройки, созданные в запросе представленном выше должны быть использованы с дополнительными тройками из онтологии, которая объявляет, что тетя класс и добавляет, по крайней мере немного метаданных о нем, как здесь:

```
# filename: ex193.ttl
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .

ab:Aunt    rdf:type    owl:Class ;
           rdfs:comment "The sister of one of the resource's parents." .
```

Классы также являются членами класса rdfs:Class или его подкласса owl:Class. Обратите внимание на сходство тройки, говорящей ""ab:Aunt является членом класса owl:Class" и тройки говорящей "?aunt является членом класса ab:Aunt."

Нет ничего, чтобы могло помешать вам взять две тройки из ex193.ttl и поместить в шаблон графа после ключевого слова CONSTRUCT в запрос ex192.rq, предварительно, как вы помните, включив объявления для rdf:, rdfs: и owl: префиксов. Затем запрос будет создать тройки и когда он создаст результирующую тройку, можно будет сказать, что ?aunt членом класса ab:Aunt. На практике, однако, когда вы говорите, что ресурс является членом определенного класса, вы, вероятно, делают это потому, что этот класс уже объявлен где-то еще.

4.4. Преобразование данных

Поскольку CONSTRUCT запросы могут создавать новые тройки, основанные на информации, извлеченной из набора, они представляют отличный способ для преобразования данных, которые используют свойства из одного пространства имен, в данные которые используют свойства от другого. Это позволяет принимать данные почти из любого места и превращать их в нечто, что можно использовать в вашей системе.

Как правило, это означает преобразование данных использующих одну схему или онтологию в данные, который использует другую, но иногда ваши входные данные не использует какой-либо конкретной схемы, и вы просто заменяете один набор предикатов на другой. В идеале, однако, существует схема для целевого формата, благодаря которой вы делаете преобразование - чтобы ваша новая версия данных соответствовала известной схеме и, следовательно, легче могла быть объединена с другими данными.

Давайте посмотрим это на примере. Мы используем файл данных ex012.ttl из главы 1 и показанный здесь:

```
# filename: ex012.ttl
@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d: <http://learningsparql.com/ns/data#> .

d:i0432 ab:firstName    "Richard" .
```

```
d:i0432 ab:lastName    "Mutt" .
d:i0432 ab:homeTel     "(229) 276-5135" .
d:i0432 ab:email       "richard49@hotmail.com" .

d:i9771 ab:firstName   "Cindy" .
d:i9771 ab:lastName    "Marshall" .
d:i9771 ab:homeTel     "(245) 646-5488" .
d:i9771 ab:email       "cindym@gmail.com" .

d:i8301 ab:firstName   "Craig" .
d:i8301 ab:lastName    "Ellis" .
d:i8301 ab:email       "craigellis@yahoo.com" .
d:i8301 ab:email       "c.ellis@usairwaysgroup.com" .
```

Серьезное приложение адресной книги будет хранить эти данные с помощью FOAF онтологии или W3C онтологии, которые представляют модели визитной карточки (vCard), это стандартные форматы файла для информационного моделирования бизнес-карты. Следующий запрос преобразует данные в визитную карточку RDF:

```
# filename: ex194.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX v: <http://www.w3.org/2006/vcard/ns#>
CONSTRUCT
{
  ?s     v:given-name   ?firstName ;
         v:family-name  ?lastName ;
         v:email        ?email ;
         v:homeTel      ?homeTel .
}
WHERE
{
  ?s     ab:firstName   ?firstName ;
         ab:lastName     ?lastName ;
         ab:email ?email .
         OPTIONAL
         { ?s   ab:homeTel   ?homeTel . }
}
```

Мы впервые узнали о ключевом слове OPTIONAL в разделе "Данные, которых может не быть". Оно служит той же цели здесь, которой оно служило в запросе SELECT: чтобы указать, что равных часть графового шаблона не должны препятствовать согласование остальной его части. В приведенном выше запросе, если входной ресурс не имеет значения ab:homeTel, но есть значения ab:firstName, ab:lastName и ab:email – они должны присутствоват в результате.

ARQ выводит это при применении запроса ex194.rq к данным ex012.ttl:

```
@prefix v: <http://www.w3.org/2006/vcard/ns#> .
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix ab: <http://learningsparql.com/ns/addressbook#> .

d:i9771
         v:email"cindym@gmail.com" ;
         v:family-name "Marshall" ;
```

```
        v:given-name "Cindy" ;
        v:homeTel"(245) 646-5488" .
d:i0432
        v:email"richard49@hotmail.com" ;
        v:family-name "Mutt" ;
        v:given-name "Richard" ;
        v:homeTel "(229) 276-5135" .
d:i8301
        v:email"c.ellis@usairwaysgroup.com" ;
        v:email "craigellis@yahoo.com" ;
        v:family-name "Ellis" ;
        v:given-name"Craig" .
```

Преобразование ab:email в v:email или ab:homeTel тв v:homeTel может показаться несущественным, но помните, что за этими префиксами выступают URI. Множество программ для RDF признают предикат http://www.w3.org/2006/vcard/ns#email, но никому за пределами этой книги не будет понятен http://learningsparql.com/ ns/addressbook#email, так что есть большая разница.

Преобразование данных может также означать нормализацию URI ресурсов, чтобы легче сочетать данные. Например, скажем, у меня есть набор данных о британских романистах, и я с помощью URI http://learningsparql.com/ns/data#HockingJoseph представляю Джозефа Хокинга. В следующей вариации CONSTRUCT запроса ex178.rq, который вытаскивал тройки об этом писатель как с DBpedia, так и из метаданных проекта Gutenberg, не осуществляется простое копирование троек, а вместо этого, он использует свой URI для писателя, в качестве субъекта всех построенных троек:

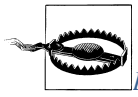
```
# filename: ex196.rq
PREFIX cat: <http://dbpedia.org/resource/Category:>
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
PREFIX gp: <http://wifo5-04.informatik.uni-mannheim.de/gutendata/resource/people/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX d: <http://learningsparql.com/ns/data#>

CONSTRUCT
{
  d:HockingJoseph       ?dbpProperty    ?dbpValue ;
                        ?gutenProperty ?gutenValue .
}
WHERE
{
   SERVICE <http://DBpedia.org/sparql>
   {
      <http://dbpedia.org/resource/Joseph_Hocking>    ?dbpProperty    ?dbpValue .
   }

   SERVICE <http://wifo5-04.informatik.uni-mannheim.de/gutendata/sparql>
   {
     gp:Hocking_Joseph    ?gutenProperty    ?gutenValue .
```

```
  }
}
```

Как в тройках графового шаблона в WHERE и в данных Turtle, в тройках графового шаблона CONSTRUCT можно для краткости использовать точки с запятой и запятые.

Результат выполнения запроса имеет тройки о http://learningsparql.com/ns/data #HockingJoseph созданные из двух источников:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix cat: <http://dbpedia.org/resource/Category:> .
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix gp: <http://wifo5-04.informatik.uni-mannheim.de/gutendata/resource/ people/> .

d:HockingJoseph
rdf:type foaf:Person ;
rdfs:comment  "Joseph Hocking (November 7, 1860–March 4, 1937) was..."@en ;
rdfs:label "Hocking, Joseph" ;
rdfs:label "Joseph Hocking"@en ;
owl:sameAs  <http://rdf.freebase.com/ns/guid.9202a...> ;
skos:subject http://dbpedia.org/resource/Category:People_from_St_Stephen-in- Brannel ;
skos:subject <http://dbpedia.org/resource/Category:1860_births> ;
skos:subject <http://dbpedia.org/resource/Category:English_novelists> ;
skos:subject <http://dbpedia.org/resource/Category:Cornish_writers> ;
skos:subject <http://dbpedia.org/resource/Category:19th-century_Methodist_clergy> ;
skos:subject <http://dbpedia.org/resource/Category:1937_deaths> ;
skos:subject < http://dbpedia.org/resource/Category:English_Methodist_clergy> ;
foaf:name "Hocking, Joseph" ;
foaf:page <http://en.wikipedia.org/wiki/Joseph_Hocking> .
```



Если разные идентификаторы URI используются для представления одного и того же ресурса в различных наборах данных (например, http://dbpedia.org /resource/Joseph_Hocking и http://wifo5-04.informatik.uni-mannheim.de/gutendata/resource /people/Hocking_Joseph в данных, полученных с помощью ex196.rq), и вы хотите, чтобы совокупные данные ссылались на то же самое, есть лучшие способы сделать это, чем изменять URI. Вы видите в одной из троек предикат owl:sameAs, говорящий что этот запрос извлекается из DBpedia – это один подход. (Кроме того, при сборе троек из многих источников, вы можете записать, когда и где вы их получили и назначить эту информацию как метаданные о графе). В данном случае, корректировка URI это просто еще один пример того, как можно использовать CONSTRUCT массированного сбора некоторых данных.

4.5. Поиск неверных данных

В разработке реляционных БД, XML и других областях информационных технологий, схема представляет собой набор правил о структурах и типах для обеспечения качества данных и повышения эффективности систем данных. Если одна из этих схем говорит, что значения количества должны быть целыми числами, вы знаете, что

количество никогда не может быть равным 3,5 или "привет". Таким образом, разработчикам, пишущим приложения для обработки данных, не нужно беспокоиться о странном поведении данных, которое будут нарушать обработку, если программа вычитает 1 из количества со значением "привет", это может привести к беде.

Приложения RDF основаны на ином подходе. Вместо предоставления шаблонов, в которые данные должны поместиться, приложения могут делать предположения о данных, RDF схема и OWL онтологии добавляют дополнительные метаданные. Например, когда мы знаем, что ресурс d:id432 является членом класса d:product3973, который имеет rdfs:label "клубника" и является подклассом класса с rdfs:label "фрукты", то мы знаем, что d:product3973 также является членом класса «фрукты».

Это здорово, но что делать, если вы хотите определить правила для троек и проверить, соответствует ли им набор данных, так чтобы приложение не заботилось о неожиданных значениях данных, нарушающих логику? OWL обеспечивает несколько способов сделать это, но это может быть довольно сложно, и вы будете нуждаться в OWL-ориентированном процессоре. Использование SPARQL для определения таких ограничений становится все более популярным, как благодаря своей простоте, так и более широкому спектру программного обеспечения (то есть, все процессоры SPARQL), которые позволяют мере определить эти правила.

В качестве бонуса, те же самые методы позволяют определить бизнес-правила, которые полностью выходят за рамки SQL в развитии реляционных БД. Они также выходят за рамки традиционных схем XML, хотя язык Schematron сделал там определенный вклад.

4.5.1. Определение правил на SPARQL

Пусть есть приложение, которое использует большое количество подобных данных, например, представленных ниже (набор получен из ex104.ttl с добавлением некоторых данных) :

```
# filename: ex198.ttl
@prefix dm: <http://learningsparql.com/ns/demo#> .
@prefix d: <http://learningsparql.com/ns/data#> .

d:item432       dm:cost 8.50 ;
                dm:amount 14 ;
                dm:approval d:emp079 ;
                dm:location <http://dbpedia.org/resource/Boston> .

d:item201       dm:cost 9.25 ;
                dm:amount 12 ;
                dm:approval d:emp092 ;
                 dm:location <http://dbpedia.org/resource/Ghent> .

d:item857       dm:cost 12 ;
                dm:amount 10 ;
                dm:location <http://dbpedia.org/resource/Montreal> .

d:item693       dm:cost 10.25 ;
                dm:amount 1.5 ;
                dm:location "Heidelberg" .

d:item126       dm:cost 5.05 ;
                dm:amount 4 ;
                dm:location <http://dbpedia.org/resource/Lisbon> .
```

```
d:emp092 dm:jobGrade 1 .
d:emp041 dm:jobGrade 3 .
d:emp079 dm:jobGrade 5 .
```

Прежде чем передавать эти данные дать на обработку приложению необходимо убедиться, что данные соответствуют определенным правилам. Ниже представлены три правила, и указаны несоответствия определенных данных из этого набора этим правилам:

• ***Все значения dm:location должны быть URI.*** Пункт d:item693 имеет dm:location значение "Гейдельберг", что является строкой, а не URI местоположения.

• ***Все значения dm:amount количество должны быть целыми числами.*** Выше, dm:amount имеет значение размере 1,5.

• ***Значение dm:approval не обязательно, если общая стоимость покупки меньше или равна 100. Если она больше 100, то покупка должна быть одобрена сотрудником со степенью работы dm:jobGrade больше, чем 4.*** Покупка d:item201 стоит более 100 (9.25x12), а одобряющий сотрудник имеет класс работы 1, покупка d:item857 также стоит более 100 и вообще не имеет одобрения.

Форма запроса ASK предполагает сопоставление образца графа с данными в наборе. Определяя образец графа для чего-то, что нарушает правило, мы можем создать запрос, который спрашивает "содержат ли эти данные нарушение этого правила?" В разделе "3.8. Фильтрация данных в зависимости от условий", мы увидели, что запрос ex107.rq находит все значения dm:location, которые не являются идентификаторами URI. Небольшое изменение превращает его в запрос, который проверяет, существует ли эта проблема во входном наборе данных:

```
# filename: ex199.rq
PREFIX dm: <http://learningsparql.com/ns/demo#>
ASK WHERE
{
  s   dm:location   ?city .
  FILTER (!(isURI(?city)))
}
```

ARQ отвечает следующее:

```
Ask => Yes
```

Другие процессоры SPARQL может вернуть xsd:boolean истинное значение. Если вы используете интерфейс к процессору SPARQL, который построен вокруг конкретного языка программирования, он, вероятно, вернет представление булевского истинного значения принятого в этом языке.

С помощью функции тип данных datatype (), о которой мы узнаем больше в главе 5, аналогичный запрос спросит есть ли какие-либо ресурсы в наборе данных с dm:amount, которые не имеют тип xsd:integer:

```
# filename: ex201.rq
PREFIX dm: <http://learningsparql.com/ns/demo#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
ASK WHERE
{
  ?item   dm:amount   ?amount .
```

```
  FILTER ((datatype(?amount)) != xsd:integer)
}
```

Значение 1.5 ресурса item693 для dm:amount соответствует этой модели и ARQ отвечает на этот запрос Ask => Yes.

Чуть более сложный запрос - проверка на соответствие бизнес-правила о необходимости согласований купли, но он сочетает в себе методы, о которых вы уже знаете: он использует OPTIONAL шаблон графа, потому что утверждение о покупке не требуется в любых условиях, и он использует ключевое слово BIND для расчета ?totalCost за каждую покупку, что нужно сравнивать с граничным значением100. Он также использует скобки и логические операторы && и ||, чтобы связать вместе все налагаемые условия:

```
# filename: ex202.rq
PREFIX dm: <http://learningsparql.com/ns/demo#>
ASK WHERE
{
  ?item   dm:cost        ?cost ;
          dm:amount   ?amount .
  OPTIONAL
  {
    ?item                    dm:approval    ?approvingEmployee .
    ?approvingEmployee  dm:jobGrade   ?grade .
  }
  BIND (?cost * ?amount AS ?totalCost) .
  FILTER ((?totalCost > 100) &&
          ( (!(bound(?grade)) || (?grade < 5 ) )))
}
```

ARQ также отвечает на этот запрос Ask => Yes.

Если бы вы проверяли набор данных на соответствие 40 правилам SPARQL, похожим на это, вы не захотели бы повторить трехступенчатый процесс чтения файла набора данных с диска, запуска на нем АЗП запроса и проверки результата 40 раз. При использовании процессора API SPARQL, такие как Jena API, или когда вы используете фреймворк для разработки продукта, вы найдете другие варианты эффективного контроля набора данных против крупной партии правил, выраженных в качестве запросов.

4.5.2. Генерация данных о нарушенных правилах

Иногда бывает удобно настроить что-то, что говорит вам о том, соответствует ли набор данных набору правил SPARQL или нет. Чаще, однако, если данные ресурса нарушают любые правила, вы хотите знать, какие ресурсы, какие правила нарушили.

Если приложение на основе RDF находит данные, которые нарушили определенные правила, и вы знаете, какие правила нарушены и где, то как бы лучше представить эту информацию? С помощью троек, конечно. Следующий вариант запроса ex199.rq идентичен оригиналу, за исключением того, что он включает в себя новую декларацию пространства имен и заменяет ключевое слово ASK на CONSTRUCT. Предложение CONSTRUCT имеет графовый шаблон из двух троек, чтобы оповестить, когда запрос находит проблему:

```
# filename: ex203.rq
```

```
PREFIX dm: <http://learningsparql.com/ns/demo#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
CONSTRUCT
{
  ?s   dm:problem   dm:prob29 .
  dm:prob29   rdfs:label   "Location value must be a URI." .
}
WHERE
{
  ?s   dm:location   ?city .
  FILTER (!(isURI(?city)))
}
```

Когда вы хотите описать что-то (в данном случае, что в исходных данных обнаружена ошибка) с помощью RDF, вы должны иметь идентификатор того, что вы описываете. Так, назначен идентификатор dm:prob29 ошибки - значение dm:location не является URI (первая тройка в теле CONSTRUCT). Далее использован классический подход RDF: назначено краткое описание ошибки как значение метки rdfs:label (вторая тройка в теле CONSTRUCT). в вторая тройка существо создан CONSTRUCT заявлении выше. Смотри раздел "3.1. Более читабельные результаты запроса" для получения подробной информации.

Выполненяя этот запроса с набором данных ex198.ttl, мы не просто спрашиваем есть ли где-то ошибка а значениях местоположения dm:location. Мы просим указать в каких ресурсах есть ошибка и что это за ошибка. Результат выполнения запроса ex203.rq дает нам эту информацию:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix dm: <http://learningsparql.com/ns/demo#> .

dm:prob29
        rdfs:label        "Location value must be a URI." .
d:item693
        dm:problem      dm:prob29 .
```

Как мы увидим в разделе "Использование существующих SPARQL словарей правил", должным образом моделируется словарный запас для идентификации проблемы объявления классов и для потенциальных проблем связанных с ними свойств. Каждый раз, когда запрос CONSTRUCT, который ищет эти проблемы, находит их - он объявляет новый экземпляр класса задачи и устанавливает соответствующие значения свойств. Сотрудничающие приложения могут использовать эту модель, чтобы выяснить, на что обращать внимание при использовании данных.

Следующий вариант запроса ex201.rq заменяет ключевое слово ASK на CONSTRUCT с графовым образцом из двух троек, чтобы создать, когда определить где проблема этого типа найдена:

```
# filename: ex205.rq
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dm: <http://learningsparql.com/ns/demo#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
CONSTRUCT
```

```
{
  ?item   dm:problem   dm:prob32 .
  dm:prob32   rdfs:label   "Amount must be an integer." .
 }
WHERE
{
  ?item   dm:amount   ?amount .
  FILTER ((datatype(?amount)) != xsd:integer)
}
```

Выполнение этого запроса показывает, какой ресурс имеет эту проблему и приводит описание проблемы:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix dm: <http://learningsparql.com/ns/demo#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

dm:prob32
        rdfs:label   "Amount must be an integer." .
d:item693
        dm:problem   dm:prob32 .
```

Наконец, вот наш последний запрос, где ASK заменен на CONSTRUCT, чтобы сообщить нам, какие ресурсы нарушили правило об утверждении расходов превышающих 100:

```
# filename: ex207.rq
PREFIX dm: <http://learningsparql.com/ns/demo#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
CONSTRUCT
{
  ?item   dm:problem   dm:prob44 .
  dm:prob44   rdfs:label   "Expenditures over 100 require grade 5 approval." .
}
WHERE
{
  ?item   dm:cost       ?cost ;
          dm:amount   ?amount .
   OPTIONAL
   {
     ?item   dm:approval   ?approvingEmployee .
     ?approvingEmployee   dm:jobGrade   ?grade .
   }
   BIND (?cost * ?amount AS ?totalCost) .
   FILTER ((?totalCost > 100) &&
           ( (!(bound(?grade)) || (?grade < 5 ) )))
}
```

Вот результаты этого запроса:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
```

```
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix dm: <http://learningsparql.com/ns/demo#> .

dm:prob44
        rdfs:label   "Expenditures over 100 require grade 5 approval." .

d:item857
        dm:problem    dm:prob44 .

d:item201
        dm:problem    dm:prob44 .
```

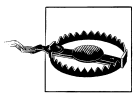
Чтобы проверить все три проблемы сразу, мы объединим три последних запросов в один, используя ключевое слово UNION. Используем разные имена переменных для хранения URI, из потенциально проблемных ресурсов, чтобы сделать связь между построенными запросами и подобранными моделями яснее. Добавим также метку о несуществующей проблеме dm:probXX просто чтобы показать, что все тройки появится на выходе были ли обнаружены проблемы или нет, потому что они жестко закодированы. Построенные тройки о проблемах, однако, появляются только тогда, когда проблемы обнаруживаются, то есть, когда двигатель SPARQL находит тройки, которые соответствуют условиям нарушения правил:

```
# filename: ex209.rq
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dm: <http://learningsparql.com/ns/demo#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
CONSTRUCT
{
?prob32item dm:problem dm:prob32 .
dm:prob32 rdfs:label "Amount must be an integer." .

?prob29item dm:problem dm:prob29 .
dm:prob29 rdfs:label "Location value must be a URI." .

?prob44item dm:problem dm:prob44 .
 dm:prob44 rdfs:label "Expenditures over 100 require grade 5 approval." .

dm:probXX rdfs:label "This is a dummy problem." .
}
WHERE
{
   {
     ?prob32item dm:amount ?amount .
     FILTER ((datatype(?amount)) != xsd:integer)
    }
  UNION
    {
     ?prob29item dm:location ?city .
      FILTER (!(isURI(?city)))
    }
  UNION
   {
    ?prob44item   dm:cost ?cost ;
                  dm:amount ?amount.
```

```
    OPTIONAL
    {
      ?item dm:approval ?approvingEmployee .
      ?approvingEmployee dm:jobGrade ?grade .
    }
    BIND (?cost * ?amount AS ?totalCost) .
    FILTER ((?totalCost > 100) &&
              ( (!(bound(?grade)) || (?grade < 5 ) )))

    }
}
```

Вот результат:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix d: <http://learningsparql.com/ns/data#> .
 @prefix dm: <http://learningsparql.com/ns/demo#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

dm:prob44
        rdfs:label"Expenditures over 100 require grade 5 approval." .
d:item432
        dm:problem dm:prob44 .
dm:probXX
        rdfs:label"This is a dummy problem." .
dm:prob29
        rdfs:label "Location value must be a URI." .
dm:prob32
        rdfs:label"Amount must be an integer." .
d:item857
        m:problem dm:prob44 .
d:item201
        dm:problem dm:prob44 .
d:item693
        dm:problem dm:prob29 ;
        dm:problem dm:prob32 .
```



Объединение нескольких правил SPARQL в одном запросе не очень хорошо масштабируется. Правильный инструмент проверки базовых правил обеспечивает способ для хранения правил отдельно, а затем их переправку, возможно, в различных комбинациях для различных наборов данных.

4.5.3. Использование существующих словарей SPARQL правил

Чтобы сохранить простыми объяснения этой книги, мы составим минимальные версии словарей которые мы будем использовать. Для серьезного приложения следует использовать существующие словари так же, как мы используем свойства vCard в реальной адресной книге. Для создания сообщений об ошибках и нарушениях ограничений в наборе данных на базе триплетов, можно использовать два словаря Schemarama и SPIN. Каждый из них был разработан для того, чтобы упростить разработку программного обеспечения для управления правилами SPARQL и нарушениями ограничений. Каждый из них включает бесплатное программное обеспечение, чтобы облегчить генерацию сообщениий об ошибках в виде троек.

Использование лексики Schemarama, запрос ex203.rq, который проверяет являются ли URI значения dm:location, может выглядеть следующим образом:

```
# filename: ex211.rq
PREFIX sch: <http://purl.org/net/schemarama#>
PREFIX dm: <http://learningsparql.com/ns/demo#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
CONSTRUCT
{
  []      rdf:type          sch:Error;
          sch:message       "location value should be a URI";
          sch:implicated    ?s.
}
WHERE
{
  ?s   dm:location   ?city .
  FILTER (!(isURI(?city)))
}
```

Этот запрос использует пару квадратных скобок, чтобы представлять пустой узел вместо префикса с подчеркиванием. Мы узнали о пустых узлах в главе 2. В этом случае, пустые узлы группируют воедино информацию об ошибке, найденной в данных.

Часть CONSTRUCT создает новый член класса Error Schemarama с двумя свойствами: сообщение об ошибке и триплет с указанием, какой ресурс имел проблему. Класс Error и его свойства являются частью онтологии Schemarama и утилита SPARQL-проверки с открытым исходным кодом, которая проверяет данные на эти правила, будет искать термины онтологии в ваших правилах. Действием утилиты по умолчанию является красиво отформатированный отчет об обнаруженных проблемах.

Можно выразить то же самое правило, используя словарный запас SPIN в этом запросе:

```
# filename: ex212.rq
PREFIX spin: <http://spinrdf.org/spin#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dm: <http://learningsparql.com/ns/demo#>
CONSTRUCT
{
   _:b0   a    spin:ConstraintViolation .
   _:b0   rdfs:comment    "Location value must be a URI" .
   _:b0   spin:violationRoot    ?this .
}
WHERE
{
  ?this   dm:location   ?city .
  FILTER (!isURI(?city)) .
}
```

Как вариант, который использует онтологию Schemarama, он создает член класса, который представляет нарушения. Это новый член класса spin:ConstraintViolation представлен пустым узлом в качестве субъекта и свойствами, которые описывают

проблему и указывают на ресурс имеющий проблему.

Обозначение и спецификация SPIN была представлена в W3C для потенциального развития в качестве стандарта. Свободное и коммерческое программное обеспечение в настоящее время доступны для использования SPIN правил.

Мы видели ранее, что SPARQL служит не только для запросов данных, хранящихся в RDF. Это означает, что вы можете написать запросы CONSTRUCT, чтобы проверить другие виды данных на соответствие правилам, таких как реляционные данные, переданные в двигатель SPARQL через соответствующий интерфейс. Это может быть довольно ценным поскольку есть много реляционных данных!

4.6. Запрос описания ресурса

Ключевое слово DESCRIBE служит для описания конкретного ресурса, и в соответствии со спецификацией SPARQL 1.1: "Описание определяется сервисом запросов." Другими словами, процессор запросов SPARQL будет решать, какую информацию он хочет вернуть, когда вы отправите запрос DESCRIBE, так что вы можете получить различные виды результатов из разных процессоров.

Например, следующий запрос спрашивает о ресуре http://learningsparql.com/ns/data#course59:

```
# filename: ex213.rq
DESCRIBE <http://learningsparql.com/ns/data#course59>
```

Набор данных в файле ex069.ttl включает в себя один триплет, где этот ресурс (там он обозначен d:course59) является субъектом и три триплета, где он объект. Когда мы просим ARQ, чтобы выполнить запрос выше против этого набора данных, мы получаем такой ответ:

```
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix ab: <http://learningsparql.com/ns/addressbook#> .

d:course59
        ab:courseTitle    "Using SPARQL with non-RDF Data" .
```

Другими словами, возвращается триплет, где этот ресурс является субъектом. Согласно документации программы по DESCRIBE: "ARQ позволяет доменно-ориентированным обработчикам самим формировать описание".

С другой стороны, когда мы отправляем следующий запрос, чтобы DBpedia, он возвращает все тройки, которые содержат наименование ресурса либо как субъект, либо как объект:

```
# filename: ex215.rq
DESCRIBE <http://dbpedia.org/resource/Joseph_Hocking>
```

Запрос DESCRIBE может не быть таким простым. Вы можете передать ему более одного ресурса URI, написав запрос, который связывает несколько значений переменной, а затем просит процессор запросов описать эти значения. Например, когда вы запустите следующий запрос ARQ к данным ex069.ttl, он опишет d:course59 и d:course85, и поскольку это ARQ, она возвращает все тройки, которые имеют эти ресурсы в качестве субъектов. Эти два курса, которые были прослушаны человеком, представленным идентификатором d:i0432 (Ричард Матт):

```
# filename: ex216.rq
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX ab: <http://learningsparql.com/ns/addressbook#>

DESCRIBE ?course WHERE
{ d:i0432    ab:takingCourse    ?course . }
```

При проверке обработчика SPARQL стоит опробовать один или два запроса DESCRIBE, чтобы получить лучшее представление о возможностях этого обработчика.

4.7. Резюме

В этой главе мы узнали:

• Как первое ключевое слово после объявлений префиксов в запросе SPARQL, называется формой запроса, и кроме SELECT есть еще три: DESCRIBE, ASK и CONSTRUCT

• Как запрос CONSTRUCT можно скопировать тройки из набора данных

• Как вы можете создавать новые тройки с CONSTRUCT

• Как запрос CONSTRUCT позволяет преобразовывать данные, использующие один словарь в данные, которые использует другой

• Как запросы ASK и CONSTRUCT могут помочь определить данные, не соответствующие правилам, которые вы укажете

• Как запрос DESCRIBE может попросить процессор SPARQL описать ресурс, и как различные процессоры могут по-разному реагировать на этот запрос для одного и того же ресурса в одном и том же наборе данных

Глава 5. Типы данных и функции

В предыдущих главах мы уже видели использование типов данных и функций в запросах SPARQL. Это дублирование темы, потому что запросы часто используют функции, чтобы получить максимальную отдачу от типов данных. В этой главе мы будем смотреть на общую картину, какую роль играет эта тема в SPARQL и какие возможности она предоставляет:

"Типы данных и запросов"

RDF поддерживает определенный набор типов, а также настраевыемые типы, и ваши запросы SPARQL могут работать с обоими.

"Функции"

Функции позволяют узнать о входных данных, создавать новые значеня из них и осуществлять контроль над типизированными данными. В этом разделе мы будем смотреть на большинство функций, определенных в спецификации SPARQL.

"Функции расширения"

Реализации SPARQL часто добавляют новые функции, чтобы сделать легче ваше развитие. В этом разделе мы увидим, как воспользоваться ими и какие возможности они предлагают.

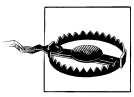
5.1. Типы данных и запросы

Представляет ли собой "197110" количество, почтовый код города, или стандарт ISO для финансовых услуг обмена сообщениями? Если мы знаем, что это целое число, то это, скорее всего, представляет собой количество. С другой стороны, если мы знаем, что это строка, то это, скорее всего идентификатор, например, почтовый индекс, идентификатор стандарта ISO или какой-то номер.

Десятилетия до семантической сети, запоминание типа данных, как некие метаданные, было одним из самых ранних способов для записи семантической информации. Знание этих дополнительных битов информации о куске данных дает лучшее представление о том, что вы можете делать с ним, и это позволяет строить более эффективные системы.

Различные языки программирования, языки разметки и языки запросов предлагают для выбора различные наборы типов данных. Когда консорциум W3C разработал спецификацию XML-схемы, они отделили часть об указании типов данных для элементов и атрибутов от части об указании элементов структуры. Типы данных, известные как "XML-схема часть 2: Типы данных", и часто сокращенно "XSD" - становятся все более популярными, нежели части 1 спецификации "Структуры". RDF использует Часть 2. Или, в более мудреных формулировках из RDF Концепций и Рекомендаций абстрактнго синтаксиса консорциума W3C: "Абстракция типов данных используемых в RDF совместимы с абстракцией, используемой в XML-схема часть 2: Типы данных".

Узел в графе RDF может быть одним из трех: URI, пустой узел или литерал. Если вы назначаете тип данных как значение xsd, то мы называем это типизированный литерал, в противном случае, это простой литерал.



В настоящее время в W3C обсуждается вопрос о том, чтобы покончить с понятием простой литерал и просто сделать тип xsd:string данных по умолчанию, так что "это" и "это" ^^ xsd:string будет означать одно и то же.

Согласно спецификации SPARQL, следующие основные типы данных являются, типизированными литералами:

- xsd:integer
- xsd:decimal
- xsd:float
- xsd:double
- xsd:string
- xsd:boolean
- xsd:dateTime

Другие типы, полученные от них, также доступны. Наиболее важным является производный тип xsd:date. (В XML-схемы Часть 2: Типы данных спецификации, это тоже примитивный тип.) Как следует из названия, это как xsd:dateTime, но вы используете его для значений, которые не включают в себя значение времени, например, "2014-10-13" вместо "2014-10-13T12:15:00".

Вот пример некоторых типизированных данных, которые мы видели в главе 2:

```
# filename: ex033.ttl
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .
@prefix d:    <http://learningsparql.com/ns/data#> .
@prefix dm:  <http://learningsparql.com/ns/demo#> .

d:item342       dm:shipped      "2011-02-14"^^<http://www.w3.org/2001/XMLSchema#date> .
d:item342       dm:quantity     "4"^^xsd:integer .
d:item342       dm:invoiced     "false"^^xsd:boolean .
d:item342       dm:costPerItem "3.50"^^xsd:decimal .
```

Вот пример тех же данных в RDF / XML:

```
<!-- filename: ex035.rdf -->

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
           xmlns:dm=http://learningsparql.com/ns/demo#
           xmlns:xsd="http://www.w3.org/2001/XMLSchema#">

<rdf:Description rdf:about="http://learningsparql.com/ns/demo#item342">
         <dm:shipped
        rdf:datatype="http://www.w3.org/2001/XMLSchema#date">2011-02-14</dm:shipped>
        <dm:quantity
        rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">4</dm:quantity>
        <dm:invoiced
        rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">false</dm:invoiced>
        <dm:costPerItem
        Description
        rdf:datatype="http://www.w3.org/2001/XMLSchema#decimal">3.50</dm:costPerItem>
        </rdf: >
</rdf:RDF>
```

Напомним несколько моментов касающихся типов данных, которые мы изучили:

• В формате Turtle идентификатор типа указывается после двух символов (^).
• В формате RDF / XML он хранится в rdf:datatype типа данных.
• Имя типа данных может быть полным URI, как показано в dm:shipped примера ex033.ttl.

• Это также может быть префиксное имя: имя с префиксом, используемом вместо URI представляющим пространство имен имя, как типы специфицированные значениями dm:quantity, dm:invoiced и dm:costPerItem представленные выше.

• Так же, как и любые другие префиксы в RDF тройках, префикс xsd: должен быть объявлен.

При выходе за кавычки литерала в формате Turtle, процессор делает определенные предположения о типе данных, если значение является словом "правда" или "ложь" или, если это число. Благодаря этому, значения в следующем примере будут интерпретироваться так же, как и в двух предыдущих примерах:

```
# filename: ex034.ttl
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix dm: <http://learningsparql.com/ns/demo#> .

d:item342       dm:shipped      "2011-02-14"^^<http://www.w3.org/2001/XMLSchema#date> .
d:item342       dm:quantity     4 .
d:item342       dm:invoiced     false .

d:item342       dm:costPerItem 3.50 .
```

Запрос ex201.rq из главе 4, воспроизведенный ниже - интересный пример того, как эти сокращенные способы представления типов могут быть использованы. Его FILTER искал значения, которые не имеют тип xsd:integer, и ни одно из значений в файле данных ex198.ttl не было явно определено как xsd:integer:

```
# filename: ex201.rq
PREFIX dm: <http://learningsparql.com/ns/demo#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
ASK WHERE
{
  ?item   dm:amount   ?amount .
  FILTER ((datatype(?amount)) != xsd:integer)
}
```

Обработчик запрос знал какие значения ?amount целыми, а какие нет, потому что любой без кавычек ряд цифр, в котором нет запятых трактуется как целое число.

Большая часть работы с типами данных в SPARQL будет включать в себя использование функций, которые более подробно описаны в следующем разделе. Прежде чем мы рассмотрим любую из них, хорошо будет узнать, как представления типизированных литералов в запросах взаимодействуют с различными видами литералов в наборе данных. Давайте посмотрим на то, что несколько запросов сделают со следующим набором данных:

```
# filename: ex217.ttl
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix dm: <http://learningsparql.com/ns/demo#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix mt: <http://learningsparql.com/ns/mytypesystem#> .
d:item1a dm:prop "1" .
d:item1b dm:prop "1"^^xsd:integer .
d:item1c dm:prop   1 .
d:item1d dm:prop  1.0e5 .
```

```
d:item2a dm:prop "two" .
d:item2b dm:prop "two"^^xsd:string .
d:item2c dm:prop "two"^^mt:potrzebies .
d:item2d dm:prop "two"@en .
```

Как вы думаете, какие пункты itemXX из ex217.ttl следующий запрос получит в результате? (Подсказка: Посмотрите ex033.ttl и ex034.ttl снова, имея в виду, что они представляют собой те же тройки).

```
# filename: ex218.rq
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX dm: http://learningsparql.com/ns/demo#

SELECT ?s
WHERE { ?s  ?p  1 . }
```

Если вы уже догадались, что d:item1b и d:item1c, то вы правы:

```
---------------
| s           |
===========
| d:item1c  |
| d:item1b  |
---------------
```

Значение d:item1a это строка "1", а потому, что объект в тройке запроса не указан в кавычках, он представляет целое число 1. Значение d:item1b заключено в кавычки, но оно имеет префикс xsd:integer после двух символов ^^, показывающий, что оно должно рассматриваться как целое.

При запуске с ARQ, следующий запрос возвращает d:item2a и d:item2b, даже если объект одной из этих троек включает обозначение типа xsd:string, а другой не делает этого:

```
# filename: ex220.rq
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>
SELECT ?s
WHERE { ?s  ?p  "two" . }
```

ARQ рассматривает их как равным "two", потому что, при работе с данными полностью хранящимися в памяти, механизм лежащий в основе Jena ARQ делает вывод о том, что нетипизированный литерал и литерал с тем же значением, введенной как xsd:string равны. (Он делает это на основе рекомендации консорциума W3C "RDF семантика", которая обеспечивает некоторые основы для возможного обновления RDF 1.1, в котором он станет официальным на более широкой основе для восприятия xsd:string как тип данных по умолчанию.) Вы можете посмотреть, что другие процессоры SPARQL, такие как те, которые используются с Fuseki и Sesame, не делают этот вывод при запросе ex217.ttl с ex220.rq и вернуть одно значение, что точно соответствует тому, что показана в триплете запроса.

Следующий запрос возвращает те же два результата при запуске с ARQ, но может вернуть одно значение, которое строго соответствует показанному в триплете запроса, с другими современными процессорами SPARQL:

```
# filename: ex221.rq
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>
SELECT ?s
WHERE { ?s   ?p   "two" ^^xsd:string . }
```

Число написанное с десятичной точкой и буквой «e», чтобы выразить экспоненту (например, 1.0e5 в ex217.ttl, которое представляет значение 100000) рассматривается как число двойной точности с плавающей запятой (xsd:double).

Остальные значения в ex217.ttl должны быть извлечены с явно указанными типами, с использованием либо полного URI, либо префикса имени URI типа данных. Например, следующий запрос получит только один пункт d:item2c:

```
# filename: ex222.rq
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX mt: <http://learningsparql.com/ns/mytypesystem#>

SELECT ?s
WHERE { ?s   ?p   "two"^^mt:potrzebies . }
```

Это интересный случай, потому что он использует ^^, чтобы указать, что значение имеет определенный тип, но это не тип xsd. RDF позволяет определить пользовательские типы данных для собственных нужд, а также этот запрос показывает, что SPARQL позволяет запрашивать их. (Мы узнаем, как запросить d:item2d, который имеетен тег en, показывающий, что знчение на английском языке, в разделе "Проверка, добавление и удаление тегов разговорного языка").

Систему мер и весов Potrzebie разработл ученый Дональд Кнут. Он опубликовал ее будучи подростком в журнале Mad Magazine в 1957 году, так что она не считается нормативной. Одному potrzebie равна толщина журнала Mad Magazine, выпуск № 26.

Использование не XSD типов в RDF в настоящее время является наиболее распространенным в данных на базе стандарта SKOS для контролируемых словарей. В SKOS концепция собственности skos:notation используется для обозначения идентификатора определяющего значение наследства от другого тезауруса и выражается в числовой последовательности (например, "920" - представляет биографии в библиотечном мире десятичной системы Дьюи). В отличие от skos:notation, концепция собственности skos:prefLabel использует более читаемое человеком имя. Например, версия AGROVOC SKOS, тезауруса терминологии продовольственной и сельскохозяйственной организации ООН о производстве пищевой продукции, объявляет подствойство skos:notation как asfaCode, защищая идентификатор данного термина от тезауруса водных наук и рыболовства (ASFA). Этот тезаурус объявляет специальный тип данных под названием ASFACode для значений этого свойства, так что вы знаете, что значение "asf4523" это не просто строка, а специализированный тип:

```
:asfaCode              rdfs:subPropertyOf     skos:notation
:an_agrovoc_uri        :asfaCode              "asf4523"^^:ASFACode
```

Другие примеры включают в себя пользовательские типы данных t:kilos и t:liters в спецификации SPARQL. В этих примерах префикс t: относится к пространству имен

http://example.org/types#, что означает, что это было сделано для целей примера.

Если вы хотите игнорировать типы и просто получить все со значением "two", вы можете просто указать в запросе рассматривать все как строки, используя функцию str(), о который мы узнаем в разделе "Функции преобразования типа узла":

```
# filename: ex223.rq
SELECT ?s
WHERE
{
  ?s   ?p   ?o .
  FILTER (str(?o) = "two")
}
```

5.1.1. Представление строк

Образцы строковых данных, которые мы видели до сих пор в файлах данных этой книги всегда были заключены в двойные кавычки, "как это". Вы также можете заключать строки в Turtle и SPARQL в апострофы или одинарные кавычки, 'как это'.

В RDF/XML представление строк символьных данных следует обычным правилам XML: они записываются, как символы между метками или заключены в одинарные или двойные кавычки в значениях атрибутов.

В Turtle и SPARQL, если вы начинаете и заканчиваете строку тремя двойными кавычками, процессоры RDF будут сохранить возврат каретки, что очень удобно для длинных блоков текста. (В SPARQL, но не в Turtle, вы можете сделать то же самое с тремя одинарными кавычками) . Если необходимо включить двойные кавычки как часть вашей SPARQL или Turtle строки, вы можете поставить обратную косую черту, а в SPARQL можно сделать то же самое с одной кавычкой.

Ниже показано несколько возможных способов для представления строк:

```
# filename: ex224.ttl
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

d:item1 rdfs:label "sample string 1" .
d:item2 rdfs:label 'sample string 2' .
d:item3 rdfs:label 'These quotes are "ironic" quotes.' .
d:item4 rdfs:label "These too are \"ironic\" quotes." .
d:item5 rdfs:label "McDonald's is not my kind of place." .
d:item6 rdfs:label """this

has two carriage returns in the middle.""" .
```

Этот простой запрос извлекает все субъекты и объекты из любого набора данных и показывает нам, как процессор SPARQL интерпретирует строки в выборочных данных представленных выше:

```
# filename: ex225.rq
PREFIX d: <http://learningsparql.com/ns/data#>
SELECT ?s ?o
```

```
WHERE { ?s  ?p  ?o }
```

При форматировании строк для вывода, ARQ использует двойные кавычки для определения строк. Он использует обратную косую черту в качестве экранирующего символа и представляет возврат каретки, как \r и перевод строки, как \n, другие процессоры SPARQL может сделать это по-другому:

```
-------------------------------------------------------------------------------
| s         | o                                                                 |
=================================================================
| d:item3 | "These quotes are \"ironic\" quotes."                                |
| d:item1 | "sample string 1"                                                    |
| d:item6 | "this\r\n\r\nhas two carriage returns in the middle." |
| d:item4 | "These too are \"ironic\" quotes."                                   |
| d:item2 | "sample string 2"                                                    |
| d:item5 | "McDonald's is not my kind of place."                                |
-------------------------------------------------------------------------------
```

Этот результат также напоминает нам, что, как строки таблицы реляционной БД, порядок набора троек не имеет значения, если только вы не сортируете их в вашем запросе с помощью ключевого слова ORDER BY. (Подробнее об этом смотри раздел "3.14.1. Сортировка данных").

5.1.2. Сравнение значений и выполнение арифметических операций

В разделе "3.8. Фильтрация данных на основе условий" показано, что операторы сравнения обеспечивают классический способ получения данных, основанных на определенных условиях. Для небольшого эксперимента мы будем использовать набор данных ex138.ttl из той же главы, но изменим его, включив метаданные к ресурсу e:date значения даты: ":00" добавляется в конце каждой даты-времени значение количества секунд, как того требует формат xsd:DateTime:

```
# filename: ex227.ttl
@prefix e: <http://learningsparql.com/ns/expenses#> .
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

d:m40392        e:description    "breakfast" ;
                e:date "2011-10-14T08:53:00"^^xsd:dateTime ;
                e:amount 6.53 .
d:m40393        e:description    "lunch" ;
                e:date "2011-10-14T13:19:00"^^xsd:dateTime ;
                e:amount 11.13 .
d:m40394        e:description    "dinner" ;
                e:date "2011-10-14T19:04:00"^^xsd:dateTime ;
                e:amount 28.30 .
```

Наш первый запрос запрашивает все данные для каждого элемента, который имеет значение e:amount меньше, чем 20:

```
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX e: <http://learningsparql.com/ns/expenses#>
```

```
PREFIX xsd: http://www.w3.org/2001/XMLSchema#

SELECT ?s ?p ?o
WHERE {
  ?s      e:amount        ?amount;
          ?p              ?o .
FILTER (?amount < 20)
}
```

Мы получаем данные, как вы уже догадались, для ресурсов d:m40393 и d:m40392:

```
-----------------------------------------------------------------------------------------
| s           | p             | o                                                       |
=========================================================================================
| d:m40393 | e:amount     | 11.13                                                      |
| d:m40393 | e:date       | "2011-10-14T13:19:00"^^xsd:dateTime |
| d:m40393 | e:description | "lunch"                                                   |
| d:m40392 | e:amount     | 6.53                                                       |
| d:m40392 | e:date       | "2011-10-14T08:53:00"^^xsd:dateTime |
| d:m40392 | e:description | "breakfast"                                              |
-----------------------------------------------------------------------------------------
```

Наш второй запрос запрашивает информацию о всех приемах пищи, которые имели место не ранее полдня 14 октября 2011 года:

```
# filename: ex230.rq
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX e: <http://learningsparql.com/ns/expenses#>
PREFIX xsd: http://www.w3.org/2001/XMLSchema#

SELECT ?s ?p ?o
WHERE {
  ?s      e:date   ?date;
          ?p       ?o .
FILTER (?date >= "2011-10-14T12:00:00"^^xsd:dateTime)
}
```

Он извлекает данные для ресурсов d:m40394 и d:m40393:

```
-----------------------------------------------------------------------------------------
| s           | p             | o                                                       |
=========================================================================================
| d:m40393 | e:amount     | 28.30                                                      |
| d:m40393 | e:date       | "2011-10-14T19:04:00"^^xsd:dateTime |
| d:m40393 | e:description | "dinner"                                                  |
| d:m40392 | e:amount     | 11.13                                                      |
| d:m40392 | e:date       | "2011-10-14T13:19:00"^^xsd:dateTime |
| d:m40392 | e:description | "lunch"                                                   |
-----------------------------------------------------------------------------------------
```

В разделе "3.12. Объединение значений и присвоение значений переменным" мы увидели как выполняется некоторые арифметические действия в следующем запросе, который вычислякт чаевые и суммарные затраты на эти блюда:

```
# filename: ex139.rq
```

```
PREFIX e: <http://learningsparql.com/ns/expenses#>
SELECT ?description ?amount ((?amount * .2) AS ?tip)
          ((?amount + ?tip) AS ?total)
WHERE
{
  ?meal e:description ?description ;
  e:amount ?amount .
}
```

Наряду с + для сложения, - для вычитания, и * для умножения, можно использовать / для деления.

Арифметические выражения особенно полезны, когда вы используете BIND, чтобы создать новое значение, как и в этом пересмотре запроса ex139.rq, который производит тот же результат при запуске на наборе данных ex138.ttl, который мы использовали с ex139.rq:

```
# filename: ex232.rq PREFIX e: <http://learningsparql.com/ns/expenses#>
SELECT ?description ?amount ?tip ?total
WHERE {
   ?meal          e:description     ?description ;
                  e:amount          ?amount .
   BIND ((?amount * .2) AS ?tip)
   BIND ((?amount + ?tip) AS ?total)
}
```

Когда для различных значений указаны разные числовые типы, вы все равно можете использовать их вместе при выполнении арифметических операций. Например, в наборе данных ex033.ttl в начале этой главы значение количество dm:quantity указано как xsd:integer, а значение dm:costPerItemcostPerItem, как xsd:decimal, но следующий запрос перемножает их вместе:

```
# filename: ex233.rq
PREFIX dm: <http://learningsparql.com/ns/demo#>
PREFIX d: <http://learningsparql.com/ns/data#>
SELECT *
}
WHERE {
  ?item          dm:quantity     ?quantity;
                 dm:costPerItem ?cost .
  BIND ( (?quantity * ?cost) as ?total )
}
```

ARQ без проблем обрабатывает этот запрос с данными ex033.ttl:

```
-------------------------------------------------
|  item        | quantity | cost  | total   |
=================================
| d:item342 | 4          | 3.50  | 14.00 |
-------------------------------------------------
```

5.2. Функции

Функции служат для выполнения математических расчетов, программной логики,

строковых операций и проверки выполнения определенных условий. Большинство функций SPARQL 1.0 из последней категории, потому что не имея эквивалента ключевого слова BIND из SPARQL 1.1, назначающего новые значения переменных, было мало оснований включают функции, которые манипулируют входными значениями и возвращают новые значения.

Из-за этого, оригинальные функции 1.0, которые спецификация называет «тестовые функции", служат для ответа на конкретный вопрос о значении. Использование FILTER обеспечивает более гибкое управление результатами запросов. Большинство из функций являются логическими, такие как regex(), и функции 1.0, которые не являются логическими, а отвечают на вопросы о характеристиках переданного им аргумента, таких как тип данных или тег какого языка может быть назначен для указанной строки.

SPARQL спецификации говорят нам, что наряду со встроенными функциями для значений тестирования "*SPARQL обеспечивает возможность вызова произвольных функций, в том числе подмножество функций кастинга XPath*". (Термин "*кастинг*" здесь относится к преобразованию одного типа данных в другой). Фраза "*возможность вызова произвольных функций*" означает, что процессор SPARQL может предложить любую функцию расширения, которую его исполнители захотят включить.

В W3C XPath язык дает способ описания наборов узлов в представлении структуры дерева XML-документа. Например, выражение XPath может ссылаться на все родственные узлы, которые предшествуют родительскому узелу текущего, так что, когда процессор, такой как XSLT обработчик анализирует XML документ, он может брать значения из этих узлов для обработки. Оригинал "1999 XPath рекомендации" определяет этот язык и некоторые функции по работе со значениями. XPath 2.0 определяет гораздо больше функций и новая XQuery спецификация ссылается на многие из них, так что W3C "XQuery 1.0 и XPath 2.0 функции и операторы" исключает из в собственной спецификации.

Спецификация SPARQL 1.0 описывает несколько основных функций, и SPARQL 1.1 предлагает широкий выбор из них, почти все из которых основаны на функциях XPath. Они не всегда имеют то же название, например, то что SPARQL 1.1 спецификация называет функция минут minutes() соответствует функции XPath fn:minutes-from-dateTime.

В спецификации SPARQL, некоторые имена функций пишутся прописными буквами, некоторые в нижнем регистре, а некоторые в смешанном виде. Мы будем писать имена функций так, как это написано в спецификации, хотя процессору SPARQL написание безразлично. Например, он не делает никакой разницы напишете ли вы функцию подстроки SUBSTR, как substr() или SUBSTR().

5.2.1. Логические функции

Функции IF() и COALESCE() оценивают одно или несколько выражений и возвращают значения, основанные на том, что они находят. Это позволяет упаковать много программной логики в краткие выражения.

Важно

Ключевые слова IF() и COALESCE() являются новой возможностью в SPARQL 1.1.

Функция IF() имеет три аргумента. Если первое условие выполняется, функция возвращает значение второго аргумента, в противном случае третьего.

Вот простой пример, который вы можете запустить без какого-либо входного файла, потому что никакие шаблоны в запросе не описаны и входные данные, будут

игнорироваться:

```
# filename: ex235.rq
SELECT ?answer
WHERE
{
  BIND ((IF (2 > 3, "Two is bigger","Three is bigger")) AS ?answer)
}
```

Функция IF() будет связываться либо со строкой "Two is bigger", либо с "Three is bigger", в зависимости от истинности выражения 2> 3. Результат очевиден:

```
-------------------------
| answer                |
=========================
| "Three is bigger"     |
-------------------------
```

Все три параметра могут быть сколь угодно сложными выражениями. Следующий пример делает более реальную работу с помощью нескольких других функций, описанных далее в этой главе. Для каждого значения местоположения dm:location в примере ex104.ttl набора данных из главы 3, создадим новую тройку, которая говорит о том, что значение является экземпляром класса dm:Place:

```
# filename: ex104.ttl
@prefix dm: <http://learningsparql.com/ns/demo#> .
@prefix d: <http://learningsparql.com/ns/data#> .

d:item432       dm:cost 8 ;
                dm:location     <http://dbpedia.org/resource/Boston> .
d:item857       dm:cost 12 ;
                dm:location     <http://dbpedia.org/resource/Montreal> .
d:item693       dm:cost 10 ;
                dm:location     "Heidelberg" .
d:item126       dm:cost 5 ;
                dm:location     <http://dbpedia.org/resource/Lisbon> .
```

В соответствие с конвенцепцией популярных объектно-ориентированных языков, имена классов RDF таких, как dm:Place, как правило, начинаются с заглавной буквы, а имена свойств, таких как dm:location начинаются с маленькой буквы.

Если мы создадим тройки из набора данных ex104.ttl со значениями местоположения dm:location в качестве субъектов, то подойдут три из этих четырех ресурса. Не подойдет dm:location "Heidelberg" в качестве значения, потому что является строкой, а субъектом тройки должен быть URI. Чтобы обойти это, следующий запрос проверяет является ли значение dm:location правильным URI, и если нет, то создает его из строки:

```
# filename: ex237.rq
PREFIX dm: <http://learningsparql.com/ns/demo#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

```
CONSTRUCT { ?locationURI    rdf:type    dm:Place . }
 WHERE
{
  ?item    dm:location    ?locationValue .
  BIND (IF(isURI(?locationValue),
                  ?locationValue,
                  URI(CONCAT("http://learningsparql.com/ns/data#",
                  ENCODE_FOR_URI(?locationValue)))
            ) AS ?locationURI
          ).
  }
```

В функциональных выражениях, вы можете вставлять пробелы там где вы хотите до и после скобок и запятых, разделяющих отдельные параметры. Это делает сложные выражения проще для чтения.

После того как запрос связывает каждое значение местоположения dm:location с переменной ?locationValue в первой тройке модели, он будет привязывать результаты функции IF() к переменной ?locationURI. Функция IF() имеет три параметра:

1. Первое выражение использует функцию isURI (), о которой мы узнем больше в следующем разделе, чтобы проверить является ли ?locationValue настоящим URI.

2. Если значение выражения первого параметра истинно, то ?locationValue будет работать в качестве субъекта создаваемой тройки, так что второй аргумент функции IF() будет связан с ?locationURI.

3. Если значение выражения первого параметра ложь, функция возвращает третий параметр: выражение, которое создает URI из locationValue?.

Функция ENCODE_FOR_URI() экранирует символы, которые могут вызвать проблемы в пути URI. Так для для "Heidelberg" не нужно ничего предпринимать но, например, для строки “Los Angeles” пробел может быть проблемой и ENCODE_FOR_URI() будет конвертировать эту строку в “Los%20Angeles”. Функция CONCAT() объединяет значение, возвращенное ENCODE_FOR_URI() со строкой "http://learningsparql.com/ns/data#", а затем функция URI() (также описываемой в следующем разделе) преобразует результат в URI.

Результат запроса формирует тройку для каждого из четырех субъектов во входном документе:

```
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix dm: <http://learningsparql.com/ns/demo#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

<http://dbpedia.org/resource/Boston>          rdf:type      dm:Place .
d:Heidelberg                                  rdf:type      dm:Place .
<http://dbpedia.org/resource/Montreal>        rdf:type      dm:Place .
<http://dbpedia.org/resource/Lisbon>          rdf:type      dm:Place .
```

Мы создали URI для Heidelberg в пространстве имен http://learningsparql.com /ns/data#", а не в http://dbpedia.org/resource/, потому что новые URI должны использовать пространства имен, которые у вас под контролем над, а не пространства построенные вокруг чужого домена.

Следует подчеркнуть, что любую из функций, о которых мы узнали, можно использовать внутри аргументов любой из них, в том числе включая вызовы IF() внутри аргументов предыдущей функции IF().

Логические значения могут быть объединены в SPARQL запросах с помощью оператора && для логического «и» и || для логического «или».

Функция COALESCE() имеет корни в мире SQL. Задайте ей столько параметров, сколько хотите, и она будет возвращать первый, который не приводит к ошибке. Это хороший способ сказать "попробуйте это, и если это не работает, попробуйте это, и если это не работает ...". В запросе SPARQL выражения параметров, которые могут или не могут работать с переменными, в зависимости от того, является ли они связанными со структурой анализируемых данных.

Например, в разделе "3.2. Данные, которых может не быть" мы видели один способ использовать ключевое слово OPTIONAL, чтобы получить значение ab:nick для каждого человека в следующих данных, если оно есть, и если это не так, чтобы получить значение ab:firstName, вместо этого:

```
# filename: ex054.ttl
@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d: <http://learningsparql.com/ns/data#> .

d:i0432 ab:firstName    "Richard" .
d:i0432 ab:lastName     "Mutt" .
d:i0432 ab:homeTel      "(229) 276-5135" .
d:i0432 ab:nick         "Dick" .
d:i0432 ab:email        "richard49@hotmail.com" .

d:i9771 ab:firstName    "Cindy" .
d:i9771 ab:lastName     "Marshall" .
d:i9771 ab:homeTel      "(245) 646-5488" .
d:i9771 ab:email        "cindym@gmail.com" .

d:i8301 ab:firstName    "Craig" .
d:i8301 ab:lastName     "Ellis" .
d:i8301 ab:workTel      "(245) 315-5486" .
d:i8301 ab:email        "craigellis@yahoo.com" .
d:i8301 ab:email        "c.ellis@usairwaysgroup.com" .
```

Следующий запрос связывает переменную ?first со значением ab:firstName и пытается привязать переменную ?nickname к значению ab:nickник, если оно есть. Функция COALESCE(), то возвращает ?nickname, если это возможно, а в противном случае возвращает первое значение ?first. Возвращаемое значение связывается со значением ?firstName для вывода:

```
# filename: ex239.rq PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT ?firstName ?last WHERE {
?s ab:lastName ?last; ab:firstName ?first .
OPTIONAL { ?s ab:nick ?nickname .
}
```

```
BIND (COALESCE(?nickname,?first) AS ?firstName) }
```

Результат показывает, что для ресурсов D:i0432, запрос нашел и использовал значение ab:nick псевдоним "Дик". Для каждого из других ресурсов, он использовал значение ab:firstName, потому что они не имеют значения ab:nick:

```
-------------------------------------------
| firstName      | last              |
===========================
| "Craig"        | "Ellis"           |
|"Cindy"         | "Marshall"        |
|"Dick"          | "Mutt"            |
-------------------------------------------
```

В приведенном выше примере использовано только два параметра в функции COALESCE(), но вы можете добавить столько, сколько необходимо. Кроме того, с помощью функции IF(), вы можете передать более сложные выражения в качестве параметров.

5.2.2. Функции проверки типов узлов и данных

Некоторые функции ожидают, что определенные их параметры будут конкретного типа. Например, вы не можете попросить функцию round() округлить строку "Hello" до ближайшего целого числа. Ваше приложение может также ожидать определенные части данных конкретного типа: если вы собираетесь сложить общую сумму, потраченную на завтраки в наборе данных отчета о расходах, как ex145.ttl, то вам надо убедиться, что каждое значение e:amount представляет собой фактическое количество, а не строку типа «5 баксов».

Для этих целей SPARQL предлагает функции проверки того, можно ли выражение квалифицировать как URI, литерал, нумерованный литерал, числовое или пустой узел. Если значение является типизированным литералом, то функция datatype() позволяет узнать его тип. Эти функции особенно ценны в правилах качества данных, которые вы создаете, чтобы определить неверные данные, как описано в разделе "4.5. Поиск неверных данных".

Функции для проверки типов узлов и типов данных также удобно использоват в директивах FILTER, если вы хотите, чтобы запрос получал тройки при соблюдении определенных условий.

Функции, которые проверяют конкретный тип узла или тип данных имеют имя, которое начинается с "is - есть" (например, IsNumeric()), и они возвращают логическое значение ИСТИНА или ЛОЖЬ. Давайте попробуем использовать эти функции со следующей выборкой данных:

```
# filename: ex241.ttl
@prefix dm: <http://learningsparql.com/ns/demo#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

d:id1 dm:location       _:b1 .
```

```
d:id2 dm:location       <http://dbpedia.org/resource/Montréal> .
d:id3 dm:amount         3 .
d:id4 dm:amount         "4"^^xsd:integer .
d:id5 dm:amount         1.0e5 .
d:id6 rdfs:label        "5 bucks" .
d:id7 dm:shipped        true .
```

Чтобы проверить тип данных в каждом из объектов этих троек, следующий запрос устанавливает значения несколько переменных в зависимости от того, что функции isBlank(), isLiteral(), isNumeric(), isIRI() и isURI() возвращают для соответствующей переменной:

```
# filename: ex242.rq
PREFIX dbr: <http://dbpedia.org/resource/>
SELECT ?o ?blankTest ?literalTest ?numericTest ?IRITest ?URITest
WHERE
{
  ?s ?p ?o .
  BIND (isBlank(?o) as ?blankTest)
  BIND (isLiteral(?o) as ?literalTest)
  BIND (isNumeric(?o) as ?numericTest)
  BIND (isIRI(?o) as ?IRITest)
  BIND (isURI(?o) as ?URITest)
}
```

Результатом является следующая таблица:

```
-------------------------------------------------------------------------------------------
| o              | blankTest | literalTest | numericTest | IRITest | URITest |
=============================================================
| 3              | false     | true        | true        | false   | false   |
| _:b0           | true      | false       | false       | false   | false   |
| "5 bucks"      | false     | true        | false       | false   | false   |
| 4              | false     | true        | true        | false   | false   |
| dbr:Montréal | false     | false       | false       | true    | false   |
| true           | false     | true        | false       | false   | false   |
| 1.0e5          | false     | true        | true        | false   | true    |
 -------------------------------------------------------------------------------------------
```

Важно

Из функций продемонстрированных выше только одна isNumeric() являются новой в SPARQL 1.1.

Есть несколько интересных вещей, касающихся результатов, которые следует отметить:

• Числа, строки, и ключевые слова true и false (написанные в нижнем регистре) все литералы. Только URI, и пустые узлы не являются ими.

• Нет разницы между isIRI() и isURI(). Они синонимы, но IRI является более технически правильным термином, а URI является наиболее часто используемым термином. Хотя http://dbpedia.org/resource/Montréal на самом деле не URI, isURI()

возвращает истину для него так же, как это делает isIRI().

Ранее в разделе "5.2.1. Логические функции программы» был хороший пример, где функция isURI() проверяла, нуждается ли строка быть преобразованной в URI.

Ни RDFS, ни OWL не обеспечивают явный способ сказать, что значение определенного свойства должны быть одного типа, но теперь у вас есть способ проверить тип с помощью этих функций и правилами SPARQL, описанными в главе 4.

Функция datatype() возвращает URI идентифицирующий тип данных переданного параметра. Следующий запрос говорит нам тип данных объекта каждой тройки, которую он читает:

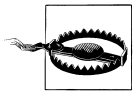
```
# filename: ex244.rq
PREFIX dbr: <http://dbpedia.org/resource/>
SELECT ?o ?datatype
WHERE
{
  ?s   ?p   ?o .
  BIND (datatype(?o) as ?datatype)
 }
```

Выполнение этого запроса с данными ex241.ttl, дает нам следующий результат:

```
-------------------------------------------------------------------------------------
| o              | datatype                                                         |
=====================================================================
| 3              | <http://www.w3.org/2001/XMLSchema#integer>   |
| _:b0           |                                                                  |
| "5 bucks"      | <http://www.w3.org/2001/XMLSchema#string>    |
| 4              | <http://www.w3.org/2001/XMLSchema#integer>   |
| dbr:Montréal |                                                                  |
| true           | <http://www.w3.org/2001/XMLSchema#boolean> |
| 1.0e5          | <http://www.w3.org/2001/XMLSchema#double>  |
-------------------------------------------------------------------------------------
```

Для каждого значения с определяемым типом данных результат запроса имеет URI XML Schema Part 2 для этого типа данных. Для URI, и пустых узлов он ничего не показывает.

Две других полезных функции для проверки переменных bound() и sameTerm(). Функция bound() говорит нам, имеет ли переменная значение, связанное с ним. Вы можете найти некоторые классические примеры того, как эта функция используется в разделе "3.3. Поиск данных, которые не отвечают определенным условиям". Функция sameTerm() возвращает булево значение, говоря нам о том, что два аргумента являются одинаковыми или нет. Мы увидим некоторые примеры ее использования в главе 7.



Функция sameTerm() возвращает истинное значение, если два аргумента являются идентичными термами, а не имеют то же самое значение. Так, в то время как sameTerm(4,4) возвращает истину, sameTerm(4,4.0) возвратит ложь.

Наибольшая полезность функции sameTerm() проявляется при сравнении двух

переменных, хранящих URI. Вызов sameTerm(?Var1,?Var2), по существу, совпадает с выражением (?Var1 = ?Var2). Если эта функция делает то же самое, что делает более очевидное выражение, то зачем она нужна? Дело в том, что с большинством обработчиков SPARQL, sameTerm() работает более эффективно и ваш запрос будет работать немного быстрее.

5.2.3. Функции преобразования типа узла

SPARQL предлагает функции для преобразования (или "замены") типа узла, и не только между Schema Part 2 XML-типами данных, таких как xsd:string и xsd:integer, но и между RDF типами узлов, строками и URI. Эти функции не могут творить чудеса, такие как замена строки "5 баксов" на целое число, но они могут заменить строку "5" на число 5 и строку "http://www.learningsparql.com" на URI.

Мы видели в главе 2, что, хотя объект тройки может быть либо URI, либо литералом, хотя лучше быть URI, потому что это может служить субъектом для других троек. Когда же URI является объектом некоторых троек и субъектом других, вы можете связать эти тройки, сделать логический вывод и получить больше информации от ваших данных.

Функция **URI()** (синоним для функции **IRI()**, как и isURI() является синонимом для isIRI()) позволяет конвертировать значения URI, если это возможно. Следующий запрос копирует входные тройки на выход, заменяя URI () версию объекта:

```
# filename: ex246.rq
BASE <http://learningsparql.com/ns/demo#>
CONSTRUCT {?s ?p ?testURI.}
WHERE
{
  ?s   ?p   ?o .
  BIND (URI(?o) AS ?testURI)
}
```

Давайте посмотрим на то, что этот запрос делает с входом ex241.ttl прежде чем обсуждать, как это работает:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix dm: <http://learningsparql.com/ns/demo#> .

d:id2    dm:location     <http://dbpedia.org/resource/Montréal> .
d:id1    dm:location     <_:5d37e0ec:12fd6bd6a74:-7ffe> .
d:id6    rdfs:label      <http://learningsparql.com/ns/5 bucks> .
```

Вот что произошло! Имейте в виду, что спецификация SPARQL 1.1 говорит нам: "Преобразование любого терма RDF [IRI() или URI() функцией], в отличие от простого литерала, xsd:string или IRI приводит к ошибке":

• Запрос не смог преобразовать целые значения объектов из d:id3 и d:id4 тройки или логическое значение из d:id7 к URI, так как для них нет выходных троек. (Эти значения являются не простыми литералами, а типизированными).

• Запрос преобразовал пустой узел с d:id1 в ... ну, это на самом деле не имеет значения, потому что пустой узел не является литералом или URI.

• URI из d: id2 вышел без изменений.

• Для преобразования строки "5 баксов" в URI, процессор обрабатывает его как относительной URI. По отношению к чему? Вначале запрос определяет базу URI с

ключевым словом BASE, при его отсутсвии ARQ формирует базу URI как file:/// и далее каталог, где находится файл запроса. Вы думаете, что добавление "5 баксов" к этой базе приведет к URI http://learningsparql.com/ns/demo#5 bucks, но потому, что функции URI() и IRI() в результате "должны привести к абсолютному IRI", в соответствии со спецификацией (таким образом, чтобы исключалась возможность использования знака фунта), он заканчивается как http://learningsparql.com/ns/5 баксов.

Важно

Функции URI() и IRI() являются новыми для SPARQL 1.1.

URI http://learningsparql.com/ns/5 баксов не может понравиться из-за наличия в нем пробела. Следующий вариант запроса использует функцию ENCODE_FOR_URI(), которую мы использовали в разделе "5.2.1. Логические функции", чтобы избежать каких-либо URI-недружественных символов, таких как пробел в «5 баксов". Функция URI() преобразует результат вызова функции ENCODE_FOR_URI() к URI:

```
# filename: ex248.rq
BASE <http://learningsparql.com/ns/demo#>
CONSTRUCT {?s ?p ?testURI.}
WHERE
{
  ?s   ?p   ?o .
  BIND (URI(ENCODE_FOR_URI(?o)) AS ?testURI)
}
```

Если вы передадите что-нибудь, кроме литерала или xsd:string функции ENCODE_FOR_URI(), ARQ выдает ошибку. Таким образом, чтобы проверить ex248.rq сделаем альтернативную ex241.ttl версию входного набора данных, который не имеют d:id1 и d:id2:

```
# filename: ex249.ttl
@prefix dm: <http://learningsparql.com/ns/demo#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

d:id3 dm:amount       3 .
d:id4 dm:amount       "4"^^xsd:integer .
d:id5 dm:amount       1.0e5 .
d:id6 rdfs:label      "5 bucks" .
d:id7 dm:shipped      true .
```

Результат игнорирует числовые и логические значения и вставляет ASCII код пробела:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix dm: <http://learningsparql.com/ns/demo#>.
```

```
d:id6    rdfs:label        <http://learningsparql.com/ns/5%20bucks> .
```

Перед передачей параметров функции URI() или IRI() хорошо подготовить его с помощью функции ENCODE_FOR_URI (), но обязательно надо убедиться, что ей не передается ничего, кроме простых литералов или xsd:string значений.

Функция str() возвращает строковое представление аргумента, переданного ей. Следующий запрос проходит объект каждой тройки, считывает его функцией str() и сохраняет результат в переменной ?testStr:

```
# filename: ex251.rq
PREFIX d: <http://learningsparql.com/ns/data#>
SELECT ?s ?testStr
WHERE
{
  ?s   ?p   ?o .
  BIND (str(?o) AS ?testStr)
}
```

При запуске с набором данных ex241.ttl, запрос дает нам следующий результат:

```
-------------------------------------------------------------------
| s      | testStr                                        |
===================================================
| d:id3 | "3"                                             |
| d:id1 |                                                 |
| d:id6 | "5 bucks"                                       |
| d:id4 | "4"                                             |
| d:id2 | "http://dbpedia.org/resource/Montréal" |
| d:id7 | "true"                                          |
| d:id5 | "1.0e5"                                         |
-------------------------------------------------------------------
```

Запрос работает довольно просто: он не возвращает ничего, если встречает пустой узел и строковое представление для чего-либо другого.

Это выглядит довольно просто, но может быть очень полезно, особенно в сочетании с функциями, описанными в разделе «Строковые функции". Например, раньше мы создавали набор данных ex249.ttl в качестве альтернативы набору ex241.ttl, потому что функция ENCODE_FOR_URI() ожидает строковый параметр, а некоторые из значений ex241.ttl не строки и, следовательно, это приведет к ошибке, если передать их этой функции. Следующий вариант ex251.rq «обертывает» параметр передаваемый в функцию ENCODE_FOR_URI() функцией str() так, что нестроковые значения не вызывают ошибки:

```
# filename: ex253.rq
BASE <http://learningsparql.com/ns/demo#>
CONSTRUCT {?s ?p ?testURI.}
WHERE
{
  ?s   ?p   ?o .
  BIND (URI(ENCODE_FOR_URI(str(?o))) AS ?testURI)
}
```

Когда мы выполним этот запрос с данными ex241.ttl, мы увидим, что процессор SPARQL ничего не сделал с пустым узлом в тройке d:id1 и сделал предсказуемые преобразования всего остального, за исключением, может быть, значения URI в тройке d:id2:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix dm: <http://learningsparql.com/ns/demo#> .

d:id5    dm:amount      <http://learningsparql.com/ns/1.0e5> .
d:id3    dm:amount      <http://learningsparql.com/ns/3> .
d:id2    dm:location    <http://learningsparql.com/ns/demo#http%3A%2F%2F
                         dbpedia%2Eorg%2Fresource%2FMontréal> .
d:id4    dm:amount      <http://learningsparql.com/ns/4> .
d:id6    rdfs:label     <http://learningsparql.com/ns/5%20bucks> .
d:id7    dm:shipped     <http://learningsparql.com/ns/true> .
```

Как же получилось, что URI в конечном итоге так выглядит? Давайте пошагово рассмотрим три функции, выполненные над URI http://dbpedia.org/resource/Montréal в BIND строке запроса ex253.rq в том порядке, в котором они выполнялись:

1. Функция str() превратила его в строку "http://dbpedia.org/resource/Монреаль".

2. Функция ENCODE_FOR_URI(), которая ожидает простой литерал в качестве входного параметра, преобразовала все символы, которые могут вызвать проблемы, если строка будет использована в части пути URI. Она сделала это путем преобразования каждого из этих символов в ASCII код.

3. Функция URI() не знала, что это значение начиналось как URI, и думала, что оно было очередной строкой, так что добавила результат вызова функции (ENCODE_FOR_URI) к базе URI объявленной в начале запроса, как запрос ex246.rq сделал со строкой "5 баксов".

Как мы можем сказать процессору запросов, чтобы он не делать все это, если значение объекта уже URI? Путем использования функций IF() и () isURI, которые мы изучили ранее в этой главе:

```
# filename: ex255.rq
BASE <http://learningsparql.com/ns/demo#>
CONSTRUCT {?s   ?p   ?testURI.}
WHERE
{
   ?s   ?p   ?o .
  BIND( IF(isURI(?o),
           ?o,
           URI(ENCODE_FOR_URI(str(?o)))
         ) AS ?testURI
      )
 }
```

В этом запросе, сначала проверяется параметр функции IF() является ли он URI, и если это так, возвращает второй параметр: ?o, без изменений. Если это не URI, третий параметр, который делает все преобразования и мы получаем разумный выход для всех данных в ex241.ttl:

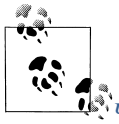
```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
```

```
@prefix d: <http://learningsparql.com/ns/data#> .
 @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix dm: <http://learningsparql.com/ns/demo#> .

d:id5   dm:amount    <http://learningsparql.com/ns/1.0e5> .
d:id3   dm:amount    <http://learningsparql.com/ns/3> .
d:id2   dm:location  <http://dbpedia.org/resource/Montréal> .
d:id4   dm:amount    <http://learningsparql.com/ns/4> .
d:id6   rdfs:label   <http://learningsparql.com/ns/5%20bucks> .
d:id7   dm:shipped   <http://learningsparql.com/ns/true> .
```

5.2.4. Преобразование типов данных

При преобразовании данного типа в другой, если вы знаете тип, в который хотите конвертировать, вы знаете, функцию, потому что ее имя совпадает с наименованием типа. Ниже приведен список функций, который должен быть знаком - это просто копия списка типов данных из начала этой главы с добавленными скобкми:

- xsd:integer()
- xsd:decimal()
- xsd:float()
- xsd:double()
- xsd:string()
- xsd:boolean()
- xsd:dateTime()



Чтобы быть последовательным с использованием схемы Часть 2 XML-типов данных, SPARQL использует функции преобразования из спецификации XPath.

Давайте попробуем преобразовать объекты троек из ex241.ttl к четырем числовым типам с помощью следующего запроса:

```
# filename: ex257.rq
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?o ?integerTest ?decimalTest ?floatTest ?doubleTest
WHERE
{
  ?s   ?p   ?o .
  BIND (xsd:integer(?o) as ?integerTest)
  BIND (xsd:decimal(?o) as ?decimalTest)
  BIND (xsd:float(?o) as ?floatTest)
  BIND (xsd:double(?o) as ?doubleTest)
}
```

Результат не принес больших сюрпризов. Преобразовалось то, что могло и остался совсем не тронутым пустой узел и URI:

```
-------------------------------------------------------------------------------------------------------------------
| o                |integerTest         | decimalTest         | floatTest                | doubleTest       |
===================================================================================================================
|3                 |3                   |"3"^^xsd:decimal     |"3"^^xsd:float            | "3"^^xsd:double  |
| _:b0             |                    |                     |                          |                  |
| "5 bucks"        |                    |                     |                          |                  |
|4                 | 4                  | "4"^^xsd:decimal    |"4"^^xsd:float            | "4"^^xsd:double  |
| dbr:Montréal     |                    |                     |                          |                  |
| true             | 1                  | 1.0                 |"1.0E0"^^xsd:float        |1.0E0             |
| 1.0e5            |                    |                     |"1.0e5"^^xsd:float        |1.0e5             |
-------------------------------------------------------------------------------------------------------------------
```

В выводе ARQ представлено большинство чисел в кавычках с ^^ указателями типа за ними.

Спецификация языка запросов SPARQL 1.1 включает в себя таблицу, которая показывает, какие преобразования типа всегда разрешены (например, целое число в строку), какие никогда не позволительны (например, DateTime в булевевское) и какие "зависят от лексического значения." В качестве примеров: строка "4" может быть преобразовано в целое, но строка "четверка" не может.

Вы можете обнаружить, что разные процессоры SPARQL справятся с последним примером по-разному. Например, в то время как xsd:decimal от 1.0e5 ничего не вернул, когда запрос выполнялся с ARQ 2.10, с Sesame 2.6.4 он возвращает 10000.

Чтобы проверить использование других трех функций преобразования дополним версию файла данных ex241.ttl:

```
# filename: ex259.ttl
@prefix dm: <http://learningsparql.com/ns/demo#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

d:id1 dm:location _:b1 .
d:id3 dm:amount 3 .
d:id4 dm:amount "4"^^xsd:integer .
d:id5 dm:amount 1.0e5 .
d:id6 rdfs:label "5 bucks" .
d:id7 dm:shipped true .
d:id8 dm:shipped "true" .
d:id9 dm:shipped "True" .
d:id10 dm:shipDate "2011-11-12" .
d:id11 dm:shipDate "2011-11-13T14:30:00" .
d:id12 dm:shipDate "2011-11-14T14:30:00"^^xsd:dateTime .
```

Следующий запрос похож на последний, за исключением того, что он пытается преобразовать объект каждой тройки в строку и логическое значение:

```
# filename: ex260.rq
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?o ?stringTest ?boolTest
WHERE
{
```

```
  ?s   ?p   ?o .
  BIND (xsd:string(?o) as ?stringTest)
  BIND (xsd:boolean(?o) as ?boolTest)
}
```

Результаты показывают, что преобразование в строку работает для всех объектов, кроме пустого узла, а преобразование в логический тип более разборчиво:

```
-------------------------------------------------------------------------------------------------------------------
| o                                        | stringTest                                | boolTest |
===================================================================================================
|3                                         |"3"^^xsd:string                            | true     |
| "true"                                   | "true"^^xsd:string                        | true     |
| "2011-11-14T14:30:00"^^xsd:dateTime      | "2011-11-14T14:30:00"^^xsd:string         |          |
| _:b0                                     |                                           |          |
| "5 bucks"                                | "5 bucks"^^xsd:string                     |          |
| "2011-11-12"                             | "2011-11-12"^^xsd:string                  |          |
|4                                         | "4"^^xsd:string                           | true     |
| "True"                                   | "True"^^xsd:string                        |          |
| true                                     |"true"^^xsd:string                         | true     |
| "2011-11-13T14:30:00"                    |"2011-11-13T14:30:00"^^xsd:string          |          |
| 1.0e5                                    | "1.0e5"^^xsd:string                       |true      |
-------------------------------------------------------------------------------------------------------------------
```

Преобразование в xsd:boolean работает для строки "true" в тройке d:id8, но не работает с идентификатором тройки d:id9 из-за заглавной буквы "T". (Если вам нужно конвертировать такие строки в логические значения, будет удобна функция LCASE(), описанная в разделе "Строковые функции"). Функция xsd:boolean() преобразует все числа, как вы можете видеть выше, в булево true, это согласуется с поведением нескольких популярных языков программирования.

Наш последний запрос демонстрирует преобразование объектов троек в значения даты-времени:

```
# filename: ex262.rq
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?o ?dateTimeTest
WHERE
{
  ?s   ?p   ?o .
  BIND (xsd:dateTime(?o) as ?dateTimeTest)
}
```

При запуске с набором данных ex259.ttl, мы получаем такой результат:

```
-------------------------------------------------------------------------------------------------------------------
| o                                        | dateTimeTest                                                |
===================================================================================================
|3                                         |                                                             |
| "true"                                   |                                                             |
| "2011-11-14T14:30:00"^^xsd:dateTime      | "2011-11-14T14:30:00"^^xsd: dateTime                        |
| _:b0                                     |                                                             |
| "5 bucks"                                |                                                             |
| "2011-11-12"                             |                                                             |
|4                                         |                                                             |
```

```
| "True"                          |                                  |
| true                            |                                  |
| "2011-11-13T14:30:00"           |"2011-11-13T14:30:00"^^dateTime   |
| 1.0e5                           |                                  |
```

Значение из d:id12 тройки появляется на выходе, потому что оно уже было значением xsd:dateTime даты и времени. Единственным входным значением, которое было преобразовано из чего-то другого в xsd:datetime было строковое значение отформатированное именно так, как ожидалось функцией: "2011-11-13T14:30:00" в тройке d:id11.

Функция STRDT() (от - STRing Data Type) создает типизированный литерал из двух ее аргументов: значения типизированного как простой литерал и URI указывающего тип.

Важно

Функция STRDT() является новой для SPARQL 1.1.

Чтобы получить общее представление о том, как эта функция работает, давайте посмотрим, что следующий запрос делает с данными ex241.ttl:

```
# filename: ex264.rq
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?o ?decimalTest
WHERE
{
  ?s   ?p   ?o .
  BIND (STRDT(str(?o),xsd:decimal) as ?decimalTest)
}
```

Вывод показывает, что в этом запросе, функция STRDT(), как правило, добавляет xsd:decimal к строковому представлению значения, имеет ли это смысл для этой строки или нет:

```
-------------------------------------------------------------------------------------------
| o                | decimalTest                                                          |
===========================================================================================
| 3                | "3"^^xsd:decimal                                                     |
| _:b0             |                                                                      |
|  "5 bucks"       |"5 bucks"^^xsd:decimal                                                |
| 4                |"4"^^xsd:decimal                                                      |
| dbr:Montréal |"http://dbpedia.org/resource/Montréal"^^xsd:decimal |
| true             | "true"^^xsd:decimal                                                  |
| 1.0e5            | 1.0e5                                                                |
-------------------------------------------------------------------------------------------
```

Например, она не имеет смысла со строками, такими как "true" и "5 bucks", но STRDT() добавляет его в любом случае. (ARQ выделяет несколько "Исключений форматов типов данных" и выдает предупреждающие сообщения для них, например для

для 1.0e5, и для Montréal URI).

Реальная ценность функции STRDT() преобразования типов, заключается в ее гибкости. Представьте себе, что интерфейс RDF вытащил из реляционной БД следующие данные:

```
# filename: ex266.ttl
@prefix im: <http://learningsparql.com/ns/importedData#> .
@prefix d: <http://learningsparql.com/ns/data#> .

d:item1  im:product "kerosene" ;
         im:amount "14" ;
         im:units "liters" .
d:item2  im:product "double-knit polyester" ;
         im:amount "10" ;
         im:units "squareMeters" .
d:item3  im:product "gold-plated chain" ;
         im:amount "30" ;
         im:units "centimeters" .
```

Числовые значения это просто строки, и единственная связь между каждым числовым значением и связанным с ним названием блока (например, "10" и "squareMeters") является то, что они объекты троек относятся с общей теме. Давайте предположим, что нам нужно преобразовать их в значения данных в пространстве имен http://learningsparql.com/ns/demo# с настраиваемыми типами данных из пространства имен http://learningsparql.com/ns/units#. Следующий запрос преобразует их с помощью STRDT(), чтобы назначить пользовательские типы данных:

```
# filename: ex267.rq
PREFIX im: <http://learningsparql.com/ns/importedData#>
PREFIX dm: <http://learningsparql.com/ns/demo#>
PREFIX u: http://learningsparql.com/ns/units#

CONSTRUCT { ?s dm:amount ?newAmount . }
WHERE
{
  ?s     im:product   ?prodName ;
         im:amount    ?amount ;
         im:units     ?units .
  BIND (STRDT(?amount,
                 URI(CONCAT("http://learningsparql.com/ns/units#",?units)))
         AS ?newAmount)
}
```

Результат показывает значения с присвоенными типами пользовательских данных:

```
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix u: <http://learningsparql.com/ns/units#> .
@prefix dm: <http://learningsparql.com/ns/demo#> .
@prefix im: <http://learningsparql.com/ns/importedData#> .

d:item2          dm:amount     "10"^^u:squareMeters .
d:item1          dm:amount     "14"^^u:liters .
```

```
d:item3                dm:amount      "30"^^u:centimeters .
```

Функция CONCAT() объединяет значение ?units в строку URI для пространства имен, функция URI() преобразует этот результат и функция STRDT() присваивает этот URI как тип данных для значений количества. Поскольку префикс u: был объявлен для блоков URI, ARQ его использует для указателей единиц измерения.

STRDT() и все остальные функции этого раздела в совокупности с функциями описанными в разделе в "5.2.2. Функции проверки типов узлов и данных" обеспечивают хороший инструментарий для очистки данных. Например, скажем, вы вытащили несколько троек из Linked Data Cloud, или использовали некоторую утилиту для преобразования электронной таблицы, некоторых XML или данных реляционных БД в триплеты. Если вы хотите преобразовать эти тройки в специфическую структуру с типами и свойствами, что ваши приложения будут намного эффективнее с использованием этих функций и правил. (Смотри раздел "Поиск плохих данных" для получения более подробной информации о правилах SPARQL.)

5.2.5. Проверка, добавление и удаление тегов разговорного языка

В разделе "2.3.3. Делаем RDF более читабельным с помощью тегов и меток" мы видели, как литерал может иметь тег, чтобы определить, какой язык или даже диалект языка он использует, например, бразильский португальский или французский швейцарский. Используя эти теги, можно назначить несколько меток и другой описательной информации к ресурсу, так что описания будут доступны на многих языках. Вот почему на запрос информации о ресурсе в DBpedia часто возвращается много значений для одного того же свойства: потому что у вас есть ответ на нескольких языках. Например, на рисунке 5-1 показан запрос на значение rdfs:label города, где находится производитель мотоциклов Ducati. В ответ получаем 13 результатов, с названием города, указанным на английском, немецком, испанском, финском и других языках.

Рисунок 5-1. Использование SNORQL для запроса в DBpedia о расположении производства Ducati

Что делать, если вы хотите описание только на одном языке? Функция lang() возвращает тег языка, прикрепленного к литералу, так что мы можем использовать ее,

чтобы в фильтре запроса указать, что мы хотим значения только на конкретном языке, например на английском:

```
# filename: ex269.rq
SELECT *
WHERE {
   :Ducati        <http://dbpedia.org/ontology/locationCity>     ?city .
  ?city           rdfs:label                                     ?cityName .
   FILTER ( lang(?cityName) = "en" )
}
```

Рассмотрим другой пример. Данные ex039.ttl из главы 2, имеют четыре из 13 rdfs:label для ресурса http://dbpedia.org/resource/Switzerland:

```
# filename: ex039.ttl
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

<http://dbpedia.org/resource/Switzerland>   rdfs:label   "Switzerland"@en, "Suiza"@es,
    "Sveitsi"@fi, "Suisse"@fr .
```

Следующий запрос извлекает все из них:

```
# filename: ex270.rq
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?label
WHERE { ?s   rdfs:label   ?label .}
```

Добавление оператора FILTER, который использует функцию lang() для проверки значений тега языка "en" указывает процессору запросов, какой язык мы хотим выбрать:

```
# filename: ex271.rq
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?label
WHERE {
  ?s   rdfs:label   ?label .
  FILTER ( lang(?label) = "en" )
}
```

Вот результат выполнения этого запроса с файлом ex039.ttl:

```
------------------------------
| label                      |
 ==================
| "Switzerland"@en   |
------------------------------
```

Что делать, если мы не хотим, чтобы "en" появлялся в результатах? Мы узнали, в разделе "5.2.3. Функции преобразования типа узела", что функция str() возвращает строковое представление аргумента, переданного ей. Это включает в себя зачистки языковых тегов, что делает ее здесь очень полезной:

```
# filename: ex273.rq
PREFIX rdfs: http://www.w3.org/2000/01/rdf-schema#

SELECT ?strippedLabel
WHERE {
  ?s   rdfs:label   ?label .
  FILTER ( lang(?label) = "en" )
  BIND (str(?label) AS ?strippedLabel)
}
```

Вот результат работы ex273.rq с теми же данными ex039.ttl:

```
------------------------------
| stripLabel                 |
 ==================
| "Switzerland"              |
------------------------------
```

Важно

При использовании ключевого слова BIND для хранения и извлечения «очищенной» метки rdfs:label необходим процессор. SPARQL 1.1.

Вот еще один файл данных из той же главы. Он включает в себя коды стран с языковым кодом, чтобы показать, какие термины на американском английском, а какие на британском английском:

```
# filename: ex037.ttl
@prefix : <http://www.learningsparql.com/ns/demo#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

:sideDish42     rdfs:label      "french fries"@en-US .
:sideDish42     rdfs:label      "chips"@en-GB .

:sideDish43     rdfs:label      "chips"@en-US .
:sideDish43     rdfs:label      "crisps"@en-GB .
```

Когда мы запустим тот же запрос ex273.rq для поиска языковых меток rdfs:label в этом файле, и также лишим их тегов языка, мы не получим ничего:

```
------------------------------
| stripLabel                 |
 ==================
------------------------------
```

Это произойдет потому, что запрос искал теги @en, а в данных используются тэги @en-US и @en-GB. Если в операторе FILTER для значения функции lang() задать @en-US или @en-GB, то получим ожидаемый результат.

К счастью, функция langMatches() обладаает большей гибкостью. Она сравнивает тег языка заданного в качестве первого аргумента со значением ее второго аргумента и возвращает логическое истину, если языки совпадают. Если второй аргумент не указывает конкретный вариант страна языка, функция не заботятся о нем. Она также не обращает

внимание на регистры, в которых записаны коды языка. Эта версия последнего запроса будет игнорировать любые коды языков:

```
# filename: ex276.rq
PREFIX rdfs: http://www.w3.org/2000/01/rdf-schema#

SELECT ?strippedLabel
WHERE {
  ?s   rdfs:label   ?label .
  FILTER ( langMatches(lang(?label),"en" ))
  BIND (str(?label) AS ?strippedLabel)
}
```

При запуске с данными ex037.ttl мы получаем все значения английского языка:

```
----------------------
| strippedLabel |
 ==============
| "crisps"         |
| "chips"          |
| "chips"          |
| "french fries" |
----------------------
```

Из-за своей гибкости, функция langMatches() лучше подходит для языкового тестирования значений, чем lang(). Например, проверить, есть ли что-то по-испански, лучше с помощью логического выражения langMatches(lang(?someVal),"es"), чем выражения (lang(?someVal) = "es"), используемого как условие фильтрации.

Скажем, нам нужно сделать «очистку» некоторых данных, и прежде мы хотим убедиться, что все rdfs:label имеют языковые теги. Для этого найдем все rdfs:label, которые не имеют тегов. Функция langMatches() является настолько гибкой, что она допускает ипользование группового символа в качестве второго аргумента, так что вы можете использовать его, чтобы проверить, какие значения имеют или не имеют языковые теги. Следующий файл данных имеет три метки rdfs:label, но только два языковых тега:

```
# filename: ex278.ttl
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema> .

d:item1   rdfs:label   "dog" .
d:item2   rdfs:label   "cat"@en .
d:item3   rdfs:label   "turtle"@en-US .
```

Выражение langMatches(lang(?label),"*") в следующем запросе вернет true для каждой метки ?label, которая имеет тег языка и false для каждой метки, у которой его нет. Символ !(), который описывает это выражение переворачивает логическое значение, так что полное выражение возвращает true для каждой метки, которая не имеет тега языка:

```
# filename: ex279.rq
```

```
PREFIX rdfs: http://www.w3.org/2000/01/rdf-schema

SELECT ?label
WHERE
{
  ?s   rdfs:label   ?label .
  FILTER (!(langMatches(lang(?label),"*")))
}
```

Когда мы выполнить этот запрос с данными ex278.ttl, он вернет одно значение без тега языка:

```
-----------
| label |
=======
| "dog" |
-----------
```

До сих пор в этом разделе мы узнали об использовании кодов языка как части критериев поиска запроса и о том как удалить код языка. Что делать, если мы хотим, добавить тег языка в строку? Мы не можем просто объединить метки и теги, потому что это специальная часть метаданных, а не дополнительные несколько символов строкового значения. Чтобы сделать это, мы используем функцию STRLANG(), которая использует параметры литерал и строку, представляющую тег языка, в качестве аргументов и возвращает литерал помеченный этим тегом.

Важно

Функция STRLANG() является новой для SPARQL 1.1.

Представьте себе, что некоторая утилита превратила таблицу эквивалентных американских и британских терминов в следующие тройки:

```
# filename: ex281.ttl
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dm: <http://learningsparql.com/ns/demo#> .
@prefix d: <http://learningsparql.com/ns/data#> .

d:cell1  dm:row 1 ;
         dm:column 1 ;
         rdfs:label "truck" .
d:cell2  dm:row 1 ;
         dm:column 2 ;
         rdfs:label "lorry" .
d:cell3  dm:row 2 ;
         dm:column 1 ;
         rdfs:label "apartment" .
d:cell4  dm:row 2 ;
         dm:column 2 ;
         rdfs:label "flat" .
d:cell5  dm:row 3 ;
```

```
        dm:column 1 ;
        rdfs:label "elevator" .
d:cell6 dm:row 3 ;
        dm:column 2 ;
        rdfs:label "lift" .
```

Утилиты, которые преобразуют файлы электронных таблиц в RDF легко найти.

Каждая строка таблицы имеет американский термин в его первом столбце, и соответствующий термин британский во втором, а следующий запрос преобразует эти данные для использования в таксономии SKOS. Потому что это создание RDF троек, это будет CONSTRUCT запрос, а не SELECT. Для каждой строки, он связывает значение rdfs:label метки из колонки 1 с переменной USTerm, а затем функций STRLANG() добавляет тег @EN-US и помещает результат в переменную ?taggedUSTerm. Подобные действия выполняются со значением из второго столбца той же строки, чтобы создать значение ?taggedGBTerm:

```
# filename: ex282.rq
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dm: <http://learningsparql.com/ns/demo#>
PREFIX skos: http://www.w3.org/2004/02/skos/core#

CONSTRUCT
{  ?rowURI   rdfs:type   skos:Concept ;
             skos:prefLabel ?taggedUSTerm, ?taggedGBTerm . }
WHERE
{
  ?cell1   dm:row    ?rownum ;
           dm:column 1 ;
           rdfs:label ?USTerm .
  BIND (STRLANG(?USTerm,"en-US") AS ?taggedUSTerm)

  ?cell2   dm:row    ?rownum ;
           dm:column 2 ;
           rdfs:label ?GBTerm .
  BIND (STRLANG(?GBTerm,"en-GB") AS ?taggedGBTerm)
  BIND (URI(CONCAT("http://learningsparql.com/ns/terms#t",str(?rownum)))
        AS ?rowURI)
}
```

Последняя директива BIND использует функцию URI(), о который мы узнали о ранее в этой главе, чтобы создать URI, который служит в качестве субъекта для трех троек, которые запрос создает для каждого значения ?rownum. Первый объект каждой тройки говорит, что что URI представляет собой SKOS концепцию и еще два назначают американские и британские skos:prefLabel этому URI. Вот результат выполнения запроса с данными ex281.ttl:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix dm: <http://learningsparql.com/ns/demo#> .
 @prefix skos: <http://www.w3.org/2004/02/skos/core#> .
```

```
<http://learningsparql.com/ns/terms#t1>
        rdfs:type skos:Concept ;
        skos:prefLabel "truck"@en-US ;
        skos:prefLabel "lorry"@en-GB .

<http://learningsparql.com/ns/terms#t2>
        rdfs:type skos:Concept ;
        skos:prefLabel "flat"@en-GB ;
        skos:prefLabel "apartment"@en-US .

<http://learningsparql.com/ns/terms#t3>
        rdfs:type skos:Concept ;
        skos:prefLabel "elevator"@en-US ;
        skos:prefLabel "lift"@en-GB .
```

Конечно, вы можете также использовать эту же функцию STRLANG(), чтобы назначить языковые теги, которые не включают обозначения стран.

5.2.6. Строковые функции

SPARQL предоставляет некоторые базовые функции для поиска и манипулирования строками текста. Они так полезны, что мы не смогли описывать их так далеко в книге, без использования некоторых их них, поэтому многие будут выглядеть весьма знакомыми. Если вы предполагаете много манипулировать строками, проверьте реализацию SPARQL, которую вы используете, чтобы увидеть, какие дополнительные строковые функции она предлагает в качестве расширений.

Важно

За исключением regex(), все функции, перечисленные в этом разделе являются новыми для SPARQL 1.1, но многие из них были популярными расширениями для процессоров SPARQL 1.0.

Чтобы изучить строковые функции, мы будем использовать этот маленький файл данных:

```
# filename: ex284.ttl
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema> .
@prefix d: <http://learningsparql.com/ns/data#> .

d:item1   rdfs:label   "My String" .
d:item2   rdfs:label   "123456" .
```

Это первый запрос демонстрирует использование STRLEN(), SUBSTR(), UCASE() и LCase() функции:

```
# filename: ex285.rq
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema>

SELECT ?label ?strlenTest ?substrTest ?ucaseTest ?lcaseTest
WHERE
{
```

```
  ?s r   dfs:label    ?label .
  BIND (STRLEN(?label) AS ?strlenTest)
  BIND (SUBSTR(?label,4,2) AS ?substrTest)
  BIND (UCASE(?label) AS ?ucaseTest)
  BIND (LCASE(?label) AS ?lcaseTest)
}
```

Когда ARQ применит этот запрос к файлу ex284.ttl данных, мы получим следующий результат:

```
-------------------------------------------------------------------------------
| label          | strlenTest | substrTest | ucaseTest      | lcaseTest    |
=======================================================
| "123456"     | 6            | "45"         | "123456"       | "123456"    |
| "My String" | 9             | "St"         | "MY STRING" | "my string" |
 -------------------------------------------------------------------------------
```

Первый столбец показывает входные строки, а остальные столбцы показать результат, что каждая функция сделала с двумя входными строками:

• Функция STRLEN() возвращает длину строки, переданной в качестве аргумента.

• Функция SUBSTR() возвращает подстроку строки, переданной в качестве первого аргумента. Второй аргумент определяет позицию начального символа и третий аргумент определяет, сколько символов нужно возвратить. Вызов функции в нашем примере определяет два символа, начиная с четвертого символа: "45" для "123456" и "St" для "My String".

• Функция UCASE() преобразует строку в верхний регистр, оставляя цифры без изменения.

• Функция LCASE() похожа на UCASE(), но преобразует строку в нижний регистре.

Следующий запрос демонстрирует четыре функции. Каждый возвращает логическое значение, которое говорит нам, удовлетворяет ли строка некоторому условию: STRSTARTS(), STRENDS(), CONTAINS() и regex().

```
# filename: ex287.rq
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema>

SELECT ?label ?startsTest ?endsTest ?containsTest ?regexTest
WHERE
{
  ?s   rdfs:label    ?label .
  BIND (STRSTARTS(?label,"12") AS ?startsTest)
  BIND (STRENDS(?label,"ing") AS ?endsTest)
  BIND (CONTAINS(?label," ") AS ?containsTest)
  BIND (regex(?label,"\\d{3}") AS ?regexTest)
 }
```

Вот то, что этот запрос делает с файлом ex284.ttl данных:

```
-------------------------------------------------------------------------
| label          | startsTest | endsTest | containsTest | regexTest |
=========================================================
| "123456"     | true         | false       | false          | true        |
| "My String" | false        | true        | true           | false       |
-------------------------------------------------------------------------
```

Будучи булевыми функциями, все они возвращают true или false, в зависимости от того, что функция нашла в строке:

• Функция STRSTARTS() проверяет, начинается ли строка в первом аргументе с содержимого строки по второму аргументу. Было установлено, что строка "123456" начинаются с "12", а "My String" не начинается.

• Функция STRENDS() проверяет, заканчивается ли строка в первом аргументе с содержимым строкои по второму аргументу. Было установлено, что строка "123456" не заканчивается "ing", а "My String" заканчивается.

• Функция CONTAINS() проверяет, включает ли строка в первом аргументе фрагмент из второго аргумента. Вызов функции СОДЕРЖИТ() обнаружил одиночный пробел в "My String", но не обнаружил в "123456".

• Функция регулярное выражение regex() является более гибким вариантом функции CONTAINS(), потому что вы можете указать регулярное выражение в качестве второго аргумента. Регулярное выражение в ex287.rq представляет три цифры подряд, которые функция нашла в "123456", но не нашла в "My String". Необязательный третий аргумент "i" заставляет эту функцию игнорировать регистр при поиске, но это не имеет значения при поиске цифровых разрядов.

Функция регулярное выражение regex() ожидает, что ее первым аргументом может быть либо xsd:string, либо простой литерал без тега языка, так что вы можете использовать функцию str(), чтобы убедиться в том, что вы передаете в качестве аргумента - например, регулярное выражение regex(str(?someVar),"jpg").

Язык, используемый для задания регулярных выражений исходит из спецификации XML-схемы Часть 2. Это примерно то же самое, что используется в языке программирования Perl. Некоторые из наиболее популярных регулярных выражений в качестве специальных символов включают точку, которая является Джокером и может заменять любой символ; \d - представляет собой любую цифре и \s - представляет собой любjq пробел (одиночный пробел, табуляция, возврат каретки или перевода строки).

Еще несколько операторов позволяют указать сколько символов вы ищете. Например, при добавлении некоторых из этих операторов после точки:

• . * представляет ноль или более символов.

• . + представляет один или более символов.

• .? представляет ноль или один символ.

• .{4} представляет именно четырех символов.

Они могут быть смешаны, например \\d{3} использовалось в ex287.rq для поиска трех цифр подряд. Хотя "123456" содержит больше цифр, но, как только функция нашла "123" в начале строки, это было то, что она искал.

Символы, показанные выше, не являются полным спектром специальных символов, которые вы можете использовать в регулярном выражении. Они могут быть гораздо сложнее, но они также могут быть гораздо проще. Например, в разделе "1.4. Поиск подстрок" мы видели, что следующий запрос извлекает тройки, где строка "Yahoo", в любом регистре, находится в объектах троек:

```
# filename: ex021.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
SELECT *
WHERE
{
?s ?p ?o .
FILTER (regex(?o, "yahoo","i"))
```

```
}
```

Регулярное выражение здесь не использует ни один из специальных символов. Даже без них, способность сделать поиск без учета регистра среди данных, указанных в остальной части запроса означает, что такое простое использование функции regex() может быть весьма удобно.

Рассмотрим функцию ENCODE_FOR_URI(), которую мы уже видели, более подробно. Она превращает любые символы в строке, которые могут вызвать проблемы, если эта строка используется в пути части URI, в знак процента, за которым следует число, представляющее ASCII код символа. Давайте посмотрим, что она сделает с данными этого файла:

```
# filename: ex289.ttl
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema> .
@prefix d: <http://learningsparql.com/ns/data#> .

d:item1    rdfs:label    "My String" .
d:item2    rdfs:label    "http://www.learnsparql.com/cgi/func1&color=red" .
```

Следующий запрос возвращает закодированную версию каждого значения ?label:

```
# filename: ex290.rq
PREFIX rdfs: http://www.w3.org/2000/01/rdf-schema

SELECT ?encodeTest
WHERE
{
  ?s    rdfs:label    ?label .
  BIND (ENCODE_FOR_URI(?label) AS ?encodeTest)
}
```

Функция заменяет пробел в "My String" на %20, и это преобразует все знаки препинания в URI с помощью ASCII кодирования:

```
-------------------------------------------------------------------------------------------------------
| encodeTest                                                                                          |
=======================================================================
| "http%3A%2F%2Fwww.learnsparql.com%2Fcgi%2Ffunc1%26color%3Dred"     |
| "My%20String"                                                                                       |
-------------------------------------------------------------------------------------------------------
```

Это особенно полезно, как мы увидим в главе 10, когда вы передаете URI или SPARQL запрос в качестве параметра для веб-услуг, таких, как конечные точки SPARQL.

5.2.7. Числовые функции

Прежде чем мы перейдем к числовым функциям, не забывайте, что вы можете использовать все основные арифметические операторы, такие как +, -, *, и / в выражениях SPARQL. Мы также видели в разделе "3.14.3. Группировка значений данных и поиск средних значений внутри групп", что AVG(), MIN(), MAX(), SUM() и COUNT() функции дают некоторые интересные варианты для работы с числовыми данными в тройках.

SPARQL спецификация также указывает, что реализации должны поддерживать abs(), round(), ceil() и floor() функции. Как и для строковых функций, стоит проверять реализацию SPARQL, которую вы используете, чтобы увидеть, не предлагает ли она какие-либо дополнительные числовые функции в качестве функций расширения.

Важно

Все числовые функции являются новыми для SPARQL 1.1.

Чтобы испробовать числовые функции, мы будем использовать этот пример данных:

```
# filename: ex292.ttl
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema> .
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix dm: <http://learningsparql.com/ns/demo#> .

d:item1   dm:amount    4 .
d:item2   dm:amount    3.2 .
d:item3   dm:amount    3.8 .
d:item4   dm:amount    -4.2 .
d:item5   dm:amount    -4.8 .
```

Следующий запрос использует каждую из четырех перечисленных выше функций:

```
# filename: ex293.rq
PREFIX dm: <http://learningsparql.com/ns/demo#>
PREFIX xsd: http://www.w3.org/2001/XMLSchema#

SELECT ?amount ?absTest ?roundTest ?ceilTest ?floorTest
WHERE
{
  ?s    dm:amount    ?amount .
  BIND (abs(?amount) AS ?absTest )
  BIND (round(?amount) AS ?roundTest )
  BIND (ceil(?amount) AS ?ceilTest )
  BIND (floor(?amount) AS ?floorTest )
}
```

Выполнение запроса с данными ex292.ttl дает такой результат:

```
-------------------------------------------------------------------------------------------------------------
| amount | absTest | roundTest           | ceilTest              | floorTest             |
=============================================================================
| -4.8       | 4.8       | "-5"^^xsd:decimal| "-4"^^xsd:decimal | "-5"^^xsd:decimal |
| -4.2       | 4.2       | "-4"^^xsd:decimal| "-4"^^xsd:decimal | "-5"^^xsd:decimal |
| 3.8        | 3.8       | "4"^^xsd:decimal  | "4"^^xsd:decimal  |"3"^^xsd:decimal   |
| 3.2        | 3.2       | "3"^^xsd:decimal  | "4"^^xsd:decimal  | "3"^^xsd:decimal   |
| 4          | 4         | 4                       | 4                       |4                        |
-------------------------------------------------------------------------------------------------------------
```

• Функция abs() возвращает абсолютное значение переданного параметра, преобразуя входные значения -4.8 и -4.2 в положительные числа.
• Функция round() округляет значения до ближайшего целого числа.
• Функция ceil() возвращает значение "потолок": ближайшее целое значение больше аргемента, если он имеет дробную часть, или сам аргумент, если это целое число.
• Функция floor() возвращает ближайшее целое меньше аргумента, если он имеет дробную часть или сам аргумент, если это целое число.

Функция round() возвращает случайное число двойной точности между 0 и 1. Она может возвращать 0, но не вернет 1. Если вы хотите вернуть число в другом диапазоне значений, то можете использовать преобразования. Например, если умножить значение round() на 11 и добавить 20, то вы получите ряд целых чисел в диапазоне между 20 и 30 включительно.

Чтобы продемонстрировать это, следующий запрос выводит два числа для каждой тройки, переданные ему в качестве входных. Переменная ?randTest1 будет иметь значение простого вызова функции rand(). Значение ?randTest2 будет случайным целое числом от 20 до 30:

```
# filename: ex295.rq
SELECT ?randTest1 ?randTest2
WHERE
{
  ?s   ?p   ?o .
  BIND (rand() AS ?randTest1)
  BIND (floor(rand()*11)+20 AS ?randTest2)
}
```

Обратите внимание, что запрос на самом деле не использовал никаких входных данных. При запуске с файлом данных ex292.ttl, который имеет пять троек, мы получаем эти пять пар случайных чисел:

```
-----------------------------------------------------
| randTest1                  | randTest2 |
=====================================
| 0.20209451122917443e0 | 29.0e0    |
| 0.04707018085243442e0 | 28.0e0    |
| 0.2190604769364065e0  | 25.0e0    |
| 0.5742086203122172e0  | 22.0e0    |
| 0.3674021731250735e0  | 21.0e0    |
-----------------------------------------------------
```

Запуская этот запрос снова сразу же, ничего не меняя, мы получаем другой набор случайных значений:

```
-----------------------------------------------------
| randTest1                  | randTest2 |
=====================================
| 0.8665625585923823e0  | 24.0e0    |
| 0.21184532852211524e0 | 22.0e0    |
| 0.18848604673741176e0 | 25.0e0    |
| 0.9411502523245124e0  | 27.0e0    |
| 0.5816330932580108e0  | 25.0e0    |
-----------------------------------------------------
```

При использовании с запрос CONSTRUCT, функция rand() может быть полезной для создания выборочных данных.

5.2.8. Функции даты и времени

Важно

Функции даты и времени являются новыми для SPARQL 1.1.

SPARQL предоставляет восемь функций для манипулирования данными даты и времени. Вы можете использовать в качестве данных типизированные литералы xsd:dateTime и, в зависимости от цели функции, литералы xsd:date и xsd:time, которые основаны на xsd:dateTime. SPARQL также предлагает функцию now(), которая дает дату и время старта запроса.

Мы видели, что типы данных SPARQL основаны на спецификации XML-схемы Часть 2 типов данных и базируются на стандарте ISO 8601. Использование ISO 8601 для представления даты и времени - 14 октября 2011, 12 часов в пятом часовом поясе к западу от Гринвича дало бы "2011-10-14T12:00:00.000-05:00 ". Вы могли бы представлять саму дату как "2011-10-14"^^xsd:date или время, как "12:00:00.000-05:00"^^xsd:time. Если части, имеющие временную зону и доли секунды, не требуется, вы можете указать "2011-10-14 T12:00 00"^^xsd:dateTime, что не вызовет сообщение об ошибке.

За исключением функции new(), все SPARQL функции даты и времени разработаны так, чтобы можно было взять конкретные биты из значений даты и времени. Давайте посмотрим, что они делают с данными онтологии Билеты:

```
# filename: ex298.ttl
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix t: <http://purl.org/tio/ns#> .

d:meeting1   t:starts   "2011-10-14T12:30:00.000-05:00"^^xsd:dateTime .
d:meeting2   t:starts   "2011-10-15T12:30:00"^^xsd:dateTime .
```

Следующий запрос вытаскивает некоторые из частей даты и времени встречи:

```
# filename: ex299.rq
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX t: http://purl.org/tio/ns#

SELECT ?mtg ?yearTest ?monthTest ?dayTest ?hoursTest ?minutesTest
WHERE
{
  ?mtg   t:starts   ?startTime .
  BIND (year(?startTime) AS ?yearTest)
  BIND (month(?startTime) AS ?monthTest)
  BIND (day(?startTime) AS ?dayTest)
  BIND (hours(?startTime) AS ?hoursTest)
  BIND (minutes(?startTime) AS ?minutesTest)
}
```

Результаты в основном предсказуемы, за исключением двух различных значений ?hoursTest:

```
-------------------------------------------------------------------------------------------
| mtg          | yearTest | monthTest | dayTest | hoursTest | minutesTest |
===========================================================================
| d:meeting2 | 2011      | 10         | 15       | 12         | 30           |
| d:meeting1 | 2011      | 10         | 14       | 17         | 30           |
-------------------------------------------------------------------------------------------
```

Процессор по умолчанию принимает часовой пояс Гринвича, потому совещание1 (meeting1) происходит в 12:30 в пяти часовых поясах западнее Гринвича, это 17:30 в Англии.

Функция seconds() возвращает значение секунд в виде десятичного числа. Мы увидим пример ее использования в ближайшее время. Во-первых, давайте посмотрим на две функции для проверки часового пояса в значении дата-время, часовой timezone() и tz():

```
# filename: ex301.rq
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX t: <http://purl.org/tio/ns#>
PREFIX xsd: http://www.w3.org/2001/XMLSchema#

SELECT ?timezoneTest ?tzTest
WHERE
{
  ?mtg   t:starts   ?startTime .
  BIND (timezone(?startTime) AS ?timezoneTest)
  BIND (tz(?startTime) AS ?tzTest)
}
```

Функция timezone() возвращает часть даты - часовой пояс, типизированный как в xsd:dayTimeDuration и tz() возвращает его версию в виде простого литерала. Вот как выглядит результат запроса ex301.rq при запуске с данными ex298.ttl:

```
-------------------------------------------------------------------------
| mtg          | timezoneTest                        | tzTest   |
=========================================================
| d:meeting2 |                                       | ""         |
| d:meeting1 | "-PT5H"^^xsd:dayTimeDuration | "-05:00" |
 -------------------------------------------------------------------------
```

Функция now() возвращает текущую дату и время, а точнее, дату и время, когда запрос начинает работать. Следующий запрос показывает нам дату начала работы запроса и демонстрирует работу функции seconds(). Этот запрос игнорирует вход, так что вы можете запустить его с любым файлом входных данных:

```
# filename: ex303.rq
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?currentTime ?currentSeconds
WHERE
{
```

```
  BIND (now() AS ?currentTime)
  BIND (seconds(?currentTime) AS ?currentSeconds)
}
```

Запрос мог бы использовать вызов функции now() в качестве аргумента функции seconds(), но использует переменную, созданный из ее значения. В любом случае, функция seconds() ожидает значение xsd:dateTime в качестве аргумента со всеми правильными ISO 8601 значениями отдельных компонент в нужных местах. Вот один из результатов работы этой функции:

```
-------------------------------------------------------------------------------
| currentTime                                          | currentSeconds |
===============================================================
| "2011-02-05T12:58:27.93-05:00"^^xsd:dateTime | 27.93          |
-------------------------------------------------------------------------------
```

5.2.9. Хэш-функции

Важно

Хэш функции являются новыми для SPARQL 1.1.

Криптографические хеш-функции SPARQL преобразовывают строку текста в шестнадцатеричное представление строки битов, которые могут служить в качестве закодированной подписи для входной строки. Например, если вы по электронной почте послали мне абзац текста, а затем отдельно послали мне результат прохождения этого текста через определенную хэш-функцию, я мог бы преобразовать принятый текст с помощью той же хэш функции и сравнить результат. Если бы результат не совпали, то я бы знал, что получил от вас не то, что вы послали.

Это популярно в данных FOAF, где адрес электронной почты является общим идентификатором человека, но боязнь спама предотвращает людей от указания их адреса электронной почты в файле FOAF. FOAF словарь включает в себя свойство foaf:mbox_sha1sum, которое хранит хэш-строку адреса электронной почты («почтового ящика»), производимую с помощью SHA-1 криптографической функции. Таким образом, вы получите достаточно уникальное значение для представления себя, не подвергая свой адрес электронной почты угрозе, когда веб-сканеры собирают незащищенные адреса для списков рассылки спама. Значение SHA-1 представляет 160-битовую строку подписи, а алгоритм MD5 использует 128-битовую строку. (Чем больше битов, тем больше безопасности). Другие криптографические хэш-функции, поддерживаемые SPARQL, которые являются вариациями более совершенного алгоритма SHA-2 используют размер битовой строки, указанный числом в их именах.

SPARQL поддерживает следующие хэш-функции:

- MD5()
- SHA1()
- SHA224()
- SHA256()
- SHA384()
- SHA512()

Чтобы продемонстрировать работу функций, мы будем принимать следующие данные, которые мы видели раньше в главах 1 и 4 и преобразовывать их в FOAF с

помощью запроса CONSTRUCT. В целях безопасности, мы не будем копировать телефонные номера, а подставим foaf:mbox_sha1sum значения вместо их адреса электронной почты:

```
# filename: ex012.ttl
@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d: <http://learningsparql.com/ns/data#> .

d:i0432 ab:firstName    "Richard" .
d:i0432 ab:lastName     "Mutt" .
d:i0432 ab:homeTel      "(229) 276-5135" .
d:i0432 ab:email        "richard49@hotmail.com" .

d:i9771 ab:firstName    "Cindy" .
d:i9771 ab:lastName     "Marshall" .
d:i9771 ab:homeTel      "(245) 646-5488" .
d:i9771 ab:email        "cindym@gmail.com" .

d:i8301 ab:firstName    "Craig" .
d:i8301 ab:lastName     "Ellis" .
d:i8301 ab:email        "craigellis@yahoo.com" .
d:i8301 ab:email        "c.ellis@usairwaysgroup.com" .
```

Запрос сохраняет результат вызова функции SHA1() в переменной ?hashEmail и использует в качестве значения foaf:mbox_sha1sum:

```
# filename: ex305.rq
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX ab: <http://learningsparql.com/ns/addressbook#>

CONSTRUCT {
  ?s     foaf:givenName       ?first ;
         foaf:familyName      ?last ;
         foaf:mbox_sha1sum ?hashEmail .
  }
WHERE
  {
   ?s    ab:firstName    ?first ;
         ab:lastName     ?last ;
         ab:email        ?email .
  BIND (SHA1(?email) AS ?hashEmail )
  }
```

Вот то, что этот запрос делает с данными ex012.ttl:

```
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix ab: <http://learningsparql.com/ns/addressbook#> .

d:i9771
         foaf:familyName "Marshall" ;
         foaf:givenName "Cindy" ;
```

```
        foaf:mbox_sha1sum "821be6ab56326d7b08246f2cb9c0f68afe0156d9" .
d:i0432
        foaf:familyName "Mutt" ;
         foaf:givenName "Richard" ;
        foaf:mbox_sha1sum "b7f191315aa6bb4a9ab56b02e334647c4b1104a0" .
d:i8301
        foaf:familyName "Ellis" ;
        foaf:givenName "Craig" ;
        foaf:mbox_sha1sum "396d3e3aef87e0fecd3cbbdd2479eb3797d7af18" ;
        foaf:mbox_sha1sum "faec1b07cf7c10e302544f22958c02c53844a4fa" .
```

Скажем, кто-то создал эти тройки и отправил их публично. У вас есть адрес электронной почты Ричарда Матт и вам интересно, тот же самый адрес указан в этом файле. Вы можете пропустить этот адрес электронной почты как аргумент функции SHA1(), используя запрос SPARQL. Вы также можете передать его с помощью короткой программы, написанной на почти любом языке программирования, поскольку поддержку SHA1 легко найти. Например, эта маленькая программа на языке Python будет делать это преобразование:

```
# filename: ex307.py
import hashlib
m = hashlib.sha1()
m.update("richard49@hotmail.com")
print m.hexdigest()
```

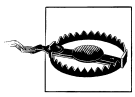
Если результатом будет "b7f191315aa6bb4a9ab56b02e334647c4b1104a0", то у вас адрес электронной почты того же Ричарда Матт.

5.3. Расширение функций

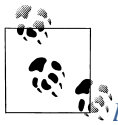
Большинство провайдеров процессора SPARQL включают больше функций чем те, которые требуются по спецификации SPARQL. Они делают это, чтобы дифференцировать свою программу от других и обеспечить собственное развитие приложений на SPARQL основе (то есть, если функция SPARQL не обеспечивает необходимого пребразования, они добавить новую функцию в их реализации SPARQL), и отправляют сообщения Рабочей Группе о том, что они считают, что полезные функции отсутствуют в SPARQL. Спецификация SPARQL 1.1 показывает, что Рабочая Группа приняла много подобных поправок; почти все функции, отмеченные в этой главе как новые в SPARQL 1.1 были функциями расширения в более чем одном процессоре, прежде чем спецификация SPARQL 1.1 была выпущена.

При работе с особым процессором SPARQL ознакомьтесь с его расширениями функций, потому что они могут расширить эффективность запросов, снизить время разработки, и, возможно, даже сократить время выполнения.

Некоторые расширения обеспечивают большую эффективность обработки, потому что они более плотно присвязанны к этой конкретной реализации. Например, функция Virtuoso bif:contains() является вариацией функции CONTAINS(), которую мы видели ранее в этой главе, и может быть гораздо быстрее, если вы используете Virtuoso.



Помните, что использование расширений стандарта делает ваше приложение нестандартным и менее переносимыми.



Потому что функции расширения приходят извне стандарта SPARQL, использование их означает, что вы должны определить пространство имен, где они берутся. Например, функции расширения ARQ находятся в пространстве имен http://jena.hpl.hp.com/ARQ/function#, так что если вы используете его функцию afn:localname, то ваш запрос должен объявить afn: префикс, также как декларировался rdfs: или dc: префикс.

Следующий запрос использует ARQ afn:localname и afn:namespace расширения для разделения локальных имен и имен пространства имен URI каждого субъекта тройки:

```
# filename: ex308.rq
PREFIX afn: <http://jena.hpl.hp.com/ARQ/function#>
PREFIX d: http://learningsparql.com/ns/data#
SELECT DISTINCT ?s ?sLocalname ?sNamespace
 WHERE
{
  ?s   ?p   ?o .
  BIND (afn:localname(?s) AS ?sLocalname )
  BIND (afn:namespace(?s) AS ?sNamespace )
}
```

При запуске с файлом данных ex012.ttl получаем такой результат:

```
-------------------------------------------------------------------------------
| s         | sLocalname | sNamespace                                 |
===============================================================
| d:i9771 | "i9771"      | "http://learningsparql.com/ns/data#" |
| d:i0432 | "i0432"      | "http://learningsparql.com/ns/data#" |
| d:i8301 | "i8301"      | "http://learningsparql.com/ns/data#" |
-------------------------------------------------------------------------------
```

Если есть какие-то новые функции, которые вы хотели бы видеть в процессоре SPARQL или которые вы используете, дайте знать об этом разработчикам. Возможно, кто-то другой также заинтересован в этих функциях и ваш голос склонит чашу весов. И, может быть, это поможет сделать эти функции достаточно популярными, чтобы быть включенными в SPARQL 1.2 или 2.0!

5.4. Резюме

В этой главе мы узнали:

- Какие типы данных SPARQL поддерживает изначально и как запросы могут использовать одновременно эти и пользовательские типы
- Опции для представления строк
- Как реализовать арифметические вычисления в SPARQL
- Функции SPARQL 1.1 для программной логики и типа узла и типа данных проверки и преобразования
- Функции SPARQL 1.1 для управления тегами языка
- Строковые, числовые, дата-время и хэш-функции в SPARQL 1.1
- Роль, которую функции расширения могут играть в запросах

Глава 6. Обновление данных с помощью SPARQL

Здорово, когда вы можете извлечь и изменить данные из коллекции, а также сформировать необходимые кросс-ссылки, но когда вы можете использовать стандартный язык запросов для добавления данных к этой коллекции, удаления и обновления своих данных, вы можете построить серьезные полноценные приложения вокруг гибкой модели данных RDF. Спецификация обновления SPARQL 1.1 описывает синтаксис и опции для этого, и, хотя это недавнее дополнение к SPARQL, несколько реализаций уже поддерживают ее. В этой главе для демонстрации различных примеров используется Fuseki, одна из простейших реализаций SPARQL обновления.


Большинство реализаций языка обновлений SPARQL вы найдете в triplestores. Будучи по сути программой управления базами данных (не реляционных), triplestore должен позволять вам запрашивать данные с помощью языка запросов SPARQL и управлять ими с помощью языка обновления.

Важно

SPARQL обновление было новым для SPARQL 1.1. В SPARQL 1.0 не было определено никакого способа, чтобы обновить данные, так triplestores с поддержкой SPARQL изначально опирались на собственные расширения, позволяя вам обновить свои данные. Вы все еще можете видеть это со старыми triplestores.

В этой главе, мы узнаем:

"Начало работы с Fuseki"

Для знакомства с Fuseki достаточно попробовать все ключевые части SPARQL обновления

"Добавление данных к набору данных"

Как добавить тройки указанные в запросе от удаленных источников к графу набора данных по умолчанию

"Удаление данных"

Как удалить данные из графа по умолчанию

"Изменение данных"

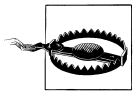
Как заменить существующие тройки в графе по умолчанию для набора данных

"Поименованные графы"

Как создать поименованные графы, добавить к ним тройки, удалить от них тройки, и заменить тройки и целые графы

6.1. Начало работы с Fuseki

Fuseki является частью проекта Jena. Он описывает себя как "SPARQL Server" и функционирует в качестве сервера веб-triplestore, который принимает SPARQL запросы, которые вы вводите через веб-формы, а также SPARQL запросы, которые вы посылаете в форме HTTP запросов. В этой главе, мы будем использовать веб-форму, а в разделе "10.4. SPARQL и HTTP", мы узнаем об использовании HTTP интерфейса.


На момент написания книги, последняя официальный релиз Fuseki выпуска 0.2.6. Диапазон возможностей Fuseki и сопроводительная документация весьма впечатляет, но если вы

используете более позднюю версию, то можете увидеть некоторые отличия от того, что описывается в этой главе.

Чтобы скачать Fuseki, загрузите с домашней странице Fuseki двоичный ZIP файл, чье имя имеет формат jena-fuseki-*- distribution.zip. После того как вы его распакуете это, вы готовы к работе. Он включает в себя сценарий оболочки (shell script) под названием fuseki-server, который будет запускать его под Linux или Mac и пакетный файл fuseki-server.bat, которыq будут делать то же самое под Windows. Вы можете узнать о параметрах командной строки Fuseki с помощью следующей команды:

```
fuseki-server –help
```

Перед запуском сервера Fuseki, можно создать подкаталог корневого каталога дистрибутива и назвать его dataDir. Затем, чтобы Fuseki стартовал в качестве сервера на машине Windows или Linux, надо запустить сценарий fuseki-server со следующими параметрами:

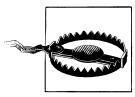
```
fuseki-server --update --loc=dataDir /myDataset
```

Эта команда включает в себя следующие параметры:

--update
Сообщает Fuseki о режиме обновления сохраненных данных. Без этого, он по умолчанию готов только для режима чтения.

--loc=dataDir
Сообщает Fuseki о хранении данных в базе данных TDB и сохранять его в каталоге dataDir, который мы только что создали. (TDB другая часть проекта Jena, предназначенной для хранения RDF). Это будет постоянная база, сохраняющая данные на вашем жестком диске, даже после того, как закрыли фусеки.

/myDataset
Имя пути набора данных, который должен начинаться с косой черты. (Это деталь Fuseki связана со спецификацией SPARQL).


Примеры в этой книге всегда используют набор данных с именем /myDataset и иногда инструктируют о стерании всего, что там хранится, прежде чем приступить к следующему шагу. Если вы создаете приложение, не связанное с примерами этой книги, то храните свои данные в другом наборе данных.

В процессе запуске сервера в окне где вы ввели команду будет прокручиваться несколько сообщений о состоянии. (Позже, когда вы закончили использование Fuseki и готовы, чтобы закрыть его, нажмите Ctrl + C).

Когда сообщения запуска перестанут появляться, Fuseki готов к использованию. Отправьте ваш браузер по адресу http://localhost:3030/, чтобы увидеть главный экран Fuseki.


Если вы предпочитаете, чтобы Fuseki использовал другой порт, кроме 3030, команда --help покажет вам, какой.

Нажмите на ссылку Панель управления главного экрана. На экране панели управления Fuseki вам нужно выбрать набор данных; /myDataset единственный созданный перед началом работы Fuseki, поэтому нажмите кнопку Select.

Это приводит к экрану Fuseki Query, как показано на рисунке 6-1. Это форма где мы будем делать наши эксперименты в этой главе.

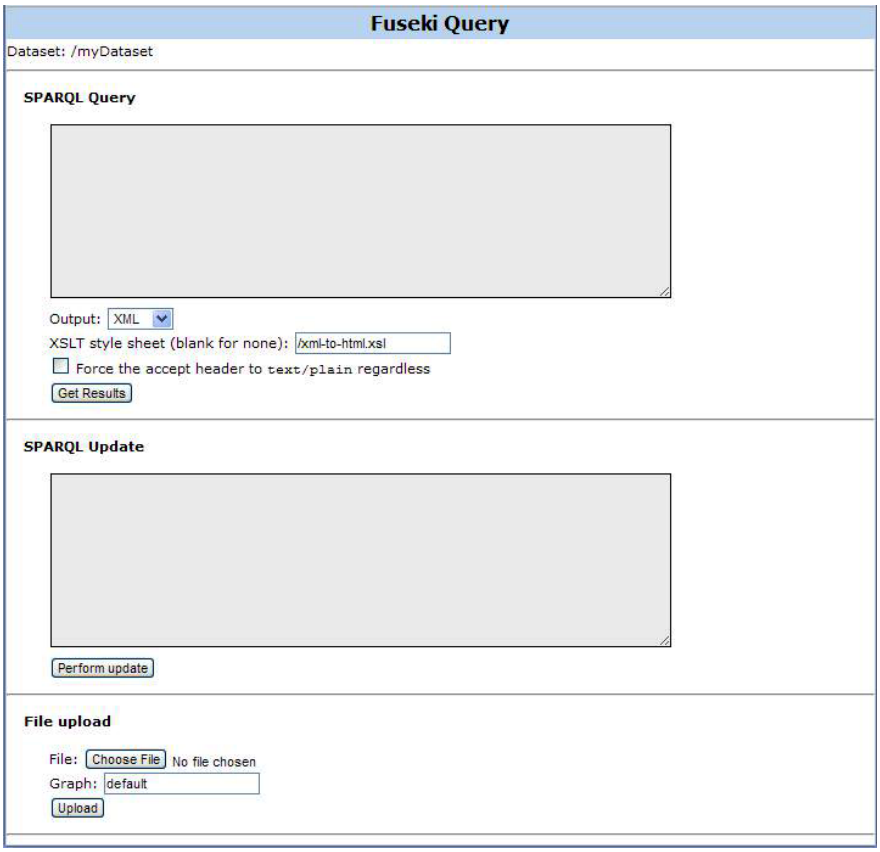


Рисунок 6-1. Экранная форма Fuseki

6.2. Добавление данных к набору данных

Большинство triplestores с интерфейсом на основе форм предлагают способ загрузки данных путем заполнения формы. Для обеспечения некоторых исходных данных для наших первых экспериментов с запросами SPARQL на обновление, используйте раздел ”File upload” (загрузки файла) в нижней части формы Fuseki для загрузки файла данных ex012.ttl, , который знаком с ранних глав этой книги:

```
# filename: ex012.ttl
@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d: <http://learningsparql.com/ns/data#> .

d:i0432 ab:firstName   "Richard" .
d:i0432 ab:lastName    "Mutt" .
d:i0432 ab:homeTel     "(229) 276-5135" .
d:i0432 ab:email       "richard49@hotmail.com" .

d:i9771 ab:firstName   "Cindy" .
d:i9771 ab:lastName    "Marshall" .
d:i9771 ab:homeTel     "(245) 646-5488" .
d:i9771 ab:email       "cindym@gmail.com" .
```

```
d:i8301 ab:firstName    "Craig" .
d:i8301 ab:lastName     "Ellis" .
d:i8301 ab:email        "craigellis@yahoo.com" .
d:i8301 ab:email        "c.ellis@usairwaysgroup.com" .
```

После нажатия кнопки ”Choice File”, выберите ex012.ttl, оставьте ”Graph” (граф) установки на "по умолчанию", и нажмите кнопку ”Upload” (загрузить). Фусеки будет читать данные и отображать короткое сообщение о том, сколько троек прочитано следующим образом:

```
Triples = 12
```

Нажатие кнопки Назад в браузере, вернет вас к форме запросов Fuseki.

Чтобы проверить, какие данные в наборе данных, введите следующую простой запрос в разделе запросов SPARQL в верхней части формы:

```
# filename: ex311.rq
SELECT *
WHERE
{ ?s   ?p   ?o }
```

При нажатии на кнопку "получить результаты" (вы можете установить в поле вывода формы либо XML, либо текст), Fuseki покажет 12 троек:

s	p	o
<http://learningsparql.com/ns/ data#i9771>	<http://learningsparql.com/ns/ addressbook#email>	“cindym@gmail.com”
<http://learningsparql.com/ns/ data#i9771>	<http://learningsparql.com/ns/ addressbook#homeTel>	“(245) 646-5488”
<http://learningsparql.com/ns/ data#i9771>	<http://learningsparql.com/ns/ addressbook#lastName>	“Marshall”
<http://learningsparql.com/ns/ data#i9771>	<http://learningsparql.com/ns/ addressbook#firstName>	“Cindy”
<http://learningsparql.com/ns/ data#i0432>	<http://learningsparql.com/ns/ addressbook#email>	“richard49@hotmail.com”
<http://learningsparql.com/ns/ data#i0432>	<http://learningsparql.com/ns/ addressbook#homeTel>	“(229) 276-5135”
<http://learningsparql.com/ns/ data#i0432>	<http://learningsparql.com/ns/ addressbook#lastName>	“Mutt”
<http://learningsparql.com/ns/ data#i0432>	<http://learningsparql.com/ns/ addressbook#firstName>	“Richard”
<http://learningsparql.com/ns/ data#i8301>	<http://learningsparql.com/ns/ addressbook#email>	“c.ellis@usairwaysgroup.com”
<http://learningsparql.com/ns/ data#i8301>	<http://learningsparql.com/ns/ addressbook#email>	“craigellis@yahoo.com”
<http://learningsparql.com/ns/ data#i8301>	<http://learningsparql.com/ns/ addressbook#lastName>	“Ellis”
<http://learningsparql.com/ns/ data#i8301>	<http://learningsparql.com/ns/addressbook#firstName>	“Craig”

При изучении способов добавления и удаления данных из набора данных, мы будем часто использовать запрос ex311.rq, чтобы проверить результаты различных запросов обновления, так что мы будем ссылаться на него как "Список всех троек по умолчанию".

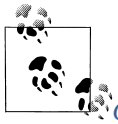
При просмотре результатов этих запросов в Fuseki, обратите внимание, как запрос становится частью URL в панели навигации браузера. Закладки этих результатов означают, что каждый раз, когда вы идете к этой закладке, вы выполняете этот запрос, так что это является удобным способом простого запуска нужных запросов.

Давайте делать некоторые обновления этих данных. Мы начнем с добавления двух троек: одна о том какое значение ab:homeTel соответствует ресурсу d:i8301 и другая, говорит о том, что ab:Person является классом.

Введите следующий запрос на обновление на панели Fuseki SPARQL Update и нажмите под ней кнопку "Выполнить обновление":

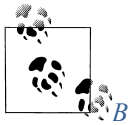
```
# filename: ex312.ru
```

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX d: <http://learningsparql.com/ns/data#>

INSERT DATA
{
  d:i8301     ab:homeTel   "(718) 440-9821" .
  ab:Person   a            rdfs:Class .
}
```



Спецификация SPARQL Update рекомендует файлам, хранящим запросы на обновление, давать расширение .ru в нижнем регистре.

После нажатия кнопки Fuseki фусеки отображает сообщение о том, что обновление успешно выполнено и следует нажать кнопку "назад" в браузере, чтобы вернуться к форме Fuseki запросов. Запрос с оператором SELECT, который вы указали в панели запросов SPARQL в верхней части формы все еще будет там, так что нажмите под ним кнопеу "Получить результаты" (или, если есть закладка результаты запроса, перейдите к этой закладке), чтобы увидеть данные, которые Fuseki сохранил: оригинальные 12 троек и две новые, вставленные ex312.ru.



В хранимых данных, вы увидите, что тройка ab:Person в роли rdfs:Class имеет предикат <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>, хотя в запросе на обновление в этой тройке вставлен предикат "a" - это потому, что "a" является сокращением для этого URI.

Прежде чем пристально рассматривать синтаксис нашего первого запроса на обновление, давайте посмотрим на другой, который делает ровно то же самое:

```
# filename: ex313.ru
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX d: <http://learningsparql.com/ns/data#>

INSERT
{
  d:i8301     ab:homeTel   "(718) 440-9821" .
  ab:Person   a            rdfs:Class .
}
WHERE {}
```

Он не имеет ключевого слова DATA после INSERT, как в ex312.ru, но у него есть WHERE и пара фигурных скобок в конце. Фактически он очень похож на запрос CONSTRUCT и по сути он идентичен: он создает новые тройки. Здесь не требуется никаких условий между фигурными скобками, но, как мы увидим, запрос INSERT обновление может включать в себя шаблон троек в этих скобках, и тройки в INSERT могут ссылаться на эти переменные. С оператором INSERT DATA не может использоваться WHERE и вы можете вставлять только тройки, а не шаблоны троек, между фигурными скобками оператора INSERT DATA.

Шаблоны троек позволяют заменить переменные в любой из трех позиций.

Почему SPARQL предоставляет два разных способа для вставки троек? Как и в запросах CONSTRUCT, использование шаблонов троек между фигурными скобками в запросах INSERT обеспечивает вам гибкость, когда вы указываете данные для вставки, например, вы можете сделать новые тройки, основанные на шаблонах размещенных между фигурными скобками оператора WHERE. (Позже в этой главе мы увидим, много примеров.) Операция INSERT DATA не позволяет использовать WHERE и переменные, тем самым упрощает работу процессора SPARQL, и он обрабатывает данные быстрее.

Разницей в скорости загрузки данных между запросом INSERT и INSERT DATA можно пренебречь, если сравнить два предыдущих примера, но хорошо помнить, что у вас есть эта возможность, когда вам нужно загрузить большое количество данных.

Если шаблоны не играют никакой роли в обновляемых данных, рекомендуется использовать INSERT DATA вместо INSERT ... WHERE {}. Запрос ex313.ru здесь приведен только для демонстрационных целей.

Прежде чем мы рассмотрим пример такой гибкости в действии, давайте рассмотрим типичный запрос CONSTRUCT. Когда следующий запрос находит любой ресурс, который имеет оба значения ab:firstName и ab:lastName, он сохраняет URI ресурса в переменной ?person и создает новую тройку указывая, что этот ресурс является членом нашего нового класса ab:Person:

```
# filename: ex314.rq
PREFIX ab: http://learningsparql.com/ns/addressbook#

CONSTRUCT
{ ?person   a   ab:Person . }
WHERE
{
  ?person   ab:firstName   ?firstName ;
            ab:lastName    ?lastName .
 }
```

Если вы используете обработчик ARQ, чтобы запустить этот запрос к данным ex012.ttl, вы увидите следующие три тройки в результате:

```
@prefix d: <http://learningsparql.com/ns/data#> .
@prefix ab: <http://learningsparql.com/ns/addressbook#> .

d:i9771   <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>   ab:Person .
d:i0432   <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>  ab:Person .
d:i8301   <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>  ab:Person .
```

Вы можете сохранить этот результат в файл и использовать его для дальнейшей работы, а запрос CONSTRUCT не имеет никакого влияния на начальный вход.

Наш следующий пример точно такой же, как ex314.rq CONSTRUCT исключением одного, что ключевое слово CONSTRUCT изменено на INSERT:

```
# filename: ex316.ru
PREFIX ab: http://learningsparql.com/ns/addressbook#

INSERT
{ ?person   a   ab:Person . }
WHERE
{
  ?person   ab:firstName   ?firstName ;
            ab:lastName    ?lastName .
 }
```

Спецификация SPARQL использует термин "запрос на обновление", а не "запрос" для этих запросов, так что пока ex314.rq это запрос, ex316.ru это запрос на обновление. Это потому, что (как мы увидим в разделе "Поименованные графы") запрос может состоять из нескольких отдельных обновлений, или, как спецификация называет их, операций.

Вставить этот запрос в Update панель SPARQL формы Fuseki и нажмите кнопку "Выполнить обновление", чтобы запустить его. Если вы затем запустите ex311.rq "список всех троек по умолчанию", вы увидите одни и те же три тройки, которые были созданы в запросе ex314.rq, будут добавлены к набору данных по запросу ex316.ru.

Когда мы начали использовать Fuseki, мы указали ему хранить данные в постоянной базе данных, так что если вы закроете Fuseki и запустите его снова после запуска запроса обновления ex316.ru, вы обнаружите, что Fuseki все еще хранит тройки, вставленные по ex312.ru и по ex316.ru.

В начале этой главы, мы использовали кнопку Fuseki ”Upload”, чтобы загрузить весь файл сразу. Еще один способ, загрузить данные с помощью SPARQL операции загрузки LOAD, который позволяет загружать весь доступный на веб набор данных одновременно. Например, по следующей операции будут загружаться RDF данные о книге Тима Бернерс-Ли "Ткачество веб" из огромной коллекции OCLC WorldCat данных о печатных работах:

```
# filename: ex546.ru
LOAD <http://worldcat.org/oclc/41238513.ttl>
```

Вставьте этот запрос в Update панель SPARQL формы Fuseki и нажмите кнопку "Выполнить обновление", чтобы запустить его. Если вы затем запустите ex311.rq "список всех троек по умолчанию", вы увидите, что вы загрузили более 100 троек в наборе данных.

6.3. Удаление данных

SPARQL Update DELETE DATA и DELETE операции соответствуют INSERT DATA и INSERT операциям. С первым оператором, вы перечисляете конкретные тройки, чтобы удалить их, со вторым вы можете сделать это, а также использовать шаблоны троек для большей гибкости.

Прежде чем взглянуть на некоторые примеры, давайте рассмотрим некоторые из данных, вставленных с помощью операции LOAD в предыдущем примере, введя следующий запрос на участке SPARQL Query на форме Fuseki:

```
# filename: ex547.rq
SELECT * WHERE
{http://www.worldcat.org/isbn/0062515861   ?p   ?o }
```

Когда вы запустите его, вы увидите предикаты и объекты из 6 (на момент написания книги) троек, которые описывают версию книги в твердом переплете. Это включает в себя предикаты и объекты из следующих двух троек, которые показывают:

• Что это издание "ткань" (cloth), промышленный термин для твердого переплета

• Тот факт, что это URI, представляет собой тот же ресурс, что и URN

(Помните, что URN тоже URI, но они редко используются отдельно от идентификационного номера ISBN книги.)

```
<http://www.worldcat.org/isbn/0062515861>
        <http://schema.org/description>
        "cloth" .
<http://www.worldcat.org/isbn/0062515861>
        <http://www.w3.org/2002/07/owl#sameAs>
        <urn:isbn:0062515861> .
```

В публичных источниках данных, эти owl:sameAs тройки обеспечивают отличную зацепку для увязки данных о конкретной теме с различными источниками.

Следующий DELETE DATA запрос удаляет эти две тройки из набора данных:

```
# filename ex548.ru
PREFIX wci: <http://www.worldcat.org/isbn/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
DELETE DATA
{
   wci:0062515861   <http://schema.org/description>   "cloth" ;
                    owl:sameAs          <urn:isbn:0062515861> .
}
```

После его запуска, попробуйте выполнить ex547.rq запрос SELECT, и вы увидите в результате на две тройки меньше, потому что их только что удалили.

Следующий запрос DELETE будет делать то же самое, что и запрос DELETE DATA в ex548.ru:

```
# filename ex549.ru
PREFIX wci: <http://www.worldcat.org/isbn/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
DELETE
{
   wci:0062515861   <http://schema.org/description>   "cloth" ;
                    owl:sameAs          <urn:isbn:0062515861> .
}
WHERE {}
```

По аналогии с запросами INSERT DATA и INSERT, DELETE DATA версия может быть быстрее, если вы работаете с большим количеством данных, и DELETE версия является более гибкой. Если вы точно знаете, какие тройки вы хотите удалить, и не будете добавлять какие-либо шаблоны троек между фигурными скобками WHERE, то лучше использовать DELETE DATA.

Давайте исследуем эту гибкость. Следующий запрос на обновление содержит шаблон из одной тройки в фигурных скобках WHERE, который для всех троек включает http://www.worldcat.org/isbn/0062515861 в качестве субъекта, а DELETE условие удаляет тройки удовлетворяющие тому же шаблону. Если перед запуском этого запроса вы запустите запрос ex311.rq, то увидите, по крайней мере три оставшихся тройки, которые соответствуют этому шаблону. Затем запустите этот запрос DELETE обновления:

```
# filename: ex550.ru
DELETE { <http://www.worldcat.org/isbn/0062515861>   ?p   ?o }
WHERE { <http://www.worldcat.org/isbn/0062515861>   ?p   ?o }
```

Повторный запуск ex311.rq показывает, что эти тройки удалены. (Вы можете видеть по крайней мере одну тройку, где этот URI, является объектом, но этот запрос был только на удаление тех троек, где он был субъектом).


Оператор DELETE также как, WHERE, CONSTRUCT и INSERT операторы, может иметь столько троек в шаблоне, сколько необходимо.

Язык SPARQL Update предлагает оператор DELETE WHERE для удаления троек, которые соответствуют определенному шаблону. Этот запрос на обновление не имеет шаблона троек после ключевого слова DELETE и предполагает, что вы хотите удалить любые тройки удовлетворяющие шаблон в WHERE. Это означает, что следующий запрос приведет к аналогичному результату, что и запрос на обновление ex550.ru:

```
# filename: ex551.ru
DELETE WHERE { <http://www.worldcat.org/isbn/0062515861>   ?p   ?o }
```

Мы увидим более интересные использования DELETE с шаблоном троек в следующем разделе о коррекции данных, которая на самом деле является операцией удаления в сочетании с операцией INSERT. Но сначала давайте посмотрим на простую, но самую мощную команду удаления всех троек: CLEAR.

Команда CLEAR удаляет тройки графа. Введите следующий запрос, чтобы удалить все тройки:

```
# filename: ex324.ru
CLEAR DEFAULT
```

После этого, запустив ex311.rq запрос вы убедитесь, что все тройки удалены.

6.4. Изменение существующих данных

Изменения в существующих тройках выполняются как операции удаления с последующими операциями вставки в одном запросе обновления. Спецификация относящаяся к этому обозначается как "DELETE/INSERT". Ее будет легче обсуждать на следующем примере:

```
# filename: ex325.ru
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX ab: <http://learningsparql.com/ns/addressbook#>

DELETE
{ ?s    ab:email    ?o }
INSERT
{ ?s    foaf:mbox    ?o }
WHERE
{ ?s    ab:email    ?o }
```

Процессор запросов SPARQL оценивает образец графа в операторе WHERE, выполняет все, что в пункте DELETE, а затем выполняет инструкции в операторе INSERT. Для запроса на обновление, представленного выше, это будет:

1. Найти все тройки с предикатом ab:email.
2. Удалить их.
3. Вставить новые тройки с тем же субъектом и объектом и предикаттом foaf:mbox.

Чтобы испытать этот запрос на обновление выполните следующие действия:

1. Если вы в настоящее время есть какие-либо тройки в графе по умолчанию, используйте команду CLEAR, описанную в предыдущем разделе, чтобы удалить их.
2. Используйте на форме Fuseki кнопки "Choose File" и "Upload" для загрузки адресной книги выборочных данных ex012.ttl.
3. Выполните запрос ex311.rq в панели запросов SPARQL Query, чтобы посмотреть, что у вас присутствует в наборе данных.
4. Выполните следующую конструкцию запроса в той же панели, чтобы увидеть, какие тройки будут созданы в INSERT/DELETE запросе на обновление ex325.ru:

```
# filename: ex326.rq
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX ab: <http://learningsparql.com/ns/addressbook#>

CONSTRUCT
{ ?s    foaf:mbox    ?o }
WHERE
{?s    ab:email    ?o }
```

Это хороший способ, чтобы увидеть, что будет добавлено в ваш набор данных, прежде чем вы на самом деле измените его.

5. Вставить запрос на обновление ex325.ru в части SPARQL Update панели и нажмите "Perform update", чтобы запустить его.
6. Запустите запрос ex311.rq снова, чтобы увидеть, как тройки теперь у вас есть. Вы увидите, что, в сущности, запрос на обновление преобразует ab:email в более известный FOAF эквивалент.

Даже несмотря на то, что удаление произойдет до вставок, графовый шаблон INSERT по-прежнему имеет всю информацию, хранящуюся в разделе WHERE.

Прежде чем писать INSERT или INSERT/DELETE запрос на обновление, использование запроса CONSTRUCT не только хороший способ получить предварительный просмотр того, что будет добавлено к данным. Он также может служить в качестве прототипа, который в конечном итоге вы можете учесть в запросе на обновление INSERT. (Помните, что при использовании веб-интерфейса Fuseki, выполнять запрос на обновление вы будете в другой части формы нежели тот, который вы будете использовать для CONSTRUCT запроса).

Давайте посмотрим на чуть более сложный пример. Следующий набор данных определяет три узла небольшой таксономии с использованием онтологии SKOS. Каждая концепция включает в себя skos:prefLabel ("предпочтительная метка") свойство, значением которого является строковый литерал:

```
# filename: ex327.ttl
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix d: <http://learningsparql.com/ns/data#> .
        d:c1  a  skos:Concept ;
                 skos:prefLabel "Mammal" .

        d:c2  a  skos:Concept ;
                 skos:prefLabel "Dog" ;
                 skos:broader d:c1 .

        d:c3  a  skos:Concept ;
                 skos:prefLabel "Cat" ;
                 skos:broader d:c1 .
```

Тройку {d:c2 skos:broader d:c1} можно понять как d:c2 шире, чем d:c1, но это в действительности означает, что d:c2 имеет skos:broader значение d:c1. Это может показаться нелогичным, но это согласуется с RDF практикой, например, d:i0432 ab:firstName "Richard" означает, что d:i0432 имеет ab:firstName значение "Ричард".

Что делать, если вы хотите назначить метаданные предпочтительным меткам, таким как "кошка" и "собака"? RDF позволяет назначать метаданные чему угодно, что можно выразить как URI, потому что вы можете создать тройки, которые имеют этот URI в качестве субъекта и затем использовать любые имена и значения свойств в качестве предикатов и объектов этих троек. Поскольку "кошка" и "собака" являются строки, вы не можете использовать их в качестве субъектов троек.

Для того, чтобы иметь возможность назначить метаданные отдельным терминам, W3C SKOS eXtension for Labels (SKOS-XL) спецификация расширяет SKOS позволяя использовать различные виды привилегированных этикеткеток: вместо того, чтобы быть строковым литералом, термин можно рассматривать как еще один ресурс (член xl:Literal класса) с собственным URI и собственными свойствами. Наиболее важным из этих свойств является xl:literalForm, которое сохраняет фактический термин, например, "кошка" или "собака". Затем этот xl:Label экземпляр может иметь, как и многие другие свойства, пары имя/значение, какие вы хотите использовать для хранения метаданных о термине. Образец данных SKOS, пересмотренный как SKOS-XL, может выглядеть следующим образом:

```
# filename: ex328.ttl
```

```
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix xl: <http://www.w3.org/2008/05/skos-xl#> .
@prefix d: <http://learningsparql.com/ns/data#>
@prefix dc: <http://purl.org/dc/elements/1.1/> .

d:c1      a      skos:Concept ;
                 xl:prefLabel d:label1 .

d:c2      a      skos:Concept ;
                 xl:prefLabel d:label2 ;
                 skos:broader d:c1 .

d:c3      a      skos:Concept ;
                 xl:prefLabel d:label3 ;
                 skos:broader d:c1 .

d:label1  a      xl:Label ;
                 xl:literalForm "Mammal" ;
                 dc:source <http://en.wikipedia.org/wiki/Mammal> .

d:label2  a      xl:Label ; xl:literalForm "Dog" ;
                 dc:source <http://en.wikipedia.org/wiki/Dog> .

d:label3  a      xl:Label ; xl:literalForm "Cat" ;
                 dc:source <http://en.wikipedia.org/wiki/Cat> .
```

Обратите внимание, что этот пример SKOS-XL включает в себя дополнительные тройки относительно источника каждого термина, используя свойства Dublin Core, чтобы показать гибкость SKOS-XL. Вы можете добавить для этих трминов все метаданные какие хотите, от любых пространств имен. Дополнительная сложность SKOS-XL помешала этой спецификации получить большую поддержку.

Если очистить данные в наборе данных Fuseki (смотри запрос на обновление ex324.ru) и загрузить данные ex327.ttl (данные SKOS, не ex328.ttl SKOS-XL) в ней, то вы затем можете запустить следующий запрос обновления в SPARQL Update панели и конвертировать сохраненные данные SKOS в данные SKOS-XL:

```
# filename: ex329.ru
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX xl: <http://www.w3.org/2008/05/skos-xl#>

DELETE
{ ?concept   skos:prefLabel   ?labelString . }
INSERT
{
  ?newURI   a                 xl:Label ;
            xl:literalForm    ?labelString .
  ?concept  xl:prefLabel      ?newURI .
 }
WHERE
{
```

```
    ?concept   skos:prefLabel   ?labelString .
    BIND (URI(CONCAT("http://learningsparql.com/ns/data#",
                                ENCODE_FOR_URI(str(?labelString)))
                ) AS ?newURI)
}
```

(Здесь не добавлено никаких исходных значений Dublin Core, которые включены в примере ex328.ttl, чтобы показать, как данные SKOS-XL могут включать дополнительные метаданные). Этот запрос обновление в операторе WHERE находит текущие использования предикта skos:prefLabel и поскольку новые xl:Label ресурсы, которые их заменяют, будут нуждаться в URI, чтобы определить их, создаются эти URI с использованием комбинации функций, которые изучались в главе 5.

Для того, чтобы создать URI для новых xl:Label, ex329.ru использует переменную ?labelString для передачи значения skos:prefLabel функции ENCODE_FOR_URI (), но можно было бы использовать также и другие методы. Здесь нет необходимости использовать это значение в URI.

Оператор DELETE удаляет тройки, которые имеют эти skos:prefLabel предикаты. Далее, оператор INSERT создает новые члены xl:Label, назначая оригинальную предпочтительную строковую метку xl:literalForm в качестве значение для каждого нового xl:Label экземпляра и делая этот xl:Label экземпляр значением xl:prefLabel того же концепта (понятия).

В созданных тройках, концепты предпочтительных меток задаются с помощью свойства xl:prefLabel, а не skos:prefLabel. Это другое свойство объявленное как часть спецификации SKOS-XL, так же, как xl:Label представляет собой новый класс определенный в этом описании.

После запуска ex329.ru INSERT/DELETE запроса на обновление, список полученный с помощью ex311.rq будет показывать, что три понятия имеют xl:prefLabel свойства, а не свойства skos:prefLabel, и значения xl:prefLabel являются ресурсами своих собственных данных. Например, концепция http://learningsparql.com/ns/data#c3 имеет xl:prefLabel значение http://learningsparql.com/ns/data#Cat, которое является xl:Label позициb, на которой располагается xl:literalForm значение «Cat»:

s	p	o
<http://learningsparql.com/ns/ data#c3>	<http://www.w3.org/1999/02/22-rdf- syntax-ns#type>	<http://www.w3.org/2004/02/skos/ core#Concept>
<http://learningsparql.com/ns/ data#c3>	<http://www.w3.org/2004/02/skos/ core#broader>	<http://learningsparql.com/ns/ data#c1>
<http://learningsparql.com/ns/ data#c3>	<http://www.w3.org/2008/05/skos- xl#prefLabel>	<http://learningsparql.com/ns/ data#Cat>
<http://learningsparql.com/ns/ data#c1>	<http://www.w3.org/1999/02/22-rdf- syntax-ns#type>	<http://www.w3.org/2004/02/skos/ core#Concept>
<http://learningsparql.com/ns/ data#c1>	<http://www.w3.org/2008/05/skos- xl#prefLabel>	<http://learningsparql.com/ns/ data#Mammal>
<http://learningsparql.com/ns/ data#c2>	<http://www.w3.org/1999/02/22-rdf- syntax-ns#type>	<http://www.w3.org/2004/02/skos/ core#Concept>
<http://learningsparql.com/ns/ data#c2>	<http://www.w3.org/2004/02/skos/ core#broader>	<http://learningsparql.com/ns/ data#c1>
<http://learningsparql.com/ns/ data#c2>	<http://www.w3.org/2008/05/skos- xl#prefLabel>	<http://learningsparql.com/ns/ data#Dog>
<http://learningsparql.com/ns/ data#Cat>	<http://www.w3.org/1999/02/22-rdf- syntax-ns#type>	<http://www.w3.org/2008/05/skos- xl#Label>
<http://learningsparql.com/ns/ data#Cat>	<http://www.w3.org/2008/05/skos- xl#literalForm>	“Cat”
<http://learningsparql.com/ns/ data#Dog>	<http://www.w3.org/1999/02/22-rdf- syntax-ns#type>	<http://www.w3.org/2008/05/skos- xl#Label>
<http://learningsparql.com/ns/ data#Dog>	<http://www.w3.org/2008/05/skos- xl#literalForm>	“Dog”
<http://learningsparql.com/ns/ data#Mammal>	<http://www.w3.org/1999/02/22-rdf- syntax-ns#type>	<http://www.w3.org/2008/05/skos- xl#Label>
<http://learningsparql.com/ns/ data#Mammal>	<http://www.w3.org/2008/05/skos- xl#literalForm>	“Mammal”

6.5. Именованные графы

SPARQL 1.1 Graph Store HTTP Protocol спецификация W3C расширяет SPARQL

протокол, чтобы для передачи сообщений о графах между клиентом и процессором SPARQL. Поскольку он включает в себя HTTP способы сказать "вот граф, для добавления к набору данных" или "удалить граф http://my/fine/graph из набора данных," он предлагает альтернативный подход к методам языка SPARQL Update для работы с целыми графами, которые описываются в этом разделе. Смотрите раздел "SPARQL и HTTP" для получения более подробной спецификации.

Чтобы получить практические навыки в создании, добавлении к, удалении из и замене троек в поименованных графах (а также удалении и замене целых графов), мы начнем с соответствующих ключевых слов SPARQL Update, используя простые фиктивные данные, в которых ресурсы менуемые d:x и d:w получают dm:tag значения "one" и "two". Затем мы рассмотрим некоторые из операций обновления с помощью более реалистичны[данные.

Чтобы обеспечить основу для наших ближайших экспериментов, следующий запрос обновление делает две операции: она очищает тройки в графе по умолчанию, а затем загружает туда две тройки.

SPARQL Update позволяет подключить несколько операций через запятую.

Выполним следующий запрос обновления в части SPARQL Update формы Fuseki:

```
# filename: ex330.ru
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>

CLEAR DEFAULT;

INSERT DATA
{
  d:x   dm:tag   "one" .
  d:x   dm:tag   "two" .
}
```

Теперь мы вставим новые тройки в конкретный именованный граф. Как в запросах именованных графов, которые мы видели в главе 3, мы имеем в виду шаблон троек из определенного графа, предшествующего шаблону графа с ключевым словом GRAPH и с именем граф. Мы будем использовать ту же операцию INSERT, которой мы пользовались ранее для вставки троек в граф по умолчанию.

Когда вы вставляете тройки в граф, который не существует, процессор SPARQL создает граф.

Следующий запрос на обновление создаст d:g1 граф и добавит указанную тройку:

```
# filename: ex331.ru
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>

INSERT DATA
{  GRAPH d:g1
```

```
  { d:x   dm:tag   "three" }
}
```

Поскольку именованные графы есть URI, как и любые другие, вы можете представить их с помощью префиксов имен. В зависимости от того, как вы решите организовать ваши данные, вы можете разместить имя графа в собственном пространстве имен, но для простоты примера мы разместим их в том же пространства имен данных http://learningsparql .com/ns/data#, где находятся и другие выборочные данные.

Следующий запрос на обновление делает то же самое, что и ex331.ru, но вместо ключевого слова DATA после INSERT, он имеет WHERE:

```
# filename: ex543.ru
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>

INSERT
{   GRAPH   d:g1
    { d:x   dm:tag   "three" }
}
WHERE {}
```

Разница между синтаксисом ex331.ru и ex543.ru похожа на разницу между запросами ex312.ru и ex313.ru, которые использовали эти два метода, чтобы вставить тройки в граф по умолчанию вместо имени одного графа. Первый использует INSERT DATA, потому что нет никакой необходимости в операторе WHERE, чтобы указать условия. Второй имеет INSERT без DATA и пустой оператор WHERE, где вы можете добавить шаблон графа для управления логикой обновления.

После запуска ex331.ru или ex543.ru в части SPARQL Update формы Fuseki, выполните следующий SELECT запрос на выборку в части SPARQL Query форм Fuseki:

```
# filename: ex332.rq

SELECT   ?g  ?s ?p ?o
WHERE
{
  { ?s   ?p   ?o }
  UNION
  { GRAPH ?g { ?s   ?p   ?o } }
}
```

Это действительно запрос на Список Всех Троек, потому что он перечисляет объединение всех троек в графе по умолчанию и всех троек в любом имени графа вместе с соответствующими именами графа. С троеками, вставленными в двух предыдущих запросах INSERTи SELECT, это запрос покажет вам следующее:

g	s	p	o
	<http://learningsparql.com/ns/data#x>	<http://learningsparql.com/ns/demo#tag>	“one”
	<http://learningsparql.com/ns/data#x>	<http://learningsparql.com/ns/demo#tag>	“two”
<http://learningsparql.com/ns/ data#g1>	<http://learningsparql.com/ns/data#x>	<http://learningsparql.com/ns/demo#tag>	“three”

Создание закладки результатов запроса ex332.rq - простой способ вернуться к запросу всякий раз, когда это необходимо. Это может быть удобно для анализа последствий оставшихся запросов обновления из этой главы.

Следующий запрос на обновление напоминает прошлый запрос на обновление INSERT, но вставляется другая тройка в тот же граф:

```
# filename: ex333.ru
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>

INSERT DATA
{  GRAPH d:g1
   { d:x    dm:tag    "four" . }
}
```

После того как вы выполните его и запустите запрос Список Всех Троек, вы увидите, что граф d:g1 теперь имеет две тройки:

g	s	p	o
	<http://learningsparql.com/ns/data#x>	<http://learningsparql.com/ns/demo#tag>	“one”
	<http://learningsparql.com/ns/data#x>	<http://learningsparql.com/ns/demo#tag>	“two”
<http://learningsparql.com/ns/ data#g1>	<http://learningsparql.com/ns/data#x>	<http://learningsparql.com/ns/demo#tag>	“three”
<http://learningsparql.com/ns/ data#g1>	<http://learningsparql.com/ns/data#x>	<http://learningsparql.com/ns/demo#tag>	“four”

6.5.1. Удаление именованных графов

Прежде чем мы узнаем, как удалить граф и добавить второй именованный граф, выполним следующий запрос на обновление и запустим запрос Список Всех Троек, чтобы увидеть, что там содержится:

```
# filename: ex334.ru
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>

INSERT DATA
{  GRAPH d:g2
   {
    d:x dm:tag "five" .
    d:x dm:tag "six" .
   }
}
```

Операция DROP удаляет граф из набора данных. Следующий запрос удаляет первый именованный граф; запустите его, а затем запустите запрос Список Всех Троек и вы увидите, что тройки графа по умолчанию и тройки именованного графа d:g2 по-прежнему существует, а тройки графа d:g1 удалены:

```
#filename: ex335.ru
PREFIX d: <http://learningsparql.com/ns/data#>

DROP   GRAPH   d:g1
```

DROP DEFAULT очищает граф по умолчанию. (Вы не можете на самом деле удалить граф по умолчанию, потому что она всегда существует, даже если он пуст). Запустите запрос ex332.rq - Список Всех Троек и вы увидите, что граф троек d:g2 по-прежнему существует, но тройки графа по умолчанию отсутствуют:

```
# filename: ex336.ru
DROP DEFAULT
```

Запустите запрос ex330.ru обновления снова для вставки некоторых троек обратно в граф по умолчанию, а затем запустите запрос ex332.rq - Список Всех Троек, чтобы убедиться, что они там. Теперь вы готовы для самого мощного запроса на обновление всех: DROP ALL, который удаляет граф по умолчанию и все именованные графы, другими словами, удаляет все.

```
# filename: ex337.ru
DROP ALL
```

Попробуйте выполнит его, а затем запустить запрос Список Всех Троек и вы увидите, как много эти два коротких слова могут сделать для набора данных.

Любая команда на любом языке, которая говорит "удалить все" представляет из себя что-то, чтобы быть осторожным с ней. Всегда стоит сделать дополнительную паузу перед непосредственным ее выполнением. SPARQL Update не предлагает никаких операций UNDO, хотя ваш triplestore, как и любая СУБД, может предложить что-то вроде этого, как часть набора команд фиксации (commit) и отката (rollback).

Далее, мы узнаем как удалить все именованные графы и оставить граф по умолчанию. Для этого запишем все тройки из последних запросов обновления INSERT обратно в Fuseki с помощью следующего запроса на обновление. Запустим его, а затем запустим запрос ex332.rq - Список Всех Троек, чтобы увидеть как шесть троек разбросаны по графу по умолчанию и двум именованным графам:

```
# filename: ex338.ru
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>

INSERT DATA
{
d:x dm:tag "one" .
d:x dm:tag "two" .

  GRAPH d:g1
  {
    d:x dm:tag "three" .
    d:x dm:tag "four" .
  }

  GRAPH d:g2
  {
    d:x dm:tag "five" .
    d:x dm:tag "six" .
  }
```

}

Команда DROP NAMED удаляет все именованные графы. Запустите следующий запрос, а затем выполните запрос Список Всех Троек. Вы увидите, что именованных графов d:g1 и d:g2 нет, но две тройки графа по умолчанию все еще там:

```
# filename: ex339.ru
DROP NAMED
```

Ранее мы узнали, как CLEAR DEFAULT очищает все тройки в графе по умолчанию. Если вы подставите ключевое слово GRAPH и имя конкретного графа вместо ключевого слова DEFAULT, вы можете очистить тройки в этом графе. Например, CLEAR GRAPH <http://mygraph> удалит все тройки из именованного графа http://mygraph. Также, CLEAR NAMED удалит трйки из всех именованных графов, а CLEAR ALL удаляет все тройки из всех именованных графов и графа по умолчанию.

Другой способ создания графов с помощью операции CREATE GRAPH, "для хранилищ, которые записывают пустые графы", в соответствии со спецификацией SPARQL Update. (Если задуматься о разнице между операциями DROP и CLEAR, то DROP удаляет полные графы, а CLEAR удаляет тройки из них, оставляя пустые графы "для хранилищ, которые записывают пустые графы")

В следующем запросе на обновление будет создан новый d:g3 граф. Если выполнить его с Fuseki 0.2.6, а затем запустить запрос ex332.rq - Список Всех Троек, то вы не увидите никаких признаков того, что Fuseki записал этот граф:

```
# filename: ex340.ru
PREFIX d: <http://learningsparql.com/ns/data#>
CREATE GRAPH   d:g3
```

После запуска запроса обновления ex340.ru, следующий запрос выполненный в панели SPARQL Query Fuseki должен перечислить все именованные графы с тройками или без каких-либо троек в них. Результат не имеет никаких признаков, что Fuseki 0.2.6 знает о d:g3 графе, но команду CREATE стоит попробовать с другими RDF-хранилищами, которые вы используете, и с будущими версиями Fuseki:

```
# filename: ex341.rq
SELECT ?g
WHERE
{  GRAPH  ?g  {}  }
```

6.5.2. Кратко о синтаксисе WITH и USING для именованных графов

Ключевое слово WITH указывает процессору SPARQL имя графа, который нужно использовать всякий раз, когда он не назван в остальной части запроса на обновление. Например, следующий запрос делает то же самое, что и запрос на обновление ex334.ru рассмотренный ранее, а именно вставляет две показанные тройки в граф d:g2:

```
# filename: ex342.ru
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>

WITH d:g2
```

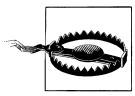
```
INSERT
{
  d:x   dm:tag   "five" .
  d:x   dm:tag   "six" .
}
WHERE {}
```

INSERT не будет работать с ключевым словом WITH, поэтому ex342.ru имеет WHERE {} в конце. Вы можете вставить шаблон троек в фигурные скобки WHERE, а затем ссылаться на переменные из этого шаблона в операторе INSERT.

Если запрос обновления только упоминает конкретное имя графа, как это делает ex334.ru с d:g2, называя граф с ключевым словом WITH, вместо ключевого слова GRAPH, то это не делает большой разницы. Когда мы получаем к обновлению и замене тройки в графах, WITH может сделать запросы менее многословным, сохраняя вас от именования графа снова и снова.

Ключевое слово USING делает для запросов обновления то же, что делает FROM для запросов выбора SELECT: он определяет граф, который анализируется условиями WHERE. Прежде, чем мы увидеть его в действии, запустим следующий запрос на обновление, который добавляет две новых тройки в граф по умолчанию. Как и две тройки в графе d:g2, эти новые тройки имеют значения объекта "five" и "six", но у них субъекты d:w вместо d:x.

```
# filename: ex343.ru
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>

INSERT DATA
{
  d:w   dm:tag   "five" .
  d:w   dm:tag   "six" .
}
```



Вы можете использовать ключевые слова USING и WITH вместе, в одном обновлении, но убедитесь, что вы понимаете, какие части вашего графа к чему относятся. Часто проще всего избежать использования их вместе.

Следующий запрос на обновление включает условие WHERE для троек с предикатом dm:tag и объектами "five" и "six". Поскольку используется ключевое слово USING, они отыскиваются в графе d:g2. Затем копии этих троек вставляются в новый граф d:g4:

```
# filename: ex344.ru
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>

INSERT
{  GRAPH d:g4
   { ?s   dm:tag   "five", "six" . }
}
```

```
USING d:g2
WHERE
{
  ?s   dm:tag   "five" .
  ?s   dm:tag   "six" .
}
```

После выполнения этого запроса, запустим запрос ex332.rq - Список Всех Троек и увидим, что две тройки вставлены в граф d:g4 и обе имеют субъекты d:x. Хотя граф по умолчанию имеет тройки, которые соответствуют шаблону WHERE (с субъектом d:w, которые были вставлены запросом на обновление ex343.ru), но ключевое слово USING специально сказало процессору SPARQL смотреть только в граф d:g2.

Если ключевое слово USING в запросе на обновление, как ключевое слово FROM в запросе SELECT, то ключевое слово USING NAMED, как FROM NAMED: оно определяет граф, на который следует ссылаться по имени. Если бы запрос на обновление ex344.ru сказал USING NAMED, а не просто USING, то процессор запросов не нашел бы эти тройки в графе d:g2, если бы его имя не было явно включено в WHERE, как это сделано в следующем запросе:

```
# filename: ex345.ru
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>

INSERT
{ GRAPH d:g4
  { ?s   dm:tag   "five", "six" . }
}
USING NAMED d:g2
WHERE
{   GRAPH d:g2
    {
      ?s   dm:tag   "five" .
      ?s   dm:tag   "six" .
    }
}
```

6.5.3. Копирование и перемещение целых графов

Операции SPARQL Update копирования COPY и перемещения MOVE позволяют копировать и перемещать тройки между именованными графами или между графом по умолчанию и именованным графом.

Можно сформировать пример данных, чтобы увидеть действие этих операторов, сначала запустив команду DROP ALL, показанную в запросе ex337.ru, а затем запустить обновление ex338.ru. Этот запрос вставляет две тройки в граф по умолчанию, дв в поименованный граф d:g1, и две в поименованный граф d:g2. Запустим запрос ex332.rq - Список Всех Троек и он покажет следующие тройки и графы, которым они принадлежат:

```
-----------------------------------------
| g      | s    | p       | o        |
=========================================
|        | d:x  | dm:tag  | "one"    |
|        | d:x  | dm:tag  | "two"    |
| d:g1   | d:x  | dm:tag  | "three"  |
```

```
| d:g1  | d:x | dm:tag | "four"  |
| d:g2  | d:x | dm:tag | "five"  |
| d:g2  | d:x | dm:tag | "six"   |
-----------------------------------------
```

Операция COPY копирует тройки из одного графа в другой, заменяя все существующие тройки в графе назначения. Если граф назначения не существует, он будет создан. Следующий запрос обновления запросить копии троек из графика по умолчанию для графа d:g2:

```
# filename: ex503.ru
PREFIX d: <http://learningsparql.com/ns/data#>
COPY DEFAULT TO d:g2
```

Довольно просто. После его выполнения, выполнив запрос Список Всех Троек, мы увидим, что "five" и "six" тройки, которые были в D:G2 граф больше не существует, и вместо них скопированы "one" и "two" тройки из графа по умолчанию:

```
-----------------------------------------
| g      | s   | p       | o       |
=========================================
|        | d:x | dm:tag | "one"   |
|        | d:x | dm:tag | "two"   |
| d:g1   | d:x | dm:tag | "three" |
| d:g1   | d:x | dm:tag | "four"  |
| d:g2   | d:x | dm:tag | "one"   |
| d:g2   | d:x | dm:tag | "two"   |
-----------------------------------------
```

Операция MOVE перемещает тройки из одного графа в другой, а также заменяет существующие тройки в графе назначения. Как и при копировании, если граф назначения не существует, он будет создан. Следующий запрос на обновление перемещает тройки из графа d:g2 в граф d:g1:

```
# filename: ex505.ru
PREFIX d: <http://learningsparql.com/ns/data#>
MOVE   d:g2   TO   d:g1
```

При запуске этого запроса с результатами предыдущего запроса COPY, мы увидим, что ничего не осталось в графе d:g2, а граф d:g1 имеет тройки, которые раньше были в d: g2:

```
-----------------------------------------
| g      | s   | p       | o       |
=========================================
|        | d:x | dm:tag | "one"   |
|        | d:x | dm:tag | "two"   |
| d:g1   | d:x | dm:tag | "one"   |
| d:g1   | d:x | dm:tag | "two"   |
-----------------------------------------
```

Можно увидеть больше примеров на использование операций COPY и MOV в SPARQL Working Group, но эти примеры в основном различные комбинации перемещения троек между графом по умолчанию и именованными графами.

6.5.4. Удаление и замена троек в именованных графах

Прежде чем тестировать некоторые запросы, которые удаляют тройки из конкретных графов, запустите запрос обновления DROP ALL (ex337.ru), чтобы очистить набор данных, затем запустите запрос на обновление ex338.ru, который добавляет тройки в два новых графа и граф по умолчанию, и наконец запустите запрос ex332.rq - Список Всех Троек и увидите данные, которые мы будем удалять и заменять.

Так же, как можно использовать DELETE DATA, когда вы точно знаете, какие тройки вы хотите удалить из графа по умолчанию, вы также можете использовать этот оператор, когда точно известно, какие тройки необходимо удалить из именованных графов. Следующий запрос удалит указанную в нем тройку из графа d:g2:

```
# filename: ex346.ru
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>

DELETE DATA
{   GRAPH  d:g2
    { d:x    dm:tag    "six" }
}
```

Мы также видели, что ключевое слово DELETE без ключевого слова DATA позволяет удалять тройки соответствующие шаблону троек. Оно также работает с именованными графами, причем можно даже использовать переменную вместо имени графа, как это сделано в следующем запросе:

```
# filename: ex347.ru

DELETE
{   GRAPH ?g   { ?s    ?p    "three" } }
WHERE
{   GRAPH ?g   { ?s    ?p    "three" } }
```

После запуска этого запроса, и получения Списка Всех Троек, можно увидеть, что тройка с объектом «three» удалена из графа d:g1.

Мы видели, что ключевое слово WITH позволяет указать процессору SPARQL имя графа, который надо использовать всякий раз, когда граф явно не назван в запросе. В то время как запрос на обновление ex347.ru было определял граф в обоих операторах DELETE и WHERE, следующий за ним запрос определяет граф только раз, и оба оператора DELETE и WHERE положения знают, что они должны использовать граф d: g1. Если вы запустите этот запрос, а затем запрос ex332.rq - Список Всех Троек, вы увидите, что тройки с объектом "four" удален из этого графа:

```
# filename: ex348.ru
PREFIX d: <http://learningsparql.com/ns/data#>

WITH d:g1
```

```
DELETE { ?s   ?p   "four"}
WHERE { ?s   ?p   "four"}
```

Замена троек в названных графах представляет собой комбинацию их удаления и вставки, как при замене тройки в графе по умолчанию. Это тот случай, когда ключевое слово WITH становится особенно полезным. Во-первых, давайте посмотрим на примере явного способа замены тройки в определенном графе:

```
# filename: ex349.ru
PREFIX d: <http://learningsparql.com/ns/data#>

DELETE
{  GRAPH d:g2 { ?s   ?p   "five" }  }
INSERT
{  GRAPH d:g2 { ?s   ?p   "cinco" }  }
 WHERE
{  GRAPH d:g2 { ?s   ?p   "five" }  }
```

Этот запрос обновления ищет тройки в графе D:G2, которые имеют "five" в качестве объекта и заменяет их тройками, имеющими тот же субъект и предикат, но объект "cinco". Запустите этот запрос, а затем запрос ex332.rq - Список Всех Троек, чтобы увидеть результат обновления данных.

Следующий запрос на обновление делает то же самое, но использует ключевое слово WITH так, что он имеет только одно упомянание графа D:g2. Без ключевого слова USING, чтобы переопределить этот выбор графа, операторы DELETE, INSERT и WHERE выполняют свои действия на этом графе:

```
# filename: ex350.ru
PREFIX d: <http://learningsparql.com/ns/data#>

WITH d:g2
DELETE
{ ?s   ?p   "five" }
INSERT
{ ?s   ?p   "cinco" }
WHERE
{ ?s   ?p   "five" }
```

Давайте попробуем некоторые запросы на обновление графа с более реалистичным набором данных. В следующем запросе на обновление удаляются все графы, по умолчанию и именованные, и создаются три новых именованных графа:

```
# filename: ex351.ru
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX g: <http://learningsparql.com/graphs/>

DROP ALL;

INSERT DATA
 {

# people
```

```
GRAPH g:people
  {
  d:i0432 ab:firstName "Richard" ;
          ab:lastName "Mutt" ;
          ab:email"richard49@hotmail.com" .

  d:i9771 ab:firstName "Cindy" ;
          ab:lastName "Marshall" ;
          ab:email"cindym@gmail.com".

  d:i8301 ab:firstName "Craig" ;
          ab:lastName "Ellis" ;
          ab:email"c.ellis@usairwaysgroup.com" .
  }

# courses
GRAPH g:courses
  {
  d:course34    ab:courseTitle   "Modeling Data with OWL" .
  d:course71    ab:courseTitle    "Enhancing Websites with RDFa" .
  d:course59    ab:courseTitle   "Using SPARQL with non-RDF Data" .
  d:course85    ab:courseTitle   "Updating Data with SPARQL" .
  }

# who's taking which courses
GRAPH g:enrollment
  {
 d:i8301        ab:takingCourse        d:course59 .
  d:i9771       ab:takingCourse        d:course34 .
  d:i0432       ab:takingCourse        d:course85 .
  d:i0432       ab:takingCourse        d:course59 .
  d:i9771       ab:takingCourse        d:course59 .
  }
}
```

Запустите запрос ex332.rq - Список Всех Троек, чтобы узнать, с чем вы будете работать тестируя следующие несколько примеров.

Теперь, давайте скажем, что у нас есть приложение, используемое студентами, чтобы записаться на курсы. Они заполняют веб-формы для поиска и подписки на эти курсы, и база реализуется как triplestore. (Мы узнаем больше о создании таких приложений в главе 10.)

Вскоре после добавления данных в ex351.ru к набору данных нашего приложения отслеживания изменений, мы слышим от отдела образования. Они говорят нам, что есть некоторые поправки к списку курсов: курс 34 теперь называется "Modeling Data with RDFS and OWL" и есть новый курс 86, который называется корректировки "Querying and Updating Named Graphs". Мы могли бы сделать обновление в соответствии с этой просьбой:

```
# filename: ex352.ru
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX g: <http://learningsparql.com/graphs/>
```

```
DELETE
{   GRAPH g:courses
        { d:course34    ab:courseTitle    ?courseTitle }
}
INSERT
{   GRAPH g:courses
        { d:course34    ab:courseTitle    "Modeling Data with RDFS and OWL" . }
}
WHERE
{   GRAPH g:courses
        { d:course34    ab:courseTitle    ?courseTitle }
};
INSERT DATA
{   GRAPH g:courses
        { d:course86 ab:courseTitle "Querying and Updating Named Graphs" . }
}
```

Здесь использованы методы, которые мы изучили ранее для обновления троек в графе. Удалили существующее название 34 курса, и затем добавили новое. Также добавили новый курс 86 и его название.

Другой подход для обновления данных состоит из двух шагов. В следующем запросе на обновление удаляется весь граф, а затем вставляется исправленная версия:

```
# filename: ex353.ru
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX d: <http://learningsparql.com/ns/data#>

DROP GRAPH <http://learningsparql.com/graphs/courses> ;

INSERT DATA
{
  GRAPH <http://learningsparql.com/graphs/courses>
  {
        d:course34 ab:courseTitle "Modeling Data with RDFS and OWL" .
        d:course71 ab:courseTitle "Enhancing Websites with RDFa" .
        d:course59 ab:courseTitle "Using SPARQL with non-RDF Data" .
        d:course85 ab:courseTitle "Updating Data with SPARQL" .
        d:course86 ab:courseTitle "Querying and Updating Named Graphs" .
  }
}
```

После использования любого из приведенных запросов, выполните следующий запрос SELECT, чтобы узнать, кто какие курсы выбрал:

```
# filename: ex354.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX g: <http://learningsparql.com/graphs/>

SELECT ?first ?last ?courseTitle
WHERE
{
  { GRAPH g:people
```

```
        { ?student   ab:firstName   ?first ;
                     ab:lastName    ?last .
        }
      }
    { GRAPH g:enrollment
        { ?student   ab:takingCourse   ?course . }
    }
    { GRAPH g:courses
        { ?course   ab:courseTitle   ?courseTitle . }
    }
}
```

Обратите внимание, как этот запрос извлекает тройки путем выявления именованных графов, где они хранятся.

Результаты этого запроса показывают, что у курса моделирования данных, выбранного Синди Маршалл название обновлено:

first	last	courseTitle
“Cindy”	“Marshall”	“Modeling Data with RDFS and OWL”
“Cindy”	“Marshall”	“Using SPARQL with non-RDF Data”
“Richard”	“Mutt”	“Using SPARQL with non-RDF Data”
“Richard”	“Mutt”	“Updating Data with SPARQL”
“Craig”	“Ellis”	“Using SPARQL with non-RDF Data”

Разработчики реляционных БД могут сравнивать именованные графы этого набора данных с таблицами реляционной БД, потому что те и другие хранят определенный набор данных и запрос задает перекрестные ссылки между ними, чтобы перечислить требуемые данные. Некоторые параллели здесь есть, но помните, что именованные графы являются гораздо более гибкими, чем таблицы. Хранение в них новых видов данных не требует каких-либо изменений схемы. И, так как на графы ссылаются с помощью такого же рода идентификаторов как и на сами данные (то есть, с помощью URI), именованные графы могут сами содержать метаданные, возложенные на них или быть метаданными других ресурсов. Использование URI, также позволяет нам перекрестные ссылки этих данных с данными из совершенно разных наборов данных.

6.6. Резюме

В этой главе мы узнали:

- Как запустить сервер Fuseki SPARQL после загрузки ZIP файл дистрибутива
- Как операция INSERT (с и без ключевого слова DATA) и ключевое слово LOAD может добавить локальные и удаленные тройки к графу по умолчанию вашего набора данных
- Как операции DELETE и DELETE DATA могут удалять тройки из графа по умолчанию
- Как совместить операции INSERT и DELETE, чтобы заменить существующие тройки в графе по умолчанию
- Как создать именованные графы, и как в них вставлять и удалять тройки, наряду с несколькими способами указывать имена обновляемых графов

Глава 7. Эффективность и отладка запроса

Запрос служит для получения определенной порции (набора) информации. Иногда, существуют альтернативные способы получения одного и того же набора информации, и некоторые способы являются более эффективными, чем другие. Когда вы оцениваете объем работы, который каждая часть вашего запроса требует от процессора SPARQL для получения результата (или, на жаргоне информатики как "дорога" каждая часть в циклах процессора), то эта оценка поможет вам создавать запросы, которые выполняются быстрее. Методы и инструменты отладки также могут помочь оптимизировать определенный запрос, а также исправить запрос, который не делает то, что от него требуется.

В этой главе, мы узнаем:

"Эффективность внутри оператора WHERE"

Оператор WHERE является сердцем любого запрос, и порядок его компонентов и выбор используемых функций могут ускорить или замедлить ход событий.

"Эффективность вне оператора WHERE"

После того, как WHERE возвращает значения из набора данных, есть несколько вещей, которые запрос может сделать с этими значеньями, и некоторые из них менее эффективны, чем другие.

"Отладка"

Отладка запросов SPARQL начинается с классических методов, которые используются с любым развивающимся языком, а также могут быть полезны специализированны подходы.

7.1. Эффективность внутри оператора WHERE

Перед тем, как процессор SPARQL сможет перечислять, сортировать, удалять или вставлять данные, описанные в запросе или запрос обновления, он как правило, должен сначала найти данные, в которых вы заинтересованы, путем сопоставления шаблона (образца) троек в операторе WHERE запроса с тройками в наборе данных, который вы запрашиваете. В то время как порядок троек в шаблоне не должен влиять на конечные результаты запроса, он может иметь большое влияние на скорость выполнения запроса. Как некоторое обоснование того, почему порядок расположения троек влияет на скорость, посмотрим на упрощенной обзор того, как процессор SPARQL продвигается по тройкам шаблона в WHERE:

1. Процессор ищет тройки в наборе данных, которые соответствуют первой тройке шаблона. Если не находит, то отказывается от поиска, если только шаблон не находится внутри оператора OPTIONAL - в этом случае процессор продолжает пытаться найти соответствия для следующих шаблонов.
2. Если процессор нашел, по крайней мере одну тройку, которой соответствовал шаблон на шаге 1, и в шаблоне были переменные, то он сохраняет соответствующие значения (или, если использовать технический термин, связывает их с этими переменными). Оператор BIND также присваивать значения переменным.
3. Если следующий шаблон в операторе WHERE использует любые переменные, которые были связаны ранее, процессор запросов заменяет эти значения в остальных частяхшабла, а затем отправляется на поиски троек, которые соответствуют этому шаблону.
4. Если не удается ничего найти и шаблон не в операторе OPTIONAL, процессор останавливает поиск. Если он находит что-либо, то процесс возобновляется на шаге 2 со следующей тройки этого шаблона.

Если запрос ничего не нашел из-за отсутствия соответствий, он повторяет второй и третий шаги, пока не получит соответствия полному графу (шаблону троек) оператора WHERE троек набора данных. Затем он применяет директивы FILTER, которые могут использовать как простые, так и сложные выражения, чтобы указать тройки для отбора из результирующего набора. И, наконец, обрабатывает остальные значения переменных в остальной части запроса за пределами оператора WHERE.

Имея в виду эти шаги, помните, это общий принцип: быстрым является запрос, который быстро определяет отсутствие соответствий шаблону графа, и если есть соответствия, быстро возвращает значения переменных для связывания. Давайте посмотрим на некоторые способы, позволяющие стимулировать этот процесс.

7.1.1. Уменьшение пространства поиска

Представьте себе, что вы потеряли очки в двухэтажном доме, который имеет 10 комнат на каждом этаже. Прежде чем отправиться, чтобы осмотреть эти 20 комнат, вы встретили друга, который спускается по лестнице и говорит вам, что он искал свои ключи в каждой комнате на верхнем этаже и точно не видел ваши очки там. Он сократил вам пространство поиска вдвое.

Если вы осуществляете поиск имени поставщика в реляционной БД или поиск по названию песни в Google, которую вы слышали по радио, то вам помогут многие из технических способов, которые были разработаны для сокращения пространства поиска, причем настолько больше, насколько это возможно. Та же идея применима к реализации поиска процессором SPARQL в коллекции троек.

Мы раскроем эту тему в ходе нашего обсуждения эффективных запросов SPARQL. Мы начнем с простого приложения этой идеи: воспользоваться типм данных. RDF не требует объявлять классы, а затем определить, какие ресурсы являются членами каких классов, но тройки, которые делают это могут упрощать поиск и ускорять его.

Например, давайте предположим, что вы ищете информацию о конкретной книге в наборе данных, который имеет информацию о книгах, играх, фильмах, картинах и музыки и использует классы DBpedia такие как dbpedia-owl:Book и dbpedia-owl:Film для иденитификации каждого ресурса. Если тройка {?resource rdf:type dbpedia-owl:Book} появляется в начале шаблона вашего запрос и следующие тройки используют ту же самую переменную ?resource, то такой шаблон сузит пространство поиска для более поздних троек и заложить основу для более быстрой работы поисковой системы.

Когда ваши данные идентифицирует класс членства для описываемых ресурсов, это может помочь ускорить запросы, которые люди делают к вашим данным.

7.1.2. OPTIONAL очень необязательны

Ключевое слово OPTIONAL удобно для извлечения значений, которые могут присутствовать, а могут отсутствовать в наборе данных и могут извлекаться не прерывая выполнение вашего запроса, но факт заключается в том, что OPTIONAL загружает процессор запросов работой, не давая возможности сокращения пространства поиска. Академические работы по оптимизации запросов SPARQL согласны с тем, что OPTIONAL является виновными в замедлении запросов и добавляют сложность в работе процессора SPARQL, обязывая его находить и возвращать соответствующие релевантные данные. Вложенные операторы OPTIONAL еще больше усугубляют проблему.

Тройки шаблона вне оператора OPTIONAL позволяют процессору SPARQL решить, стоит ли продолжать дальнейший поиск и поэтому перемещение дополнительных условий после них может смягчить последствия OPTIONAL.

Обработчики запросов, как правило, оптимизированы для вас, так что перемещение частей шаблона вперед или назад может не иметь никакого влияния на производительность запросов. Различные процессоры запросов имеют различные подходы к этому. Спросите разработчиков процессора SPARQL об их стратегии оптимизации. Некоторые считают, что их подход имеет преимущество над другими и, следовательно, возможно, не захотят обсуждать эту тему подробно, но для процессоров SPARQL с открытым исходным кодом, это является классический темой обсуждения.

Лучшей оптимизацией является - просто избежать использование OPTIONAL, когда это возможно. В ASK запросе, о котором мы впервые узнали в главе 4, оператор OPTIONAL не имеет никакого влияния на ответ (ответ только yes или no), поэтому он будет ничего не делать, но замедлит выполнение запроса. (OPTIONAL с FILTER или MINUS может повлиять на результат ASK запроса, поэтому этот совет применим только к простым использованиям OPTIONAL).

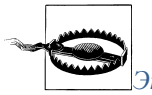
7.1.3. Порядок троек имеет значение

Операторы OPTIONAL являются не единственными составляющими шаблона троек, порядок которых может повлиять на время выполнения запроса, даже порядок простых троек может это сделать. Вы знаете, что тройка с тремя несвязанными переменными будет соответствовать всем тройкам в хранилище данных, а тройка без каких-либо несвязанных переменных будет соответствовать только одной тройке в хранилище. Чем меньше троек в шаблоне, которые соответствуют тройкам набора данных, тем больше он сужает пространство поиска, и тем быстрее процессор запросов может закончить свою работу.

Положение переменной в тройке (то есть, является ли она субъектом, предикатом или объектом) также может служить информацией об относительном числе троек хранилища, которые могут ей соответствовать. Знание этого дает вам больше возможностей для выявления троек, которые быстро уменьшают пространство поиска и заслуживают того, чтобы быть в начале запроса.

Документация для C#dotNetRDF библиотеки с открытым исходным кодом предоставляет некоторые подробности о том, как оптимизатор запросов SPARQL перестраивает тройки в шаблоне, основываясь на том, где тройка имеет переменные. dotNetRDF пытается определить тройки шаблона с более высокой селективностью ранее, чтобы уменьшить пространство поиска. Это делается путем ранжирования троек шаблона в порядке, указанном в следующем списке, который описывает, какие части каждой тройки имеют фиксированные значения, вместо того чтобы быть переменной:

1. Subject-Predicate-Object
2. Subject-Predicate
3. Subject-Object
4. Predicate-Object
5. Subject
6. Predicate
7. Object



Этот список предназначен для dotNetRDF версии 1.0.0 и может измениться. Помните, что это дает общие эвристики на основе типовых шаблонов и не обязательно оптимально для любой комбинаци набора данных и запроса обращенного к нему. Поэтому процессоры в запросе часто применять дополнительные методы оптимизации, такие как анализ статистики данных. Тем не менее, список

предоставляет отличный пример, чтобы думать о потенциальном влиянии различных шаблонов троек на время выполнения запроса.

Тройка без переменных является наиболее селективной, потому, что ей будет соответствовать только один триплет. Наименее селективными (кроме, конечно, тройки с тремя переменными) является те, которые имеют переменные на месте субъекта и предиката и определенным значением в позиции объекта. Рейтинг тройки с известным субъектом (позиция 5 в списке dotNetRDF) выше тройки с известным предикатом (позиция 6), поскольку тройки с известным субъектом и переменной в положении предикатов обычно сужают пространство поиска быстрее, чем наоборот.

Обсуждение приведенное выше, не затронуло потенциальную роль именованных графов, но те же самые принципы будут применимы: если процессор запросов знает, что он пытается отыскать тройки в конкретных именованных графах, то он имеет более узкую область поиска и может сделать свою работу быстрее. С другой стороны, если у вас используется ключевое слово GRAPH для переменной вместо имени графа, то вы задаете подсистеме запросов больше работы.

7.1.4. Фильтры: где и что

Наряду с перемещением тройной модели вокруг в шаблон графа, перемещая фильтр себе ранее, также могут уменьшить пространство поиска для последующих троек, пока все переменные, которые его ссылки были связаны перед фильтром сообщении.

Иногда, бережное использование тройных моделей может позволить вам опустить фильтр заявление и сделать запрос быстрее. Например, следующий запрос работает, но может использовать некоторое улучшение. При выполнении против SPARQL конечной точки DBpedia, он успешно перечисляет все фильмы, которые снимались как Аль Пачино и Роберт Де Ниро: