Алгоритмы интеграции СУБД PostgreSQL с семантическим веб

Левшин Дмитрий Владимирович, Марков Александр Сергеевич
Московский Государственный Университет им. М.В. Ломоносова
E-mail: levshin@nicevt.ru, markov@nicevt.ru

Аннотация

В данной статье рассматривается интеграция СУБД PostgreSQL с семантическим веб. Для выполнения общей задачи проводится анализ известных систем для работы с семантическим веб, использующих базы данных, и возможностей данной СУБД для решения задачи, предлагаются алгоритмы интеграции и осуществляется реализация одного из предложенных алгоритмов.

1 Введение

Данная статья посвящена интеграции PostgreSQL [1] - разработанной Калифорнийским университетом в Беркли открытой объектно-реляционной системы управления базами данных (СУБД) - с семантическим веб.

Автором идеи семантического веб [2] считается Тим Бернерс-Ли. История концепции уходит корнями в середину 90-х годов XX века, первые детализированные публикации относятся к 1998 году. С 1999 года проект семантической паутины развивается под эгидой Консорциума Всемирной паутины (W3C). Семантический веб становится все более популярным: концепция активно пропагандируется и внедряется многими проектами с открытым исходным кодом, внедряется крупными компаниями и корпорациями.

Семантический веб определяется как расширение Всемирной паутины (World Wide Web), такое что в нем информация снабжена точно определённым смыслом, позволяющим человеку и машине успешно взаимодействовать. Внедрение семантического веб предоставит широкие возможности для машинной обработки информации. В частности, семантический веб позволит поисковым системам получать ответы на более сложные запросы, увеличит эффективность выполняемого поиска.

Основные форматы семантического веб (RDF, RDFS, OWL и SWRL) были разработаны людьми с академическим образованием и считаются трудными для понимания рядовыми пользователями Интернета. Поэтому актуальной задачей является разработка программ, упрощающих пользователю использование возможностей семантического веб. В то же время в СУБД PostgreSQL, имеющей большое

распространение, нет возможностей для работы с данными семантического веб, что и послужило основной причиной исследований и реализаций в данной работе.

Хотя в самой СУБД нет средств для работы с данными в форматах семантического веб, PostgreSQL является удобным полигоном для проведения исследований, так как является открытой системой и позволяет достаточно легко добавлять новые возможности. При разработке алгоритмов интеграции принималось решение, что SQL и стандарты не модифицируются, требуется внедрять данные семантического веб с использованием тех возможностей, которые уже имеются в PostgreSQL.

Для выполнения общей задачи проводится анализ существующих решений и предложений для работы с семантическим веб, определяются те возможности PostgreSQL, которые могут использоваться для решения поставленной задачи. Проведенный анализ позволил разработать алгоритмы и методы интеграции PostgreSQL с семантическим веб. На основе разработанных методов и алгоритмов была выполнена реализация интеграции семантического веб с PostgreSQL, которая позволит пользователям данной СУБД работать с данными в форматах семантического веб.

1.1 Форматы семантического веб

Приведем кратко информацию об основных форматах семантического веб. Для того чтобы Семантическая Сеть могла функционировать, нужно было определить язык, при помощи которого пользователи могли бы выражать информацию и правила вывода. В феврале 2004 г. в качестве стандарта W3C для создания понятного компьютеру описания ресурсов в семантической паутине был утвержден формат RDF [3] (Resource Description Framework, система описания ресурсов).

При определении языка задания метаданных в Семантической сети требовалось решить две следующие проблемы: он должен был позволять определять выражения произвольной сложности, но в то же время эти выражения должны иметь форму достаточно простую для понимания машиной. Поэтому в RDF все утверждения описывается в виде триплетов (троек) вида «предмет (субъект) – свойство (глагол) – значение свойства (объект данного утверждения)». Кроме того, различные приложения должны однозначно понимать эти выражения, поэтому для обозначения ресурсов решено было использовать URI (Uniform Resource Identifier). В каждом триплете субъект и предикат являются ресурсами, а объект может быть или ресурсом, или некоторым литералом в формате UNICODE.

RDF является основой Semantic Web, представляя только базовые возможности. Более богатые возможности доступны при использовании расширений RDF. Расширение RDF - RDF Schema (RDFS) [4] - предоставляет возможности определения специфичных для

приложений классов и свойств, иерархии классов, указания того, какой тип должны иметь субъект и объект для данного свойства. Эта информация может восприниматься как ограничения на классы и свойства или же средство для вывода новых утверждений.

Язык веб онтологий OWL (Web Ontology Language) [5] может использоваться, чтобы явно представлять значения терминов и отношения между этими терминами в словарях. SWRL (Semantic Web Rule Language) [6] является расширением OWL, добавляя возможность определения Хорно-подобных правил. Предлагаемые правила имеют вид импликации между предпосылкой (телом) и следствием (заголовком), состоящими из нуля или более унарных и бинарных атомов. Например:

parentOf(?x,?y) & Man(?x) → fatherOf(?x,?y)

parentOf(?x,?y) & brotherOf(?x,?z) → uncleOf(?z,?x)

2 Обзор существующих решений

Среди существующих на сегодня решений для работы с семантическим веб можно выделить Protégé [7] – разработанное в отделении Медицинской Информатики Школы Медицины Стэнфордского Университета средство с открытыми кодами, его коммерческий аналог TopBraid Composer [8], а также решения, предложенные в Oracle.

2.1 Protégé и TopBraid Composer

Разработка Protégé исторически определялась направленностью на биомедицинские приложения, но при этом система является независимой от области применения и используется многими приложениями из различных областей. TopBraid Composer определяется как профессиональная среда разработки для RDF, RDFS, OWL, SWRL и языка запросов SPARQL [9] в соответствии со стандартами W3C и предоставляет полный набор возможностей для покрытия всего жизненного цикла разработки семантических приложений.

Оба средства, Protégé и TopBraid Composer, обладают схожей функциональностью и пользовательскими интерфейсами. Они позволяют создавать и редактировать файлы в поддерживаемых форматах RDF, RDFS, OWL, правила SWRL и SPARQL запросы с помощью удобного графического интерфейса. Кроме того, рассматриваемые средства могут использоваться как среда времени исполнения для выполнения правил, запросов, механизмов рассуждений. С помощью доступных механизмов рассуждений они позволяют выполнять классификацию и проверку логической целостности на основе OWL. Для выполнения правил и вывода дополнительных отношений между ресурсами в TopBraid Composer используется внутренний движок правил Jena. Предоставляется возможность выполнения SPARQL запросов только над явно заданными данными, или с

учетом имеющихся правил, выполняя логический вывод во время исполнения запроса. Допускается выполнение SPARQL запросов как над RDF хранилищами, так и над интегрированными реляционными базами данных. Рассматриваемые средства позволяют расширять свою функциональность с помощью специально определенных плагинов.

Главным отличием этих двух средств заключается в том, что Protégé является открытой системой с академическими истоками в различных исследовательских проектах, а TopBraid Composer – коммерческая платформа. Поэтому говорится, что TopBraid Composer стабильнее (содержит меньше ошибок), чем Protégé. Кроме того, многие полезные возможности для Protégé предоставляются третьей стороной, а в TopBraid Composer они встроены сразу. Например, в TopBraid Composer встроены свободно распространяемый механизм рассуждений для OWL DL Pellet и движок правил, а в Protégé требуется дополнительно устанавливать Racer. TopBraid Composer реализован как плагин к Eclipse и использует некоторые достоинства данной платформы для предоставления таких возможностей, как графическая визуализация и UML импорт. Кроме того, история исследований по Protégé насчитывает более 10 лет, поэтому API данного средства не оптимизировано для RDFS и OWL. Разработка TopBraid Composer,напротив, была оптимизирована с учетом стандартов семантического веб.

Открытость Protégé позволяет изучить ее схему на основе [10]. Protégé разделено на 2 части: модельную и представления. Модель Protégé – механизм внутреннего представления онтологий и баз знаний. Компонент представления Protégé предоставляет пользовательский интерфейс для отображения и обработки нижележащего уровня. Рассмотрим модельную часть.

Модель Protégé основана на простой, гибкой метамодели, сопоставимой с объектно-ориентированными и фреймовыми системами. Для запросов и обработки моделей Protégé предоставляет открытый Java API. Метамодель Protégé сама является онтологией, что позволяет достаточно просто расширять ее, например, для обработки UML и OWL. Поддержка OWL реализуется с помощью специального плагина над Protégé, который поддерживает RDFS, OWL Lite, OWL DL и часть OWL Full.

Как показано на рисунке 1, OWL плагин расширяет модель Protégé и предоставляет расширенный Java API для доступа и обработки OWL онтологий. В то время как основной API предоставляет доступ к классам, свойствам и элементам классов онтологий, OWL плагин расширяет этот API Java классами, специально созданными для различных типов классов OWL. Этот API инкапсулирует внутренние преобразования и таким образом защищает пользователя от подверженного ошибкам низкоуровневого доступа. Данный

API можно расширить, определив специальные классы для расширений OWL, например, SWRL.

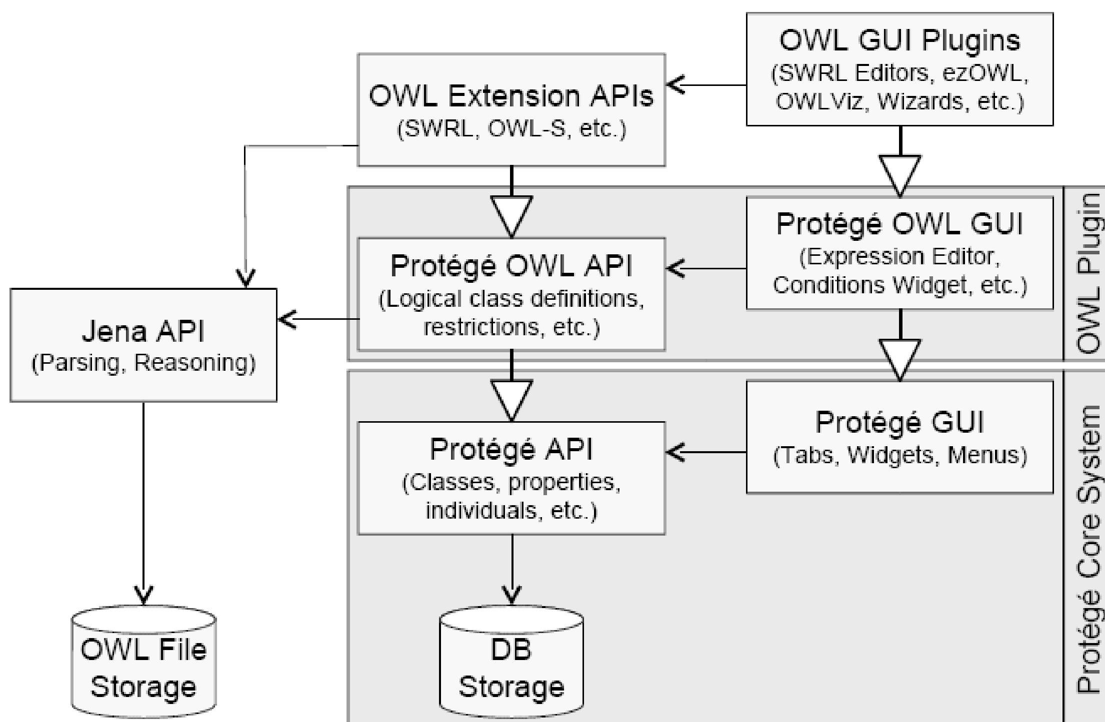

Рисунок 1

OWL плагин предоставляет полное соответствие между его расширенным API и стандартной библиотекой разбора OWL Jena. После того как онтология была загружена в Jena модель, OWL плагин генерирует соответствующие объекты Protégé. Затем система все время хранит модель Jena в памяти и синхронизует ее со всеми изменениями, произведенными пользователями. Таким образом, когда пользователь создает класс Protégé, в то же время создается класс Jena с таким же именем. Наличие вторичного представления онтологии в терминах Jena объектов означает, что пользователь постоянно может вызвать произвольные основанные на Jena услуги, такие как интерфейсы классификаторов, языков запросов или средства визуализации. Соответствие Jena также облегчает встраивание существующих и будущих сервисов семантического веб в OWL плагин.

Оба средства, Protégé и TopBraid Composer, используют Jena [11] – открытую java среду для создания приложений семантического веб. Среда Jena предоставляет программное окружение для RDF, RDFS, OWL и SPARQL, а также включает основанный на правилах движок логического вывода. Она включает RDF и OWL API, чтение и запись RDF в форматах RDF/XML, N3 и N-Triples, хранение в памяти и в базах данных, а также движок запросов SPARQL.

Рассмотрим схему хранения в реляционных базах данных, используемую в Jena, на основе [12]. В Jena1 схема состоит из таблицы утверждений, таблицы ресурсов и таблицы литералов, как показано на рисунке 2. Таблица утверждений содержит заданные триплеты и утверждения реификации, ссылаясь на таблицы ресурсов и литералов для субъектов, предикатов и объектов. Для того чтобы отличать литеральные значения объектов от URI ресурсов используются два столбца. Таблица литералов хранит все значения литералов, а таблица ресурсов – URI ресурсов в RDF графе. Такая схема очень эффективна с точки зрения использования пространства, так как различные вхождения одинаковых URI ресурсов и значений литералов сохраняются один раз. Однако операции поиска в Jena требовали множественные операции соединения над таблицами утверждений, ресурсов и литералов.

Таблица утверждений

Subject	Predicate	ObjectURI	ObjectLiteral
201	202	null	101
201	203	204	null
201	205	101	null

Таблица литералов

Id	Value
101	Jena2
101	The description - a very long literal that might be stored as a blob.

Таблица ресурсов

Id	URI
201	mylib:doc
202	dc:title
203	dc:creator
204	hp:JenaTeam
205	dc:description

Рисунок 2

Поэтому в Jena2 (втором поколении Jena) для уменьшения времени выполнения операций поиска была предложена ненормализованная схема, в которой URI ресурсов и значения литералов, длина которых не превосходит некоторого предела, хранятся прямо в таблице утверждений. Чтобы отличать ссылки базы данных от литералов и URI, значения столбцов кодируются с префиксами, которые обозначают тип значения. Отдельные таблицы литералов и ресурсов используются только для длинных литералов и URI. Пример ненормализованной схемы показан на рисунке 3.

Ненормализованная схема позволяет уменьшить время выполнения операций поиска в Jena за счет сокращения числа операций соединения, но увеличивает используемое дисковое пространство. Чтобы сократить пространство для хранения RDF данных предложено несколько подходов. Во-первых, часто используемые префиксы сохраняются в таблице утверждений не напрямую, а как ссылки отдельную таблицу префиксов. При этом предполагается, что таблица префиксов, будет достаточно малой и

сможет постоянно хранится в памяти. Поэтому использование данной таблицы не повлечет дополнительных обращений к дискам. Во-вторых, URI и литералы слишком большой длины сохраняются в отдельные таблицы. При этом предел длины может настраиваться, и ее изменение позволяет определять приложению, что является приоритетом – время или пространство.

Таблица утверждений

Subject	Predicate	Object
mylib:doc1	dc:title	Jena2
mylib:doc1	dc:creator	HP Labs - Bristol
mylib:doc1	dc:creator	Hewlett-Packard
mylib:doc1	dc:description	101
201	dc:title	Jena2 Persistence
201	dc:publisher	com.hp/HPLaboratories

Таблица литералов

Id	Value
101	The description - a very long literal that might be stored as a blob.

Таблица ресурсов

Id	URI
201	hp:aResource-WithAnExtreme-lyLongURI

Рисунок 3

В Jena1 все графы хранятся в одной таблице утверждений. В Jena2 различные графы можно помещать в несколько таблиц в зависимости от того, какие графы часто используются вместе, а какие отдельно.

Кроме того, в Jena2 для хранения свойств, которые часто встречаются вместе, предлагается использовать специально определенные таблицы свойств. Таблица свойств хранит все вхождения определенного свойства в графе, т.е. данное свойство не встречается ни в одной другой таблице, используемой для графа. Свойства, которые имеют максимальную кардинальность, равную единице, можно хранить вместе в одной таблице. Одна строка в такой таблице будет содержать значения данных свойств для одного субъекта.

2.2 Предложения Oracle

Описанные в [13] решения могут служить образцом комплексного подхода к решению проблемы. Отмечается, что большинство подходов к разработке схемы эффективных и масштабируемых запросов к данным RDF, основанные на определении новых языков запросов к RDF данным, связаны со следующими недостатками: 1) они вызывают трудности при интеграции с SQL запросами, используемыми в приложениях баз данных, и 2) становятся неэффективными при преобразовании данных SQL к форматам соответствующих новых языков. Поэтому предлагается подход, основанный на SQL, который позволяет устранить эти проблемы.

В сам язык SQL не вносится изменений, для запросов к RDF данным вводится табличная функция RDF_MATCH со следующей функциональностью:

- возможность поиска по произвольно заданному образцу среди RDF данных, включающих процессы логического вывода, основанных на правилах RDFS
- возможность включать наборы определенных пользователем правил как дополнительные источники ресурсов.

Функция RDF_MATCH имеет следующий формат:

```
  RDF_MATCH (
    Pattern       VARCHAR,
    Models        RDFModels,
    RuleBases     RDFRules,
    Aliases       RDFAliases,
  )
RETURNS AnyDataSet;
```

Первым аргументом в качестве своего значения задается шаблон графа для поиска. В нем используется синтаксис в стиле SPARQL, причем переменные имеют префикс ‘?’. Второй аргумент специфицирует одну или более моделей, применяемых при поиске. Под моделью понимается набор триплетов, относящихся к одной области. Третий аргумент специфицирует базу правил (если она имеется). Значение этого аргумента NULL указывает на то, что база правил отсутствует. Четвертый аргумент специфицирует определенные пользователем псевдонимы пространства имен (если они имеются). Значение NULL указывает на то, что пользователь не определил никаких псевдонимов, однако умалчиваемые псевдонимы, такие как rdf:, остаются доступными в любом случае. Результатом функции RDF_MATCH является таблица, набор столбцов которой определяется шаблоном графа. Для каждой переменной в шаблоне графа присутствует столбец с таким же именем (без знака ‘?’ в начале), что достигается использованием типа AnyDataSet. Для переменных, значения которых могут быть литералами, в таблице дополнительно присутствуют столбцы, указывающие их тип. Подстановками значений соответствующих столбцов в шаблон графа будет получен подграф, полученный из указанной модели и правил.

При обработке RDF данных с использованием SQL в одном запросе наряду с RDF данными могут использоваться и остальные таблицы, указывая их в списке выборки вместе с функцией RDF_MATCH. Для дальнейшей обработки результатов могут применяться стандартные конструкции SQL, включая WHERE, ORDER BY конструкции и другие. Обращение к этой табличной функции в дальнейшей обработке перезаписывается как SQL запрос, тем самым устраняется проблема процедурного переопределения с табличными функциями времени выполнения. Это также не препятствует оптимизации

перезаписанного запроса в комбинации с остальными запросами. Результирующий запрос выполняется достаточно эффективно благодаря использованию B-деревьев и материализации специализированных представлений на отношения субъект-свойство.

Для хранения триплетов RDF данных используются две таблицы: IdTriples (триплеты в формате идентификаторов) и UriMap (пары из целочисленных идентификаторов и значений URI или литералов). Схема похожа на используемую в Jena1. При вставке триплета в базу данных для каждого элемента триплета в таблице UriMap выполняется поиск соответствующих идентификаторов. Если для какого-то элемента не был найден идентификатор, для него генерируется новый уникальный идентификатор и полученное соотношение вставляется в таблицу UriMap. Полученная тройка идентификаторов вставляется в IdTriples. Подобная нормализация используется, потому что URI (или литералы) обычно повторяются неоднократно. Кроме того, это способствует эффективной обработке запросов благодаря компактности хранения. Для того чтобы поддерживать соответствие между множественными представлениями одного и того же значения (например, целое 123 и плавающее 12.3E+1), каждый литерал приводится к канонической форме. Каноническим литералом для каждого значения становится первый введенный литерал с этим значением. Для соотнесения остальных литералов с равными значениями к канонической форме используется специальный флаг в UriMap, отмечающий канонический литерал. Кроме того, для предопределенных типов все значения разбиваются на семьи, ассоциированные со своими пространствами значений (например, семейство чисел). Для преобразования лексического представления UriMap в каноническую форму используется специально определенная функция и индекс.

При выполнении RDF_MATCH полученный шаблон преобразуется в запрос с самосоединениями к данным таблицам. Для эффективного выполнения такого запроса используются B-деревья и материализованные представления над таблицами IdTriples и UriMap.

Предложенный подход поддерживает RDFS и правила, определенные пользователем. По наличию рекурсивности заданные пользователем правила можно классифицировать следующим образом:

- Правила без рекурсии: атомы из тела не могут быть выводимы ни из данного правила, ни из любого другого правила, содержащего атомы из заголовка данного правила.
- Правила с простой рекурсией: определение транзитивности и симметричности свойств, определенных пользователем.
- Правила с произвольной рекурсией, отличной от предыдущих двух категорий.

Для обработки правил функция RDF_MATCH замещает ссылки к таблице IdTriples на генерируемый SQL текст с подзапросами или табличными функциями, которые производят соответствующие явные и логически выводимые триплеты.

Для реализации правил без рекурсии вместо IdTriples в списке выборки подставляется подзапрос, сгенерированный с учетом подходящих правил. Например:

```
SELECT ...
FROM (
-- (?x ChairpersonOf ?c) => (?x ReviewerOf ?c)
 SELECT t1.SubjectID, 14 PropertyID, t1.ObjectID
 FROM IdTriples t1 WHERE t1.PropertyID = 56
 UNION
-- явные ReviewerOf триплеты
 SELECT t1.SubjectID, t1.PropertyID, t1.ObjectID
 FROM IdTriples t1 WHERE t1.PropertyID = 14
) t1;
```

Кроме того, для повышения скорости выполнения предлагается индексация баз правил, предварительное вычисление выводимых триплетов и сохранение в отдельных таблицах. Симметрия тоже обрабатывается как простой запрос. Однако для обработки транзитивности одним запросом используются иерархические запросы (например, с использованием предложений START WITH и CONNECT BY NOCYCLE в Oracle). В частности, для поддержания правил логического вывода RDFS требуется вычислять транзитивное замыкание двух транзитивных свойств RDFS: rdfs:subClassOf (правило rdfs11) и rdfs:subPropertyOf (правило rdfs5).

Стоит особенно отметить, что в данной системе не поддерживаются рекурсивные правила. Поэтому, в частности, пользователям запрещено расширять словарь RDFS, чтобы удостовериться в том, что результаты логического вывода могут быть использованы в едином SQL запросе.

2.3 Выводы

Таким образом, обзор показал, что основным подходом при работе с семантическим веб с помощью баз данных является применение специальных API. Однако отмечается, что такой подход обладает рядом недостатков, которых позволяют избегать решения, основанные на применении SQL. В частности, схема, предложенная Oracle, может быть реализована в любой реляционной СУБД, где поддерживаются табличные функции, материализованные представления на соединения и B-деревья. Однако данное решение позволяет работать только с RDF и RDFS, а также простыми пользовательскими правилами.

Основной используемой схемой является нормализованная, при которой все триплеты хранятся в отдельной таблице в виде троек целых чисел, ссылающихся на

другие таблицы, содержащие значения URI и литералов. Данное решение является эффективным по использованию дискового пространства, но для уменьшения времени выполнения запросов предлагаются изменения схемы, такие как использование таблиц свойств, хранение в нормализованном виде только тех значений литералов и URI, длина которых превосходит определенный предел. Использование данной схемы и ее изменений для хранения данных и табличных функций для выполнения запросов может быть полезным при решении поставленной задачи.

3 Исследование специфики возможностей СУБД PostgreSQL для выполнения интеграции

Стоит подробнее остановиться на некоторых возможностях PostgreSQL, которые могут быть полезны при выполнении интеграции. Для того чтобы СУБД смогла выполнять логический вывод утверждений на основе загруженных правил, можно использовать два механизма – триггеры и система правил. Для выполнения запросов к RDF данным может быть полезным использование табличных функций, эффективный доступ к данным может быть обеспечен с помощью индексных структур. Табличные функции и функции триггера могут быть написаны на C или одном из процедурных языков.

3.1 Триггеры

Триггер определяет, что база данных автоматически выполняет определенную функцию, когда выполняется операция некоторого типа. Триггер может быть определен для выполнения до (в этом случае сначала вызывается функция триггера, а затем выполняется вызвавшее триггер событие) или после (наоборот, сначала выполняется вызывающее триггер событие, а затем выполняется функция триггера) INSERT, UPDATE или DELETE операции, для каждого кортежа (в этом случае функция триггера будет вызываться для каждой вставляемой, обновляемой или удаляемой строки) или для всего SQL выражения (функция триггера вызывается один раз). Когда выполняется событие триггера, функция триггера выполняется в соответствующее время для обработки этого события. Функция триггера должна быть создана до создания триггера. После этого триггер создается с помощью команды CREATE TRIGGER. Одна и та же функция может использоваться для нескольких функций. Для написания триггера могут использоваться процедурные языки или C.

3.2 Система правил и представления

Использование правил некоторым образом похоже на применение триггеров. В обоих случаях определяется некоторое событие (INSERT, UPDATE или DELETE, но

правила могут вызываться и на SELECT) и действие которое выполняется при выполнении этого события. Но между ними есть множество различий. Во-первых, у правил можно определять помимо событий еще условия выполнения правила при помощи конструкции WHERE (кроме правил на SELECT). Во-вторых, при определении триггера выполняемым действием является вызов определенной функции, а у правил - SQL команды или NOTHING (не делать ничего). Но главным отличием между триггерами и правилами заключается в способе их выполнения.

Вызов триггеров происходит уже на этапе выполнения запроса. Система правил изменяет сам запрос, после чего выполняется уже измененный запрос. Система правил расположена между парсером и планировщиком, она получает на вход дерево запроса (внутреннее представление SQL выражения, которое строится для выполнения оптимизации его выполнения), построенное парсером, и определенные пользователем правила в виде деревьев запросов с некоторыми дополнительными условиями и создает ноль или более деревьев запросов. Ноль деревьев запросов может быть получено, если вместо данного дерева запроса (если правило было определено как INSTEAD) подставляем правило, у которого действие определено как NOTHING. Более одного дерева запроса может получиться, когда к текущему дереву запросов (пусть оно имеет некоторое условие cond) применяется правило с некоторым условием rule_cond, при этом будет получено два дерева: одно будет иметь условие cond and not rule_cond и в него не будет подставлено правило, второе будет иметь условие cond and rule_cond и в него будет подставлено правило. При подстановке правил в исходное дерево запросов система правил смотрит, на какое событие они определены, для какой таблицы, при каких условиях и другую информацию в определении правила. Результат работы системы правил - деревья запросов, которые могли быть получены и парсером для некоторых SQL выражений. Такой способ выполнения может быть очень полезен, например, для представлений.

Представления - это виртуальные таблицы, реальных экземпляров этих таблиц не существуют. Для выполнения запросов или модификации представлений используется система правил. Для определения правил используется команда CREATE RULE, а для представлений - CREATE VIEW.

Стоит, однако, отметить, что на выполнение правил накладываются ограничения, связанные с рекуррентными вызовами. Запрещается определение правил, определенных на вставку, которые выполняют вставку измененных значений в ту же таблицу. Кроме того, если правило r1 имеет условием вставку в таблицу T1 и выполняемое действие – вставку в T2, а правило r2, наоборот, определено на T2 и выполняет вставку в T1, то при

выполнении вставки в любую из этих двух таблиц будет получена ошибка. Для того чтобы обойти эти ограничения, связанные с рекурсией, можно вместо одного из правил использовать триггеры. Поэтому для выполнения задачи логического вывода представляется полезным использование обоих механизмов – триггеров и системы правил.

3.3 Процедурные языки

Для создания пользовательских функций в PostgreSQL используется команда CREATE FUNCTION. С ее помощью можно создавать функции, написанные на SQL, C, а также на процедурных языках. При выполнении функции, написанной на процедурном языке, сервер базы данных не имеет встроенного знания о том, как ее интерпретировать, для этого он вызывает специальный обработчик, который знает все детали этого языка. Обработчик может сам выполнять всю работу по лексическому и синтаксическому анализу и выполнению функции или служить "клеем" между PostgreSQL и существующей реализацией языка программирования. Обработчик сам является функцией на языке C, скомпилированной в разделяемый объект и загруженной в базу данных, как любая другая функция на C в PostgreSQL.

В настоящее время в PostgreSQL представлены четыре процедурных языка: PL/pgSQL, PL/Tcl (для написания функций на Tcl), PL/Perl (позволяет писать функции на Perl, но с некоторыми ограничениями) и PL/Python (для написания функций на Python). Остальные процедурные языки могут быть определены пользователем.

PL/pgSQL - загружаемый процедурный язык для PostgreSQL, который:

- может использоваться для создания функций и триггеров
- добавляет к SQL контрольные структуры
- может выполнять сложные вычисления
- наследует все определенные пользователем типы, функции и операции
- прост в использовании

За исключением вычислительных функций для определенных пользователем типов и преобразований ввода/вывода, все, что может быть определено в C функциях, может быть сделано и с PL/pgSQL.

Для того чтобы использовать процедурные языки, пользователи должны сначала установить эти языки в свою базу данных. Для этого можно воспользоваться командой CREATE LANGUAGE или createlang из командной строки shell. Если требуется использовать какой-либо процедурный язык, кроме указанных выше, сначала нужно создать в базе данных функцию-обработчик для языка.

3.4 Методы индексирования. GiST (Generalized Search Tree)

Эффективный доступ к данным является одной из важнейших задач базы данных. Для больших баз данных, которые не помещаются в оперативную память, эффективность доступа к данным определяется, в основном, количеством обращений к диску, поэтому основной задачей СУБД является минимизация этих обращений. Обычно, это достигается использованием индекса, вспомогательной структуры данных, предназначенной для ускорения получения данных, удовлетворяющих определенным поисковым критериям. Индекс позволяет уменьшить количество дисковых операций необходимых для считывания данных с диска.

В PostgreSQL, кроме стандартных методов индексирования на основе B-деревьев, R-деревьев и хэш, реализован метод индексирования GiST [14] (Generalized Search Tree, Обобщенное поисковое дерево), предложенный профессором Беркли Джозефом Хеллерстейном. Преимущество GiST заключается в том, что с его помощью реализуется не только расширяемость типов (то есть возможность использования индексов для определенных пользователем типов данных), но и расширяемый набор запросов. Реализация расширяемого набора запросов является важной особенностью, так как позволяет поддерживать запросы естественные для этих типов. Достигается это благодаря тому, что при определении GiST структуры можно использовать произвольные предикаты, а не только операции сравнения, как у остальных индексных структур. С использованием GiST определение новых типов данных с поддержкой индексирования и естественных запросов становится достаточно простой задачей.

3.5 Табличные функции

Подход, предложенный Oracle, опирается на использовании табличных функций. При интеграции PostgreSQL с семантическим веб для выполнения запросов также возможно использовать табличные функции. Табличные функции – это функции, результатом выполнения которых является набор строк скалярного типа или составного типа данных (табличные строки). Они могут использоваться, как и таблицы, представления или подзапросы, в WHERE предложениях. Столбцы, возвращенные табличными функциями, могут быть включены в SELECT, JOIN или WHERE предложениях так же как и столбцы таблиц, представлений или подзапросов. В случаях, когда требуется, чтобы табличная функция возвращала множество строк, столбцы которых зависят от вызова функции, она может быть определена как возвращающая значения псевдотипа record. При использовании такой функции в запросе должна быть

явно указана структура возвращаемой строки, чтобы система знала, как выполнять парсинг и строить план запроса.

4 Предложение и обоснование методов и алгоритмов

4.1 Основная схема

Для выполнения разбора документов в форматах семантического веб предлагается разработать модуль, который будет подсоединяться к базе данных с помощью интерфейса доступа, поддерживаемого в PostgreSQL. При вызове модуля предполагается указывать входной файл и название базы данных, в которую будет загружаться извлеченная информация. Модуль будет производить разбор полученного файла, строить триплеты и загружать их в указанную базу данных. Для правил вывода OWL и SWRL модуль будет генерировать правила в базе данных. Семантику RDFS предлагается реализовывать с помощью правил базы данных, которые создаются при инициализации системы, основываясь на правилах rdfs1 – rdfs13 в [15].

Для хранения триплетов предлагается использовать таблицу pg_rdf_triples (subject_id, property_id, object_id, isinferenced). В данной таблице все триплеты хранятся в форме троек целых чисел, которые взаимно однозначно ставятся в соответствие каждому загруженному URI и литералу. Столбец isinferenced принимает значение 0, если соответствующий триплет был явно задан, и больше 0, если был логически выведен.

Для установления соответствия между ресурсами, литералами и целыми числами создается таблица pg_rdf_values (value_id, text_value, value_type, literal_type), где value_id – уникальный целочисленный идентификатор, генерируемый системой, text_value – текстовое представление значения ресурса или литерала, value_type указывает тип значения, literal_type – тип литерала (равен NULL, если кортеж соответствует не типизированному литералу). Такая нормализация позволяет сократить размеры таблицы pg_rdf_triples, так как для запоминания каждой тройки используется не символьные строки, которые могут иметь достаточно большую длину, а целочисленные идентификаторы, длина которых постоянна и практически всегда меньше, чем у соответствующих строк. Такой подход может давать выигрыш в размерах используемой памяти, так как URI многих ресурсов, определенные в словарях rdf:, rdfs:, owl:, swrl:, dc: и т.д., могут встречаться довольно часто. Все значения ресурсов и литералов в таблице pg_rdf_values хранятся в строковом представлении.

Однако для типизированных литералов может понадобиться преобразование из строкового представления к их типу. Например, если известно, что “+1” и “1” являются литералами целочисленного типа и требуется провести их сравнение, лексикографическое

сравнение даст неверный результат. Для получения верного результата сначала оба литерала нужно с использованием определенной функции преобразовать к общему виду, а затем сравнить. Поэтому для хранения информации о типах литералов используется таблица pg_rdf_type (typname, typinput, typoutput), где typname – имя типа, typinput, typoutput определяют функции для преобразования строкового представления в значения требуемого типа и обратно. Информация о типе ресурса хранится в таблице pg_rdf_valtypes (value_id, value_type, type_description).

Для выполнения запросов к RDF данным вводится табличная функция RDF_QUERY, параметром которой является текстовая строка, представляющая шаблон RDF графа. Переменные в шаблоне обозначаются знаком ‘?’, стоящим в начале. Кроме того, в начале текстовой строки можно указывать используемые префиксы для краткости задания запросов. Стоит отметить, что данный формат задания запросов к RDF данным соответствует спецификации SPARQL. Количество столбцов в результате выполнения функции соответствуют числу переменных в шаблоне RDF графа, а названия столбцов – именам переменных. Такой гибкости, а также возможности использования результатов запросов к RDF данным в более сложных запросах позволяет добиться использование для этих целей табличной функции. Запросы с использованием `RDF_QUERY` имеют следующий вид:

```
SELECT * FROM RDF_QUERY('(?a hasFather ?b)
     (?b married Ann)') AS res (a text, b text);
```

Для ответа на запросы пользователей к RDF данным с учетом имеющихся правил SWRL и онтологий необходима реализация логического вывода. Рассмотрим два алгоритма выполнения логического вывода триплетов, неявно заданных с помощью правил.

4.2 Статический алгоритм

Выполнение логического вывода можно производить на этапе загрузки триплетов в базу данных, для чего правила должны генерироваться таким образом, чтобы условием их срабатывания являлась вставка нового триплета в таблицу pg_rdf_triples. Например, если во входном файле имелось правило следующего вида P(?x,?y) → R(?x,?y), то модуль разбора должен сгенерировать правило:

```
CREATE RULE <rulename> AS ON INSERT TO pg_rdf_triples
     WHERE NEW.property_id = get_id(‘P’)
DO INSERT INTO pg_rdf_inferenced
     SELECT NEW.subject_id, get_id(‘R’), NEW.object_id;
```

Отметим, что результатом выполнения правила является вставка выведенных триплетов в специально определенную таблицу pg_rdf_inferenced (subject_id, property_id, object_id, mark). Данная таблица вводится, так как в PostgreSQL запрещается определять правила, условием которых является вставка в таблицу, а действием является также вставка в эту же таблицу. Кроме того, на таблице pg_rdf_inferenced возможно определить триггер, который разрешает вставку в таблицу только новых триплетов, то есть тех, которых нет ни в pg_rdf_triples, ни в pg_rdf_inferenced, что позволяет избежать бесконечного выполнения вывода одних и тех же триплетов.

После загрузки триплетов в базу данных вызывается специально определенная функция, которая выполняет следующие действия:

1. удаляет из таблицы все кортежи, значение столбца mark которых равно 1
2. изменяет значение столбца mark всех оставшихся кортежей на 1 (значение данного столбца по умолчанию при вставке равно 0)
3. если таких кортежей в таблице нет, то выполнение функции завершается
4. иначе функция вставляет все новые триплеты в pg_rdf_triples
5. выполняет все действия, начиная с пункта 1, так как вставка этих триплетов в pg_rdf_triples может снова вызвать срабатывание правил и вывод новых триплетов.

Стоит отметить, что использование столбца mark позволяет на каждой итерации рассматривать только те кортежи, которые были получены на предыдущей итерации, отбрасывая те, которые были получены раньше и уже вставлены в pg_rdf_triples.

Так как логический вывод выполняется на этапе загрузки, выполнение запросов является очень простой задачей: необходимо просмотреть хранимые данные и выбрать только те тройки, которые соответствуют шаблону запроса. Поэтому преимуществом данного метода является высокая скорость выполнения запросов, однако он обладает рядом существенных недостатков:

- загрузка онтологий, правил вывода возможна только до загрузки триплетов
- требуется постоянное хранение всех триплетов, полученных логическим выводом
- изменение или удаление триплетов становится трудной задачей, так как может привести к появлению в базе данных триплетов, выведенных из измененных триплетов (возможно, не напрямую), которые стали неверными.

Для того чтобы избежать указанных недостатков, вводится следующий алгоритм.

4.3 Динамический алгоритм

Логический вывод проводится на этапе выполнения запроса. Выполнение всех имеющихся правил в этом случае может означать достаточно длительное время выполнения запроса. Поэтому выполнение данного алгоритма состоит из двух этапов:

- определение правил, которые могут использоваться для получения результата
- выполнение тех правил, которые были отобраны на первом этапе

Рассмотрим подробнее оба этапа.

4.3.1 Определение правил

Пусть шаблон RDF графа при запросе содержит тройку с некоторым предикатом P(x,y), тогда для выполнения данного запроса необходимо активизировать все правила базы данных, результатом которых является вставка триплета с данным предикатом. Однако активизации правил базы данных только для предикатов, фигурирующих в шаблоне RDF графа, будет недостаточно для получения результата. Предположим, что для данного предиката существует некоторое SWRL правило R(?x,?y) → P(?x,?y), такое, что существуют также правила, заголовок которых содержит предикат R. В таком случае для получения всех триплетов с предикатом P может потребоваться и активизация всех правил, которые содержат в заголовке предикат R.

Условием срабатывания правил базы данных является некоторое событие, например, вставка кортежа в таблицу. Поэтому для выполнения логического вывода предлагается создать таблицу, на которой будут определены правила базы данных, условием выполнения которых будет вставка в данную таблицу кортежей, являющихся шаблонами цели. То есть если в результате данного запроса необходимо получить триплеты с предикатом P, то в данную таблицу должен быть вставлен кортеж с этим предикатом. Исходя из соображений, указанных выше, для каждого правила, заголовок которого содержит предикат P, в таблицу также должны быть вставлены кортежи, содержащие предикаты из тела правила. Для определенных предикатов в таблицу также должны быть вставлены кортежи, соответствующие предикатам из тела правил, заголовок которых содержит эти предикаты. Процесс должен продолжаться до тех пор, пока не будут определены все те шаблоны целей, на основе которых нужно выполнять правила базы данных для получения результата запроса.

Необходимость такого процесса можно пояснить на следующем простом примере: пусть имеется триплет Q(a,b) и правила R(?x,?y) → P(?x,?y) и Q(?u,?v) → R(?u,?v), пусть также имеется запрос с шаблоном ‘(?v1 P ?v2)’. Если при таких условиях будет выполняться только правило базы данных, выполняющее вставку триплетов с предикатом

P, то результат запроса будет пустым, так как второе правило не будет принято во внимание. Если же будет выполнен процесс определения правил, то результат запроса будет соответствовать триплету P(a,b).

Для определения правил базы данных создаются две таблицы pg_rdf_oldprop (property_id, arg1_id, arg2_id) и pg_rdf_newprop (property_id, arg1_id, arg2_id). После завершения процесса определения правил в таблице pg_rdf_oldprop должны находиться все кортежи, соответствующие шаблонам целей тех правил, которые должны быть выполнены.

Процесс определения правил начинается с построения кортежей шаблонов цели по заданному шаблону RDF графа по следующим правилам:

- каждой тройке в шаблоне соответствует один кортеж
- в каждой тройке субъекту соответствует столбец arg1_id, свойству - property_id, объекту - arg2_id
- если какой-либо элемент тройки является переменной, то значение соответствующего столбца равно NULL
- если же элемент является URI или литералом, то значением столбца является соответствующий целочисленный идентификатор из таблицы pg_rdf_values

Например для шаблона '(?a hasFather ?b)(?b married Ann)' будут созданы два кортежа <get_id(hasFather), NULL, NULL> и <get_id(married), NULL, get_id(Ann)>. Будем говорить, что шаблон цели t1 поглощает шаблон цели t2, если выполняются следующие условия: 1) оба шаблона содержат одинаковые предикаты и 2) первый (второй) аргумент шаблона t1 равен NULL или первому (соответственно, второму) аргументу шаблона t2. Например, <P,a, NULL> поглощает <P, a, b> и <P, a, NULL>, а <P, a, b> поглощает только самого себя.

Построенные кортежи вставляются в таблицу pg_rdf_newprop, на которой определен триггер. Триггер запрещает вставку в данную таблицу любых кортежей. Для каждого вставляемого кортежа проверяется, существует ли в таблице pg_rdf_oldpop кортеж шаблона, поглощающего данный. Если такого кортежа нет, то триггер выполняет вставку полученного кортежа в таблицу pg_rdf_oldpop. Таким образом удается избежать вставки в таблицу pg_rdf_oldpop кортежей, которые уже есть в ней или для которых присутствуют более общие (поглощающие) шаблоны.

Для того чтобы определение шаблонов выполнялось быстрее можно расширить функциональность триггера таким образом, чтобы он не сразу вставлял в pg_rdf_oldprop полученный шаблон, а пытался сначала строить поглощающий его шаблон. Триггер, проверив, что для кортежа шаблона tn нет поглощающего его в pg_rdf_oldprop, будет

определять число всех таких кортежей, предикат которых равен предикату нового шаблона tn, первый (второй) аргумент совпадает с первым (вторым) аргументом шаблона tn («совпадает» означает, что либо оба аргумента имеют значение NULL, либо равны), а вторые (первые) аргументы отличаются. Если число таких кортежей равно некоторому заранее заданному числу N, то все эти N кортежей удаляются из таблицы pg_rdf_oldprop, кортеж tn также не вставляется в таблицу, а выполняется вставка кортежа, полученного из tn заменой второго (первого) аргумента на NULL. Для построенного шаблона может опять выполняться триггер. Обработка шаблонов, предикатом которых является rdf:type, отличается: для них не выполняется обобщение второго аргумента, т.е. значения типа. Наличие большого числа шаблонов с одинаковым значением предиката, но различными аргументами в большинстве случаев невыгодно из-за того, что большинство правил не зависит от значений аргументов шаблона. Для rdf:type, напротив, важным является значение типа, и его замена на NULL может повлечь активизацию большого числа правил базы данных, не нужных для получения результата.

Однако возможны ситуации, когда предложенный механизм построения поглощающего шаблона не позволяет избежать накопления таких шаблонов, например, в случае, когда выполняется вставка шаблонов с одинаковыми предикатами, но различными значениями аргументов. Поэтому триггер может дополнительно отслеживать число всех шаблонов с тем же предикатом, что и у вставляемого шаблона. Если это число равно некоторому заранее заданному числу L, то все они удаляются из pg_rdf_oldprop, выполняется вставка шаблона с данным предикатом и значением NULL вместо обоих аргументов.

На таблице pg_rdf_oldprop определены правила базы данных, которые позволяют определить все необходимые шаблоны. Правила базы данных могут строиться двумя различными способами:

1. для каждого полученного правила R(?x,?y) → P(?x,?y) на этапе разбора входного файла генерируется правило базы данных следующего вида:

   ```
   CREATE RULE <rulename> AS ON INSERT TO pg_rdf_oldprop
        WHERE NEW.property_id = get_id('P')
   DO INSERT INTO pg_rdf_newprop
        SELECT get_id('R'), NEW.arg1_id, NEW.arg2_id;
   ```

   Если тело правила состоит из нескольких атомов, то генерируется столько правил базы данных, сколько атомов. Кроме того, если в заголовке правила встречается атом, у которого один или оба аргумента не переменные, а постоянные значения, то в указанном выше правиле базы данных изменяется действие. Если заголовок

правила содержит постоянные значения, также соответствующим образом изменяются условие и действие сгенерированного правила базы данных. Приведем пример, иллюстрирующий приведенные замечания. Для правила R(?x,a) → P(b,?x) будет построено правило базы данных следующего вида:

```
CREATE RULE <rulename> AS ON INSERT TO pg_rdf_oldprop
     WHERE NEW.property_id = get_id('P')
     AND (NEW.arg1_id = get_id('b') OR NEW.arg1_id IS NULL)
DO INSERT INTO pg_rdf_newprop
     SELECT get_id('R'), NEW.arg2_id, get_id('a');
```

2. Использовать специальную таблицу pg_rdf_depend, в которую для каждого полученного правила будут заноситься зависимости между шаблонами. Так, для правила R(?x,?y) → P(?x,?y) при разборе входного файла в таблицу должен будет вставлен кортеж, в котором будет содержаться информация о том, что шаблон <get_id('P'), NULL, NULL> зависит от шаблона <get_id('R'), NULL, NULL>. При вставке в таблицу pg_rdf_oldprop нового кортежа будет срабатывать специально определенное правило, которое для данного шаблона будет искать в pg_rdf_depend информацию о том, есть ли кортежи шаблонов, от которых зависит новый шаблон или поглощающий его. Если такие шаблоны будут найдены, то они будут вставляться в pg_rdf_newprop. Например, если в pg_rdf_oldprop был вставлен шаблон <get_id('P'), NULL, NULL>, то правило вставит в pg_rdf_newprop кортеж <get_id('R'), NULL, NULL>. Отметим, что в данном случае для определения шаблонов можно использовать не правило базы данных, а триггер.

Таким образом, с помощью правил базы данных или триггеров автоматически будут определяться все необходимые шаблоны целей.

Число правил, загруженных в базу данных конечно, число предикатов, используемых в правилах также конечно. Количество шаблонов, содержащих каждый определенный предикат, которые могут быть вставлены в таблицу pg_rdf_oldprop, ограничено числом порядка L N. Для rdf:type число таких шаблонов ограничено числом порядка N. Поэтому выполнение данного этапа может быть оценено как O(C N + M L N), где M – число предикатов, кроме rdf:type, участвующих в правилах, C – число различных значений типов, участвующих в запросе.

Стоит отметить, что предложенный динамический алгоритм накладывает на шаблоны RDF графа, задаваемого при запросах в функции RDF_QUERY следующие ограничения: значение предиката в каждой тройке не может быть переменной, значение объекта в тройках с предикатом rdf:type также не может быть переменной.

4.3.2 Выполнение правил

Для выполнения собственно логического вывода в базе данных создается таблица задач pg_rdf_tasks. При выполнении разбора входных файлов для онтологий и правил SWRL генерируются соответствующие правила на таблице pg_rdf_tasks. Например, если во входном файле имелось правило следующего вида P(?x,?y) → R(?x,?y), то модуль разбора должен сгенерировать правило:

```
CREATE RULE <rulename> AS ON INSERT TO pg_rdf_tasks
     WHERE NEW.property_id = get_id('R')
DO INSERT INTO pg_rdf_inferenced
     SELECT tr.subject_id, get_id('R'), tr.object_id
     FROM pg_rdf_triples tr WHERE property_id = get_id('P');
```

Данный пример показывает, что правила базы данных определены таким образом, что условием их срабатывания является вставка в pg_rdf_tasks кортежа шаблона, соответствующего атому из заголовка исходного правила. Поэтому вставка в данную таблицу кортежа со значением property_id, соответствующим некоторому предикату P, означает, что СУБД будет выполнять все правила для получения триплетов с данным предикатом P.

Исходя из указанных соображений, выполнение логического вывода состоит из следующих шагов:

1. выполняется вставка в pg_rdf_tasks каждого кортежа, полученного на предыдущем этапе
2. определяется, был ли получен хоть один новый триплет. Если да, то все триплеты из таблицы pg_rdf_inferenced (subject_id, property_id, object_id) вставляются в pg_rdf_triples и удаляются из pg_rdf_inferenced и выполняется переход на шаг 1, иначе – шаг 3
3. останов

Использование таблицы pg_rdf_inferenced так же обусловлено ограничениями на рекурсию в определении правил, как и в случае статического алгоритма. Кроме того, триггер, определенный на данной таблице, разрешающий вставку только новых триплетов, упрощает выполнение шага 2: для того, чтобы определить, были ли получены на данной итерации новые триплеты, достаточно проверить, есть ли в данной таблице хоть один кортеж.

4.3.3 Модификации алгоритма

Для увеличения скорости выполнения запросов можно предложить следующие модификации динамического алгоритма:

- Преимуществом статического алгоритма являлось то, что выполнение логического вывода на этапе загрузки данных делало выполнение запросов простой и быстрой задачей. Поэтому после выполнения каждого запроса предлагается не удалять все триплеты, полученные в результате логического вывода. В таком случае время выполнения следующего запроса может быть сокращено за счет того, что часть триплетов, которые требуются для получения результата, были уже получены на предыдущем запросе. В частном, случае выполнение правил для такого же запроса будет состоять всего из одной итерации. Если же пользователь хочет внести изменения в хранимые триплеты, он может удалить все выведенные триплеты с помощью специально определенной функции clearinf().
- Во вспомогательной таблице pg_rdf_usedid определяется столбец iter_num, в который после каждой итерации заносится ее порядковый номер, начиная с 1. При вставке кортежей, полученных логическим выводом в таблицу триплетов pg_rdf_triples значение столбца isinferenced равно номеру итерации, на которой был получен данный кортеж. В этом случае на каждой итерации при выполнении правил можно выбирать только те кортежи, из которых могут быть получены новые тройки. Например, для правила R(?x,?y) → P(?x,?y) генерировать следующее правило базы данных:

```
CREATE RULE <rulename> AS ON INSERT TO pg_rdf_tasks
     WHERE NEW.property_id = get_id(‘R’)
DO INSERT INTO pg_rdf_inferenced
     SELECT tr.subject_id, get_id(‘R’), tr.object_id
     FROM pg_rdf_triples tr, pg_rdf_usedid ui
     WHERE tr.property_id = get_id(‘P’)
     AND tr.isinferenced >= ui.iter_num;
```

Знак ‘>=’ используется вместо ‘=’, так как результаты предыдущих запросов, у которых значение столбца isinferenced может быть достаточно большим, могли быть не удалены, а они тоже должны быть учтены при выполнении логического вывода.

- В предложенном выше алгоритме выполнения правил на каждой итерации в таблицу задач вставляются все определенные кортежи шаблонов цели. Однако некоторые правила базы данных требуется выполнять не на всех итерациях, а иногда только на первой итерации. Например, если для шаблона с предикатом P имеется только одно правило R(?x,?y) → P(?x,?y), при чем триплеты с предикатом R заданы только явно (т.е.

загружены из файла и не могут быть выведены), то кортеж данного шаблона имеет смысл вставлять в таблицу задач только на первой итерации. Вставка этого кортежа на следующих итерациях не принесет получения новых триплетов, но повлечет дополнительные обращения к хранимым данным.

Чтобы избежать этих дополнительных обращений можно предложить следующие изменения динамического алгоритма. При загрузке правил в базу данных в таблицу зависимостей pg_rdf_depend заносится информация о том, как шаблоны зависят друг от друга. При вставке шаблона в таблицу pg_rdf_oldprop ему в соответствие ставится уникальный целочисленный идентификатор. Для этого в данной таблице добавляется новый столбец key_id. После того как все шаблоны были определены, заполняется таблица зависимостей шаблонов данного запроса pg_rdf_taskdep(anc_id, con_id). Для любых двух кортежей t1 и t2 из pg_rdf_oldprop в pg_rdf_taskdep заносится кортеж <t1.key_id, t2.key_id>, если в таблице зависимостей pg_rdf_depend есть информация о том, что шаблон, соответствующий t2, зависит от шаблона, соответствующего t1.

В таблице pg_rdf_oldprop создаются два булевых столбца oldhasinf и newhasinf. Перед каждой итерацией для всех шаблонов в pg_rdf_oldprop значение столбца newhasinf устанавливается равным FALSE. Во время выполнения каждой итерации при вставке нового триплета в pg_rdf_inferenced у шаблона, которому соответствует данный триплет, значение столбца newhasinf изменяется на TRUE. После завершения итерации таблица pg_rdf_oldprop обновляется следующим образом: столбец oldhasinf кортежа t1 получает значение TRUE, если в таблице pg_rdf_taskdep существует кортеж td такой, что td.con_id = t1.key_id и существует кортеж t2 в таблице pg_rdf_oldprop такой, что t2.key_id = td.anc_id и t2.newhasinf IS TRUE. Другими словами, столбец oldhasinf для некоторого шаблона будет равен TRUE, если существует некоторое правило, заголовок которого - атом, соответствующий данному шаблону, а в теле атом, для которого был получен хотя бы один новый триплет. У всех остальных кортежей в pg_rdf_oldprop значение столбца oldhasinf устанавливается равным FALSE. Перед первой итерацией значение столбца oldhasinf устанавливается равным TRUE для всех кортежей.

На каждой итерации в таблицу pg_rdf_tasks вставляются только те кортежи из pg_rdf_oldprop, у которых столбец odlhasinf имеет значение TRUE, то есть существует хотя бы одно правило, которое позволяет получить новые триплеты по данному шаблону.

5 Реализация интеграции СУБД PostgreSQL с семантическим веб

5.1 Описание реализованной системы

Для выполнения разбора документов в форматах семантического веб и загрузки построенных троек и сгенерированных правил в базу данных был создан модуль на языке C. Для написания модуля был выбран язык C, так как в дистрибутивы PostgreSQL включена библиотека libpq для доступа к PostgreSQL из программ, написанных на C.

Для выполнения логического вывода был реализован динамический алгоритм с предложенными модификациями, так как он не обладает указанными недостатками статического метода.

Функция RDF_QUERY была написана на процедурном языке PL/Perl, так как с помощью Perl удобно выполнять операций над строками. На вход функции подается текстовая строка, представляющая собой шаблон RDF графа. По данной строке функция строит шаблоны триплетов и выполняет их вставку в pg_rdf_newprop, генерирует запрос, с помощью которого пользователю выдается результат запроса после выполнения логического вывода. Для осуществления логического вывода из функции RDF_QUERY вызывается написанная на PL/pgSQL функция inference. Использование PL/pgSQL было обусловлено тем, что данный язык позволяет выполнять команды SQL и использовать условные конструкции и циклы для реализации итеративного процесса.

На рисунке 4 отмечены таблицы, создаваемые для интеграции СУБД PostgreSQL с семантическим веб. Кроме таблицы pg_rdf_triples для хранения RDF триплетов, были созданы таблицы pg_rdf_typetrip для хранения триплетов с предикатом rdf:type и pg_rdf_disjoint – с предикатом owl:disjointWith. Использование этих таблиц позволяет сократить выделяемую для хранения триплетов с указанными предикатами память. Кроме того, при выполнении правил, связанных с использованием данных предикатов, появляется возможность не обращаться к единой таблице, хранящей все триплеты с произвольными предикатами, а сократить область поиска триплетами с соответствующим предикатом. Стоит отметить, что правила, связанные с rdf:type встречаются часто, а определение индивидуальности ресурсов (owl:differentFrom) связано с использованием триплетов с owl:disjointWith. Для того чтобы в системе присутствовала возможность создания и других таблиц для хранения триплетов с определенными предикатами, над указанными тремя таблицами определено представление pg_rdf_tr, объединяющее с помощью операции UNION все хранимые триплеты. В правилах баз данных для доступа к триплетам с произвольными предикатами используется данное представление. Поэтому при добавлении новой таблицы для хранения триплетов нужно переопределить

представление соответствующим образом и определить правило или триггер над pg_rdf_triples, чтобы определенные триплеты вставлялись в новую таблицу. Для доступа к RDF данным без выполнения логического вывода над pg_rdf_tr определено представление pg_rdf_view.

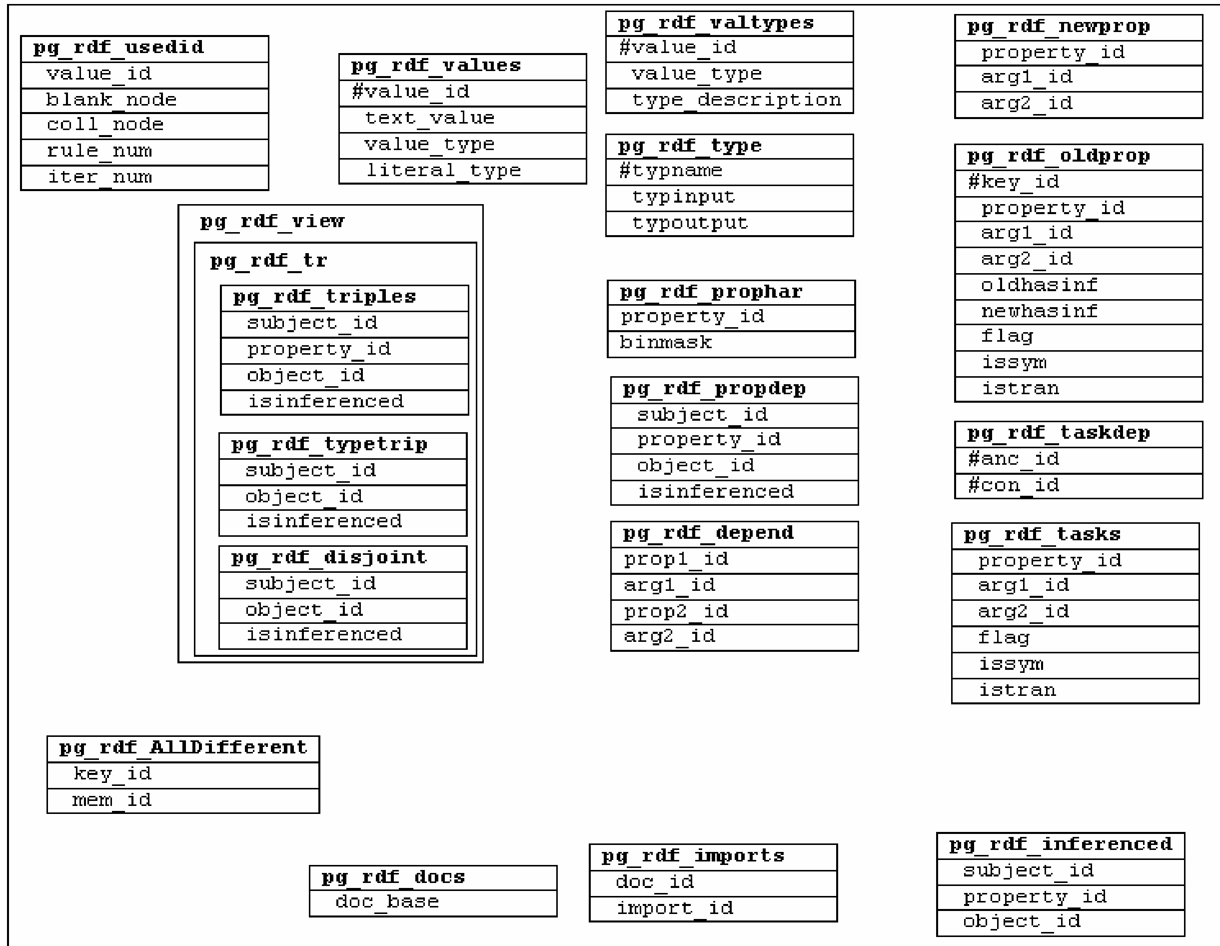

Рисунок 4

Таблица pg_rdf_prophar используется для хранения информации о свойствах предикатов. Для каждого предиката, заданного в таблице своим целочисленным идентификатором (property_id), изменяются значения четырех бит маски binmask: если binmask & 1 = 1, то предикат обладает свойством транзитивности (owl:TransitiveProperty),если binmask & 2 = 2 – свойством симметричности (owl:SymmetricProperty), если binmask & 4 = 4 – функциональной зависимости (owl:FunctionalProperty), если binmask & 8 = 8 – обратной функциональной зависимости (owl:InverseFunctionalProperty).

В таблице pg_rdf_oldprop для всех шаблонов с предикатами, обладающими свойством транзитивности (симметричности) и только для них значение столбца istran (issym, соответственно) принимает значение TRUE. Для таких шаблонов выполняется транзитивное замыкание или симметричное отображение, соответственно. После каждой

итерации для таких шаблонов столбец oldhasinf принимает значение TRUE не только в случае, указанном в предложенном алгоритме, но и если столбец newhasinf данного шаблона равен TRUE. Столбец flag используется для указания того, что для предиката данного шаблона существуют подсвойства (rdfs:subPropertyOf) или обратное свойство (owl:inverseOf). При вставке в pg_rdf_tasks шаблонов с определенным значением данного столбца выполняются соответствующие правила базы данных.

При определении шаблонов используется не отдельное правило базы данных для каждого правила, а таблица pg_rdf_depend. Кроме того, для указания подсвойств и обратных свойств используется отдельная таблица pg_rdf_propdep, которая позволяет сократить используемую память, а также применяется на этапе логического вывода. Принято решение использовать не отдельные правила базы данных, а таблицу зависимостей, поскольку она позволяет строить таблицу зависимостей запроса pg_rdf_taskdep.

Таблица pg_rdf_AllDifferent используется для хранения информации полученной из элемента owl:AllDifferent, все ресурсы из одного элемента хранятся с одинаковым значением key_id. При определении шаблонов, если первый раз был получен шаблон с предикатом owl:differentFrom, то для каждой пары кортежей t1, t2 с одинаковыми значениями key_id в pg_rdf_triples вставляется кортеж с данным предикатом, свойством и объектом, равными t1.mem_id и t2.mem_id.

Для отслеживания импортов используются таблицы pg_rdf_docs, в которой хранится значение base для каждого загруженного документа, и pg_rdf_imports, указывающая для каждого документа (doc_id) импортируемые документы (import_id). Наличие такой информации позволяет при загрузке документа проверять, присутствуют ли в базе данных импортируемые документы. Если эти документы отсутствуют, выдается соответствующее сообщение.

На рисунке 5 изображена общая схема использования реализованной системы. Команды создания таблиц, правил, триггеров и функций, используемых для интеграции PostgreSQL с семантическим веб сохранены в отдельном SQL файле rdf_analyze.sql. Поэтому для того, чтобы добавить в базу данных возможность выполнения запросов к RDF данным, пользователь должен выполнить следующие действия:

1. при необходимости создать базу данных с помощью команды createdb. Например: `$ createdb rdf_analyzeDB`
2. Загрузить в выбранную базу данных языки PL/Perl и PL/pgSQL:
   ```
   $ createlang plperl rdf_analyzeDB
   $ createlang plpgsql rdf_analyzeDB
   ```

3. выполнить команды, сохраненные в SQL файле, с помощью команды psql. Например: `$ psql rdf_analyzeDB` </home/postgres/RDF/SQL/rdf_analyze.sql

RDF, OWL или SWRL документ
rdf_analyze
вставка триплетов и генерация правил
интерфейс libpq
rdf_analyze.sql
psql
БД
запросы с помощью RDF_QUERY
создание таблиц, правил, триггеров и функций

Рисунок 5

Для того чтобы использовать модуль разбора, сначала нужно выполнить компиляцию его исходных кодов с помощью команды make. После чего загрузка триплетов и правил в базу данных из файлов выполняется в командной строке shell командой rdf_analyze с указанием в качестве параметров базы данных, в которую будет выполняться вставка, и файла, содержащий информацию в форматах семантического веб. В указанную базу данных заранее должны быть добавлены возможности для работы с семантическим веб, как указано выше.

5.2 Демонстрационная задача

Для демонстрации возможностей реализации использовалась онтология семейных отношений, в которой используются SWRL правила, доступная в [16]. Особенностью данной онтологии является то, что большинство триплетов задано неявно с помощью правил SWRL и конструкций OWL. На рисунке 6 отображены отношения родства, заданные явно в онтологии. На рисунке отношение «иметь Ребенка» (hasChild) отражается тонкой стрелкой, отношение «иметь Супруга» (hasConsort) – толстой пустой стрелкой; стрелки ведут от субъекта триплета к объекту. В прямоугольники с прямыми углами помещены ресурсы, про которые известно, что они имеют мужской пол (hasSex Male), в

прямоугольники со скругленными концами – женский пол (hasSex Female). Кроме того, в онтологии явно заданы триплеты с предикатом hasParent («иметь Родителя»), удаление которых из базы данных не изменяет времени выполнения проведенных запросов и результатов, так как эти триплеты могут быть выведены из утверждений, отраженных на рисунке 6. Для каждого ресурса указаны его уникальный идентификатор и значение свойства name («имя»), например, F01 и Mary, соответственно.

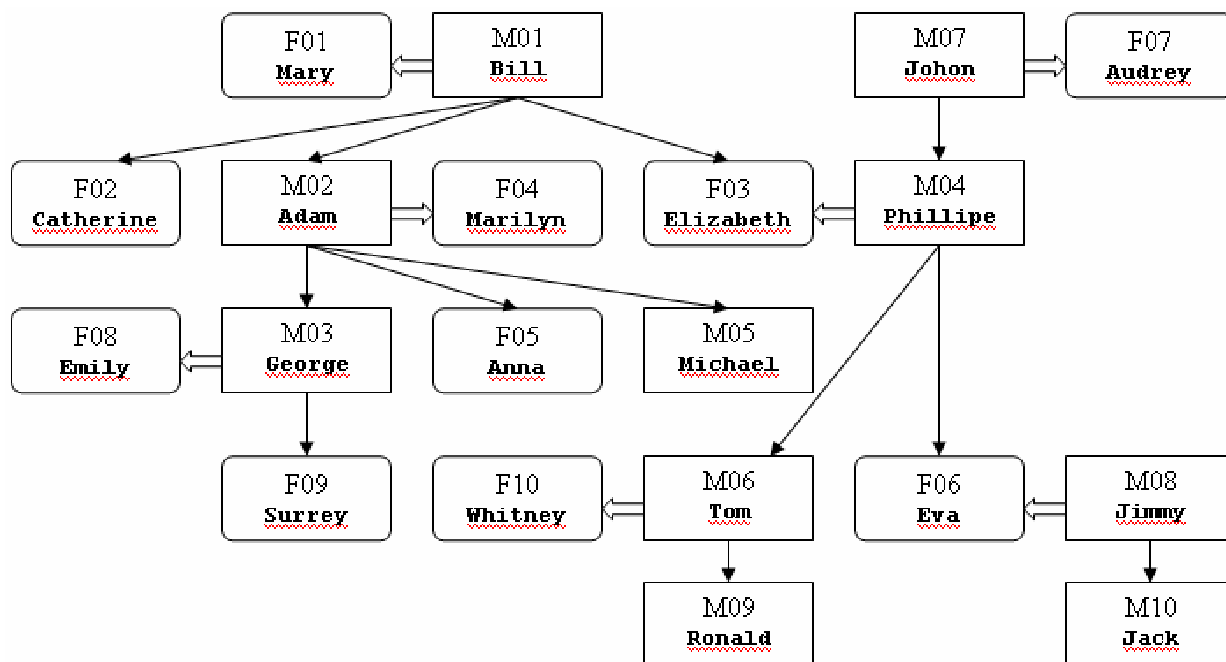

Рисунок 6

С помощью реализованной системы выполняются запросы к хранимым данным на определение отношений hasSibling («иметь Брата или Сестру», запрос Q1), hasSon («имтеь Сына», Q2) и hasDaughter («иметь Дочь», Q3), hasFather («иметь Отца», Q4) и hasMother («иметь Мать», Q5), hasDescendent («иметь Предка», Q6), hasBrother («иметь Брата», Q7) и hasSister («иметь Сестру», Q8), hasAunt («иметь Тетю», Q9) и hasUncle(«иметь Дядю», Q10). Например, запрос Q4 выглядит следующим образом:

```
SELECT n1,n2 FROM RDF_QUERY(
'(?s http://a.com/ontology#hasFather ?m)
 (?s http://a.com/ontology#name ?n1)
 (?m http://a.com/ontology#name ?n2)')
AS res (s text, m text,n1 text,n2 text);
```

Серии из десяти указанных запросов выполнялись:

1. для реализации динамического алгоритма без модификаций,
2. для реализации алгоритма с использованием столбца isinferenced в правилах базы данных,

3. с использованием столбца isinferenced и определением используемых шаблонов после каждой итерации,
4. без использования clearinf (т.е. не удаляя результаты логического вывода при выполнении предыдущих запросов) после выполнения запросов.

	Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**
1	4548,089	4586,605	4590,082	4649,636	4631,810	5289,772
2	4027,332	3837,798	3841,228	3873,910	3836,865	4060,827
3	1955,127	1706,378	1692,431	1792,438	1674,307	1803,400
4	1890,977	641,051	658,196	646,781	642,206	892,902
рез (стр)	14	14	10	12	12	54
рез (%)	100	100	100	100	100	100
	Q7	**Q8**	**Q9**	**Q10**	**среднее**	
1	4518,293	4573,696	4617,119	4541,548	4654,665	
2	3833,515	3841,377	3817,312	3847,415	3881,758	
3	1668,277	1691,868	1680,943	1692,480	1735,765	
4	665,792	654,797	656,510	653,717	800,293	
рез (стр)	7	7	10	4	-	
рез (%)	100	100	100	100	100	

Таблица 1

В таблице 1 приведено время (мс) выполнения всех запросов и среднее время выполнения запросов для указанных четырех способов выполнения запросов, а также для всех запросов – число строк в результирующих таблицах и процентная доля полученных результатов от ожидаемых. Запросы выполнялись на системе Pentium IV 2.6 ГГц, 512 Мб RAM в Mandrake Linux 10.1, версия PostgreSQL 8.2.1. Полученные результаты показывают, что предложенные модификации позволяют сократить время выполнения запросов. Определение шаблонов после каждой итерации позволило получить достаточно большой выигрыш по времени, благодаря тому, что выполнение логического вывода при данных запросов состояло из 9 итераций. Сохранение триплетов, полученных логическим выводом на предыдущих запросах, позволило сократить число итераций последующих запросов. Рассмотренные реализации отличаются временем выполнения, а полученные результаты запросов верны и полны, то есть получены только верные результаты и при том все.

На примере данной онтологии можно показать преимущество предложенного подхода, основанного на использовании возможностей SQL, а не специальных API, - к результатам запросов к RDF данным можно применять всю мощь языка SQL. Например, можно привести следующие примеры:

- К результатам RDF запросов можно применять агрегатные функции. Например, можно определить, число всех триплетов, соответствующих отношению hasDescendent:

```
SELECT count(*) FROM RDF_QUERY('
(?s http://a.com/ontology#hasDescendent ?m)
```

```
 (?s http://a.com/ontology#name ?n1)
 (?m http://a.com/ontology#name ?n2)')
AS res (s text,m text,n1 text,n2 text);
```

- Вызовы функции RDF запросов могут указываться в произвольных запросах везде, где могут использоваться обычные таблицы или представления, - в списках выборки, в условиях WHERE предложений и др.

emp_name	dep_id
Tom	1
Marilyn	1
Bill	1
Anna	1
Jack	2
Donald	2
Catherine	2
Elizabeth	3
Eva	3
Adam	3
Ken	5
George	5
Michael	5
Ronald	5

```
SELECT e1.* FROM
RDF_QUERY(
'(?s http://a.com/ontology#hasBrother ?m)
 (?s http://a.com/ontology#name ?n1)
 (?m http://a.com/ontology#name ?n2)')
AS res (s text, m text,n1 text,n2 text),
 emp e1,emp e2
 WHERE e1.emp_name = n1 AND e2.emp_name = n2
 AND e1.dep_id = e2.dep_id;

 emp_name  | dep_id
-----------+--------
 Elizabeth |      3
 George    |      5
 Michael   |      5
(3 rows)
```

Рисунок 7

На рисунке 7 приведен пример такого использования. Пусть определена таблица emp (emp_name, dep_id), в которой для каждого служащего некоторой организации хранятся его номер и номер отдела, к которому он принадлежит. С помощью функции RDF_QUERY можно определить имена тех служащих, которые служат в одном отделе со своим братом. Таблица emp указана слева на рисунке 7, а запрос и его результат – справа.

- К результатам RDF запросов можно применять операции над множествами. Например, определение братьев или сестер может быть выполнено, не только указав предикат hasSibling, но и с помощью операции объединения:

```
SELECT n1,n2 FROM (
SELECT * FROM RDF_QUERY(
'(?s http://a.com/ontology#hasBrother ?m)
 (?s http://a.com/ontology#name ?n1)
 (?m http://a.com/ontology#name ?n2)'
) AS res (s text,m text,n1 text,n2 text)
UNION
```

```
SELECT * FROM RDF_QUERY(
'(?s http://a.com/ontology#hasSister ?m)
 (?s http://a.com/ontology#name ?n1)
 (?m http://a.com/ontology#name ?n2)'
) AS res (s text,m text,n1 text,n2 text)) AS foo;
```

Полученные результаты будут одинаковые, но время выполнения запроса с использованием операции объединения больше, так как в данном случае функция RDF_QUERY вызывается два раза.

Таким образом, на примере данной онтологии видно, что выполненная реализация позволяет выполнять логический вывод на основе правил SWRL и средств OWL, а также применять к результатам RDF запросов средства SQL.

5.3 Сравнение с существующими системами

Проведенный ранее анализ существующих систем показал, что большинство из них основывается на специально определенных API. Однако такой подход связан с некоторыми недостатками, такими как неэффективность при преобразовании форматов данных, трудности при интеграции с SQL запросами. Предложенный подход и выполненная реализация основываются на использовании возможностей SQL и не имеют указанных недостатков, благодаря чему становится возможным применение к результатам RDF запросов всей мощи языка SQL. Существующей системой для запросов к RDF данным, также основанной на SQL, является решение, предложенное Oracle. Однако предложенная в данной статье система позволят работать не только с форматами RDF, RDFS и простыми пользовательскими правилами, как у Oracle, а также с форматами OWL и SWRL.

6 Результаты. Дальнейшая работа

Анализ известных решений для работы с семантическим веб, показал, что большинство решений основывается на использовании специально определенных API. В то же время применение для этих целей возможностей SQL позволяет получить определенные преимущества при интеграции СУБД с семантическим веб.

Анализ возможностей PostgreSQL для интеграции данной СУБД с семантическим веб, показал, что для выполнения логического вывода можно использовать триггеры и систему правил, для запросов могут быть полезными табличные функции, а также доступны различные методы индексирования.

Основываясь на определенных возможностях СУБД, предложено два алгоритма решения поставленной задачи. Статический алгоритм предполагает осуществление логического вывода во время загрузки данных. Динамический алгоритм, напротив, основан на осуществлении логического вывода всякий раз при выполнении логического

вывода. Отмечается, что статический алгоритм обладает рядом недостатков, связанных с модификацией данных, которых нет у динамического алгоритма. Однако динамический алгоритм связан с временными затратами при выполнении запросов, на сокращение которых направлены предложенные модификации.

Выполнена реализация интеграции СУБД PostgreSQL на основе предложенного динамического алгоритма, так как он лишен проблем статического алгоритма. Было показано, что реализованная система позволяет работать с существующими форматами семантического веб – RDF, RDFS, OWL и SWRL, а также применять к результатам RDF запросов мощь языка SQL.

Дальнейшие работы по интеграции СУБД с семантическим веб могут быть направлены на повышение эффективности выполнения запросов. В достижении данной цели полезным может оказаться исследование более эффективных схем хранения данных, а также алгоритмов выполнения логического вывода, позволяющих выполнять меньшее число правил базы данных без изменений в полученных результатах.

Литература

1. PostgreSQL: The world's most advanced open source database [HTML] (http://www.postgresql.org/)
2. T. Berners-Lee, J. Handler, O. Lassila. The Semantic Web. Scientific American, May 2001, pp. 34–43
3. F. Manola, E. Miller, eds RDF Primer W3C Recommendation 10 February 2004 [HTML] (http://www.w3.org/TR/2004/REC-rdf-primer-20040210/)
4. D. Brickley, R.V. Guha, B. McBride RDF Vocabulary Description Language 1.0: RDF Schema W3C Recommendation 10 February 2004 [HTML] (http://www.w3.org/TR/2004/REC-rdf-schema-20040210/)
5. M. K. Smith, C. Welty, D. L. McGuinness OWL Web Ontology Language Guide W3C Recommendation 10 February 2004 [HTML] (http://www.w3.org/TR/2004/REC-owl-guide-20040210/)
6. I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean SWRL: A Semantic Web Rule Language Combining OWL and RuleML W3C Member Submission 21 May 2004 [HTML] (http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/)
7. The Protégé Ontology Editor and Knowledge Acquisition System [HTML] (http://protege.stanford.edu/)
8. TopBraid Composer [HTML] (http://www.topbraidcomposer.com/)

9. E. Prud'hommeaux, A. Seaborne SPARQL Query Language for RDF W3C Working Draft 26 March 2007 [HTML] (http://www.w3.org/TR/2007/WD-rdf-sparql-query-20070326/)
10. H. Knublauch, R. W. Fergerson, N. F. Noy, M. A. Musen. The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications. In Proceedings of the 3rd International Semantic Web Conference (ISWC), 2004
11. Jena Semantic Web Framework [HTML] (http://jena.sourceforge.net/)
12. K. Wilkinson, C. Sayers, H. Kuno, D. Reynolds Efficient RDF Storage and Retrieval in Jena2, First International Workshop on Semantic Web and Databases, pp. 131-151, 2003
13. E. I. Chong, S. Das, G. Eadon, J. Srinivasan An Efficient SQL-based RDF Querying Scheme. In Proceedings of the 31st International Conference on Very Large Data Bases, 2005, pp. 1216 – 1227
14. О. Бартунов, Ф. Сигаев Написание расширений для PostgreSQL с использованием GiST [HTML] (http://www.sai.msu.su/~megera/postgres/talks/gist_tutorial.html)
15. P. Hayes, B. McBride RDF Semantics W3C Recommendation 10 February 2004 [HTML] (http://www.w3.org/TR/2004/REC-rdf-mt-20040210/)
16. Protégé Ontologies Library [HTML] (http://protege.cim3.net/cgi-bin/wiki.pl?ProtegeOntologiesLibrary)