УДК 004.047

Алгоритмы обработки запросов к дедуктивным базам данных и реализация алгоритма QSQ

© А.И. Выборнов, А.В. Дубанов

МГТУ им. Н.Э. Баумана, Москва, 105005, Россия

Рассмотрены существующие алгоритмы обработки запросов к дедуктивным базам данных, основанных на языке Datalog. Наиболее эффективный из них — рекурсивный алгоритм «запрос-подзапрос» (QSQR) — реализован в составе прототипа интерфейса программирования приложений.

Ключевые слова: *база данных, Datalog, алгоритм QSQ, дерево доказательства.*

Введение. Необходимой частью любой системы управления базами данных (СУБД) является язык запросов к базе данных (БД). В настоящее время наибольшее распространение получил «язык структурированных запросов» (Structured Query Language — SQL), который ориентирован на реляционную модель данных. Однако существуют языки запросов к БД, работающие в терминах математической логики. Подобные языки часто называют дедуктивными. Запрос к БД, сформулированный на таком языке, часто оказывается более кратким и понятным по сравнению с запросом на языке SQL.

Наиболее распространенным дедуктивным языком запросов является Datalog. Язык удобен для описания отношений наследования, семантических сетей и иных моделей из различных предметных областей, где наиболее выразительным способом представления данных является граф [1]. Применение дедуктивных языков определяется следующим обстоятельством: с помощью реляционных языков очень сложно выразить любой рекурсивный запрос [2], такой как транзитивное замыкание бинарного отношения.

Бинарное отношение — отношение двух переменных $R(a,b)$. Данное отношение называют транзитивным, если для любых трех элементов a, b и c выполнение отношений $R(a,b)$ и $R(b,c)$ влечет выполнение отношения $R(a,c)$. Под транзитивным замыканием бинарного отношения R понимают наименьшее транзитивное отношение, включающее R [3]. На языке Datalog такой запрос может быть записан следующим образом:

$$p(X,Y) :- p(X,Z), p(Z,Y).$$

Здесь и далее: p — предикат, задающий отношение; X, Y и Z — переменные.

Транзитивное замыкание бинарного отношения в запросах может возникать при решении задач широкого круга предметных областей. Так, характерным примером является анализ молекулярных и реакционных графов в био- и хемоинформатике.

Существует ряд реализаций СУБД с дедуктивным языком запросов. Большая их часть реализована в виде интерфейса программирования приложений (Application Programming Interface — API) для различных функциональных языков программирования общего назначения [4]. Для языков, поддерживающих парадигму объектно-ориентированного программирования (Java, C++) и широко применяемых для промышленной разработки приложений, существует небольшое число реализаций [5, 6].

Таким образом, сохраняется актуальность СУБД с дедуктивным языком запросов, способных работать с большими объемами данных с API на наиболее широко применяемых языках программирования.

В данной работе авторами был реализован прототип СУБД с языком запросов Datalog и API на языке Java и выполнена оценка ее применимости к большим объемам данных. Были рассмотрены существующие алгоритмы вычисления программ на Datalog и выбран алгоритм, наиболее подходящий для реализации.

Язык Datalog. Данный язык дедуктивных запросов является подмножеством языка логического программирования Prolog. Однако программа на языке Datalog полностью декларативна, в то время как программы на языке Prolog на практике имеют двойное значение — декларативное (логическое) и императивное (процедурное). Программа на языке Prolog может определять порядок вычислений, в то время как порядок вычислений в программе на языке Datalog определяется реализацией последнего. Предикаты в Datalog могут рассматриваться как чистые (значение предиката однозначно определяется подстановкой переменных, вычисление значений не сопровождается побочными эффектами). Поэтому в дальнейшем будем говорить именно о *вычислении* программы, а не о ее *выполнении*. Поскольку Datalog не получил широкого распространения в практике программирования, рассмотрим его более подробно.

Факт в языке Datalog представляет собой предикат с множеством аргументов (доменом), состоящим из констант. Факт, отображающий отношение p от констант a и b, на языке Datalog записывается как $p(a,b)$.

Факт является предикатом, значение которого всегда истинно.

Предикаты p_1 и p_2 унифицируемы, если существует такая подстановка θ, что $p_1 \cdot \theta = p_2 \cdot \theta$ (здесь запись $p_1 \cdot \theta$ означает подстановку фактических аргументов θ вместо формальных аргументов предиката p). Например, предикат $p(a,b,b)$ унифицируем с предикатами: $p(a,X,X)$, так как существует подстановка $\theta = \{X / b\}$ (здесь и далее символ «/» обозначает подстановку), и $p(X,Y,Z)$ — подстановка $\theta = \{X / a, Y / b, Z / b\}$. В свою очередь, предикат $p(a,b,b)$ не унифицируем с предикатом $p(b,X,X)$.

Множество фактов являются БД. Факт языка Datalog можно рассматривать как строку таблицы реляционной БД. Таким образом, набор фактов можно представить набором реляционных таблиц.

Отношения между фактами задаются с помощью правил. Под правилом понимают строку вида *голова* : – *тело*. В качестве тела выступает набор предикатов с доменом из констант и переменных, а в качестве головы — единственный предикат также с доменом из констант и переменных. При этом все переменные, присутствующие в голове, должны использоваться в теле правила. Правило является импликацией вида *тело* $\Rightarrow$ *голова* и читается следующим образом: голова правила истинна для всех тех наборов переменных, для которых истинен каждый предикат из тела правила. Например, правило

$$p(X,Y) : - p(X,Z), p(Z,Y).$$

задает транзитивное замыкание бинарного предиката *p*. Если факты $p(a,b)$ и $p(b,c)$ — истинны, то на основе этого правила может быть установлена истинность факта $p(a,c)$. Таким образом, правила в языке Datalog эквивалентны дизъюнктам Хорна — дизъюнкции предикатов с не более чем одним положительным предикатом. При этом отрицательные литералы задают тело, а положительный — голову. Соответственно, факт представляет собой дизъюнкт Хорна с единственным положительным дизъюнктом. Запрос представляет собой дизъюнкт Хорна с единственным отрицательным предикатом и задается с помощью единственного предиката с доменом из переменных и констант. Результатом вычисления запроса является множество фактов, унифицируемых с запросом. Например, запрос:

$$? : - p(a,X).$$

вернет все факты, которые хранятся в БД либо выводятся из фактов БД по правилам и которые унифицируемы с предикатом $p(a,X)$.

Программа на языке Datalog представляет собой набор правил и запросов. Вычисление программы на языке Datalog сводится к выводу фактов, унифицируемых с запросом, на основе правил программы

и фактов в БД. Результат не зависит от порядка правил в теле программы.

Отсюда Datalog, как язык запросов к БД, намного более удобен, чем Prolog. Но из изложенного вытекает и относительный недостаток языка Datalog: поиск всех фактов, удовлетворяющих запросу, требует больших затрат времени. Таким образом, необходим эффективный алгоритм вычисления программы на Datalog.

Методы вычисления программ на языке Datalog. Теоретические основы вычисления программ на Datalog разрабатывались с 1970-х годов. Они базируются на математической логике и логическом программировании. Возможны три основных метода вычисления программ на Datalog: метод резолюций (resolution), прямой вывод (top-down evaluation) и обратный вывод (bottom-up evaluation). Кратко рассмотрим принципы, достоинства и недостатки этих методов.

Метод резолюций получил наибольшее распространение в области логического программирования. Данный метод позволяет доказать, что некоторое утверждение ложно. Поясним метод резолюций на примере. Пусть даны некоторая БД и запрос:

$$?:-p(a,X).$$

Этому выражению соответствует дизъюнкт Хорна, записанный вместе с квантором $\forall$:

$$\forall X \neg p(a,X).$$

В методе резолюций предпринимается попытка опровержения этого утверждения путем генерации контрпримеров посредством поиска таких X, для которых дизъюнкт Хорна $\neg p(a,X)$ будет ложным. Так будут получены все значения X, для которых дизъюнкт $p(a,X)$ будет истинным, эти же значения и станут ответом на запрос.

Метод резолюций используется в реализациях языка Prolog. Для этого метода актуальна проблема останова, т. е. определения итерации, после которой необходимо прекратить порождать контрпримеры. В Prolog-системах алгоритм резолюции может бесконечно порождать контрпримеры. Задача своевременного останова при этом возлагается на программиста. Предложен ряд решений данной задачи. Наиболее полно и эффективно она решена в алгоритмах обратного вывода [7].

Несмотря на то что метод резолюции предоставляет очень мощный способ вычисления логических программ, он не является приемлемым для языка Datalog, поскольку дает ответы на запросы по одному, в то время как от него требуется получать все факты, удовлетворяющие запросу.

Прямой вывод подразумевает последовательный вывод новых фактов на основе правил, БД и ранее выведенных фактов. Данный метод также называют методом вывода «снизу вверх» (восходящий алгоритм).

Алгоритм прямого вывода достаточно прост и состоит в следующем. Поместим все факты из БД в множество F, а все правила из программы — в множество R. Пусть функция $getFacts(\text{arg}_1, \text{arg}_2)$ принимает два аргумента (arg_1 — множество фактов, arg_2 — правило) и возвращает множество фактов, которые можно получить из arg_2, используя факты из arg_1. Тогда с помощью псевдокода этот алгоритм можно записать так, как показано на рис. 1.

```
1  F' ← ∅
2  while F ≠ F'
3          F' ← F
4          for ∀r ∈ R
5                  F ← F ∪ getFacts(F', r)
```

Рис. 1. Псевдокод, поясняющий метод прямого вывода

Для получения необходимого ответа следует отфильтровать полученное множество F по запросу.

Очевидно, что данный способ вычисления программы на Datalog неэффективен, так как порождает множество фактов, которые не потребуются для ответа на запрос, и использует абсолютно все факты из БД. Для повышения эффективности требуется сократить число рассматриваемых фактов за счет тех, которые не нужны для вычисления программы (нерелевантных фактов). Прямой вывод такого сокращения обеспечить не может.

Обратный вывод. В отличие от прямого при обратном выводе вычисление начинается с запроса. Метод состоит в построении дерева доказательства для каждого ответа на запрос. В контексте обратного вывода под деревом доказательства понимают дерево, обладающее следующими свойствами:

• узлы дерева представляют собой либо факт, либо правило, содержащееся в программе;

• если узел является правилом или фактом из БД, он не имеет листьев;

• если узел является не содержащимся в БД фактом, его листья представляют собой такие правила и факты, что при условии истинности фактов из правила будет следовать факт, записанный в этом узле.

Задача обратного вывода состоит в построении таких деревьев для каждого факта, являющегося ответом на запрос. Данная схема реализуется с помощью деревьев поиска, отличающихся от дерева доказательства следующим:

• вместо фактов используются предикаты, включающие и константы и переменные;

• во всех предикатах дерева переменные с одинаковыми именами имеют одно и то же значение.

При обратном выводе корнем дерева является предикат запроса. Он образует дерево поиска нулевой глубины. Для всех правил программы, голова которых унифицируема с корнем, подбираются такие предикаты, чтобы при подстановке их в правило с головой, заданной корневым узлом, получалось верное утверждение. В результате получаем несколько деревьев первого уровня. Аналогично обрабатываем все листья деревьев первого уровня, затем второго и т. д. Ввиду бесконечности этого процесса используются различные условия останова (как правило, динамические).

После того как будут порождены все необходимые деревья поиска, в каждом из них необходимо заменить переменные в листьях на факты из БД, что выполняется обходом дерева снизу вверх. Таким образом, получаем деревья доказательства, содержащие ответ на запрос. В общем случае одному дереву поиска может соответствовать несколько деревьев доказательства.

При реализации большинства алгоритмов обратного вывода построение деревьев поиска и доказательства используется не явно.

Данный подход достаточно ресурсоемок, однако дает возможность сократить количество обращений к БД, а значит, и время вычисления результатов запроса. Наиболее эффективной реализацией обратного вывода является ***алгоритм QSQ*** (Query-SubQuery — «запрос-подзапрос») [3].

Алгоритм QSQ представляет собой дальнейшее развитие алгоритма обратного вывода. Главными достоинствами алгоритма QSQ являются обеспечение доступа к минимальному числу фактов из БД, необходимых для ответа на запрос, и, соответственно, минимальное количество обращений к БД. Рассмотрим этот алгоритм более подробно.

При получении ответа на искомый запрос, а именно при обработке правил программы, в ней могут встретиться предикаты, требующие уточнения. Они могут не только содержаться в БД, но и быть выведены из правил программы. Для объяснения способа вычисления таких предикатов введем понятие подзапроса.

Подзапрос — запрос для конкретизации требующего уточнения предиката в теле правила, заключающийся в запуске QSQ для данно-

го предиката. Это определение применяется рекурсивно: подзапрос для подзапроса тоже является подзапросом.

Алгоритм QSQ использует два множества: P — множество ответов и Q — множество текущих подзапросов. Множество P содержит ответы как на главный запрос, так и на промежуточные подзапросы, а множество Q — все рассматриваемые или уже рассмотренные к данному моменту запросы и подзапросы. На каждом шаге алгоритм QSQ хранит состояние, представленное парой множеств: P и Q. Алгоритм генерирует новые ответы и новые подзапросы, подлежащие ответу. Алгоритм QSQ может быть реализован в двух вариантах — итеративном и рекурсивном.

Рекурсивный QSQ (QSQR) отдает предпочтение обработке подзапросов. При обнаружении нового подзапроса алгоритм рекурсивно запускает этот подзапрос, а процесс отыскания ответов откладывается до того момента, к которому данный подзапрос будет уже полностью обработан.

Итеративный QSQ (QSQI) отдает предпочтение продуцированию ответов. При обнаружении нового подзапроса его выполнение откладывается до тех пор, пока не будут порождены все возможные факты, которые могли бы быть получены к данному моменту.

Далее будет рассмотрен QSQR, у которого на прикладных задачах результат лучше, чем у QSQI. Для каждого правила, голова которого соответствует начальной цели, процесс вычисления начинается при пустом P и Q, содержащем только начальный запрос. На каждом шаге рекурсии множества P и Q дополняются.

Процесс завершается, когда не порождаются новые подзапросы и новые ответы. На момент окончания работы алгоритма множество P будет содержать минимально необходимые факты, требуемые для получения ответа на запрос. В свою очередь ответ может быть получен путем фильтрации множества P по запросу. Более подробное изложение данного алгоритма можно найти в [7].

Как было отмечено выше, QSQ требует доступ только к минимально необходимому числу фактов из БД и порождает только те факты, которые требуются для вывода ответа на запрос. Все это позволяет успешно применять QSQ при вычислении программ на Datalog, работающих с большими объемами данных.

Прототип дедуктивной СУБД. ***Аппаратное и программное обеспечение.*** Разработка и тестирование прототипа СУБД были осуществлены на персональном компьютере с процессором Intel Core i7-3770K (3,5 ГГц) и 16 Gb RAM под управлением операционной системы Windows 7. Кроме того, были использованы компилятор JavaSE-1.7 и библиотека sqLite JDBC 3.7.2 (для применения вспомогательной БД).

Особенности реализации алгоритма QSQ. Для реализации на языке Java авторы модифицировали алгоритм QSQ следующим образом.

Пусть алгоритм QSQ принимает на вход запрос *request* (в дальнейшем будем употреблять обозначение *request* как для запроса, так и для подзапроса). Если $request \notin Q$, то в множество P добавляются все факты БД, унифицируемые с *request*. Затем обрабатываются все правила, голова которых унифицируема с *request*. Процесс обработки правил повторяется до тех пор, пока не перестанет изменяться множество *P*.

Процесс обработки правила состоит в следующем. Сначала порождается список подстановок $list_0$, состоящий из единственной пустой подстановки. Затем последовательно обрабатываются все предикаты правила.

Возьмем предикат $pred_i$ и применим к нему каждую из подстановок списка $list_{i-1}$. В результате получим множество предикатов $temp_i$. Найдем множество фактов, унифицируемых с хотя бы одним предикатом из $temp_i$. В результате получаем множество ответов ans_i. Сопоставим каждый факт из ans_i с $pred_i$ и получим список подстановок $list_i$. Затем таким же образом обработаем следующий предикат правила $pred_{i+1}$, и так до тех пор, пока не будут обработаны все предикаты этого правила. Схема данного процесса показана на рис. 2.

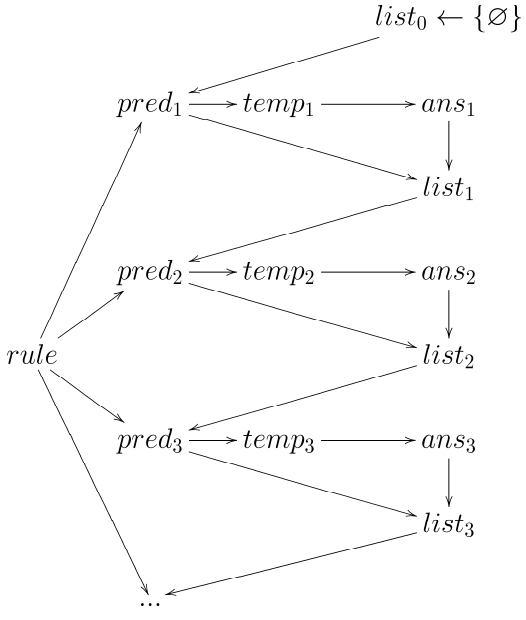

Рис. 2. Обработка предикатов правила

Пусть *list* — список подстановок, полученный при обработке последнего предиката. Необходимо применить каждую подстановку из *list* к голове правила. Получим результат обработки правила, который необходимо добавить к множеству *P*.

Во время обработки правила происходит вызов подзапросов. В процессе формирования ans_i из $temp_i$ возникает проблема в определении всех унифицируемых фактов для предикатов, требующих уточнения. Именно для этих предикатов и запускается подзапрос.

В практической реализации алгоритма QSQ важным является порядок обработки предикатов тела правила. Оптимальной считается обработка предикатов тела правила в порядке убывания количества констант [7], поэтому авторы выполняли сортировку предикатов тела каждого правила перед вычислением.

Для того чтобы рассмотреть реализацию алгоритма QSQ в деталях, введем некоторые дополнительные понятия.

Подстановку θ_1 будем называть более общей по отношению к θ_2 тогда и только тогда, когда существует подстановка θ, такая что $\theta_1 \cdot \theta = \theta_2$. Наиболее общим унификатором двух предикатов будем называть более общий по отношению ко всем остальным унификатор.

В алгоритме QSQ выполняется проверка двух предикатов на унифицируемость. Данную задачу мы решаем поиском наиболее общего унификатора с помощью алгоритма *produceMostGeneralUnifier* [7], представленного ниже. В этом алгоритме функция $size(p)$ возвращает количество аргументов в предикате *p*, функция $name(p)$ — имя предиката *p*, а p_i обозначает *i*-й аргумент *p*. При успешном выполнении возвращается наиболее общий унификатор, в противном случае — *null*. Данный алгоритм представлен на рис. 3.

В некоторых случаях в ходе выполнения этого алгоритма может возникнуть ситуация, когда переменная в составе правила (например, переменная X в правиле $p(X,Y) :- P(Y,X)$) и переменная с тем же именем в предикате (например, переменная X в факте $p(a,X)$) имеют различное значение. Во избежание ошибок мы выполняем переименование всех переменных следующим образом: если переменная уже встречалась ранее в текущем предикате, правиле или запросе, ей назначается то же имя, что и при предыдущем появлении переменной с таким же именем. В противном случае для переменной назначается имя, отличное от всех ранее употреблявшихся.

В алгоритме QSQ используются упомянутые выше множества P (множество ответов) и Q (множество текущих подзапросов), а также функции, описанные ниже.

```
PRODUCEMOSTGENERALUNIFIER(p, p')
 1  if NAME(p) ≠ NAME(p') or SIZE(p) ≠ SIZE(p')
 2          return null
 3  else
 4          θ ← ∅; Инициализируем θ пустой подстановкой
 5          for i ← 1 to SIZE(p)
 6                  if p_i · θ ≠ p'_i · θ
 7                          if p_i · θ является переменной
 8                                  θ ← θ ∪ {p_i/p'_i}
 9                          else if p'_i · θ является переменной
10                                  θ ← θ ∪ {p'_i/p_i}
11                          else
12                                  return null
13          return θ
```

Рис. 3. Алгоритм поиска наиболее общего унификатора

Функция *select*(*predicate*) является аналогом операции выборки σ из реляционной алгебры, которой в языке SQL соответствует операции *SELECT* [8]. Это обстоятельство позволило применить СУБД sqLite для временного хранения фактов. Данная функция возвращает множество фактов из БД, унифицируемых с предикатом *predicate*.

Функция *getRules*(*predicate*) для головы каждого из правил и *predicate* ищет унификатор θ и возвращает все правила, для которых θ существует, предварительно применяя θ как к голове, так и к телу правила.

Функция *next*(*body*) возвращает предикат из списка предикатов *body*. Первый вызов функции возвращает первый предикат списка, второй вызов — второй предикат списка и т. д.

Функция *processRule*(*rule*) производит обработку правила *rule*. Псевдокод данной функции приведен на рис. 4, а на рис. 5 — псевдокод алгоритма QSQ с использованием перечисленных функций.

Рассмотрим этапы выполнения алгоритма при вычислении следующей программы:

$$p(X,Y) :- p(X,Z), p(Z,Y).$$

$$? :- p(a,X).$$

```
processRule(Request)
 1  head, body ← rule
 2  list ← {∅}
 3  while p ← next(body)
 4          list′ ← ∅
 5          for each θ ∈ list
 6                  if предикат p требует уточнения
 7                          QSQ(p)
 8                          A ← {x ∈ P | x унифицируем с p}
 9                  else
10                          A ← select(p · θ)
11                  list′ ← {list′} ∪ {θ′ | ∀p′ ∈ A: p · θ′ = p′}
12          list ← list′
13  P ← head · list
```

Рис. 4. Функция обработки правила

```
QSQ(Request)
1  if Query ∉ Q
2          Q ← Q ∪ {Request}
3          P ← select(Request)
4          P′ ← P
5          repeat
6                  for each rule ∈ getRules(Request)
7                          processRule(rule)
8          until P ≠ P′
```

Рис. 5. Псевдокод реализации алгоритма QSQ

Сначала множества Q и P — пустые. Далее вызывается функция QSQ для запроса $p(a,X)$. Так как $p(a,X) \notin Q$, начинается выполнение алгоритма. Сначала в Q добавляется запрос, а в P — результаты, уже хранящиеся в БД: $Q = \{p(a,X)\}$, $P = \{p(a,b)\}$. Функция *getRules* возвращает единственное правило: $p(a,Y) :- p(a,Z), p(Z,Y)$. Начинается обработка правила.

Рассматриваем предикат $p(a,Y)$. Данный предикат требует уточнения, поэтому для него вызывается алгоритм QSQ, который сразу же завершается, поскольку в Q уже содержится такой предикат. Следовательно, Q и P — не изменились. Теперь, получив все факты

из P, унифицируемые с $p(a,Y)$: $\{p(a,b)\}$, получаем подстановку $\theta' = \{Y / b\}$. Обработка предиката $p(a,Y)$ завершена, $list = \{\{Y / b\}\}$.

Рассматриваем предикат $p(Y,Z)$. Применяя к нему подстановку из *list*, получаем предикат $p(b,Z)$. Данный предикат требует уточнения, поэтому для него вызывается алгоритм QSQ. После выполнения алгоритма имеем: $Q = \{p(a,X), p(b,Z)\}$, $P = \{p(a,b), p(b,c)\}$. Теперь, получив все факты из P, унифицируемые с $p(b,Z)$: $\{p(b,c)\}$, получаем подстановку $\theta' = \{Y / b, Z / c\}$. Обработка предиката окончена, $list = \{\{Y / b, Z / c\}\}$.

Обработка всех предикатов завершена. Применив подстановку из *list* к голове правила, получаем множество $p(a,c)$, являющееся ответом на запрос.

Интерфейс программирования приложений. Прототип СУБД с языком запросов Datalog был реализован в виде библиотеки на языке Java. Для работы с этой библиотекой предоставляется API, который включает набор классов, реализующих базовые структуры языка Datalog (аргумент предиката, предикат, факт, правило, запрос), набор классов для работы с программой на языке Datalog (включая контейнер для трех множеств — запросов, множество правил и фактов) и собственно интерпретатор языка Datalog, включающий реализацию рекурсивного алгоритма QSQ.

Результаты тестов. Предварительные тесты показали, что полученная реализация дедуктивной СУБД обладает приемлемой производительностью только в случаях, не порождающих глубокую рекурсию. Тест с глубокой рекурсией, заключающийся в построении транзитивного замыкания множества вида: $p(a_1,a_2), p(a_2,a_3), \ldots, p(a_{n-1},a_n)$ при n порядка 1000, требует времени, которое в рамках прикладного применения можно считать бесконечным. Но при более простых запросах, не подразумевающих глубокую рекурсию, алгоритм показал приемлемое время работы, не превышающее 1 с на БД, содержащей порядка 100 000 фактов.

Исходя из этих результатов, мы считаем нецелесообразным использование Datalog в качестве основного языка запросов для работы с БД больших объемов. Но, на наш взгляд, применение Datalog удобно и оправдано как для запросов, не использующих глубокую рекурсию, так и для более сложных запросов к выборкам, предварительно сделанным из реляционных БД традиционным способом.

Заключение. Было показано, что Datalog вряд ли может быть рекомендован в качестве основного языка для работы с БД. В то же время его использование целесообразно для некоторых типов запросов. Таким образом, развитие дедуктивных СУБД с языком запросов

Datalog, на наш взгляд, является перспективным, хотя их применение будет ограничено специальными случаями.

Кроме того, следует подчеркнуть, что реализация алгоритма QSQR и последующее использование API на императивном объектно-ориентированном языке (Java) не вполне удобно и не позволяет написать краткий и наглядный код. В то же время применение в алгоритме рекурсии, конструкций *for...each* и реализация отдельных частей в виде функций делает удобным его использование на функциональных языках программирования. Декларативность языка Datalog создает благоприятные условия для его реализации в виде предметно-ориентированного языка в рамках таких языков, как диалекты Lisp. В пользу этого утверждения говорит доступность такой реализации [6].

Исходный код проекта доступен по адресу: https://bitbucket.org/art_vybor/datalogdb

Авторы не ограничивают его использование в академических целях.

ЛИТЕРАТУРА

[1] Huang S.S., Green T.J., Loo B.T. Datalog and Emerging Applications: An Interactive Tutorial. Athens, *ACM SIGMOD'11*, 2011, june 12–16.

[2] Дейт К.Дж. *Введение в системы баз данных.* Птицина К.А., ред. 8-е изд. Москва, Вильямс, 2005.

[3] Karvounarakis G. Dept. of Computer and Information Science. *Datalog: Encyclopedia of Database Systems*. Philadelphia, University of Pennsylvania.

[4] Racket documentation. *Datalog: Deductive Database Programming*. URL: http://docs.racket-lang.org/datalog.

[5] IRIS Reasoner. URL: http://iris-reasoner.org/foundations.

[6] Małuszyński J., Szałas A. Partiality and Inconsistency in Agents' Belief Bases. *Proceedings of KES-AMSTA 2013*. *Frontiers of Artificial Intelligence and Applications.* Amsterdam, IOS Press, 2013, vol. 252, p. 3–17.

[7] Чери С., Готлоб Г., Танка Л. *Логическое программирование и базы данных.* Калиниченко Л.А., ред. Москва, Мир, 1992.

[8] Гарсиа-Молина Г., Ульман Д., Уидом Д. *Системы баз данных.* Варакина А.С., ред. Москва, Вильямс, 2003.

Статья поступила в редакцию 26.06.2013

Ссылку на эту статью просим оформлять следующим образом:

Выборнов А.И., Дубанов А.В. Алгоритмы обработки запросов к дедуктивным базам данных и реализация алгоритма QSQ. *Инженерный журнал: наука и инновации,* 2013, вып. 11. URL: http://engjournal.ru/catalog/it/hidden/1059.html

Выборнов Артем Игоревич родился в 1993 г. Студент 4-го курса кафедры «Теоретическая информатика и компьютерные технологии» МГТУ им. Н.Э. Баумана. Специализируется на разработке компиляторов и программировании баз данных. e-mail: art-vybor@ya.ru

Дубанов Александр Вячеславович родился в 1975 г., окончил Московскую медицинскую академию им. И.М. Сеченова в 1998 г. Канд. биол. наук, доцент кафедры «Теоретическая информатика и компьютерные технологии» МГТУ им. Н.Э. Баумана. Автор 20 научных работ в области применения вычислительных методов и разработки программного обеспечения для медико-биологических исследований. e-mail: qrcs@mail.ru