DOI: 10.25586/RNU.V9187.20.02.P.051 УДК 004.4'4

Е.А. Буйволов

ПРИМЕНЕНИЕ DYNAMIC SQL ПРИ РЕФАКТОРИНГЕ ХРАНИМЫХ ПРОЦЕДУР СУБД SYBASE В КРУПНОЙ СТРАХОВОЙ КОМПАНИИ

Разработана методология оптимизации рефакторинга хранимых процедур, содержащих в себе логику проведения медико-экономического контроля по случаям лечения граждан за период времени. Приведены примеры упрощения кода, уменьшения объема кода, использования динамических входных данных и вывода. Проведен анализ упрощения поддержки кода и оптимизации времени выполнения медико-экономического контроля.
Ключевые слова: Dynamic SQL, рефакторинг, автоматизация, парсинг, методология, транслятор, хранимые процедуры.

E.A. Buivolov

DYNAMIC SQL APPLICATION TO REFACTOR DBMS SYBASE STORED PROCEDURES WITHIN AN INSURANCE COMPANY

This article reviewing building of refactoring optimization methodology, created in process of medical and economical control of treatment for period. This article includes example of code simplifying and decreasing, using dynamic input and output data. Analysis of simplify the code support and optimization of medical and economical control lead-time are given as result.
Keywords: Dynamic SQL, refactoring, automation, parsing, methodology, translator, stored procedures.

Введение

Несмотря на то что технология Dynamic SQL не является новой и не является де-факто стандартом работы с данными в реляционных БД, в настоящий момент существует большое число компаний, в которых на первых стадиях рефакторинга и переноса логики данный инструмент может стать оптимальным решением. Критерием выбора Dynamic SQL может послужить использование хранимых процедур для обработки данных и формирования выборок, отсутствие ORM (Object-Relational Mapping) и решений с использованием внешних языков программирования. В компаниях, не вкладывающихся в разработку, нередки ситуации, при которых все силы небольшого IT-отдела направлены на поддержание функционирования базы данных и хранимых процедур. При использовании устаревших СУБД поддержка такого кода весьма сложна. Наем новой команды разработчиков не всегда возможен, но разработка решения на внешнем языке программирования в любом случае необходима. В настоящей статье приводится описание эффективных техник при рефакторинге хранимых процедур, а также создании формального языка, описывающего бизнес-процессы внутри компании с использованием Dynamic SQL, при задействовании лишь ресурсов СУБД. Работа выполнена с использованием СУБД Sybase ASA 9.02, которая поддерживает два SQL-диалекта: WatcomSQL и T-SQL.

© Буйволов Е.А., 2020

При анализе литературы на данную тему стоит отметить статью [10], которая обращается к основным трудам, связанным с трансляцией SQL в реляционную алгебру, в статье приведен пример пре-процессинга SQL-запросов, введена система трансляции запросов, которая помогает создать объектно ориентированную модель запроса. Было использовано средство для построения компиляторов YACC (Yes Another Compiler Compiler) [1] для генерации парсера, преобразовывающего входной SQL-запрос в синтаксическое дерево на основе SQL-грамматики. Результатом работы стала система конструкций в ООП (объектно ориентированное программирование) стиле, имплементирующая SQL запросы. В статье [7] приведена простая идея трансляции естественного языка в SQL-запросы. Стоит отметить, что разработка трансляторов и лингвистических моделей стала развиваться практически сразу после введения реляционной модели Кодда и языка SQL, но все еще находит применение в современных IT-решениях. Например, ORM-фрэймворк Hibernate оперирует ООП-сущностями, но позволяет обращаться к БД посредством трансляции логики в реляционную модель [4]. В статье [5] рассматривается создание языка реляционной алгебры и реляционного исчисления ЯРАИ, описаны используемые надстройки транслятора и обоснована причина использование транслятора в поставленной задаче. В статье [8] приводится идея расширения реляционной алгебры рекурсивными структурами, иначе говоря, каскада, который, стоит отметить, применяется в фрэймворке Hibernate. Статья схожа с предыдущими и демонстрирует, что SQL и реляционная модель предоставляют достаточную функциональность для расширения алгебры и создания трансляторов и надязыков, способных покрывать необходимые задачи. В [6] приведен обзор методов лексической оптимизации запросов, а в [3] пример динамической компиляции выражений в SQL-запросах для PostgreSQL с использованием SeqScan и LLVM.

При рефакторинге хранимых процедур можно обратиться к любой классической книге по рефакторингу БД, например к [9].

Целью настоящей работы является выработка методологии эффективного рефакторинга с использованием Dynamic SQL.

Задачами данной работы являются: повышение гибкости и качества кода, удаление дублирования и ошибок, уточнение бизнес-процессов, подготовка проекта к переносу на внешний ЯП, создание формального языка над SQL, определяющего бизнес-процессы компании.

Проблематика

Процедура МЭК в страховой компании в сфере обязательного медицинского страхования (ОМС) направлена на выявление случаев лечения граждан, противоречащих нормативным документам – ГТС (генеральное тарифное соглашение), предоставляемому территориальным фондом ОМС (ТФОМС). Обработка представляет покрытие тестовых случаев блоками SQL-кода, буферизацию выборок по определенным критериям в результате последовательного выполнения хранимых процедур и дальнейший анализ результатов. Суммарный объем исходных данных составляет около 50 млн кортежей, состоящих из сотен атрибутов. Объем выходных данных составляет около 15 тысяч кортежей, состоящих из менее чем 10 атрибутов.

Из проблем, влияющих на объем кода и сложность поддержки, стоит выделить следующее:

1. При работе с изменяющимся техническим заданием (нормативными документами) необходимо редактирование входных параметров, что приводит к дублированию SQL-запросов для разных входных параметров.

2. Установление соответствия между нормативными документами и проводимым анализом может распространяться не только на буферный период тестирования, но и на предыдущие временные промежутки с соблюдением предыдущего технического задания, соответствующего времени проведения лечения, что также приводит к дублированию запросов для различных исходных таблиц.

3. При возникновении исключительных случаев необходимо предусмотреть подобный случай во всех повторяющихся конструкциях, что непременно влечет за собой ошибки и, как следствие, финансовые потери компании.

4. При недостатке кадров работа может привести к стрессу, снижению производительности труда, к снижению качества кода, к увеличению сложности поддержки.

5. Для покрытия одного тестового случая необходимо составить SQL-запрос объемом примерно от 10 до 100 строк кода. Каждый год нормативные документы вмещают в себя по примерной маркировке более четырехсот тестовых случаев [2]. Таким образом, суммарный объем актуального кода (за последние три года) в репозитории на момент старта рефакторинга составлял чуть более 35 тысяч строк кода, а время выполнения хранимых процедур составляло около четырех суток.

Предлагаемое решение

На основании проведенного анализа можно выделить следующие основные проблемы:

1. Логика подвержена изменению во времени и связана с данными своего временного промежутка.

2. Необходима возможность гибкого изменения значений и количества входных параметров.

3. Необходима централизация точек входа и выхода данных, а также централизация логики повторяющихся конструкций.

Из вышесказанного можно сделать вывод о том, что в компаниях, наращивающих объемы данных и имеющих изменчивое техническое задание (ТЗ), использование хранимых процедур и реализация логики в SQL-запросах является неверным решением, так как такой подход ведет к увеличению времени обработки данных, ошибкам и сложности сопровождения. Одним из вариантов решения может быть создание приложения на языках программирования, обеспечивающих простоту связывания логики и данных, функциональный подход, тестирование функциональности. Для решения поставленной задачи был выбран язык программирования Java и микросервисная архитектура с использованием Spring и Docker/GitLab для CI/CD.

На рисунке 1 представлено возможное отображение конструктора запросов на frontend, в то время когда сам запрос будет собираться в backend из элементов Dynamic SQL.

В случае небольшой команды было принято решение использовать Dynamic SQL для внедрения связывания логики и данных и функционального стиля на стадии рефакторинга хранимых процедур. В процессе рефакторинга повторяющиеся конструкции были вынесены в логические элементы: элементы конструктора SQL. Таким образом, в рамках данной работы был создан собственный, декорирующий SQL, формальный язык внутри языка, имплементирующий бизнес-логику.

Рис. 1. Пример конструктора SQL-запроса в графическом интерфейсе

Алгоритм

Для достижения поставленной цели решено группировать общий по назначению код, используя процедуры-декораторы, позволяющие передавать входные (in) и выходные (out) параметры, таким образом, алгоритм динамического формирования SQL в первой итерации можно свести к следующему:

1. Объявить переменную @query, которая будет принимать Watcom SQL-код для выполнения.
2. Объявить все дополнительные переменные, содержащие исходные данные для задачи.
3. Сформировать запрос, основываясь на бизнес-логике и техническом задании, следующим образом:

a) начать запрос с ключевой фразы where или join;
b) добавить все необходимые ограничения по проверяемым полям;
c) выполнить отказ неверного лечения путем выполнения сформированного запроса.

Во второй итерации входные данные желательно сделать внешними (например, использовать таблицы, которые могут динамически формироваться в рамках Java-сервиса):

1. Объявить курсор и записать в него данные, с которыми будет проведена работа.
2. Повторить первую часть алгоритма для каждой итерации курсора.

Для динамического формирования SQL-запроса были разработаны конструкции – элементы SQL-конструктора (в таблице 1 приведена часть разработанных конструкций).

Таблица 1

Элементы конструктора SQL

`and (inout @query text)`	Добавляет к исходной строке логическое «и»

`filter(inout @query text, in @column_name text, in @condition text)`	Добавляет к исходной строке условия по определенной колонке. Например, вызов
	`call filter('', 'profile', '78, like 8%, not 56, not7, ^r, @profile_table(pr; rec:п1)')`
	
	вернет
	`(exists (select 1 from profile_table where profile = 'pr' and rec in ('п1')) and profile like '8%' or profile in ('r') and profile not in ('56','7') and profile in ('78'))`

`join(inout @query text, in @table_name text)`	Используется в начале составления запроса вместо where для добавления ключевого слова «join» в запрос
`not(inout @query text)`	Добавляет к исходной строке логическое «не»
`or(inout @query text)`	Добавляет к исходной строке логическое «или»

Окончание табл. 1

`period_amount_greater(inout @query text, in @quantity int, in @test_table text, in @keys_values_condition text)`	Логическая проверка на наличие услуги с определенными параметрами чаще, чем разрешено в указанный период (например, 1 услуга в день). Осуществляется через подзапрос с проверкой @keys_values_condition в конструкции filter.
`profile(inout @query text, in @profile_list text, in @sign text default null, in @dateout text default null)`	Добавляет к исходной строке in-условие для профилей лечения
`where(inout @query text)`	Используется в начале составления запроса для добавления ключевого слова «where» в запрос
`refuse(in @logical text, in @period text, in @marker varchar (50), in @refuse_code varchar (5), in @remark text)`	Процедура производит конкатенацию строк и дальнейшее выполнение запроса. Отвечает за маршрутизацию результатов
`subquery(inout @query text, in @subquery text)`	Добавляет к исходной строке подзапрос, на который распространяются все вышеописанные конструкции (с переменной @subquery можно работать так же, как и с @query)

Также были разработаны дополнительные процедуры работы со строкой (табл. 2).

Таблица 2

Дополнительные процедуры для работы со строкой

`parse_simple_condition(inout @out_condition text, in @column_name text, inout @in_condition text)`	Связывает колонку и простые условия типа not, ^, like
`parse_table_condition(inout @out_condition text, in @column_name text, inout @in_condition text)`	Рекурсивная процедура. Связывает таблицу и колонку, добавляя условия внутрь подзапроса по таблице через привязку по спец. символу @. Рекурсивно разрезает строку на левую и правую часть, и условие

`split_condition_to_out_vars(`
`in @correlation_name text,`
`in @test_table text,`
`in @keys_values_condition text,`
`out @profile_condition text,`
`out @reciever_condition text,`
`out @period_condition text,`
`out @diagnosis_condition text,`
`out @case_condition text,`
`out @vid_condition text,`
`out @history_condition text,`
`out @refuse_compare_condition text,`
`out @begin_condition text,`
`out @end_condition text,`
`out @direction_condition text,`
`out @form_code_condition text,`
`out @serv_card_type_condition text,`
`out @serv_prvs79_condition text,`
`out @cross_condition text,`
`out @ref_condition text)`
	Разбивает все существующие условия для подстановки в тело запроса (актуальные условия возвращаются через out-параметры)

Рассмотрим подробнее алгоритм работы конструкции filter (рис. 2), позволяющей эффективно покрывать все необходимые ограничения любой сложности и уровня вложенности.

```
set @out_condition  = '';
// Парсинг сложного табличного выражения, если используется ключевой символ @.
call parse_table_condition(@out_condition, @column_name, @condition);
if (Проверка на null) then
// Парсинг всех простых условий.
    call parse_simple_condition(@out_condition, @column_name, @condition);
end if;

Конкатенация вывода.
```

Рис. 2. Алгоритм процедуры filter. Псевдокод

Как видно из кода, процедура вызывает две вспомогательные процедуры, которые последовательно обрабатывают входную строку и генерируют необходимые условия проверки. Для этого рассмотрим код каждой из процедур (рис. 3–4).

```
Ицинициализация кэш-таблицы.
Цикл

       Обнуление индексов разреза строки
       Ставим срез с позиции символа @;
       if (Строка начинается с 'not @') then
         Определяем значения разреза строки;
         Ставим флаг на условие отрицания;
       Else
         Определяем значения разреза строки;
       end if;

       if (Символ @ присутствует) then
         Урезаем строку до символа после @;
         if (В подстроке есть '(') then
            Обрезаем подстроку до закрывающей скобки;
            Извлекаем название таблицы;
            В обрабатываемое условие срезаем все, что относится к условии в скобках
       после названия таблицы;

            while (Обрабатываемое условие содержит @) LOOP
                  Буферизуем предыдущие значения за счет записи в кэш-таблицу;
                  call s_parse_table_condition(буфер, ключ, обрабатываемое условие);
            END LOOP;

            Цикл по буферу, подставляем простое условие;
                  call s_parse_simple_condition(буфер, ключ, условие из буфера);
            END LOOP;
            Конкатенация результата в подзапрос типа select * from имя_таблицы where
       сумма условий из буфера;
         else
            Конкатенация результата в подзапрос типа select * from имя_таблицы;
         end if;

         Склейка исходной строки обратно без учета обработанного;
       else
            Покидаем цикл;
       end if
Конец цикла;
```

Рис. 3. Алгоритм parse_table_condition. Псевдокод

```
Инициализация входных значений через разбиение на строки через ‘;’ и на key/value с
помощью ‘:’;
Инициализация буферов условий (in, not in, like, not like);

Цикл по входным значениям:

    Проверка на null
        if (Содержится '%^%') then
            Меняем переменную оператора на ‘or’;
            Заменяем символ на пустой символ;
        else
            Иначе переменная оператора = ‘and’;
        end if;
        if (Содержится '%not%') then
            Ставим флаг отрицания;
            Заменяем not на пустой символ;
        else
            Снимаем флаг отрицания;
        end if;
        if (Содержится '%like%') then
            Меняем переменную оператора на ‘like’;
            Заменяем like на пустой символ;
            Собираем запрос для текущей итерации;
        else
            Меняем переменную оператора на ‘in’;
            if (Содержится '%@%') then
                Ничего не делаем;
            else
                if (Переменная оператора = ' and ') then
                    if (Флаг отрицания снят) then
                        Вставляем значение в буферную in таблицу;
                    else
                        Вставляем значение в буферную not in таблицу;
                    end if;
                else
                   Собираем запрос для текущей итерации;
                end if;
            end if;
        end if;
Конец цикла;

Конкатенация вывода с учетом буферных таблиц и собранных запросов;
```

Рис. 4. Алгоритм parse_simple_condition. Псевдокод

Практический результат

На примере алгоритма возьмем исходный код (рис. 5).

Этому же коду соответствует случай «3 раза в день», в котором меняется только проверка на amount и количество карт подзапроса. Как можно заметить, такой код содержит ряд проблем:

1. Повторяющиеся блоки и дублирование кода.
2. Фиксированные таблицы для выборки.
3. Фиксированный вывод данных.
4. Возможность для разной реализации и отсутствие формальной системы.

```
1   insert into #буфер
2   select … поля …
3   from исходная_таблица a
4   where (profile in (… сет профилей …)
5      or (profile in (… сет профилей …)
6     and a.dateout >= '2019-01-01')
7       )
8     and GET_SUM_N (a.ID_service) < a.sum
9     and dateout between @min_dateout and @d_end
10    and not exists (select 1 from вывод where id_object = id_service)
11    and not exists (select 1 from вывод
12      join исходная_таблица z on id_object = z.id_service
13      where a.id_case = z.id_case
14    and ref_type = '71э');
15
16  insert into вывод
17  select id_account,                                      -- id счета
18         id_service,                                      -- id услуги
19         sum - GET_SUM_N(ID_service),                     -- сумма отказа
20         '71э',                                           -- тип отказа
21         'Услуга предоставляется не более 2 раз в день', -- примечание
22         '2zd n!'                                         -- маркировка теста
23  from #буфер a
24  where amount > 2
25    and ctrldate between @d_beg and @d_end;
26  commit;
27
28  insert into вывод
29  select …дублирование полей выше…
30  from #буфер a
31  where a.ctrldate between @d_beg and @d_end
32    and (select sum (amount) amount from #буфер b where b.surname = a.surname
33    and b.name1 = a.name1
34    and b.birthday = a.birthday
35    and b.profile = a.profile
36    and b.reciever = a.reciever
37    and b.dateout = a.dateout
38    and (b.ctrldate < a.ctrldate or (b.ctrldate = a.ctrldate
39    and b.cod_z < a.cod_z))) + a.amount > 2;
40  commit;
```

Рис. 5. Исходный код Watcom SQL

Применяя описанный выше алгоритм, получим элегантное решение проблемы (рис. 6).

```
1   set @subquery = '';
2   call profile(@subquery, 'сет профилей');
3   call or(@subquery);
4   call profile(@subquery, 'доп. сет профилей', '>=', '20190101');
5
6   set @query = '';
7   call where(@query);
8   call subquery(@query, @subquery);
9   call and(@query);
10  call period_amount_greater(@query, 'day', 2);
11  call refuse(@query, 'Тестовый период', 'Марк. Количество в день', 'Код', 'Прим.');
```

Рис. 6. Код Watcom SQL, подверженный рефакторингу с помощью Dynamic SQL

К плюсам результата, продемонстрированного на рисунке 6, можно отнести:

1. Контролируемый вход и выход (тестовый период).
2. Относительная простота дополнения кода.
3. Повторное использование компонентов и отсутствие дублирования.
4. Вынесение кода в логические блоки и упрощение понимания бизнес-логики.
5. Простота переноса конструкций на любой язык программирования.

К минусам можно отнести:

1. Сложность подхода.
2. Написание дополнительных функций для преобразования строки.

Однако при выполнении проверки «3 раза в день» необходимо скопировать существующий код. Но в тесте, направленном на проверку количества услуг в день, можно обнаружить зависимость между профилями лечения и количеством дней, таким образом определив входные параметры. Теперь вынесем все тестовые данные во внешнюю таблицу. В рассматриваемом случае все входные данные были помещены в форму web-приложения и записаны в БД с дальнейшей возможностью редактирования пользователем через web. Реализуя вторую итерацию алгоритма, получим код на рисунке 7.

```
1   open cursor_amount with hold;
2   call benchmark_start(@marker);
3   z_amount:
4       LOOP
5   fetch next cursor_amount
6       into @this_id, @this_prof, @this_amount, @this_condition, @this_mee;
7   if SQLCODE <> 0 then
8       leave z_amount;
9   end if;
10
11  set @marker = 'Кратность. Строка №';
12  set @refuse_code = '71э';
13  set @remark = 'Услуга предоставляется не более ' || @this_amount || ' раза в ';
14  if (@this_condition like '%day%') then
15      set @remark = @remark || 'день';
16  elseif (@this_condition like '%month%') then
17      set @remark = @remark || 'месяц';
18  elseif (@this_condition like '%year%') then
19      set @remark = @remark || 'год';
20  end if;
21
22  set @query = '';
23  call where(@query);
24  call profile(@query, @this_prof);
25  call and(@query);
26  call period_amount_greater(@query, @this_amount, @subquery_test_table,
27  @this_condition || '; profile: equals; compare: dateout');
28  if (@this_mee = 1) then
29      call medical(@query, @test_period, @marker || @this_id, @refuse_code, @remark);
30  else
31      call refuse(@query, @test_period, @marker || @this_id, @refuse_code, @remark);
32  end if;
```

Рис. 7. Код Watcom SQL, рефакторинг с использованием курсора и Dynamic SQL

По первому впечатлению кажется, что кода стало больше, однако данная конструкция покрывает сразу все случаи дублирования лечения в день, месяц и год (1, 2, 3, 4, 6, 11 и т.д.), уменьшая таким образом количество кода с 40*N*, где *N* – количество тестовых случаев до 32 строк. Это на практике позволяет уменьшить объем строчек для покрытия данного тестового случая на порядок.

К сожалению, сверить планы запросов невозможно из-за устаревшей СУБД (рис. 8). Однако путем замера времени выполнения были получены одинаковые результаты.

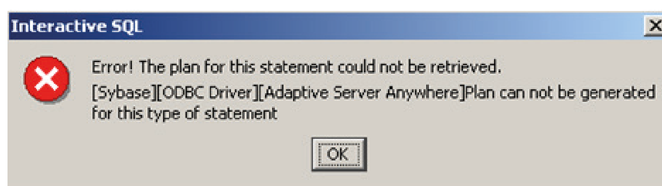

Рис. 8. Ошибка создания плана запроса для СУБД Sybase ASA 9.02

Методология

В рамках проведенной работы была выработана методология эффективного рефакторинга хранимых процедур в бизнес-модели страховой компании с использованием Dynamic SQL. В нее входит:

1. Формирование четкого ТЗ.
2. Обнаружение повторяющегося кода, если итоговый элемент обеспечивает покрытие всех тестовых случаев, вынесение элемента в отдельную Dynamic SQL-процедуру.
3. При создании элементов конструктора SQL необходимо предусмотреть возможность добавления входных параметров таким образом, чтобы не нарушать существующую инфраструктуру кода.
4. Необходимо отделять данные от логики.
5. Использование debug является единственно эффективным средством рефакторинга при работе с устаревшими СУБД.
6. Использование версионирования, git и регулярного code review с коллегами.

Использование DataGrip или других подобных инструментов как средство рефакторинга.

Заключение

В результате проделанной работы была разработана методология эффективного рефакторинга в экстремальных условиях. Был создан формальный язык, состоящий из Dynamic SQL-конструкций. В результате использования разработанной методологии в первый месяц рефакторинга одним сотрудником удалось централизовать выходные данные процедур, а в последующие три месяца описать всю оставшуюся необходимую логику и покрыть ей большую часть хранимых процедур. Таким образом, методология оказалось крайне эффективной, с учетом того, что до применения подобной методики объем кода только увеличивался с одновременным увеличением времени выполнения хранимых процедур.

Данный подход позволил сократить объем строк в 2,5–4,3 раза (в зависимости от бизнес-группы хранимой процедуры). Время выполнения сократилось с четырех суток до

двадцати двух часов, в основном благодаря нахождению ошибок: использование временных непроиндексированных таблиц, дублирование кода, выполняющего одну задачу, избыточность исходных данных и их разрозненность, уточнение ТЗ и оптимизация за счет уточнения выборок. За счет гибкости и централизации точек входа и выхода данных стало возможным проведение повторного МЭКа за различные промежутки времени. Проект подготовлен к переносу на Java и по оценке IT-отдела компании потребует значительно меньших усилий при сопровождении. Проект введен в эксплуатацию и активно используется, предоставляя возможность без найма сотрудников осуществлять разработку внешнего решения.

Литература

1. Введение в Lex & Yacc. URL: http://mech.math.msu.su/~vvb/FormLang/LexYacc/lexyacc.htm (дата обращения: 03.05.2020).
2. Генеральное тарифное соглашение ТФОМС. URL: https://spboms.ru/page/mo (дата обращения: 03.05.2020).
3. Динамическая компиляция выражений в SQL-запросах для СУБД PostgreSQL / Е.Ю. Шарыгин, Р.А. Бучацкий, Л.В. Скворцов, Р.А. Жуйков, Д.М. Мельник // Труды Института системного программирования РАН. 2016. № 28 (4). С. 217–240. doi.org/10.15514/ISPRAS-2016-28(4)-13.
4. *Карпова Т.С., Малышева С.Ю.* Расширение систем электронного тестирования на примере SQL-запросов // Интеллектуальные технологии на транспорте. 2019. № 1. С. 33–40.
5. *Малиёв А.А.* Разработка языка реляционной алгебры и реляционного исчисления и транслятора на язык SQL // Вестник Воронежского государственного технического университета. 2009. № 12. С. 154–155.
6. *Мендкович Н.А., Кузнецов С.Д.* Обзор развития методов лексической оптимизации запросов // Труды Института системного программирования РАН. 2012. Т. 23. С. 195–214.
7. *Сабинин О.Ю., Горбатов Н.В.* Разработка алгоритма трансляции предложений естественного языка в запросы на языке SQL // Теоретическая и прикладная наука. 2019. № 5. С. 414–418.
8. *Соколова В.В., Замятина О.М., Новосельцев В.Б.* Расширение реляционной алгебры рекурсивными структурами // Известия Томского политехнического университета. 2010. № 5. С. 163–165.
9. *Ambler S., Sadalage P.* Refactoring Databases, Addison-Wesley Professional, 2006. 384 p.
10. *Xu S., Hong M.* Translating SQL into Relational Algebra Treeusing Object-Oriented Thinking to Obtain Expression of Relational Algebra // International Journal of Engineering and Manufacturing. 2012. Vol. 2, Iss. 3. P. 53–62.

Literatura

1. Vvedenie v Lex & Yacc. URL: http://mech.math.msu.su/~vvb/FormLang/LexYacc/lexyacc.htm (data obrashcheniya: 03.05.2020).
2. General'noe tarifnoe soglashenie TFOMS. URL: https://spboms.ru/page/mo (data obrashcheniya: 03.05.2020).
3. Dinamicheskaya kompilyatsiya vyrazhenij v SQL-zaprosakh dlya SUBD PostgreSQL / E.Yu. Sharygin, R.A. Buchatskij, L.V. Skvortsov, R.A. Zhujkov, D.M. Mel'nik // Trudy Instituta sistemnogo programmirovaniya RAN. 2016. № 28 (4). S. 217–240. doi.org/10.15514/ISPRAS-2016-28(4)-13.

4. *Karpova T. S., Malysheva S. Yu.* Rasshirenie sistem elektronnogo testirovaniya na primere SQL-zaprosov // Intellektual'nye tekhnologii na transporte, 2019. № 1. S. 33-40.
5. *Maliyov A.A.* Razrabotka yazyka relyacionnoj algebry i relyacionnogo ischisleniya i translyatora na yazyk SQL // Vestnik Voronezhskogo gosudarstvennogo tekhnicheskogo universiteta, 2009. № 12. S. 154-155.
6. *Mendkovich N.A., Kuznecov S.D.* Obzor razvitiya metodov leksicheskoj optimizacii zaprosov // Trudy Instituta sistemnogo programmirovaniya RAN, 2012. T. 23. S. 195–214.
7. *Sabinin O.Yu., Gorbatov N.V.* Razrabotka algoritma translyacii predlozhenij estestvennogo yazyka v zaprosy na yazyke SQL // Teoreticheskaya i prikladnaya nauka. 2019. № 5. S. 414–418.
8. *Sokolova V.V., Zamyatina O.M., Novosel'cev V.B.* Rasshirenie relyacionnoj algebry rekursivnymi strukturami // Izvestiya Tomskogo politekhnicheskogo universiteta. 2010. № 5. S. 163–165.
9. *Ambler S., Sadalage P.* Refactoring Databases, Addison-Wesley Professional, 2006. 38 p.
10. *Xu S., Hong M.* Translating SQL into Relational Algebra Treeusing Object-Oriented Thinking to Obtain Expression of Relational Algebra // International Journal of Engineering and Manufacturing. 2012. Vol. 2, Iss. 3. P. 53–62.

DOI: 10.25586/RNU.V9187.20.02.P.062 УДК 004.023+656

С.Е. Вечерская, М.В. Нагорняк

ОСНОВНЫЕ ТРЕНДЫ ОРГАНИЗАЦИИ ТРАНСПОРТНОЙ ЛОГИСТИКИ В УСЛОВИЯХ НЕФИНАНСОВОГО КРИЗИСА

Проанализированы особенности управления транспортной логистикой в контексте и в зависимости от типа кризиса. Выявлены закономерности поведения отрасли, определяющие направленность выбора решений по оптимизации управления.
Ключевые слова: управление, оптимизация, логистика, кризис.

S.E. Vecherskaya, M.V. Nagornyak

MAIN TRENDS IN THE ORGANIZATION OF TRANSPORT LOGISTICS IN CONDITIONS OF NON-FINANCIAL CRISES

The features of transport logistics management in the context and depending on the type of crisis are analyzed. The patterns of industry behavior that determine the direction of the choice of solutions for optimizing management are identified.
Keywords: management, optimization, logistics, crisis.

Введение

Отрасль транспортной логистики относится к сфере неторгуемых товаров. Однако в отличие от большинства прочих неторгуемых товаров сферы услуг транспортную логи-

© Вечерская С.Е., Нагорняк М.В., 2020