Санкт-Петербургский Государственный Университет

Математико-механический факультет

Кафедра системного программирования

Автоматизированная трансформация динамических SQL запросов при реинжиниринге информационных систем

Магистерская диссертация

студента 661 группы

Григорьева Семёна Вячеславовича

Научный руководитель	.....	ст. преподаватель Я.А. Кириленко
	/подпись/	
Рецензент	.....	программист ООО "ИнтеллиДжей Лабс"
	/подпись/	А.А. Бреслав
“Допустить к защите”	.....	д.ф.-м.н., проф. А.Н. Терехов
заведующий кафедрой	/подпись/	

Санкт-Петербург

2012

SAINT PETERSBURG STATE UNIVERSITY

Mathematics & Mechanics Faculty

Software Engineering Chair

Automated transformation of dynamic SQL queries in information system reengineering

by

Grigoriev Semen Vjacheslavovich

Master's thesis

Supervisor	.....	Senior Lecturer J.A. Kirilenko
	/Signature/	
Reviewer	.....	developer at "JetBrains" A.A. Breslav
	/Signature/	
"Admitted to proof"	.....	Professor A.N. Terekhov
Head of Chair	/Signature/	

Saint Petersburg

2012

Содержание

1 Введение

Многие современные языки программирования позволяют обращаться к реляционным базам данных, динамически формируя запросы непосредственно во время выполнения программы. Кроме того, современные СУБД предоставляют возможность вызова динамического SQL, которая формально введена в стандарте SQL/92 [17]. Операторы динамического SQL, в отличие от операторов встроенного SQL, формируются не на этапе компиляции, а на этапе выполнения приложения как строковые выражения.

При реинжиниринге приложений, содержащих такие конструкции, необходимо помнить, что порождаемые строки – тоже код на некотором формальном языке программирования и его надо обрабатывать соответствующим образом. Например, при миграции хранимого SQL-кода, который содержит динамически формируемые конструкции, необходимо гарантировать, что после обработки кода хранимых процедур соответствующие значения будут вычисляться к корректным SQL операторам.

Задача осложняется тем, что, зачастую, при реинжиниринге информационных систем (ИС) производится переименование объектов (таблиц, колонок, процедур в случае SQL) и/или их удаление (удаление мёртвого кода, удаление неиспользуемых объектов). В этом случае надо гарантировать не только синтаксическую корректность результатов трансляции динамически формируемых конструкций, но и корректность по отношению к переименованию и удалению объектов.

Возможно, что существуют ИС, подлежащие ренижинирингу, но обладающие следующими важными с точки зрения процесса миграции свойствами. Во-первых, разработчики исходной системы практически не пользовались динамическим SQL в хранимых процедурах. Во-вторых, клиентские приложения созданы с использованием ORM-библиотек, поддерживающих и исходную БД, и целевую БД, или хотя бы не изобилуют SQL запросами. При реинжиниринге таких систем перевод (адаптация) динамических запросов не является ресурсоемкой задачей и может быть выполнена вручную, без автоматизации трансформации. Как вспомогательное средство контроля – для тестирования корректности формируемых динамических запросов – может быть использован один из инструментов, статически

проверяющий корректность запроса [3].

В реальных системах, с которыми приходилось сталкиваться авторам, дела обстоят намного сложнее. В данной работе задача трансляции динамически формируемых строк решалась в рамках проекта по миграции базы данных с MS-SQL Server 2005 [2] на Oracle 11gR2 [1]. Этот проект обладал следующими характеристиками: приблизительно 2,6 миллиона строк кода, более 3000 динамических запросов, из которых больше 60% могут принимать более одного значения. Кроме того, исходная ИС могла взаимодействовать с несколькими другими системами. Как следствие этого, части строк приходят извне и не могут быть вычислены статически, а часть передаётся через таблицы в базе. Например, таким образом организуется очередь, что тоже усложняет статический анализ.

Важной особенностью является то, что логика формирования динамического запроса может быть очень сложной. Например, в данном проекте, напечатанные полностью вычисленные значения динамических запросов – это около 6Гб текста (без значений, формируемых в циклах), при объёме исходного кода 300Мб.

Один из возможных подходов к решению такой задачи – статически вычислить новые значения для всех переменных, участвующих в формировании кода. Для этого необходимо статически вычислить все возможные значения для каждого запроса. Затем необходимо провести синтаксический разбор этого множества. В результате разбора получится множество деревьев – лес, над которым нужно провести необходимые преобразования и на основе полученной информации сформировать новые значения для переменных. Для примера рассмотрим следующий код на T-SQL [23].

```
....
IF @X = @Y
    SET @TABLE = '#table1'
ELSE
    SET @TABLE = 'table2'
SET @S = 'SELECT x FROM ' + @TABLE + '  WHERE ISNULL(n,0) > 1'
EXECUTE (@S)
....
```

В данном случае, для переменной `@S` статически вычислимы два значения:

1. `‘SELECT x FROM #table1  WHERE ISNULL(n,0) > 1’`
2. `‘SELECT x FROM table2  WHERE ISNULL(n,0) > 1’`

Результатом синтаксического разбора будет лес из двух деревьев, соответствующих этим значениям. Далее, при переводе данного кода на PL-SQL [20] необходимо провести следующие преобразования, касающиеся динамического запроса.

- Непосредственно трансляция запроса.
- Данный запрос возвращает набор данных (является SELECT-ом), по этому он должен быть преобразован в динамический курсор.
- Должны быть переименованы таблицы, участвующие в запросе, если это необходимо.

После трансляции мы должны вычислить новые значения для всех переменных, участвующих в формировании запроса, и получить следующий результат:

```
....
IF lv_X = lv_Y
THEN
    lv_TABLE := ‘tt_table1’;
ELSE
    lv_TABLE := ‘new_table2’;
END IF;
lv_S := ‘SELECT new_x FROM ‘ || lv_TABLE || ‘  WHERE NVL(n,0) > 1’;
OPEN new_cursor FOR lv_S;
....
```

В общем случае задача статического вычисления всех значений переменных неразрешима, так как, например, множество порождаемых строк может быть бесконечным. Такая ситуация может возникнуть при наличии

конкатенации в цикле. Так же переводимая система может взаимодействовать с другими системами и получать данные от них и, как следствие, не все значения могут быть вычислены статически.

По этой причине в большинстве случаев обработка динамически формируемых строк подразумевает автоматизацию процесса, а не полностью автоматическую обработку. Это означает, что в случае, если по каким-либо причинам автоматическая обработка не возможна, происходит ручное вмешательство. Однако, для упрощения ручной работы необходимо максимально точно обнаруживать сложные участки и сообщать о них всю необходимую информацию.

В рамках данной работы исследованы алгоритмы работы с динамически формируемыми строками, предложен механизм для автоматизации трансляции динамического SQL, описан опыт реализации предложенного механизма на примере трансляции хранимых процедур из T-SQL в PL-SQL. Данный опыт может быть полезен не только в проектах по реинжинирингу, но и при создании интегрированных сред разработки с поддержкой рефакторинга.

2 Постановка задачи

Главная цель данной работы – это разработка средства автоматизации трансформации динамически формируемых строк.

В качестве основной задачи предлагается рецить задачу автоматизации трансляции динамического SQL из T-SQL в PL-SQL.

Для её решения необходимо:

- исследовать возможности автоматической обработки динамически формируемых строк.
- разработь инструмента, который должен позволить автоматизировать трансформацию динамических SQL-запросов.
- реализовать этот инструмент и интегрировать его в проект по реинжинирингу.

Кроме этого, разрабатываемый инструмент должен обладать следующими свойствами.

- Статическая проверка синтаксической корректности результатов трансляции.
- Сохранение и последующее воспроизведение результатов трансляции динамического SQL без запуска трансляции. Так как трансляция динамического SQL – это очень ресурсоёмкий и длительный процесс, то необходимо иметь возможность сохранить результаты его работы не зависимо от результатов общей трансляции и воспроизводить при последующих запусках инструмента не запуская собственно трансляцию динамического SQL.
- Сохранение изменений при ручной корректировки результатов трансляции их и воспроизводимость.

3 Обзор

В рамках обзора основной интерес представляют инструменты для трансформаций, особенно трансляции, хранимого SQL-кода, которые, возможно, поддерживают преобразования динамического SQL, а также инструменты и подходы к работе с динамически формируемыми строками.

3.1 Промышленные инструменты миграции баз данных

Миграция БД – актуальная задача, для решения которой существует ряд промышленных инструментов. В силу особенностей решаемой задачи, нас интересуют инструменты для трансляции хранимого кода БД. Такими являются PL-SQL Developer [10], SwisSQL [22], SQL Ways [24]. Эти инструменты применяются для трансляции хранимого SQL-кода, однако, ни один из них не обрабатывает динамический SQL.

3.2 Инструменты анализа динамически формируемых строк

Задача статической обработки динамически формируемых строк достаточно актуальна. Многие приложения работают с базами данных через различные интерфейсы, позволяющие формировать запросы во время выполнения и исполнять их напрямую. При этом, для крупных систем задача обеспечения надёжности и безопасности системы серьёзно усложняется. По этому одно из основных направлений – это работа со встроенным SQL: проверка корректности формируемых запросов и защита систем от SQL-инъекций [14]. Также развивабтя инструменты для проверки валидности генерируемого HTML.

Для работы с динамически формируемыми строками существует ряд инструментов, основные особенности которых представлены ниже.

- **Java String Analyzer (JSA) [4]** – это инструмент для анализа формирования строк и строковых операций в программах на Java. Для каждого строкового выражения он строит конечный автомат, пред-

ставляющий приближённое значение всех значений этого выражения, которые могут быть получены во время выполнения [7].

- **Alvor [3] [5]** – это плагин к среде разработки Eclipse, предназначенный для статической валидации SQL-выражений встроенных в код на Java. Он может использоваться или в режиме разового запуска на всём исходном коде или в режиме инкрементального анализатора, который работает в процессе написания кода. Найденные в коде SQL-запросы проверяются на основе SQL-грамматики. Так же они могут проверяться исполнением в указанной тестовой базе.

- **PHP string analyzer (Phasa) [21]** – это статический анализатор для строк, порождаемых программами на PHP. Он аппроксимирует значения таких строк некоторой контекстно-свободной грамматикой. Может использоваться, например, для валидации динамически генерируемых программой на PHP Web страниц.

- **SAFELI [14]** — инструмент статического анализа, предназначенный для определения возможности SQL-инъекций в Web-приложениях на этапе компиляции. SAFELI работает с MSIL байткодом ASP.NET приложений.

- **A Static Analysis Framework for Database Applications [9]** . Этот Инструмент может проводить анализ скомпилированого кода, использующего ADO.NET для доступа к данным. Может применяться для определения возможных мест для SQL-инъекций, определения "узких мест" по производительности запросов, проверки ограничений на данные.

Все эти инструменты предназначены для различных валидаций динамически формируемых строк. В большинстве случаев это проверка синтаксической корректности. Ни один из существующих инструментов не предназначен для трансляции динамически формируемых строк.

3.3 Подходы к обработке динамически формируемых строк

Обработка динамических строк в общем случае может состоять из лексического и синтаксического анализа, где первый может отсутствовать как явно выделенный шаг. Рассмотрим существующие подходы.

- Лексический анализ:
  - С применением FST [6] – на основе входного автомата строится новый над нужным алфавитом.
  - Scannerless технологии – нет отдельной стадии лексического анализа. Грамматика обрабатываемого языка задаётся непосредственно над алфавитом символов.
- Синтаксический анализ:
  - Абстрактная интерпретация — основана на постепенной интерпретации некоторого представления множества возможных значений формируемой строки. Данный подход подробно описывается в статьях [18] [8]
  - На базе синтаксического анализа [11]. Основная идея этого подхода заключается в том, что обрабатываются не явно вычисленные значения строк, а их более компактные представления: регулярное выражение, конечный автомат, data-flow уравнение. Большинство из рассмотренных инструментов применяют именно этот подход с различными модификациями.

3.4 Вывод

В результате обзора существующих решений было выяснено, что современные промышленные инструменты для миграции баз данных не обрабатывают динамические запросы, а инструменты для работы с динамически формируемыми строками не предназначены для их трансформации,в частности, трансляции, а только для различных валидаций.

В качестве основы для реализации выбран подход на основе синтаксического анализа с явно выделенным лексическим анализом, так как он поз-

воляет переиспользовать уже разработанные лексер, грамматику и транслятор, что немаловажно для проекта, в котором эти компоненты уже разработаны.

4 Реализация

Разработка велась в рамках промышленного проекта по миграции базы данных с MS-SQL Server 2005 на Oracle 11gR2. Так как этот проект разрабатывался на платформе .NET [19] и основным языком программирования был F# [12], то эти же язык и платформа были выбраны для решения нашей задачи.

Так как разработка велась в рамках проекта, который к моменту начала работ уже длительное время развивался, то логичным решением было постараться максимально переиспользовать уже реализованные алгоритмы и компоненты:

- грамматика MS-SQL;
- лексер MS-SQL;
- транслятор MS-SQL в PL-SQL;
- различные статические анализы и обходы.

4.1 Формат статистики

Для проведения различных оценок и контроля процесса, особенно в промыленном проекте, полезно иметь статистику по текущему состоянию уровня автоматизации. При обработке динамического SQL собиралась, кроме всего прочего, базовая статистика синтаксического разбора динамических запросов в следующем формате:

Всего:	Общее количество динамических запросов.
Содержат DDL:	Количество запросов, содержащих DDL.
Успешно:	Количество полностью успешно обработанных запросов.
Частично успешно:	Количество частично успешно обработанных запросов.

С лексическими ошибками:	Количество запросов с лексическими ошибками.
С ошибками парсера:	Количество запросов с синтаксическими ошибками.
С другими ошибками:	Количество запросов с другими ошибками.
Не разобраны:	Количество неразобранных запросов.
С лексическими ошибками:	Количество вызовов с лексическими ошибками.
С ошибками парсера:	Количество вызовов с синтаксическими ошибками.
Экспоненциальный рост количества стеков:	Количество запорсов со сложной логикой построения, для которых количество состояний растёт очень быстро.
С другими ошибками:	Количество запросов с другими ошибками.
Процент успешных:	Процент успешно прошедших синтаксический разбор.
Процент частично успешных:	Процент частично успешно обработанных запросов.
Процент с не пустым лесом:	Процент запросов, для которых в результате разбора был получен непустой лес.
Процент содержащих DDL:	Процент запросов, содержащих DDL.

К частично успешным запросам относятся запросы, при разборе которых были получены ошибки, но при этом лес оказался не пустым. Это самая сложная для анализа категория, так как ошибка парсера, в данном случае,

может быть „ложной тревогой“, поскольку при выполнении такое значение могло никогда не порождаться.

Далее мы будем часто приводить такую статистику для демонстрации результатов.

4.2 Терминология

Главная особенность абстрактного анализа – не вычислять все возможные значения динамически формируемой строки, а работать с компактным представлением множества значений. В данном случае это будет граф, являющийся представлением результата протягивания констант. Таким образом каждому пути в графе соответствует некоторое возможное значение динамического запроса. Будем говорить, что данный путь ***порождает*** соответствующее значение.

Будем говорить, что ***запрос содержит лексические или синтаксические ошибки***, если в графе, соответствующем этому запросу, существует хотя бы один путь, порождающий значение с лексической или синтаксической ошибкой соответственно.

В рамках данной задачи основной сируктурой данных для лексического и синтаксического анализа является граф. Его можно воспринимать как аналог потока входных символов и потока токенов соответственно. Поэтому, мы будем говорить о ***токенизации*** или ***лексическом ананлизе*** графа, подразумевая под этим некоторый процесс, который переведёт граф, содержащий строки, в граф, содержащий токены. Аналогичным образом мы будем говорить о ***синтаксическом анализе*** (***разборе***) графа, подразумевая процесс, на выходе которого мы получем некоторое множество синтаксических деревьев или ***лес***, соответствующих входному графу. Каждое дерево соответствует некоторому значению запроса, порождённому некоторым путём в графе. Таким образом дерево соответствует пути в графе.

Под ***обратной ссылкой*** будем понимать ссылку из узла в дереве, построенном для динамического запроса, на узел в исходном дереве. Информацию, необходимую для установления соответтвия между узлом исходного дерева и некоторым объектом на этапе абстрактного анализа, будем называть ***привязкой***

4.3 Абстрактный анализ

Основной подход к работе с динамически формируемыми строками – абстрактный анализ (лексический и синтаксический [11]), который заключается в том, что можно не строить явно множество всех порождаемых строк, а работать с более компактным представлением: регулярное выражение, data-flow уравнение. При этом производятся различные загрубления по двум основным причинам:

- В общем случае задача сводится к проверке вложенности языков, которая не всегда разрешима [6].
- Объём вычислений при решении задачи трансляции динамически формируемых строк в общем случае очень велик, однако, часто можно предложить упрощения, существенно снижающие объём вычислений и дающие приемлимый по качеству результат.

Серьёзным отличием от решаемых ранее задач была необходимость трансляции. При валидации генерируемых строк производится только синтаксический анализ, что позволяет объединять состояния парсера как это происходит в GLR-алгоритмах [16] анализа. При трансляции это не возможно, так как неизбежно появляются семантические вычисления и понятие состояния существенно усложняется. В общем случае необходимо вычислить все семантические действия для всех значений динамической строки.

Так же трансляция требует сохранения точной привязки к исходному дереву для того, чтобы вычислить новые значения для переменных, участвующих в формировании строки, в то время как многие инструменты валидации указывают только точку выполнения запроса и причину ошибки.

Было принято решение работать с графом, представляющим результат протягивания констант. Так как в результате работы важно получить новые значение для переменных, а их повторение на синтаксическую корректность не влияет, то из графа были исключены циклы и заменены на единичное повторение. Это позволило существенно упростить последующую работу, так как граф, при таких упрощениях, становится DAG-ом. Далее, если идёт речь о графе, то этот граф так же DAG, если не оговорено противное.

При отладке алгоритмов часто оказывается нужным получить наглядное представление значений, которые может принимать выражение. Представление в виде регулярного выражения оказалось удобным для визуальной оценки сложности запроса, а в дальнейшем и для оценки корректности результатов трансляции. Для целей отладки наиболее подходящим оказалось представление в виде графа. Для получения визуального представления графов были реализована утилита, которая генерировала файлы для инструмента Graphviz [15] в формате DOT с описанием графа. Инструмент Graphviz использовался для раскладки и визуализации графов.

Для отладки оказалось удобным и, в большинстве случаев, достаточным использовать графы, полученные на двух этапах работы алгоритма:

1. Граф, построенный непосредственно по результатам протягивания констант.

2. Граф, полученный после лексического анализа, непосредственно перед синтаксическим анализом.

Также для промышленного инструмента важным механизмом является механизм сообщение об ошибках. Основным критерием часто является точность указания места в коде, где произошла ошибка. Техническое решение, позволившее существенно упростить отладку и сделать сообщения об ошибках более удобными, – это установить в качестве координат узлов координаты литералов из которых они породились, а в качестве родителя для корня дерева – execute из основного дерева. После этого стало возможным применение общего механизма вычисления абсолютной и относительной, с привязкой к началу процедуры, координат ошибки.

Серьёзной проблемой оказались вопросы, связанные с корректностью формируемых запросов. В статистике такие запросы отмечены как частично успешные. Эта категория самая сложная для анализа. С одной стороны, можно сделать предположение, что все пораждаемые значения корректны, так как транслируемая система применяется на практике и её текущее поведение можно считать корректным. В такой ситуации ошибка означает ошибку в нашем инструменте. С другой стороны, это предположение не всегда верно по двум причинам. Первое – знание семантики разработчиком

позволяет ему утверждать, что при выполнении кода некоторые значения никогда не будут получены, однако при статическом анализе они получаются и оказываются некорректными. Второе – в долго живущих системах очень много мёртвого кода, который может быть принципиально некорректным. В случае с базами данных ситуация осложняется тем, что в мигрируемой базе могли быть сохранены различные тесовые процедуры. Часть таких вопросов была снята после удаления мёртвого кода. Однако, осталось большое количество запростов, вопрос о корректности которых пришлось решать вручную.

Трансляция динамического SQL производится полностью до трансляции основного скрипта. С одной стороны это позволяет упростить сохранение привязки, так как нет необходимости следить за преобразованиями узлов в основном дереве. С другой – информация, полученная в результате обработки динамического SQL необходима для корректной трансляции основного скрипта. Таким образом, новые значения вычисляются для переменных в исходном дереве и только после этого производится трансляции основного скрипта.

Отдельной задачей является правильное упорядочивание различных анализов и проходов по дереву, так как существует достаточно сильная зависимость между многими из них и непосредственно обработкой динамического SQL. Например, превращение динамического запроса в курсор может повлиять на сигнатуру процедуры, в которой находится этот запрос. В тоже время, в динамическом запросе существуют вызовы процедур, которые требуют трансляции, для которой необходимо знать сигнатуру вызываемой процедуры.

Так же важно отметить, что верификация результатов трансляции производится до трансляции основного скрипта. Это позволяет полностью переиспользовать алгоритм протягиваник констант. При этом делается важное предположение, что трансляция основного скрипта корректна и, если до неё запрос формировался правильно, то и после он так же сформируется правильно.

Такм образом, общая схема работы инструмента выглядит следующим образом:

1. разбор основного скрипта
2. протягивание констант
3. трансляция динамического SQL
    (a) абстрактный лексический анализ
    (b) абстрактный синтаксический анализ
    (c) трансляция
    (d) вычисление новых значений для переменных, участвующих в формировании запроса
    (e) печать новых значений
4. протягивание констант
5. валидация результатов трансляции динамического SQL
6. трансляция основного скрипта.

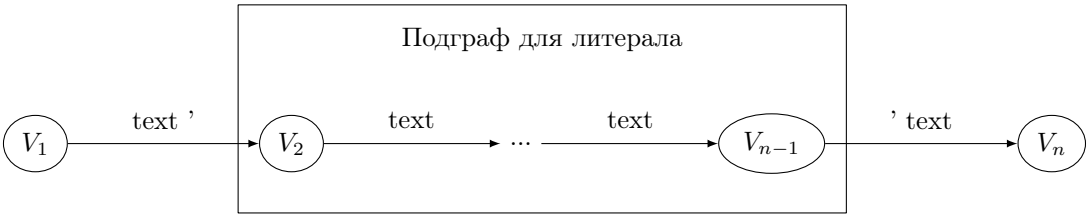
Далее будут более подробно рассмотрены шаги, необходимые для трансляци динамического SQL.

4.3.1 Абстрактный лексический анализ

Было принято решение выделить синтаксический анализ в отдельный шаг, так как это позволяло переиспользовать уже разработанную грамматику T-SQL. Также, важно, что такой подход позволял практически полностью переиспользовать для лексического анализа уже разработанный лексер.

Основная идея реализованного абстракного лексического анализа заключается в том, что в предположении, что текст, соответствующий одному токену, не разрываются на несколько переменных, лексический анализ сводится к токенизации строки на ребре графа. Надо отметить, что исходный лексер был реализован так, что в некоторых случаях требовалось знание контекста (в основном заглядывание вперёд) для определения типа токена. Однако в нашем случае информации на ребре хватало для корректной токенизации.

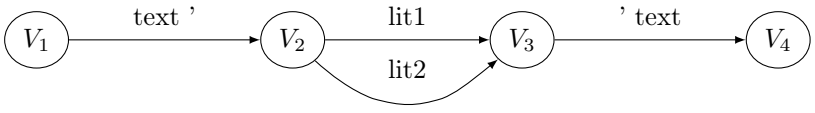
В большинстве случаев такое допущение оказалось верным. Исключение составили „разорванные“ литералы, То есть такие литералы, открывающая и закрывающая кавычка которых находится в разных литералах в исходном коде. Результат протягивания констант для таких ситуацию будет выглядеть следующим образом.



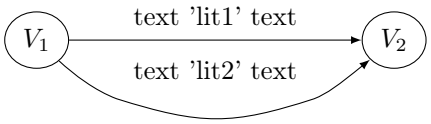
Такие литералы были основным источником лексических ошибок. В качестве решения данной проблемы было предложено поступить следующим образом:

1. явно построить все пути в графе, при разборе которого возникла данная проблема,
2. провести конкатенацию значений на построенных путях,
3. провести токенизацию,
4. минимизировать полученный граф.

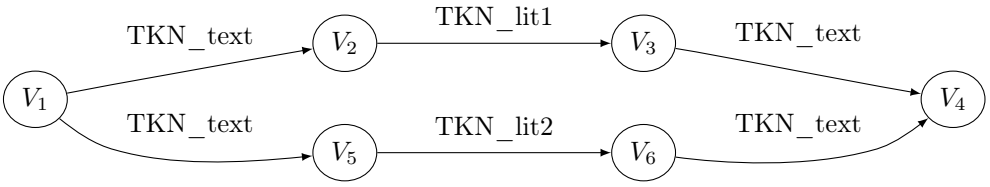
Рассмотрим данные преобразования на примере. Пусть на входе имеется следующий граф.



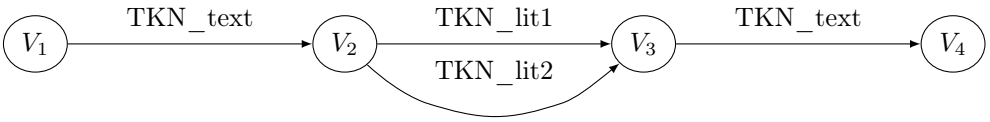
Токенизация такого графа не сможет завершится успешно, так как на ребрах $V_1 \to V_2$ и $V_3 \to V_4$ есть незавершённые литералы. Чтобы решить эту поблему, построим всё множество путей явно.



Токенизация такого графа уже пройдёт успешно и будет получен следующий результат.



Следующим шагом нужно провести минимизацию полученного графа. Результат приведён ниже.



К сожалению, данное решение не работает на графах с большим количеством ветвлений, так как при явном вычислении путей их получается очень много. Но даже если ввести искусственное ограничение и явно вычислять все пути только в графах, в которых таких путей не больше n (для достаточно большого, но „разумного“ n), то это существенно улучшает статистику обработки.

Пример изменения статистики для n=5000.

	Было	Стало
Всего:	3029	3029
Содержат DDL:	551	553
Успешно:	1469	1899
Частично успешно:	418	358
С лексическими ошибками:	120	21
С ошибками парсера:	339	357
С другими ошибками:	0	0

Не разобраны:	591	219
С лексическими ошибками:	444	55
С ошибками парсера:	565	179
Экспоненциальный рост количества стеков:	26	40
С другими ошибками:	0	0
Процент успешных:	59.28%	76.7%
Процент частично успешных:	16.87%	14.46%
Процент с не пустым лесом:	76.15%	91.16%
Процент содержащих DDL:	18.19%	18.26%

4.3.2 Абстрактный синтаксический анализ

При реализации абстрактного синтаксического анализа изменения касаются только интерпретатора таблиц. Сами же таблицы остаются стандартными. Таким образом, можно реализовать свой интерпретатор для существующего генератора таблиц [11].

Так как основной парсер MS-SQL был реализован с помощью инструмента YaccConstructor [25] [26] с фронтендом YARD [27] и бэкендом FsYacc [13], который является стандартным генератором LALR(1) анализаторов для языка программирования F#, то было решено реализовать абстрактный синтаксический анализ на базе FsYacc-а. Важным, при этом, является то, что FsYacc входит в пакет F# Power Pack (`http://fsharppowerpack.codeplex.com/`), исходники которого открыты.

Таким образом можно переиспользовать уже разработанную грамматику и лексер и воспользоваться генератором FsYacc для получения таблиц. Необходимо только реализовать новый интерпретатор, за основу которого можно взять интерпретатор из FsYacc-а.

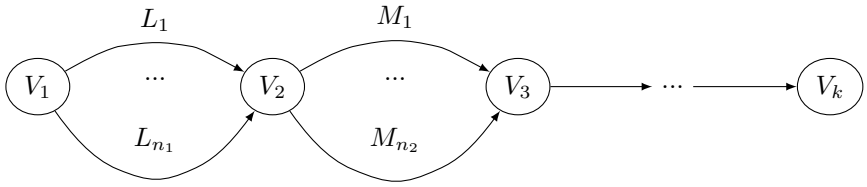
Кроме непосредственно преобразования интерпретатора для абстрактного анализа необходимо реализовать механизм протягивания и сохранения привязки узлов к узлам в исходном дереве и отключить восстановление от ошибок.

Использование механизма восстановления после ошибок в абстрактном синтаксическом анализе, особенно в рамках задач по реинжинирингу, не целесообразно, так как статически сложно сказать, является ли данная ситуация действительно ошибочной. Это связано с тем, что при появлении ошибки мы не знаем, действительно ли это ошибка или же просто такое значение никогда не могло быть получено во время выполнения. При применении восстановления могут быть получены дополнительные результаты, о корррректности которых судить очень сложно. В большинстве случаев они окажутся лишними. Поэтому ошибочные состояния заносятся в отдельный список ошибок для последующего ручного анализа и удаляются из множества состояний и в дальнейшем парсером не обрабатываются.

Основнаа алгоритма абстрактного синтаксического анализа сводится к вычислению всех возможных состояний парсера в каждой вершине графа [6]. То есть в основе лежит обход графа с поиском неподвижной точки. В нашем случае можно упростить задачу. Так как граф является DAG-ом, то можно просто обойти все вершины один раз в порядке N нумерации.

Серьёзной проблемой являются большие запросы, формируемые с использованием множества условий, которые могут в общем случае порождать при статическом анализе очень много вариантов. Условно такая ситуация называется „экспоненциальным взрывом“ и подлежит внимательному изучению. В некоторых случаях удаётся эквивалентно переформулировать исходный код, а иногда просто вручную добавить уточнения о заведомо ложных/„несовместных“ ветках. Ситуации с „экспоненциальным взрывом“ отмечены в статистике отдельно.

На практике такая ситуация встречается достаточно часто. При этом, граф не обязательно должен быть очень сложным, чтобы его обработка потребовала много ресурсов. Для примера рассмотрим следующий граф.



В данном случае для вершины V_3 будет плучено $N = n1 * n2$ состояний. Предположим, что N оказалось большим, но все состояния были вычис-

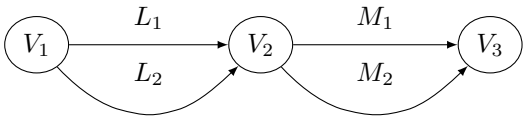
лены. Пусть, также, "хвост" графа от вершины V_3 до V_k является линейным участком и k достаточно велико. Графы такого вида достаточно часто встречаются на практике. При обработке такого графа, при переходе по каждой дуге, начиная с исходящей из V_3, необходимо обработать N состояний. В результате требуется обработать $N * (k-3)$ состояний, что потребует больших ресурсов.

Таким образом, задача уменьшения количества обрабатываемых состояний очень актуальна для систем с большим количеством запросов со сложной логикой построения.

Борьба с „экспоненциальным взрывом“ основана на следующей идее. Результатом трансляции должны стать новые значения для переменных, участвующих в формировании динамического запроса. Поэтому можно строить не все возможные деревья, а минимальное множество деревьев, покрывающее все переменные. То есть можно рассмотреть не все возможные пути в графе, а только минимальное множество путей, покрывающих все дуги. В общем случае эта задача сводится к задаче о покрытии, которая является Np-сложной. Но даже приближённо решая её тривиальным жадным алгоритмом, удаётся добиться существенного снижения скорости роста количества состояний.

Нужно понимать, что нельзя вычислить такой набор путей статически для графа, полученного после лексического анализа. Основная проблема заключается в том, что при статическом вычислении заранее не известно, сможет ли данный путь корректно разобраться парсером и породить дерево. В случае, если путь окажется некорректным, то будет потеряна часть информации.

Рассмотрим такой пример. Пусть абстракный синтаксический анализ должен обработать следующий граф.



Одно из возможных множеств путей для такого графа при статическом вычислении может быть таким: $\{(L_1; M_1); (L_2; M_2)\}$. Однако, они могут оказаться синтаксически не корректными и в результате синтаксического ана-

лиза не будет получено ниодного дерева. Корректными могут оказаться, например, пути $\{(L_1; M_2); (L_2; M_1)\}$. Таким образом, при разборе первого множества не будет получено ни одного дерева, а при разборе второго – два.

По этому вычисление набора путей происходит итеративно. Во время синтаксиеского анализа для каждой вершины в которую входит более чем одна дуга происходит фильтрация состояний в этой вершине. Происходит это следующим образом. Изначально результирующее множество состояний пусто. В него постепенно добавляются новые состояния. При этом рассматривается подграф, образованный предками рассматривоемой вершины. Состояние должно быть добавлено, если верно одно из условий:

- Новому состоянию соответствует путь, который покрывает хотя бы одну дугу, ещё не покрытую путями, соответствующими состояниям, уже добавленным в результирующее множество.
- Новое состояние соответствует состоянию парсера, отсутствующему в результирующем множестве. При этом оно может не добавлять новых дуг в покрытие.

Таким образом получается результирующее множество, которое по мощьности не более, чем исходное, содержит состояния, задающее такое же множество состояний парсера, как и исходное множество и, при этом, соответствующие пути покрывают все дуги в соответствующем подграфе.

После применения такой фильтрации удалось достичь следующих результатов:

- уменьшилось количество деревьев;
- уменьшилось количество “экспоненциальных взрывов”;
- улучшилась производительность;

	Было	Стало
Всего:	3122	3122
Успешно:	2181	2253

Частично успешно:	408	522
С лексическими ошибками:	283	289
С ошибками парсера:	354	468
С другими ошибками:	0	0
Не разобраны:	533	347
С лексическими ошибками:	140	134
С ошибками парсера:	280	305
Экспоненциальный рост количества стеков:	253	41
С другими ошибками:	0	0
Процент успешных:	69.86%	72.17%
Процент частично успешных:	13.07%	16.72%
Процент с не пустым лесом:	82.93%	88.89%

Но фильтрация не всегда гарантирует, что разбор динамического запроса завершится за разумное время, так как асимптотика в худшем случае не меняется и остаётся экспоненциальной. В такой стуации критерием разумности является ограничение, накладываемое тем, что обработка всего скрипта должна проходить в рамках ночного тестирования.

Техническое решение этой проблемы – снятие обработки динамического запроса по таймауту. На разбор динамического запроса выделяется определённое время, которое задаётся в конфигурации инструмента. Если разбор не завершился за это время, то он прекращается и результат считается не успешным. В статистике – это категория „Экспоненциальный рост количества стеков“.

4.3.3 Сохранение привязки

Результатом работы алгоритма должны быть ноыве значения переменных, учатвующих в формировании динамического запроса. Для того, чтобы была возможность вычислить эти значения, необходимо сохранять при-

вязку элементов динамичксуогоязапроса к исходному дереву. Существует несколько основных типов узлов, к которым привязываются части динамических запросов. Это непосредственно литерал, определение переменной, формальный параметр процедуры и имя колонки в таблице.

Сохранение привязки начинается с протягивания констант, когда вместо строкового значения сохраняется пара, содержащая значение и узел в дереве из которого взято данное значение. В результате протягивания констант получается граф, каждой дуге которого соответствует такая пара. При упрощении такого графа привязка соответствующим образом перещитывается. Например, происходит конкатенация значенй при склейке последовательно идущих рёбер в линейном участке. Таким образом, после необходимых упрощений сохраняется информация о соотношении дипозона в строке на дуге графа и узла в оригинальном дереве.

Во время лексичечкого анализа такого графа токен, появившийся с некоторой дуги, получает в качестве привязки элемент, который соответствует диопозону строки с соответствующей дуги. Проблему составляют составные литералы, в случае с которыми для одного токена может существовать несколько дуг в графе, после токенизации которых он получился. Как следствие, у одного токена появляется несколько обратных ссылок. Узлы, которым соответствует несколоко узлов в исходном дереве существенно усложняют их дальнейшую обработку.

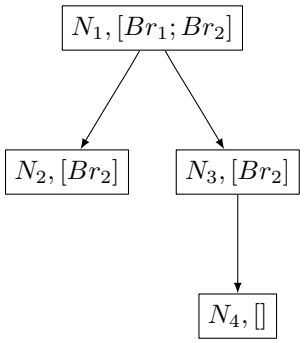
Вычсление обратных ссылок происходит непосредственно при синтаксичечком анализе. Вычисление происходит при свёртке аналогично тому, как вычисляется привязка к исходному тексту. При этом тянуть привязку надо до перврго узла, сыновья которого ещё не имеют такой же привязки. Идеальный случай, когда у одного узла одна обратная ссылка. Ситуация заметно усложняется если это не так. Классический пример такого случая – разорваный литерал. Для соответствующего ему узла обратных ссылок будет столько, сколько переменных участвовало в его формировании.

После того как дерево построено совершается ещё один проход, кокторый корректирует обратные ссылки таким образом, чтобы конкретная обратная ссылка стояла как можно выше в дереве и при этом все сыновья узла с этой ссылкой, включая его самого, содержали бы обратную ссылку

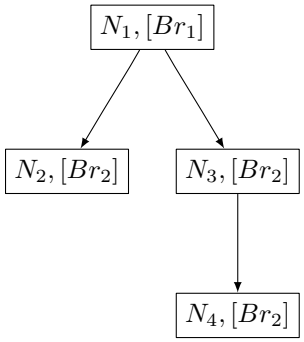
на тот же узел или этот узел содержал уникальную ссылку.

Корректеровка проводится после синтаксического разбора, так как произведение необходимых вычислений по хожу анализа серьёзно бы его замедлило. При этом, при вычислении „на лету“, большое количество вычислений было бы сделано зря, например, для состояний которые будут отфильтрованы. По этому, было решено выполнять корректировку отдельным шагом после синтаксического разбора.

Например, после синтаксического анализа могли быть получены следующие обратные ссылки.



В результате корректировки будет получен следующий результат.



При дальнейшей работе необходимо следить за корректностью обратных ссылок. Например, не забывать переносить их при трансляции из старых узлов в новые.

4.3.4 Трансляция

Трансляция леса, полученного для динамических запросов не должна ни чем отлиаться от трансляции основного скрипта. По этому в основе лежит

транслятор, используемый для трансляции основного скрипта. Он запускается над деревом, собраным из леса, полученного для всех динамических запросов. Склеивание леса в одно дерево позволяет выиграть в производительности. Правда, данное решение может быть пересмотрено при решении вопроса о параллельной обработке динамических запросов. Но этот вопрос требует отдельного изучения.

При трансляции динамического SQL переиспользовался механизм основной трансляции. Однако в некоторых случаях требовались доработки. Одним из таких случаев является идентификация динамических объектов.

Существуют объекты, которые не передаются как аргументы при вызовах, но при этом видны по стеку вызовов. В MS-SQL ярким примером являются локальные временные таблицы. Созданная в процедуре, такая таблица видна далее в вызываемых процедурах без явной передачи в качестве параметра. При статическом анализе можно попытаться в точке использования идентифицировать такую таблицу, просматривая все вызывающие процедуры по графу вызовов. Но это часто оказывается неточным или, даже, невозможным, так как для процедуры, в которой происходит использование объекта, может существовать несколько вызывающих процедур, в которых создаются таблицы с одинаковыми именами. При работе с динамическим SQL часто используется следующий шаблон. В вызывающей процедуре создаётся временная таблица, имя которой передаётся в вызываемую в качестве параметра. В вызываемой процедуре формируется запрос к этой таблице. В таком случае, для более точного переименования таблиц необходимо производить идентификацию таблиц, имена которых передаются как аргументы, в контексте вызывающей процедуры.

Также, отдельной обработки при трансляции требовали конструкции , которые возвращали результат (например, `SELECT`)и вызов процедуры `SP_EXECUTESQL`,которая предназначена для выполнения динамических запросов и имеет следующую сигнатуру.

```
sp_executesql [ @stmt = ] statement
[
  { , [ @params = ] N'@parameter_name data_type [ OUT | OUTPUT ][ ,...n ]' }
     { , [ @param1 = ] 'value1' [ ,...n ] }
```

]

Аргументы процедуры:

- `[ @stmt= ] statement` – строка, содержащая инструкцию или пакет T-SQL.
- `[ @params= ] N'@parameter_namedata_type [ ,... n ] '` – строка, содержащая определения всех параметров, внедренных в `@stmt`. Строка должна представлять собой константу либо переменную в строковом формате. Определение каждого параметра состоит из имени параметра и типа данных.
- `[ @param1= ] 'value1'` – значение для первого параметра, определенного в строке параметров.

Видно, что из параметров данной процедураы формируется выражение на T-SQL, которое за тем выполняется. Часто это выражение может оказаться блоком, который возвращает результат. В таком случае создать динамический курсор не получится. В качестве решения данной проблемы можно передавать во внутрь переменную курсорного типа, которая будет инициализироваться внутри динамического запроса. При этом результат будет соответствующим образом заполнен и виден снаружи.

Сам по мебе, вызов системной процедуры `SP_EXECUTESQL`тоже должен быть транслироватн в `EXECUTE IMMEDIATE`, как и большинство других динамических запросов. При этом параметры этой процедуры передаются с использованием конструкции `USING`.

Конструкция `SELECT`,возвращающая результат, должна преобразовываться в `OPEN`<cursor> `FOR`<query>. Где <query> является кооректной трансляцией исходного запроса и не требует дальнейших доработок.

В случае, если динамический запрос состоит не из единственной инструкции, а является сложным блоком, то необходим анализ, выявляющий конструкции, пораждающие курсор.

Важно, что такие преобразования, ведущие к появлению новых курсоров, нужны для анализа курсоров в основном скрипте, результаты которого влияют, кроме всего прочего, на сигнатуры процедур, в которых встретился такой запрос.

4.4 Печать новых значений для переменных

Печать новых значений для переменных оказалась сложной задачей, требующей отдельного изучения. Основные проблемы, с которыми пришлось столкнуться в рамках данной работы, а также возможные пути решения изложены ниже.

Часто имена таблиц и колонок передаются в процедуру в качестве параметров. Наряду с этим, так же передаются части условий для различных фильтраций и объединений. В нашем проекте, в подавляющем большинстве случаев, значения подставлялись непосредственно в точке вызова процедуры. Поэтому было достаточно реализовывать только протягивание параметров вызовов процедур, а не полностью межпроцедурное протягивание. Однако, даже просто протягивание параметров вызова иногда приводит к „экспоненциальному росту стека“. Например, когда процедура содержит достаточно сложный запрос, зависящий от параметров вызова, и вызывается большое количество раз с различными аргументами из различных процедур. При этом, полностью оказываться от протягивания аргументов не всегда корректно, так как знание точных значений может уточнить трансляцию. Кроме этого, часто в качестве аргументов передаются составные части различных условий, которые требуют полноценной трансляции.

Бинарные операции, запятые, скобки, которые часто находятся на границе литералов, не имеют явного представления в дереве и, как следствие, не имеют привязки к исходному дереву. Это вызывает большие проблемы при их печати обратно в переменные.

Существуют конструкции, которые имеют сложные проекции трансляции, сильно изменяющие структуру конструкции. Ярким примером является конструкция MS-SQL `UPDATE... FROM`, которая транслируется в `MERGE`.

При этом сильно изменяется структура запроса и становится невозможным ограничится только вычислением новых значением для старых переменных. Неизбежно появляются новые переменные, за исключением тривиальных случаев, когда изменяющейся частью запроса являемся имя таблицы.

Ещё одной серьёзной проблемой являются разорванные литералы, особенно со сложной логикой формирования. Для таких литералов сложно

при печати для его частей определить, какие переменные участвовали в его формировании и вычислить новые значения для них.

На данный момент реализован механизм, позволивший корректно обработать примерно 45% запросов. Он основан на следующих основных идеях.

- Для запросов, которые могут принимать единственное значение, можно явно печатать это значение в одну переменную.
- Можно считать, что и те запросы, для которых множественность появляется только в результате протягивания аргументов процедур, так же принимают единственное значение. Но для таких запросов нужен чуть более сложный алгоритм печати.
- При печати новых значений для аргументов процедур надо внимательно следить за тем, чтобы не напечатать лишнего, так как изменять аргументы процедур нельзя.

4.5 Валидация результатов трансляции

Необходимо статически гарантировать корректность результата трансляции динамического SQL. Это связано с несколькими особенностями. Основная из них заключается в том, что не всегда можно быстро реализовать формально корректный алгоритм тарнсформации динамически формируемых строк, если вообще возможно. Часто бывает выгоднее реализовать некоторый набор эвристик, работающих в рамках конкретной задачи, что негативным образом сказывается на формальной корректности алгоритма. При этом для больших систем, как правило, не существует полного тестового покрытия и разработать его не представляется возможным. Однако, так как самым „узким“ местом оказалась печать новых значений для переменных, участвующих в формировании запроса, а алгоритм трансляции является хорошо протестированным и проверить его корректность проще, то простая проверка синтаксической корректности результатов трансляции даёт достаточную оценку качества трансляции.

Валидация основана на предположении, что для конкретного динамического запроса количество деревьев в результате абстрактного разбора соответствующим парсером до и после трансляции должно быть одинаково.

Для валидации необходимо только разработать грамматику целевого языка в необходимом объёме. Остальные механизмы (протягивание констант, абстрактный лексический
синтаксический анализы) могут быть полностью переиспользованы.

Используя возможности YaccConstructor-а удалось так же частично переиспользовать грамматику, так как синтаксисически T-SQL и PL-SQL очень похожи. Это существенно упростило работу.

YaccConstructor предоставляет механизм схожий с условной компиляцией, когда те или иные участки грамматики отфильтровываются на этапе лексического анализа на основе условий, заданных пользователем. Для задания таких условий существует следующая синтаксическая конструкция:

```
#if <literal>
<code>
#else
<code>
#endif
```

Литерал может быть задан пользователем при запуске инструмента. Для этого используется опция `-D`. Соответственно, если он задан, то остаётся часть в первом блоке (блоке `then`), иначе остаётся часть в блоке `#else`. Блок `#else` не является обязательным.

Пример использования.

```
if_stmt:
    ‘‘IF’’ expr
#if pl
    ‘‘THEN’’
#endif
    stmt_block
    ‘‘ELSE’’
    stmt_block
#if pl
    ‘‘END’’ ‘‘IF’’
#endif
```

```
;
```

Соответственно, если при запуске генератора пользователь указал `-D pl`, то будет построен транслятор для следующей грамматики:

```
if_stmt:
    ''IF'' expr
    ''THEN''
    stmt_block
    ''ELSE''
    stmt_block
    ''END'' ''IF''
;
```

В противном случае транслятор построится для грамматики:

```
if_stmt:
    ''IF'' expr
    stmt_block
    ''ELSE''
    stmt_block
;
```

4.6 Ручная доработка результатов трансляции

В некоторых случаях может потребоваться ручная доработка результатов трансляции. В большинстве случаев требуется корректировка значений конкретных литералов. Необходимо, чтобы исправления не требовалось вносить после каждой новой генерации кода, а результаты исправлений были автоматически воспроизводимы при последующих запусках.

Для такой корректировки реализован механизм, позволяющий описать исправления в виде <старое значение литерала>/<новое значение литерала>, с возможностью указать координаты литерала, требующего исправлений.

Описание изменений производится в формате XML и сохраняется в файле, доступном инструменту миграции. После трансляции динамического

SQL загружаются данные из этого файла и производится корректировка результата. Такой подход позволяет проводить ручные исправления только один раз и в дальнейшем вносить их автоматически.

Однако, на практике такой подход оказался неудобен, так как правка значений происходит параллельно с развитием инструмента трансляции, поэтому для поддержания XML файла в актуальном состоянии требуется много дополнительных ресурсов.

5 Заключение

В ходе выполнения данной дипломной работы были получены следующие результаты:

- Изучены алгоритмы работы с динамически формируемыми строкам.
- Предложена модификация абстрактного анализа, применимая для трансляции динамически формируемых строк.
- Сформулирован ряд ограничений и упрощений, позволяющих упростить задачу трансляции динамически формируемых строк.
- Разработан и реализован механизм автоматизации трансформации динамического SQL из MS-SQL в PL-SQL в рамках проекта по миграции базы данных с MS-SQL Server 2005 на Oracle 11gR2.
- Реализованы следующие компоненты:
  - библиотека абстрактного лексического и синтаксического анализа, расширенного для нужд трансляции;
  - компонента печати новых значений для переменных, участвующих в формировании запроса;
  - компонента статической валидации результатов трансляции динамического SQL.
- Реализован способ ручной доработки результатов трансляции с автоматически воспроизводимым результатом и механизм валидации результатов трансляции.
- Представленный механизм позволил автоматически обработать 45% запросов, в том числе те, что должны преобразовываться в курсор, а также реализовать переименование используемых объектов БД.

5.1 Направления дальнейшего развития

Для решения проблемы с лексическими ошибками было предложено явно вычислять все пути в графе. Для больших графов это не представляется

возможным, поэтому лексические ошибки нужно локализовывать в графе и явно вычислять пути только в небольшом подграфе.

Многие операции, например, синтаксический анализ запросов, можно выполнять параллельно для каждого запроса. Однако, на практике анализ сложных запросов требует большого объёма памяти, что делает параллельный запуск таких задач невыгодным. Необходимо проанализировать возможности для распараллеливания и, если это будет разумным, реализовать.

При печати сложных запросов состоящих из многих литералов возникают проблемы с печатью некоторых символов (бинарные операции, скобки, запятые, пробелы), находящихся на границе литералов или представляющих отдельный литерал. В некоторых случаях удаётся придумать эвристики, но в большинстве всё же требуется ручная доработка. Возможно, эта проблема решится, если все такие символы будут представлены в синтаксическом дереве. Так же стоит попробовать подход предложенный А. Москалём для сохранения истории пораждения при работе с макросами. Это должно упростить обработку сложных литералов.

При печати новых значений для переменных, которые участвовали в формировании запроса, структура которого сильно изменилась после трансляции возникают проблемы, связанные с тем, что в результирующем запросе появляются участки, требующие для своего формирования новых переменных, значения которых зависят от логики компоновки исходного запроса. Автоматическое создание таких переменных и заполнение их значений согласованно с логикой построения запроса является отдельной сложной задачей.

Список литературы

[1] Документация Oracle Database 11g Release 2. http://www.oracle.com/technetwork/database/enterprise-edition/overview/index.html.

[2] Документация по Microsoft SQL Server 2005. http://download.microsoft.com/download/3/4/5/345f3d86-7f16-426e-902f-9b39466dec98/ss05-product_guide.pdf.

[3] Сайт проекта Alvor. http://code.google.com/p/alvor/.

[4] Сайт проекта Java String Analyzer. http://www.brics.dk/JSA/.

[5] Aivar Annamaa, Andrey Breslav, Jevgeni Kabanov, and Varmo Vene. An interactive tool for analyzing embedded sql queries. In Kazunori Ueda, editor, *Programming Languages and Systems*, volume 6461 of *Lecture Notes in Computer Science*, pages 131–138. Springer Berlin / Heidelberg.

[6] Aivar Annamaa, Andrey Breslav, and Varmo Vene. Using abstract lexical analysis and parsing to detect errors in string-embedded dsl statements. In Marina Walden and Luigia Petre, editors, *Proceedings of the 22nd Nordic Workshop on Programming Theory*.

[7] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium (SAS)*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003. Available from http://www.brics.dk/JSA/.

[8] Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. Static analysis of string values. In *Proceedings of the 13th international conference on Formal methods and software engineering*, ICFEM'11, pages 505–521, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] Arjun Dasgupta, Vivek Narasayya, and Manoj Syamala. A static analysis framework for database applications. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 1403–1414, Washington, DC, USA, 2009. IEEE Computer Society.

[10] Сайт проекта PL/SQL Developer. http://www.allroundautomations.com/plsqldev.html.

[11] Kyung-Goo Doh, Hyunha Kim, and David A. Schmidt. Abstract parsing: Static analysis of dynamically generated string output using lr-parsing technology. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 256–272, Berlin, Heidelberg, 2009. Springer-Verlag.

[12] Дистрибутивы и документация по языку программирования F#. http://www.research.microsoft.com/fsharp.

[13] Дистрибутивы и документация по инструменту FsYacc. http://fsharppowerpack.codeplex.com/wikipage?title=FsYacc.

[14] Xiang Fu, Xin Lu, Boris Peltsverger, Shijun Chen, Kai Qian, and Lixin Tao. A static analysis framework for detecting sql injection vulnerabilities. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '07, pages 87–96, Washington, DC, USA, 2007. IEEE Computer Society.

[15] Дистрибутивы и документация по инструменту Graphviz. http://www.graphviz.org/.

[16] Dick Grune and Ceriel J. H. Jacobs. *Parsing techniques: a practical guide*. Ellis Horwood, Upper Saddle River, NJ, USA, 1990.

[17] ISO. *ISO/IEC 9075:1992: Title: Information technology — Database languages — SQL*. 1992. Available in English only.

[18] Soonho Kong, Wontae Choi, and Kwangkeun Yi. Abstract parsing for two-staged languages with concatenation. In *Proceedings of the eighth international conference on Generative programming and component engineering*, GPCE '09, pages 109–116, New York, NY, USA, 2009. ACM.

[19] Сайт платформы .NET. http://www.microsoft.com/NET/.

[20] Документация по языку PL-SQL. http://www.oracle.com/technetwork/database/features/plsql/index.html.

[21] Сайт проекта PHP string analyzer. http://www.score.is.tsukuba.ac.jp/~minamide/phpsa/.

[22] Сайт проекта SwissSQL. http://www.swissql.com/.

[23] Документация по языку Transact-SQL. http://msdn.microsoft.com/en-us/library/ms189826(v=sql.90).aspx.

[24] Сайт проекта SQL Ways. http://www.ispirer.com/products.

[25] Сайт проекта YaccConstructor. http://recursive-ascent.googlecode.com.

[26] Улитин Константин Андреевич. Инструмент реинжиниринга спецификаций трансляций. Дипломная работа, Санкт-Петербургский Государственный Университет. Математико-механический факультет, 2011. Available from http://code.google.com/p/recursive-ascent/wiki/Downloads?tm=2.

[27] Чемоданов Илья Сергеевич. Генератор синтаксических анализаторов для решения задач автоматизированного реинжиниринга программ. Дипломная работа, Санкт-Петербургский Государственный Университет. Математико-механический факультет, 2007. Available from http://code.google.com/p/recursive-ascent/wiki/Downloads?tm=2.