Правительство Российской Федерации

Федеральное государственное автономное образовательное учреждение

высшего профессионального образования

«Национальный исследовательский университет
«Высшая школа экономики»

Факультет Бизнес-информатика

Отделение Программной инженерии

Кафедра Управление разработкой программного обеспечения

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

На тему: Система автоматизированного разделения кода прикладных программ между клиентским и серверным компонентами

|  |
| --- |
| Студентка группы №471ПИСоозарь М.О.\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Соозарь М.О. /«31» мая 2013 г. |
| Руководитель ВКРдоцент каф. УРПО, к.т.н.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Брейман А.Д. /«31» мая 2013 г. |

Москва, 2013

**Аннотация**

Оглавление

[**Введение** 4](#_Toc357771780)

[**Глава 1. Способы и средства для оптимизации работы с БД** 7](#_Toc357771781)

[1. Обзор системы 7](#_Toc357771782)

[2. Методы оптимизации работы с БД 8](#_Toc357771783)

[2.1. Стандартные SQL оптимизаторы 8](#_Toc357771784)

[2.2. Секционирование 9](#_Toc357771785)

[3. Программные продукты для улучшения производительности БД 10](#_Toc357771786)

[4. Актуальность разработки 15](#_Toc357771787)

[5. Задание на разработку 16](#_Toc357771788)

[**Глава 2. Основные принципы работы системы** 18](#_Toc357771789)

[**2.1. Предпосылки: исходный код** 18](#_Toc357771790)

[**2.2.** **Модель исходной системы** 19](#_Toc357771791)

[**2.3.** **Выбор методов для переноса** 19](#_Toc357771792)

[**2.4.** **Схема работы системы** 21](#_Toc357771793)

[**2.5.** **Генерация RMI интерфейса, клиентской и серверной частей** 22](#_Toc357771794)

[**2.6.** **Модификация клиентской части** 23](#_Toc357771795)

[**2.7.** **Модификация серверной части** 23](#_Toc357771796)

[**Глава 3. Модули и библиотеки системы** 25](#_Toc357771797)

[**3.1. Средства разработки** 25](#_Toc357771798)

[**3.2. Javaparser** 25](#_Toc357771799)

[**3.3. Выявление методов** 26](#_Toc357771800)

[**3.4. Генерация RMI классов и интерфейса** 28](#_Toc357771801)

[**3.5. Модификации вызова методов и сборка** 32](#_Toc357771802)

[**Глава 4. Результаты исследований и направление развития** 33](#_Toc357771803)

[**4.1. Эксперименты** 33](#_Toc357771804)

[**4.2. Возможная область применения системы** 34](#_Toc357771805)

[**4.3. Итоги проведённого исследования** 35](#_Toc357771806)

[**4.4. Дальнейшие исследования: синхронизация** 36](#_Toc357771807)

[**4.6. Выделение менее крупных единиц переноса** 37](#_Toc357771808)

[**4.7. Выявление большего количества кандидатов для переноса** 37](#_Toc357771809)

[**Использованные источники:** 38](#_Toc357771810)

# **Введение**

Множество современных приложений используют базы данных для накопления самых разных видов информации, которые могут включать пользовательскую статистику, информацию о счетах, введённые запросы и ключевые слова, - всё то, что необходимо сохранить, получить, обработать для выполнения поставленных задач.

Хотя такие программы могут различаться по своей структуре, решаемым задачам, и визуальному представлению, все они сталкиваются с одинаковыми проблемами:

1. Высокая чувствительность ко времени выполнения.

 В трёхуровневой архитектуре, в рамках которой и будут проводиться исследования, запрос клиента проходит следующий цикл: отправка на сервер обработки клиентских запросов, преобразование в соответствующий запрос к базе данных, передача на сервер базы данных, обработка и получение результата, отправка результата на сервер приложений, обработка и преобразование результата в требуемый клиентом вид, отправка результата и отображение результата в клиентском приложении.

Такая структура позволяет сделать систему более безопасной и надёжной, однако, на сервер приложений, сервер базы данных и канал связи между ними ложится высокая нагрузка. Реляционная база данных предполагает наличие блокировок, то есть пока с данной таблицей выполняется какая-то операция, она становится недоступной для ожидающего потока транзакций.

Клиенты, в свою, очередь, ожидают выполнения своих операций практически мгновенно, а в условиях конкуренции между схожими программными продуктами, скорость может стать ключевым фактором для выбора одной и отказа от других систем.

1. Изначальная неоптимальность кода и, как следствие, растрата ресурсов системы и снижение времени выполнения запросов.

Разработчики фокусируются в первую очередь на том языке, на котором программируется сервер приложений. Запросы к базе данных пишутся так, чтобы получить некоторую долю информации, и затем быстро (не в плане производительности, а в плане написания кода) обработать её. Языки программирования имеют намного больше встроенных средств, чем, к примеру, SQL, и обработчики информации на них писать занимает значительно меньше времени.

Целью данного исследования является создание системы, которая будет уменьшать время выполнения клиентских запросов, за счёт разделения исходного кода сервера приложений на 2 компоненты и переместить часть обработки данных ближе к месту их хранения.

Для достижения данной цели необходимо выполнить ряд задач:

1. Создать систему анализа исходного кода программы. Исходя из набора файлов с кодом сервера приложений, должны выявляться подходящие для преобразования фрагменты кода – получающие информацию из БД, обрабатывающие её и имеющие конечный результат.
2. Создать систему разделения, которая по выбранным методам, создаст классы и методы для удалённого вызова и отредактирует первоначальный код их вызова.
3. Создать окружение для исходных программ, которое будет обеспечивать синхронизацию передаваемых данных и передачу результата, а также будет собирать статистические данные о времени выполнения для последующего анализа результатов.
4. Создать или выбрать одну из существующих баз данных и программу на которой будет проводиться тестирование системы.
5. Провести тестирование: программы до применения системы и после её применения.

При успешных результата экспериментов, с помощью данной системы возможно будет уменьшить время выполнения клиентских запросов в сложных приложениях не прибегая при этом к услугам разработчиков.

В результате система пытается соединить достоинства двух- и трёхуровневой архитектуры, приблизить данные к месту их обработки, распределить нагрузку между сервером БД и сервером приложений, сохранив при этом безопасность и надёжность.

Несмотря на то, что используемые модели архитектуры, построение графа зависимостей, преобразование запросов в хранимые процедуры – применяются достаточно давно и хорошо известны, приложение этих знаний к программам работающим с БД является достаточно новым. На данную тематику вышло 2 публикации: первая – летом 2012 года в журнале, посвящённом большим базам данных «The VLDB journal», и в начале 2013 года в MIT, обе написаны группой из студентов и профессоров MIT и университета Cornell [1,2]. Именно эти публикации и послужили основой для данной работы.

С учётом представленных в вышеуказанных работах результатов, данное направление оптимизации работы приложений вероятно будет активно развиваться, но на данный момент работы по данной тематике практически отсутствуют.

# **Глава 1. Способы и средства для оптимизации работы с БД**

## Обзор системы

Основной задачей разрабатываемой системы является повышение производительности систем состоящий из сервера приложений и сервера баз данных, а именно уменьшение времени между отправкой клиентского запроса и получением соответствующего результата.

В ходе работы система будет анализировать исходный код, выявлять подходящие для переноса в качестве хранимой процедуры участки, и добавлять их в БД.

Первоначально должен производиться анализ исходного кода сервера приложений. Система формирует список методов пригодных для переноса на другой сервер. Подходящими можно считать методы, в которых получается и затем обрабатывается некоторая информаций из БД, и в данных методах модифицируются только поля объектов класса, в котором содержится данный метод.

Также, предварительно будут собраны данные о работе системы, время выполнения различных транзакций. С помощью этого параметра можно будет оценить эффективность работы системы до разделения и после.

По итогам работы анализатора составляется список методов и необходимого набора параметров, которые необходимы для генерации файлов взаимодействия.

 Разделитель по созданному списку выделяет нужные методы, генерирует интерфейс, клиент и сервер для RMI-взаимодействия, классы-результаты для передачи данных.

В код сервера приложений вместо перенесённых строк будут добавлены обращения к сгенерированным классам для удалённого вызова заданных процедур.

Упомянутые библиотеки обеспечивают синхронизацию объектов, передаваемых в качестве параметров и отправку результата. Часть, располагающаяся на сервере БД при получении сигнала вызывает заданную процедуру применив её к переданному в качестве параметра объекту, получив ответ преобразует его в нужную форму и возвращает вместе с изменёнными значениями параметров.

В результате вместо прежнего взаимодействия сервер приложений, обращающийся по JDBC к серверу БД, будет получена более сложная цепочка: сервер приложений – компонент взаимодействия на первом сервере – компонент взаимодействия на втором сервере – вызов метода – возврат результата и синхронизация параметров через компоненты взаимодействия.

Несмотря на добавление передачи параметров из-за того, что часть данных будет обрабатываться внутри БД нагрузка на канал связи между серверами должна снизиться. В целом, прирост производительности ожидается в следствие следующих факторов:

* Снижение количества избыточных пересылок данных. Данные будут пересылаться, обрабатываться и возвращать результат только один раз. Если в перенесённом на другой сервер коде больше чем 1 обращение к БД, то время выполнения сократится минимум на время пересылки данных туда и обратно.
* Использование удалённого вызова процедур вместо обычных запросов.

Для анализа будет взято 2 случая: стандартное приложение без изменений и разделённое приложение.

1. Методы оптимизации работы с БД

Существует множество различных способов улучшения производительности систем работающих с базами данных и самих баз данных. Так как цель разрабатываемой системы именно сокращение времени выполнения клиентских запросов, то далее будут рассмотрены средства и программные продукты с таким же предназначением.

### Стандартные SQL оптимизаторы [3]

В СУБД обычно присутствует стандартный оптимизатор SQL запросов, его цель – подобрать наилучшую стратегию выполнения запроса для выбранных данных. Первоначально присланный запрос отправляется на разбор синтаксическому анализатору, затем производится построение планов выполнения. Выбираемые планы выполнения в некоторых случаях можно скорректировать добавляя специальные команды в запрос или с помощью программных средств.

Оптимизатору необходимо подобрать приемлемо эффективный план в условиях жёстких временных ограничений. Для того, чтобы подобрать максимально эффективный план выполнения, необходимо было бы построить все возможные варианты и затем провести их оценку и выбрать наиболее лучший, но такой подход требует больших временных затрат, поэтому оптимизатор может сгенерировать и выбрать не самую эффективную стратегию.

При оценке планов выполнения учитывается количество вводов-выводов, требуемая память и затраты процессора. Помимо этого, учитывается количество обрабатываемых записей.

После проведения оптимизации план может сохраняться в память или в специальный кэш, чтобы использовать его снова, если поступит тот же запрос. Планы удаляются из памяти, если произошли какие-то существенные изменения в базе, также они могут быть удалены, когда выполняется некая задача, требующая больших объёмов памяти. При изменении настроек БД, весь кэш планов выполнения может быть очищен.

Таким образом, встроенный оптимизатор запросов выполняется всегда и является одним из ключевых средств улучшения производительности БД.

* 1. Секционирование [4]

Секционирование – это разделение хранящихся в БД объектов, например, таблиц на несколько меньших частей, с разным физическим расположением на диске.

Существует 2 основных вида секционирования:

* Горизонтальное. Разные части таблицы содержат разные её строки.
* Вертикальное. Столбцы таблицы находятся в разных файлах. Такой подход может быть оправдан, если в некоторых столбцах хранится большее количество информации, чем в других, а запрашиваются эти данные реже.

Таблицы могут быть разделены по следующим критериям:

1. Диапазон значений. Например, данные за разные периоды времени находятся в разных частях таблицы.
2. Список значений. Принадлежность к одному из разделений определяется по равенству какого-либо параметра заданной величине.
3. Хэш-разделение. Разделение по значению хэш-функций
4. Композитное. Последовательно применённые несколько из вышеперечисленных критериев.

Такой подход может значительно ускорить обработку запросов в больших таблицах. Если блокировки применяются на уровне секций, а не таблиц, то это поможет уменьшить количество конфликтов блокировок, следовательно, транзакции будут меньше ожидать снятия блокировки и выполняться быстрее.

### Программные продукты для улучшения производительности БД

* 1. Продукты компании Embarcadero – DB Optimizer и J Optimizer

Есть категория приложений, в названии которых хоть и присутствуют слова «оптимизация» или «оптимизатор», однако они занимаются графической визуализацией загрузки сервера баз данных: использованием ЦП, оперативной памяти и прочее, - с некоторой долей анализа, который выделяет слабые места в работе текущей системы. То есть цель данного продукта указать *разработчикам* участки кода, которые необходимо исправить.

Примером такого продукта является DB Optimizer от компании Embarcadero. Отличительная черта данного продукта – диаграмма визуальной настройки SQL. Этот модуль позволяет переписывать код запросов, хранимых процедур с помощью выбора в графическом интерфейсе всех необходимых параметров выражения из появляющихся диалоговых окон. Также, данная система замеряет нагрузку на сервер базы данных, визуализирует полученные данные и формирует отчёт о собранных данных с некоторыми рекомендациями. Помимо этого, в DB Optimizer встроен модуль проведения стресс-тестирования SQL-кода.

Разрабатываемая в качестве выпускной квалификационной работы система предполагает наличие модуля, определяющего загруженность системы, однако количество анализируемых показателей может отличаться от системы DB Optimizer. Основное различие предлагаемого решения от программного продукта компании Embarcadero состоит в том, что последнее нацелено именно на удобство визуального редактирования кода разработчиком, в то время как в разрабатываемой системе переписывание кода вручную не предполагается как таковое.

Компания Embarcadero предлагает и другой программный продукт – J Optimizer, который анализирует деятельность приложений разработанных на Java SE и Java EE. По сути данная система осуществляет мониторинг за деятельностью приложения, отслеживает количество используемой памяти и ресурсов ЦП, а также следит за деятельностью отдельных компонентов приложения, таких как JDBC, RMI и так далее.

При рассмотрении данного продукта целью было понять какого рода оптимизацию Java-приложений он производит. Однако, как оказалось, никакой иной деятельности кроме отслеживания большого набора показателей, не происходит.

Сходство разрабатываемой системы и рассмотренных ранее программных продуктов только в наличии модуля мониторинга, однако и тут есть различие. В рамках данной работы для сравнения производительности приложений до и после разделения кода в качестве основного показателя будет использоваться время между отправкой запроса пользователем и получения им ответа.

* 1. Qure Optimizer компании DB Sophic

Qure Optimizer является частью системы Qure управления производительностью БД и приложений работающих с БД. Данный программный продукт производит анализ самой базы данных, в процессе которого создаётся набор рекомендаций, которые проранжированы по степени возможного улучшения производительности. Затем система проводит тестирование, чтобы получить данные о реальных изменениях. На вход оптимизатор принимает файл трассировки SQL, затем применив изменения к копиям объектов БД повторяет активность, указанную во входном файле и сравнивает результаты.

Сравнение результатов происходит по следующим показателям: продолжительность выполнения, количество чтений с диска, чтение из кэша, загрузка процессоров и количество записываемой информации.

В качестве рекомендаций могут быть изменения к схемам, переписывания SQL выражений и тому подобное. При этом каждая рекомендация сопровождается детальным текстовым обоснованием и, если это возможно, сгенерированным файлом, который эту рекомендацию применит к исходной БД.

Могут присутствовать рекомендации по добавлению, изменению или удалению индексов. Удаление индексов применяется только к избыточным или дублирующимся, а не неиспользованным. Для создания таких рекомендаций анализируются не отдельные транзакции, а все произведённые в ходе тестирования операции.

Для рекомендаций по переписыванию отдельных SQL выражений производится также проверка корректности как самого выражения, так и совпадения типа возвращаемого результата с оригинальным. В рекомендациях к схемам и других видах рекомендаций могут присутствовать возможные пропущенные ограничения, неиспользуемые столбцы, отсутствующие выражения с Join, использование Select \* вместо указание конкретно требуемой информации.

Помимо файлов с применением рекомендаций также создаются файлы производящие отмену любого из применённых предложений.

Для первичного или повторного анализа работы системы предоставляется бесплатный продукт Qure Analyzer, который может производить поиск слабых участков в системе опираясь на файл трассировки SQL. Также данная программа может производить сравнительный анализ по нескольким входным файлам.

Данная система управления производительностью автоматически производит очень детализированный анализ существующей базы данных и посылаемых к ней запросов, ищет ошибки в самих принципах построения БД и предоставляет список возможных улучшений, что может значительно повысить производительность системы. Однако это решение не полностью исключает возможность избыточной передачи данных между сервером приложений и сервером БД. В случае, если сервер приложений производит изменения и отправляет результат обратно в базу данная система, вероятно, сможет выявить подобное поведение и предложить использование альтернативного запроса, который изменяет данные прямо внутри БД. Однако в случае, если переработка полученной информации на сервере приложений не даёт информации о том, какие именно данные были отброшены/изменены и затем переданы клиентам, то данная система, вероятно, не сможет предоставить каких-либо рекомендаций.

Qure Optimizer и предлагаемое решение могут использоваться вместе. Оптимизатор улучшит структуру самой базы данных первоначально, а после добавления хранимых процедур могут появиться рекомендации и по их изменению. Однако в таком случае изменять внесённые процедуры придётся вручную.

Сходным функционалом обладает система Database Tuning Advisor для Microsoft SQL Server.

* 1. SQL optimizer for Oracle

Данный программный продукт компании Dell собирает данные о выполняемых выражениях из Системной глобальной области в БД Oracle (*англ. System Global Area, SGA).* Данная область оперативной памяти является общей для всех процессов в базе. Она содержит словарных кэш базы, буферный кэш данных, буфер журнала выполненных действий, разделяемый участок памяти, участок памяти для Java и участок памяти, содержащий пользовательскую глобальную область (*англ. User Global Area, UGA)*.

SQL Optimizer собирает данные не только из базы, но и может анализировать приложение, обращающееся к ней, выделяя из него отдельные запросы.

Для каждого обрабатываемого запроса предлагается несколько альтернативных вариантов, которые могут выполняться быстрее. Помимо этого, проанализировав нагрузку системы в целом могут быть предложены изменения в индексах.

Отличительная особенность данного средства в возможности управлять планами выполнения SQL выражений. Система управления базами данных создаёт несколько планов выполнения для различных объёмов запрашиваемых данных и применяет наиболее подходящий для выполнения запроса.

Несмотря на то, что данные рассматриваемая система собирает автоматически, замену оригинального SQL выражения на один из предложенных вариантов необходимо будет сделать вручную. Улучшение в производительности при применении одной из альтернатив необходимо также выявлять разработчику. То есть для систем с большим количеством разнообразных запросов количество времени, которое необходимо потратить на проверку каждого из вариантов выполнения запросов, будет значительным.

Рассмотренная система может улучшить скорость выполнения отдельных SQL-выражений, однако она не затрагивает логику самого приложения и не поможет сократить количество избыточных пересылок данных по сети. Совместное использование предлагаемого решения и данного программного продукта представляется возможным, так как в исходном коде, разумеется присутствуют SQL-запросы. Таким образом разрабатываемая система будет переносить часть кода на сервер БД, а SQL optimizer сможет улучшать сам текст запросов, если он задан в явном виде.

* 1. PureQuery

PureQuery позволяет писать приложения с использованием статического SQL. Данная система требует начальной подготовки кода к сбору данных – добавление специальной аннотации к SQL запросам передаваемым через JDBC. Для аннотированных методов система автоматически генерирует оптимизированную версию исполнения.

Затем, в процессе выполнения приложения pureQuery сохраняет посылаемые к БД запросы, а также различные данные к ним, например, параметры. Полученные данные сохраняются в специальные файлы и добавляются к БД.

Данный метод подходит для тех задач, где параметры можно чётко фиксировать, иначе возникнет избыточность передаваемых данных и необходимость их дополнительно обрабатывать.

## Актуальность разработки

В большинстве современных систем, работающих с БД, набор параметров к запросам не фиксирован, а потенциально создаваемая избыточность негативно повлияет на производительность системы в целом. Для больших таблиц выигрыш, полученный за счёт использование статического SQL может стать несущественным из-за обработки лишних данных.

Также стоит отметить, что для pureQuery разработчикам придётся переделывать код программы, чтобы система смогла получить данные об отправленных запросах.

Большинство из приведённых методов решения и программных продуктов требуют вмешательства разработчика для проведения оптимизации. Кроме того, большинство из них нацелены только на одну часть системы – сервер базы данных, при этом работы с кодом приложения, в котором могут быть недоработки не производиться практически никакой.

Данная же система делает упор на нахождение некоторых недоработок с точки зрения производительности в сервере приложений. Если исходить из предположения, что разработчику быстрее, проще и удобнее написать несложный SQL-запрос и провести обработку результата с использованием средств языка высокого уровня, следует, что избыточность в запросах вероятнее всего будет.

Разрабатываться данная система будет на Java и для этого же языка программирования. Во-первых, некоторые СУБД поддерживают технологию написания хранимых процедур на Java. Во-вторых, данный язык программирования широко применяется для написания именно серверов приложений.

База данных исходного приложения должна использовать SQL и иметь интерфейс для JDBC подключения. Наиболее вероятно, что будет использоваться PostgreSQL, так как данная система поддерживает написание функций на Java с использованием модуля PL/Java, а также является свободно распространяемой СУБД.

Синхронизацию данных и параметров можно обеспечить двумя способами: передача всех объектов, использующихся в процедуре, в качестве параметров или синхронизация локальных стека и кучи. Первый способ был выбран в силу более простой реализации, нежели во втором варианте, который потребует работы с JVM и возможного внесения модификаций. Если же возникнут случаи, при которых синхронизация через параметры будет невозможна, то второй вариант останется единственным возможным решением.

## Задание на разработку

Ознакомиться с технологией принципами построения системы разделения кода «Pyxis», разрабатываемой группой из MIT и университета Cornell.

Провести исследование методов и программных продуктов, нацеленных на оптимизацию работы приложений с базами данных.

Создать приложение для автоматизированного разделения кода, с целью повысить производительность приложений, работающих с БД, содержащее следующие модули:

1. Анализатор исходного кода, определяющий положение частей приложения после разделения
2. Разделитель кода и генератор классов взаимодействия
3. Библиотеки времени выполнения, обеспечивающие взаимодействие и синхронизацию данных
4. Профилировщик, собирающий данные о нагрузке на систему и времени выполнения клиентских запросов.

Провести испытания программы и сделать сравнительный анализ 2 случаев: стандартная реализация с использование JDBC для отправки запросов и приложение после разделения кода с помощью разработанной системы.

Проанализировав полученные результаты сделать выводы об эффективности подобного подхода, о случаях, в которых выигрыш в производительности от использования системы будет наибольшим и наименьшим.

# **Глава 2. Основные принципы работы системы**

## **2.1. Предпосылки: исходный код**

Основная идея, на которой строится данная система, заключается в том, что исходный код сервера приложений не оптимален и в нём присутствуют определённые конструкции, порождающие избыточную пересылку данных. Примерами таких конструкций могут быть:

1. Последовательное выполнение нескольких запросов для получения одного общего результата.

Подобные конструкции обычно могут быть заменены одним более сложным SQL-запросом.

1. Циклическое выполнение запросов

Часто применяются в случае обновления нескольких строк таблицы. Стандартный SQL не предполагает выполнения циклов, они могут присутствовать в различных расширениях от производителей СУБД, например в PL/SQL.

Наличие подобных конструкций в коде может быть объяснено рядом факторов:

* Больший опыт разработки на высокоуровневых языках программирования, чем на транзакционном SQL
* Меньшие временные затраты на разработку, так как высокоуровневые языки предоставляют возможности для более простой и быстрой разработки, также можно подключить дополнительные библиотеки, если стандартных средств языка недостаточно.

Основной причиной наличия не оптимального кода является тот факт, что разработчику необходимо переводить данные из одной модели в другую, а именно, из объектно-ориентированной модели для клиента и сервера приложений в реляционную для сервера базы данных. Не соответствие принципов хранений данных порождает некоторую избыточность в самой БД и более сложные SQL-запросы, а также получаемый результат запросов требует более сложных преобразований для выделения необходимой информации, чем в случае использования одной и той же модели данных во всех частях системы.

## **Модель исходной системы**

Данная методика рассчитана на приложения с трёхуровневой архитектурой: клиент – сервер приложений – сервер базы данных. Так как программа нацелена на устранение излишних затрат на пересылку данных между 2 и 3 компонентом этой модели, то применять её двухуровневой, будет нецелесообразно.

Способ взаимодействия между клиентом и сервером приложений значения не имеет, однако желательно отсутствие в сообщениях должен SQL кода или его частей, которые потом будут пересланы на сервер БД.

Несмотря на то, что для исследования выбраны приложения, работающие с БД PostgreSQL, конкретный вид используемой приложением СУБД не важен. На сервер приложений и сервер БД накладываются только следующее ограничение:

* Взаимодействие с БД происходит посредством использования JDBC и классов пакета *java.sql* .

JDBC позволяет не привязываться к определённой специфике СУБД, а также при её замене модификации кода минимальны, если вообще присутствуют

## **Выбор методов для переноса**

Критерием для определения метода, как подходящего для переноса может являться использование в нём классов, объектов и методов из пакета *java.sql*.

В качестве единицы переноса выбран метод. Для методов сделано следующее допущение: внутри метода изменяются только поля объекта, от которого вызывается метод.

При переносе функций, отвечающих этому критерию синхронизация копий становится достаточно простой задачей. После выполнения функции удалённо, необходимо подставить результат выполнения удалённой функции вместо результата выполнения обычной, а также объекту источнику присвоить значение объекта, на котором выполнялась функция удалённо.

Сигналом для добавления метода к подходящим для перемещения на другой сервер может быть: наличие объектов принадлежащих к типам *java.sql.Connection, java.sql.ResultSet, java.sql.Statement*, наличие возвращаемого значения принадлежащего к этим же типам, а также выполнение методов: *getConnection, prepareStatement, createStatement, execute, executeQuery, executeUpdate*.

Для выбранных методов необходимо запомнить единицу компиляции, в которой он располагается, название и поля класса и сигнатуру метода. Единица компиляции (Compilation Unit) является наиболее крупным объектом, с которым синтаксический анализатор языка может работать, представляет собой содержимое одного Java-файла.

Из единицы компиляции можно будет извлечь, какие классы необходимо импортировать, чтобы объявить удалённый аналог выбранного метода, а также пакет, в котором располагается данный класс.

Название класса может являться типом объекта, для которого вызывается метод, полям класса необходимо будет передать новые значения, если они изменились в процессе удалённого выполнения. Из сигнатуры метода необходимо выделить параметры, тип возвращаемого значения, наличие выбрасываемых исходным методом исключений.

С помощью всех этих данных можно определить метод-посредник таким образом, чтобы модификации кода рядом с точкой вызова, а также тела метода, были минимальны. Например, если не учесть какое-либо выбрасываемое исключение, то необходимо будет окружить код конструкцией *try/catch* в точке вызова метода, методах посредниках как на клиенте, так и на сервере.

В данной системе удалённый вызов процедуры осуществляется не напрямую, а через классы-посредники, которые представляют собой клиент и сервер интерфейса удалённого доступа.

Под клиентом в данном случае подразумевается сервер приложений, а под сервером – сервер БД. Это обусловлено тем, что в RMI именно клиент инициирует удалённый вызов, что соответствует логике сервера приложений.

## **Схема работы системы**

Система представляет из себя 2 модифицированные копии одной первоначально программы, расположенные на разных серверах. Вызовы методов, перенесённых на другой сервер будут заменены на вызов класса-посредника на клиенте. Данный класс передаёт необходимые параметры и вызывает класс-посредник на сервере, который в свою очередь выполняет заданный метод на заданном объекте и возвращает результат.

Введение удалённого вызова методов (в данном случае посредством Java RMI) требует создания 3 основных объектов:

* Интерфейс, содержащий в себе объявление всех методов, которые будут вызываться удалённо. Данный интерфейс является общим звеном для клиента и сервера. Обращение к серверу выглядит как вызов метода у объекта, реализующего этот интерфейс. Сервер в свою очередь должен иметь класс, реализующий этот интерфейс и транслировать объект этого класса.
* Клиент, направляющий запросы к серверу. В случае данной системы – клиент имеет только статические методы, таким образом не нужно будет создавать лишних объектов в каждый момент вызова удалённой процедуры, соответственно этот объект не нужно будет передавать в методы.
* Сервер, являющийся реализацией интерфейса. Экземпляр класса будет доступен удалённо, результат выполнения методов этого экземпляра будет передаваться клиенту.

2 класса-посредника нужны для обеспечения соединения, также они позволяют не загромождать первоначальный код программы (например добавлением *try{..}catch (RemoteException)* вокруг каждого удалённого вызова), также методы в них отличаются по количеству параметров и типу возвращаемого значения от выбранных на перенос методов. К параметрам добавлен объект-источник если таковой есть.

 Тип возвращаемого значения представляет собой пару (объект с 2 полями) объект-источник и результат выполнения метода. Так как после выполнения метода значения некоторых полей могут измениться, объекты-источники необходимо приводить в соответствие.

Соответственно, по сигнатуре метода и классу, в котором он содержится необходимо сгенерировать класс для представления результата работы удалённого метода. В качестве имени берётся «Res» и к нему приписывается название метода. Типом поля source становится класса, содержащий метод, типом поля result становится результат выполнения метода. Названия полей являются одинаковыми у всех классов-результатов, отличается только название самого класса и тип полей. Также, в данный класс необходимо импортировать класс-источник с указанием пакета, в котором он содержится.

## **Генерация RMI интерфейса, клиентской и серверной частей**

2 вышеописанных класса и интерфейс имеют сходную структуру, поэтому процесс генерации всех 3 объектов выглядит следующим образом:

1. Задать импорт классов, взяв их из единиц компиляции, в которых содержатся переносимые методы, импортировать также классы, содержащие эти методы, так как методы применяются на объектах этих классов. Добавить импорт нескольких RMI классов, например, *java.rmi.Registry.*
2. Задать пакет. Название пакета для всех 3х классов будет одинаковым: «rmicall».
3. Создать объявление классов, задаются имя и модификаторы. Дальнейшие шаги предполагают редактирование содержимого классов.
4. Создание методов соединения на клиенте и на сервере, в интерфейсе такой метод не нужен. Эти методы создают или получают Registry, задают менеджер безопасности (класс SecurityManager), начинают прослушивать определённый порт.
5. Далее происходит процесс генерации переносимых методов. Проходя по списку в каждом классе генерируется сигнатура метода, на этом же этапе генерируется класс-результат для удалённого вызова метода.

В теле метода на клиенте происходит удалённый вызов метода и возвращение результата.

В теле метода на сервере к объекту-источнику, применяется метод. Затем создаётся пара-результат, в которую идёт результат работы метода и объект-источник.

Методы добавляются в списки членов классов или интерфейса.

1. Классы или интерфейс добавляются в соответствующую единицу компиляции, которая записывается в файл с названием, соответствующим имени класса или интерфейса.

## **Модификация клиентской части**

Модификация клиентской копии программы должна обеспечить согласование объектов-источников до выполнения удалённого запроса, а также предоставлять результат выполнения метода (если такой имеется) в первоначальную точку вызова.

Вызовы выбранных методов должны быть заменены на получение результата работы метода по RMI. Создаётся соответствующий объект-пара, которому присваивается результат работы метода. Сам вызов метода заменяется на доступ к полю *result*, добавляется присвоение объекту-источнику значения поля *source* класса-результата.

Также, к первоначальным файлам классов сервера приложений добавляются сгенерированный RMI-интерфейс и непосредственно класс с RMI-клиентом.

## **Модификация серверной части**

Модификация серверной части, за исключением добавления туда класса для RMI взаимодействия и RMI-интерфейса, не требуется, так как там происходит только вызов методов от заданных объектов.

Таким образом, серверная часть представляется из себя копию классов первоначального сервера приложений, RMI-интерфейс и класс представляющий собой сервер для удалённого вызова методов. Последний также является запускаемым.

# **Глава 3. Модули и библиотеки системы**

## **3.1. Средства разработки**

В качестве целевого языка программирования была выбрана Java из-за её распространённости.

Данная система разделения кода также написана на Java.

Для семантического разбора исходного кода была выбрана библиотека *Javaparser*, особенности которой описаны в разделе 3.2.

Сборка клиентского и серверного приложения будут осуществляться с помощью Apache Maven, так как данная система умеет обрабатывать зависимости и может загрузить недостающие библиотеки из центрального репозитория.

В качестве БД для тестирования системы была выбрана СУБД PostgreSQL. Данная система является свободно распространяемой и обладает большим функционалом, в том числе поддержкой процедур на высокоуровневых языках программирования.

## **3.2. Javaparser**

Большая часть классов созданной системы так или иначе используют функциональность библиотеки *javaparser.* Данная библиотека позволяет анализировать исходных код на Java, строит абстрактное синтаксическое дерево (АСД), позволяет модифицировать вершины этого дерева, а также создавать новые узлы пользуясь шаблонами конструкций.

Как было описано в предыдущей главе, наиболее крупной обрабатываемой структурой является единица компиляции, которая выстраивается на основе *.java-*файла.

Парсер содержит классы для обработки всех конструкций языка, а также классы-визитеры (Visitor), в которых присутствуют методы для просмотра различных типов узлов (Node). Классы, представляющие конструкции языка, являются наследниками класса Node. Прямой переход между узлами невозможен, что в некоторых случаях осложняло задачу. Так, например, если в данный момент просматривается метод, то нельзя получить узел уровнем выше – то есть класс, в котором он содержится.

Для того, чтобы посетить узел какого-либо типа, необходимо создать класс-наследник одного из Visitor’ов и переопределить метод для соответствующего типа узлов.

Существуют несколько основных видов конструкций:

* Объявление (Declaration) – для выражений import, объявления классов, полей, методов.
* Выражение-конструкция (Statement) используется для условных операторов, блоков *try-catch*, циклов и блоков выражений. Блоком выражений является, например, тело метода
* Простое выражение (Expression) – бинарные и унарные операторы, вызов метода, доступ к полю, присваивание. Также, одно из наиболее полезных при генерации кода Name expression. В случае, когда часть какой-либо конструкции компоновать из объектов было достаточно сложно, часть выражения можно было заменить подобной конструкцией, использовав её как выражение-строку.

Если в одном из методов, просматривающих узел, есть обработка конструкций какого-то вида, то переопределённые методы для этих конструкций не буду вызываться, если вызов не задан явно.

Javaparser изначально написан для версии языка 1.5, но также совместим с 1.6, грамматика Java 7 не поддерживается, это накладывает ограничение на код программы, которую необходимо разделить.

## **3.3. Выявление методов**

Из-за особенностей javaparser получать метод и необходимый к нему набор данных приходится в 2 этапа.

Как уже говорилось ранее, во многих случаях невозможно перейти от вершины, относящейся к внутренней части метода к самому методу. Также нет возможности поиска по набору выражений, чтобы узнать, присутствует ли там конструкция заданного типа.

Последовательный перебор всех конструкций в рамках одного метода возможен, однако выделить определённый тип выражения будет достаточно сложно. Пример сложной конструкции:

*Int a= SomeClass.b + (++j) \*o.CallSomeMethod(parameter);*

Данная строчка содержит: бинарные выражения (операторы + и \*), обращение к полю (*SomeClass.b),* вызов метода (*o.CallSomeMethod(parameter)*, унарное выражение (++j), объявление переменной (int a) и выражение присваивания (=).

Данная конструкция получит тип объявление переменной (*VariableDeclarationExpr).* Для того, чтобы получить выражение типа вызов метода (*MethodCallExpr)* путём перебора всех выражений в рамках метода, необходимо будет переопределить методы visit для всех вышеназванных типов конструкций, а так как видов конструкций, частью которых может быть вызов метода достаточно много, то эффективнее будет разбить данный процесс на 2 более простых этапа. К тому же переопределение будет частично повторять возможности самого *javaparser*.

В данном случае необходимо воспользоваться тем, что каждая вершина АСД хранит в себе номер начальной и номер завершающей строки. Так как АСД строится для единицы компиляции, то оба шага должны выполняться последовательно на *одном и том же* файле.

1. Необходимо обойти все вершины типа «вызов метода». Если вызываемый метод принадлежит к методам из списка JDBC методов, то запоминается номер начальной строки.
2. Производится обход вершин «объявление метода» и «объявление класса». Если номер строки лежит в интервале [*номер\_строки\_начала\_метода; номер\_строки\_завершения\_метода*], вершина «объявление метода», добавляется в список. Для классов такая же процедура.

Данная функциональность не может быть реализована в рамках одного класса, так как *javaparser* обходит дерево в глубину.

В результате мы получаем списки методов, соответствующих им единиц компиляции, классов, всё то, что необходимо для генерирования классов взаимодействия.

## **3.4. Генерация RMI классов и интерфейса**

Возможности *javaparser* по генерации классов сводятся лишь к тому, что можно инстанциировать классы, представляющие собой различные виды выражений. Не смотря на то, что в классах есть несколько конструкторов, кроме конструктора по умолчанию, создание объектов некоторых типов выглядит достаточно громоздко из-за наличия в параметрах списков объектов. Для создания подобных объектов приходится создавать минимум один, зачастую несколько списков, содержащих в себе всего по 1 элементу.

На начальном этапе рассматривалась возможность генерации классов по шаблону путём работы со строками, однако несмотря на недостатки *javaparser* при создании классов, использование его функциональности значительно упростило процесс генерации классов.

Для того, чтобы сгенерировать успешно компилирующийся класс с корректным функционалом необходимо понимать структуру создаваемого объекта. Фактически, создаётся АСД.

Как уже упоминалось ранее, синтаксическое дерево целиком представляет из себя единицу компиляции.

Пусть единица компиляции будет вершиной дерева. На следующем уровне будут 3 узла, представляющие собой импорт классов и пакетов, непосредственно пакет (а точнее, его название) и объявление класса или интерфейса.

Compilation Unit



Class or interface declaration

Import

Package

Таким образом, изначально создаются 3 пустые единицы компиляции: для интерфейса, для RMI клиента и сервера.

1. Импортируются все пакеты, которые импортированы в классах, содержащих методы для переноса на БД. Дополнительно, необходимо импортировать следующие классы и интерфейсы из пакета *java.rmi: Remote, RemoteException, Registry и LocateRegistry, –* добавляются во все 3 объекта, на сервер также добавляем *UnicastRemoteObject.*

Дополнительно необходимо импортировать все классы проекта с указанием их пакетов.

1. В качестве пакета задаётся “rmicall”. Название значения не имеет, главное, чтобы у всех 3х классов он был одинаков.
2. Задаётся название и модификаторы генерируемого класса или интерфейса. Дальнейшие действия будут разветвлением поддерева класса. Также необходимо задать классы и интерфейсы, от которых пронаследуются генерируемые. Класс-сервер будет наследником интерфейса
3. Объявление полей. Поле присутствует только у класса-клиента, его тип – Registry. К этому полю будет обращение в каждом методе класса, а инициализироваться оно должно только 1 раз – при поиске удалённого соединения.
4. Создание методов соединения на клиенте и на сервере. В указанных методах присутствует установка *SecurityManager,* если он не задан, а также инициализация объекта типа Registry, в случае клиента – он ищет соединение, а в случае сервера – создаёт его, также задаётся порт, по которому будут связываться части программы.
5. Создание копий переносимых методов и классов-результатов.

Имена методов во всех 3 классах будут совпадать call+имя\_переносимого\_метода. К параметрам исходного метода добавляется объект класса, в котором содержится метод, также у методов совпадают с исходным выбрасываемые исключения.

Типом возвращаемого значения у всех методов будут пары <объект-источник, возвращаемое значение метода>, более подробно описанные в Главе 2 п.4.

Так как у интерфейса нет тела метода, то в этот момент объект типа «декларация метода» добавляется к списку членов интерфейса.

Тело метода-клиента представляет собой 3 конструкции: объявление объекта-результата, *try…catch* блок и возврат результата (return). Удалённый вызов процедуры может выбросить исключение типа *RemoteException*, поэтому его необходимо окружить соответствующим *try…catch*-блоком.

В теле метода на сервере инициализируется объект-пара, затем полю *result* этого объекта присваивается результат выполнения метода на переданном в параметрах объекте, соответственно полю *source* присваивается значение этого объекта после выполнения метода. Данный метод возвращает сгенерированный объект-пару.

Пример: пусть в классе SomeClass для переноса выбран метод со следующей сигнатурой:

 *public String[] getAllNames(int start\_pos) {тело метода}*

Для данного метода будет сгенерирован класс-результат:

 *public class ResgetAllnames{*

 *public SomeClass source;*

 *public String[] result;*

*}*

В интерфейсе будет создана следующее объявление метода:

*public ResgetAllnames callgetAllnames(Some class obj, int start\_pos) throws RemoteException;*

В классе RMI клиента метод будет выглядеть следующим образом:

*public static ResgetAllnames callgetAllnames(SomeClass obj, int start\_pos){*

*Resgetallnames res=null;*

*try{*

 *res=rmivar.callgetAllNames(obj, start\_pos);*

*}catch(RemoteException e){*

 *e.printStackTrace();*

*}*

*return res;*

*}*

*rmivar* – переменная типа *RMIinterface* и является полем класса-клиента.

На сервере метод будет выглядеть следующим образом:

*public ResgetAllnames callgetAllnames(SomeClass obj, int start\_pos) throws RemoteException{*

*ResgetAllnames res=new ResgetAllnames();*

*res.result=obj.getAllnames(start\_pos);*

*res.source=obj;*

*return res;*

*}*

Названия методов на клиенте и сервере совпадают только для удобства. Класс-клиент не является наследником RMI интерфейса, все методы у него статические.

Название параметров у исходного метода и сгенерированных совпадают, так как были взяты из объявления самого метода.

В серверный класс также добавляется метод main, содержащий вызов метода на установку соединения.

1. Сгенерированные методы добавляются в класс, класс добавляется в единицу компиляции и затем содержимое каждой единицы компиляции записывается в файл с названием, соответствующим названию класса или интерфейса.

## **3.5. Модификации вызова методов и сборка**

Модификации клиентского кода описаны в главе 2 п.6. Перед выполнением модификаций, исходный код копируется в папку, в которой хранится код RMI сервера. Файлы с модифицированным кодом размещаются вместе с классами RMI клиента. Далее генерируется xml-файл для сборки кода в 2 jar-файла, при этом в архиве для серверной части исполняемым указывается сгенерированный класс RMI-сервера. Далее вызывается maven, который по описанной в XML-файле спецификации собирает архивы.

Таким образом, результатом работы системы будут 2 исполняемых jar-файла, один для запуска на сервере приложений, другой – на сервере баз данных. При этом jar на сервере БД должен запускаться первым, так как он создаёт возможность удалённого подключения.

# **Глава 4. Результаты исследований и направление развития**

## **4.1. Эксперименты**

Обозначенные в предыдущих главах допущения касательно переносимого кода не позволяют использовать какие-либо стандартные тесты для проверки работы систем, взаимодействующих с БД.

Был разработан собственный набор тестов и 2 программы, в которых присутствуют ошибки, обозначенные в главе 2 п.1, а именно: последовательное выполнение нескольких различных запросов к БД с целью получения одного результата и выполнение запросов в цикле.

Для большинства запросов видно небольшое улучшение в производительности, однако эти улучшения достаточно незначительны и могут считаться в рамках статистической погрешности.

Для запросов в циклах улучшения в производительности более заметны, хотя по-прежнему не очень велики.

При обоих видах запросов получили пусть и не самые значительные, но всё же улучшения.

Стоит отметить, что эти результаты получены в модели с ограничениями на переносимые методы.

В дальнейшем предполагается исследовать поведение системы при необходимости синхронизировать объекты с достаточно большим количеством полей.

Поскольку переносятся достаточно большие участки кода данные результаты можно считать корректными только для мало- или средненагруженных серверов БД.

## **4.2. Возможная область применения системы**

Вышеописанные ошибки могут встречаться при разработке систем с ORM, которые предоставляют доступ к БД в виде объектов. При создании подобных систем разработчик не всегда хорошо понимает их устройство и из-за этого может, допустим, не применять методы к коллекции целиком, а выполнять обработку для каждого отдельного объекта.

Однако для того, чтобы система могла использоваться в реальных условиях, необходимо будет провести дальнейшие доработки: снятие ограничений на переносимые методы, обеспечение синхронизации любых, в том числе и сложных объектов, содержащих примитивы синхронизации и данные, разделяемые с другими объектами.

## **4.3. Итоги проведённого исследования**

 В рамках выпускной квалификационной работы необходимо было создать систему, которая уменьшала бы время выполнения клиентских запросов за счёт выполнения некоторых участков кода на том же сервере, где располагается БД.

Такой подход может сократить время на обработку запроса за счёт устранения излишних пересылок данных между сервером базы данных и сервером приложений.

В рамках данного исследования была разработана система, которая:

1. Анализирует исходный код программы и выявляет методы, подходящие для выполнения на другом сервере.
2. В процессе работы генерирует классы, обеспечивающие взаимодействие компонент системы по RMI, а также классы-результаты для запросов.
3. Автоматически компилирует созданные классы и собирает Jar файлы для выполнения на сервере БД и сервере приложений.

В результате экспериментов было выявлено, что временя выполнения запросов, в которых выполнялось несколько последовательных обращений к БД, сокращалось незначительно. Для того, чтобы установить является ли это улучшение значимым или оно в рамках статистической погрешности необходимо провести больше экспериментов. Сокращение времени выполнения запросов, содержащих циклические обращения к БД, было более существенным.

В созданной модели присутствовал ряд существенных ограничений на исходный код разделяемого приложения. Для того, чтобы система могла полноценно применяться к реально работающим программам, необходимо провести дальнейшие исследования способов сокращения времени выполнения клиентских запросов.

## **4.4. Дальнейшие исследования: синхронизация**

Возможен способ синхронизации программ на клиенте и сервере целиком. Для этого необходимо будет внести некоторые модификации в JVM и передавать между частями системы локальные стек и кучу.

 **4.5. Дальнейшие исследования: хранимые процедуры**

Первоначально в рамках данного исследования предполагалось использование процедур на Java вызываемых из PostgreSQL.

PL/Java позволяет выполнять процедуры, написанные на Java, как функции в СУБД PostgreSQL. Для этого необходимо зарегистрировать данную функцию в базе, указав возвращаемый результат, имя и путь к Jar-файлу, класс и название метода в этом классе.

В процессе исследования от применения этой технологии пришлось отказаться по следующей причине:

PostgreSQL запускает отдельную копию JVM, в которой выполняется содержимое архива. Осталось непонятным, как и возможно ли получить доступ к выполняющемуся там извне. Подгружать функции из архива означает, что данные в них будут не синхронизированы.

Также для параметров функции необходимо указывать тип в терминах SQL, то есть сложные объекты, возможно, придётся передавать списком их полей.

Такая возможность была бы полезна для методов или фрагментов методов, которые высчитывают какие-либо показатели и при этом не изменяют никаких полей объектов.

В данном исследовании представлялось необходимым использовать или удалённый вызов методов, или вызов хранимых процедур на Java. При этом первый способ оставляет больше возможностей по синхронизации и манипулированию с данными.

## **4.6. Выделение менее крупных единиц переноса**

В данной системе единицей переноса был метод целиком, однако в методах кроме обращений к БД могут выполняться какие-то дополнительные функции, вычисление которых можно не переносить на сервер БД, а оставить в текущем методе. В таком случае, необходимо перенести только отдельные выражения из метода. Для этого надо проанализировать зависимости по управлению и по данным, чтобы выявить необходимые для корректного выполнения запроса переменные.

## **4.7. Выявление большего количества кандидатов для переноса**

На данном этапе система рассматривает только методы, содержащие в себе какие-то части из определённого набора выражений, связанные с обращением к БД по JDBC. Однако, если выделить методы, занимающиеся только работой с базой и добавить их вызов в группу конструкций, которые являются индикаторами взаимодействия с БД, то количество методов-кандидатов увеличится.

Однако, если давать методу статус работающего с базой по наличию только 1 выражения, то таким способом в списке могут оказаться все методы исходной программы.

Следовательно, необходимо будет разработать критерий причисления метода, к группе работающих с БД, а затем анализировать выражения вокруг вызова этого метода, с помощью графа зависимостей программы [5] дабы выявить целесообразность переноса данных конструкций на другой сервер.

# **Использованные источники:**

1. Cheung, A., *Automatic Partitioning of Database Applications*. [Электронный ресурс] /

 Cheung, A., Aurden, O., Madden, S., Myers, A.C. *The VLDB Journal,* vol 5, 2012

<http://vldb.org/pvldb/vol5/p1471_alvincheung_vldb2012.pdf> [дата обращения 24.05.13]

1. Cheung, A., *StatusQuo: Making Familiar Abstractions Perform using Program Analysis /* Cheung, A., Madden, S., Myers, A.C., Solar-Lezama, A, 2012.

MIT’s Computer Science and Artificial Intelligence Laboratory (CSAIL) website: <http://people.csail.mit.edu/akcheung/papers/cidr13.pdf> [дата обращения 13.01.13]

1. Nevarez, B. *The SQL Server Query Optimizer*. [Электронный ресурс] / *Simple talk journal,* 2011.

<https://www.simple-talk.com/sql/sql-training/the-sql-server-query-optimizer/> [дата обращения: [20.04.13]

1. Swaminathan, G. *Strategies for Partitioning Relational Data Warehouses in Microsoft SQL Server.* [Электронный ресурс] / Guy W., Ozer St., Marathe Ar., Miller J, Ruthruff M., Goldstein B. *Microsoft TechNet, 2005.*

<http://technet.microsoft.com/en-us/library/cc966457.aspx>

1. Horwitz, S. *Interprocedural Slicing Using Dependence Graphs* / Horwitz, S., Reps, T., Binkley, D.[Электронный ресурс] *ACM Transactions on Programming Languages and Systems, Vol. 12, No. 1*, *1990* .