Санкт-Петербургский Государственный Университет
Математико-механический факультет

Кафедра информационно-аналитических систем
Направление 02.04.03 «Математическое обеспечение и администрирование информационных систем»
Профиль «Информационные системы и базы данных»

Рахимзянов Данис Ильшатович

Автоматический рефакторинг для целостной оптимизации Java-приложений

Магистерская диссертация

Научный руководитель:
д. ф.-м. н., профессор Новиков Б. А.

Рецензент:
Ведущий руководитель проекта в T-Systems RUS Порошин И. А.

Санкт-Петербург
2016

SAINT-PETERSBURG STATE UNIVERSITY

Department of Analytical Information Systems
Program «Software and Administration of Information Systems»
Profile «Information Systems and Databases»

Rakhimzyanov Danis

Automated refactoring for holistic optimization of Java applications

Graduation Thesis

Scientific supervisor:
professor Novikov Boris

Reviewer:
Senior project manager at T-Systems RUS Poroshin Ilya

Saint-Petersburg
2016

Оглавление

Введение

Поддержание программного кода в приличном состоянии зачастую является непростой задачей. Безусловно, существуют различные процедуры и техники для этого, начиная от руководств стиля (style guide), которым должны следовать разработчики, заканчивая статическими анализаторами и строгой инспекцией кода (code review).

В данной работе нас интересует класс приложений, в которых используются реляционные базы данных. Не у каждого программиста есть отчётливое представление о работе с системами управления базами данных. Тестируя программный код на небольших объёмах данных, разработчик может получить впечатление, что работа с СУБД практически не отличается от работы с коллекцией объектов в оперативной памяти. Широко используемая парадигма объектно-ориентированного программирования и различные `Object Relational Mapping`-библиотеки хорошо вписываются в теорию «идеального» кода, но на практике, на реальных объёмах данных, такое слепое следование общепризнанным концепциям программирования может создать значительные проблемы с производительностью, что можно видеть в работах [1] и [2]. Мы хотим проектировать приложения так, чтобы работа с базами данных была эффективна. Как этого добиться? Об этом и пойдёт речь далее.

Рефакторинг – неотъемлемая часть культуры программирования, и подразумевает как рутинные задачи, так и сложную интеллектуальную аналитическую работу. Многие разработчики ищут золотую середину между идеальным кодом и «техническим долгом». Таким образом, тема автоматического рефакторинга набирает всё большую популярность в виду роста объёма исходного кода приложений.

Мы задались вопросом, можно ли каким-то способом повысить эффективность взаимодействия с БД, не переписывая приложение «с нуля». Различные среды разработки, такие как `IDEA` и `Eclipse` предлагают мощные инструменты для рефакторинга кода, но они не касаются анализа и оптимизации фрагментов кода, содержащих SQL-запросы. В виду отсутствия подобных решений, было решено исследовать возмож-

ность анализа и автоматического рефакторинга вычислительно неэффективных фрагментов программного кода.

В связи с тем, что наша цель – повысить эффективность приложения за счёт улучшения взаимодействия с БД, в нашей работе рассматриваются следующие задачи:

- Определить и уметь находить вычислительно неэффективные фрагменты кода;
- На основе статического и динамического анализа выдавать советы по улучшению исходного кода;
- Производить трансформацию (рефакторинг) в автоматическом режиме;

Что касается первой подзадачи, то было решено начать с простых вещей, например, однотипных запросов в цикле, постепенно переходя к анализу более сложных случаев. При решении второй подзадачи под динамическим анализом подразумевается замеры времени выполнения методов и общего времени выполнения приложения, а также построение и простейший анализ «графа вызовов». Третья подзадача является не только самой сложной по реализации, но и самой важной. Для её реализации было решено использовать open-source библиотеку Java Parser [9], которая переводит исходный код приложения в абстрактное синтаксическое дерево.

Для решения задач, были выбраны следующие технологии: драйвер `JDBC` и СУБД `Oracle 11G`. Версия `Java` не так принципиальна на данном этапе нашего исследования, но использовался `JDK` версии 1.8.

1. Постановка и анализ задачи

В данной главе мы детально рассмотрим задачи, намечанные во введении, а именно, приведём общее описание процесса автоматического рефактиринга программного кода и каждой из подзадач, касающихся синтаксического анализа исходного кода, поиска вычислительно неэффективных фрагментов и оценки производительности до и после процесса трансформации.

1.1. Общая идея

Основная идея нашего исследования – проверить возможность автоматического рефакторинга вычислительно неэффективных фрагментов в коде приложений, использующих реляционные базы данных. Для этого была спроектирована и реализована внутренняя структура для хранения информации о методах, цикла и SQL-запросах. Эта информация заполняется по мере синтаксического разбора исходного кода анализируемого приложения. На основе этой внутренней структуры производится поиск и трансформация «проблемных» шаблонов. Таким образом, был спроектирован алгоритм автоматического рефакторинга, состоящий из нескольких этапов:

1. Проверка работоспособности приложения и замеры времени выполнения;
2. Разбор исходного кода и построение абстрактного синтаксического дерева;
3. Синтаксический анализ методов, циклов и содержащихся в них SQL-запросов;
4. Поиск вычислительно неэффективных фрагментов программного кода и «выдача» советов по улучшению кода;
5. Рефакторинг проблемных фрагментов;

6. Итоговая проверка работоспособности и замеры времени выполнения;

На рисунке 1 в общих чертах изображён план всех этапов процесса рефакторинга. Мы рассмотрим подробнее каждый из них в следующих главах. Внутренняя структура, в которой хранится вся информация о методах, SQL-запросах и циклах представлена в главе 1.3.

1.2. Метрики и оценка качества

Существуют различные метрики для программного кода: количественные (SLOC), метрики сложности потока управления программы и/или данных (цикломатическая сложность [6] и её вариации, например, [7]), метрики надёжности и другие. На наш взгляд, наиболее репрезентативны из них методы, основанные на анализе управляющего графа программы. Любая работа должна быть целесообразной и рентабельной, особенно если это касается рефакторинга. Для оценки качества было решено использовать следующие методы:

1. Цикломатическая сложность;
2. Время работы методов, содержащий SQL-запрос;
3. Общее время выполнения приложения;
4. JUnit-тесты для проверки бизнес-логики;

Выбор данного набора можно рассматривать как комплексную метрику для оценки произведённого нами рефакторинга, так как по отдельности они могут быть необъективными и нерепрезентативными. Что касается цикломатической сложности, то это количество линейно-независимых «путей» через исходный код. Так как наш рефакторинг SQL-фрагментов не всегда подразумевает избавление от циклов и других ветвлений, то остаётся только следить за тем, чтобы сложность приложения не увеличивалась (но она может и уменьшаться, например, в случае коррекции INSERT-SELECT запросов). Таким образом,

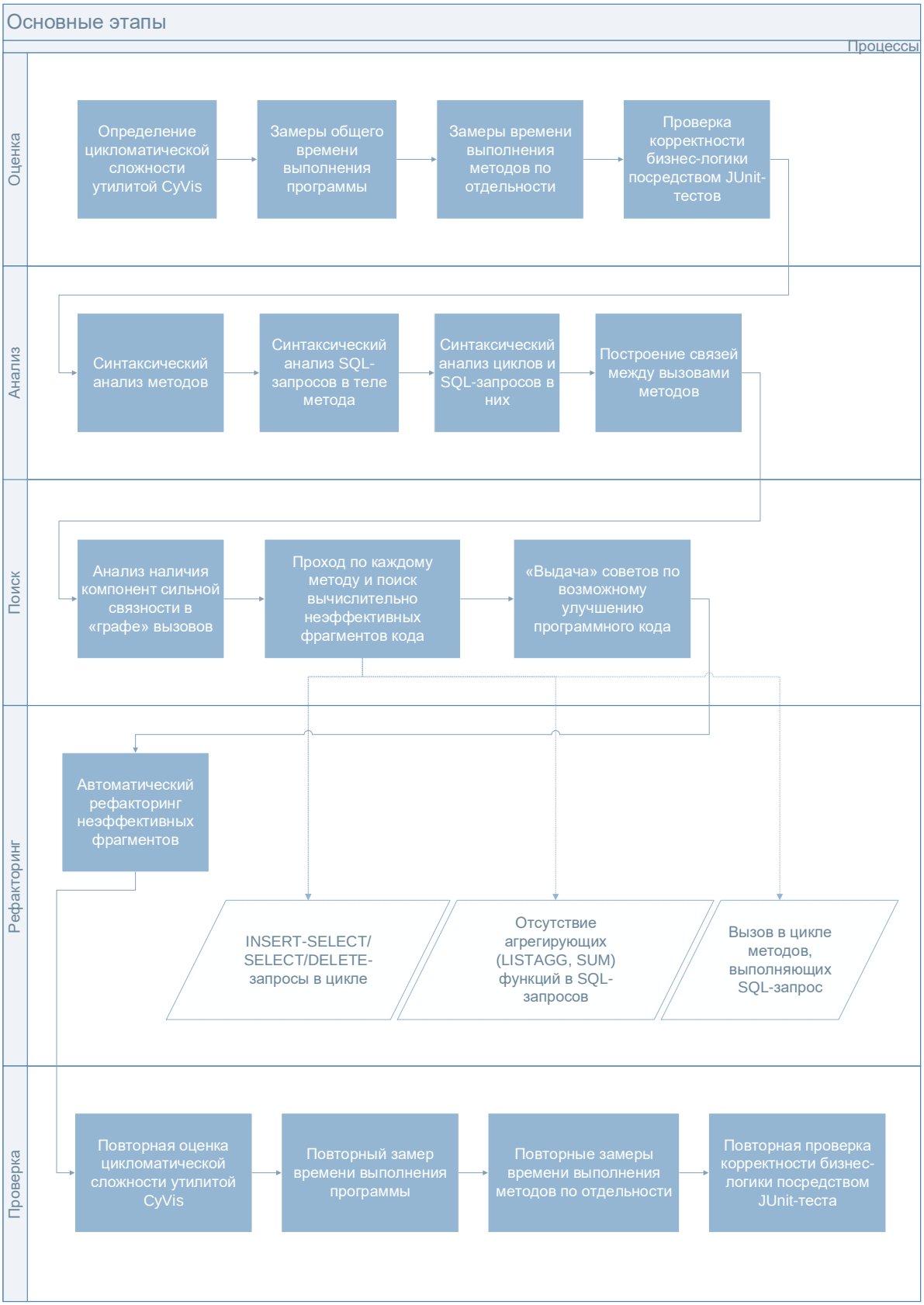

Рис. 1: Общая идея

оценка цикломатической сложности скорее будет играть роль при проведении рефакторинга более сложных синтаксических конструкций.

Для замера времени работы методов, содержащих SQL-запросы придётся либо вручную встраиваться в исходный код, например, посредством библиотеки `log4j` и/или `Java Interceptors`, либо анализировать вывод профайлеров, например, входящих в IDE. В [1] авторы анализировали историю запросов в базу данных `Oracle`, но на наш взгляд самодостаточные SQL-оптимизаторы справятся с анализом логов лучше.

Оценка работоспособности до и после рефакторинга должна быть произведена, например, посредством JUnit-тестов. Из совокупности этих четырёх оценок можно будет судить об успешности автоматического рефакторинга. Если цикломатическая сложность и результаты тестов должны оставаться на том же уровне, то в идеальном случае мы ждём уменьшение времени выполнения методов, содержащих SQL-запросы.

Для замера цикломатической сложности и других метрик программного кода существует как коммерческие, так и бесплатные утилиты (Eclipse metric plugin [12], Checkstyle [13]). Наше внимание привлёк инструмент `CyVis` [11], который позволяет наглядно увидеть оценки сложности каждого метода/класса.

На данный момент оценка производительности производится в «ручном» режиме, но ничего не мешает в будущем автоматизировать также и этот процесс.

1.3. Разбор исходного кода

Синтаксический анализ программного кода – не самая благородная задача, которая отсылает нас к вопросу функционирования компиляторов. Регулярные выражения несколько примитивны для этой задачи в виду «оторванности» от семантики разбираемого текста, и как следствие, отсутствует возможность корректного разбора вложенных конструкций. Следующей идеей было использование грамматики языка Java для генерации парсера посредством `flex`, `bison` или `ANTLR`. Но это был бы слишком «низкий» уровень разбора исходного кода, когда

было бы трудно воссоздать связь между кодом и бизнес-логикой.

Альтернативой вышеперечисленному является анализ байт-кода, в который транслируется исходный текст программы. Однако этот метод требует соответствующего технического бэкграунда в области работы виртуальной Java-машины. В конечном итоге было решено использовать библиотеку `Java Parser` [9], которая трансформирует исходный код Java-приложения в абстрактное синтаксическое дерево (далее – АСД). Грубо говоря, это такое дерево, узлы которого – операторы языка программирования, а листья – соответствующие им операнды. Таким образом, мы можем обрабатывать вложенные конструкции, а в каждом узле этого дерева мы можем получить фрагмент кода, в котором, в свою очередь, можем с помощью регулярных выражений и сторонних библиотек найти и обработать необходимые синтаксические конструкции.

Отметим, что существует альтернативная библиотека `Roaster` [10], которая также даёт возможность парсинга исходного кода, но она более всего удобна для генерации Java-классов «с нуля».

Сам процесс парсинга выглядит относительно просто:

1. Реализуем абстрактный класс `VoidVisitorAdapter`, переопределив в нём метод `visit`. Данный метод содержит всю логику по выделению неэффективных фрагментов кода и их коррекции;

2. Создаём для файла с Java-классами `CompilationUnit`-объект;

3. «Обходим» каждый метод в `CompilationUnit`-объекте посредством ранее реализованного класса;

При обходе каждого метода у нас будет доступ к его содержанию как в виде фрагмента кода, так и в виде узлов АСД. Итерировать между узлами можно посредством методов и классов, входящих в библиотеку, например, `getParentNode` и `getChildrenNodes`. Реализацией класса узлов (`Node`) являются различные `Statement`-классы (например: `ForStmt`, `WhileStmt`, `ExpressionStmt` и другие) с помощью которых можно обрабатывать различные синтаксические конструкции.

Что касается первого пункта, то было реализовано следующее: при проходе по `Statement`'ам в теле текущего метода запоминается инициализация переменных (например, `String`, `Integer`), коллекций (`Map`, `List`, `ResultSet`), а также циклов (`For`, `While`). Каждый `Statement` проверяется на наличие SQL-выражения. Парсинг SQL-запросов был выполнен посредством регулярных выражений, так как это было наиболее оптимально с точки зрения простоты имплементации. Но были предприняты попытки использования фреймворков для генерации SQL-запросов из наших вспомогательных структур, которые в целом закончились успешно.

На рисунке 2 изображены Java-классы нашей вспомогательной структуры и их отношения. Во время анализа программного кода данные объекты инициализируются и далее используются для поиска «проблемных» фрагментов в исходном коде.

Далее разберём каждый из этапов немного подробнее.

1.4. Синтаксический анализ методов

Как было сказано ранее, мы можем рассматривать тело методов как последовательную структуру и итерировать по ней. Таким образом, разбор программного кода начинается со следующих шагов:

1. На вход нашему приложению подаётся путь до файлов с исходными кодами исследуемого Java-приложения;

2. Далее мы рассматриваем каждый Java-класс и методы в них, сохраняем информацию о них во вспомогательные структуры, в объекты типа `Method`. Данный процесс включает в себя анализ SQL-выражений и циклов, а также сохранение информации об инициализации Java-переменных и их связи с аттрибутами SQL-выражений, если таковые имеются;

3. После первичного анализа всех методов, проводится анализ вызовов методов (объекты типа `MethodCall`);

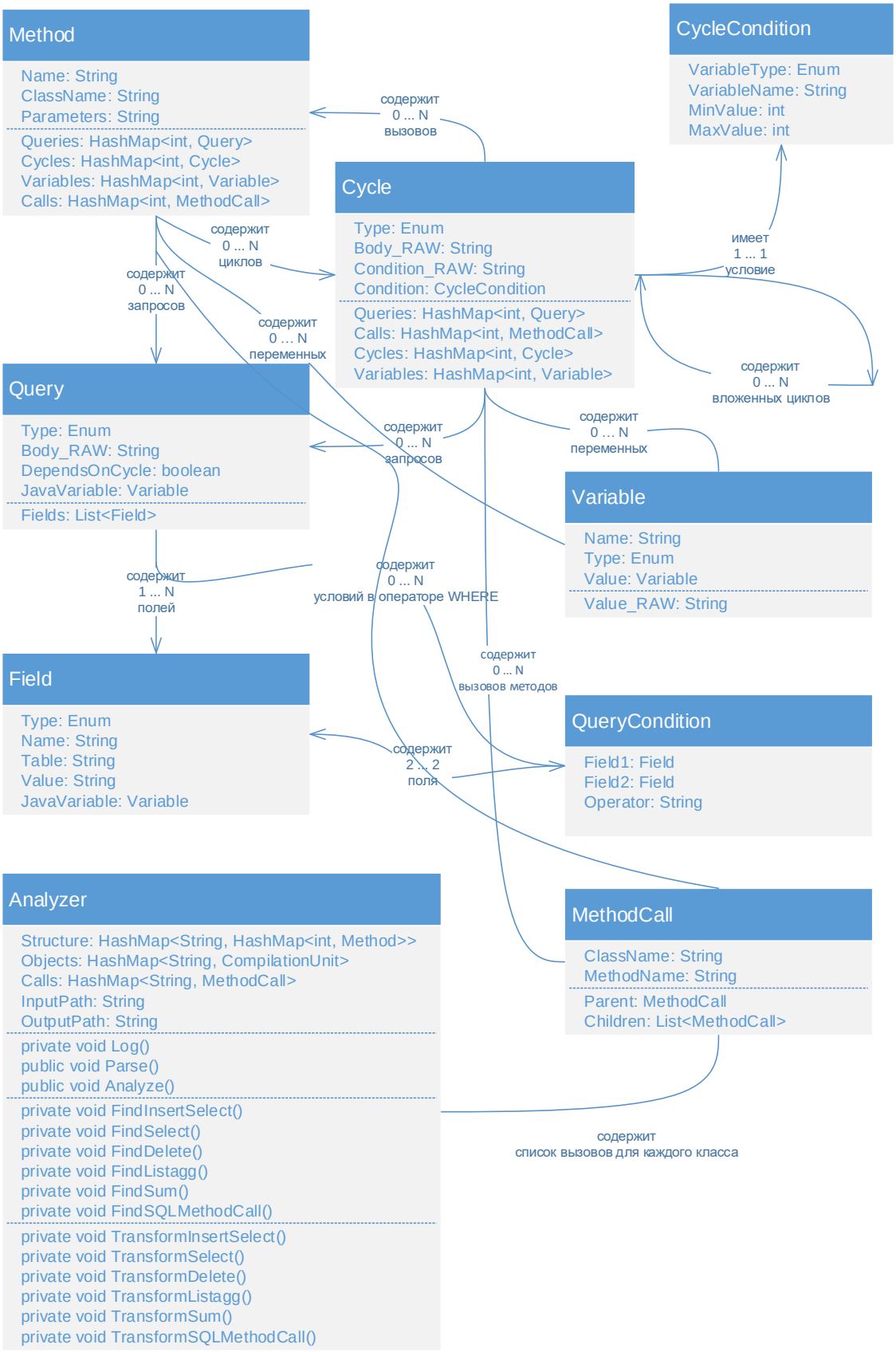

Рис. 2: Диаграмма классов и их отошений

```
[File: Test.java]
[Classname: Test]
...

[Method: SaveResults(int[] arr)]
...

[Pattern found: INSERT-SELECT]
      WHILE-cycle at LINE 12
      INSERT-query at LINE 15
      SELECT-query at LINE 11
[Solution: Can be transformed to one INSERT-SELECT query]
...
```

Рис. 3: Пример содержимого лог-файла при нахождении потенциального INSERT-SELECT запроса

Поиск различных неэффективных фрагментов имплементирован как методы класса `Analyzer`. В начале анализа исходного кода создаётся лог-файл, в который записывается имя класса, метода, номера строк и соответствующие SQL-запросы. Далее данное действие в описаниях алгоритмов называется ЛОГИРОВАНИЕ. На рисунке 3 показан пример содержимого лог-файла. После нахождения фрагмента сразу же вызывается соответствующий метод для рефакторинга (далее в описании – ТРАНСФОРМАЦИЯ). Далее произведённые изменения сохраняются по новому пути `OutputPath`, данному в начале инициализации нашего объекта `Analyzer`.

1.5. Синтаксичекий анализ SQL-запросов

Разбор SQL-выражений производится как при анализе методов, так и при парсинге циклов. Напомним, что мы рассматриваем стандартный JDBC-синтаксис для работы с SQL-выражениями. Процесс парсинга достаточно прост для SELECT и DELETE-запросов, но для INSERT и UPDATE случаев возможны два варианта: там присутствует конкатенация тела запроса с Java-переменными; либо маппинг производится посредством метода `PrepareStatement`. И в зависимости от ситуации

парсинг и рефакторинг таких выражений, очевидно, различается. В своём исследовании мы рассматривали только первый случай. В будущих работах планируется добавить анализ второго случая.

В рамках этой подзадачи был реализован следующий алгоритм:

1. Итерируем по `Statement`'ам в теле метода (цикла);
2. Посредством регулярного выражения производим поиск SQL-запроса. Если `Statement` содержит SQL-запрос, то создаём новый объект класса `Query`, заполняем его аттрибуты, парсим названия колонок и WHERE-оператор, таким образом, создавая объекты типа `Field` и `QueryCondition`;
3. Проверяем связь SQL-запроса с условиями цикла и сохраянем информацию об этом, если таковая имеется;
4. Сохраняем новый `Query`-объект в методе (цикле);

1.6. Синтаксический анализ циклов

Процесс разбора циклов был реализован следующим образом:

1. Итерируем по `Statement`'ам и производим поиск ключевых слов FOR и WHILE посредством регулярных выражений;
2. Если это цикл, то создаём новый объект типа `Cycle`, заполняем его аттрибуты и создаём объект `CycleCondition` из соответсвующих условий цикла;
3. Анализируем тело цикла на наличие вложенных циклов. Мы не рассматриваем ситуации, где вложенность больше одного цикла;
4. Производим поиск SQL-запросов, отмечаем их зависимость от условий цикла, если такая связь имеется. Данный маркер является наиболее ярким примером необходимости рефакторинга данного участка кода;
5. Сохраняем новый `Cycle`-объект в методе;

1.7. Анализ «графа вызовов» методов

После первичного анализа программного кода приложения, у нас есть информация о всех классах и методах, а также о циклах и SQL-выражениях в них. Мы предположили, что информация о вызовах в циклах методов, которые выполняют SQL-запросы, может быть полезной для обнаружения потенциальных JOIN-запросов. Для этой цели, при последующем проходе по методам мы запоминали связи методов между собой (объект `MethodCall`).

В статье [1] граф вызовов с содержанием информации об SQL-запросах является результатом выполнения утилиты `AppSleuth`. Мы же в своём исследовании строим некое подобие графа вызовов (объекты типа `MethodCall`) и используем эту информациию для автоматического рефакторинга случая, описанного в главе 2.1.3.

В будущем планируется преобразование этой информации к структуре графа в библиотеке `WebGraph` [14] для поиска компонент сильной связности среди методов, содержащих SQL-вызовы. Эта информация может быть использована для поиска потенциальных JOIN-запросов.

2. Реализация трансформации

В этой главе мы опишем основные алгоритмы по поиску «проблемных» (вычислительно неэффективных) фрагментов кода и их рефакторингу. Считаем, что к этому моменту вся информация о методах, циклах и запросах сохранена в нашу внутреннюю структуру. При этом каждый из объектов мы можем однозначно идентифицировать по номеру строки в исходном файле.

2.1. Поиск и рефакторинг неэффективных фрагментов программного кода

После сохранения информации во вспомогаетльные структуры, мы готовы к анализу «проблемных» фрагментов. Нас интересуют циклы и SQL-запросы в них. Было решено исследовать возможность автоматического рефакторинга пары относительно простых случаев и одного сложного случая, связанного с `Push Up Method`-рефакторингом, а именно мы рассмотрим:

1. Типичные INSERT-SELECT/SELECT/DELETE запросы в цикле;
2. Отсутствие агрегирующих функций (LISTAGG, SUM) в цикле;
3. Вызов в цикле методов, содержащих SQL-запрос;

В будущем планируется расширить данный список. Далее рассмотрим далее каждый из случаев по отдельности.

2.1.1. Типичные запросы в цикле

Рассмотрим следующий простой случай: если внутреннему INSERT-запросу в цикле предшествует SELECT-запрос (в том же цикле или вне его), и их аттрибуты «пересекаются», а именно, при вставке используются аттрибуты из первоначальной выборки, то это потенциальный INSERT-SELECT-запрос. Далее рассмотрим следующие случаи:

1. В цикле нет другой логики, кроме INSERT-запроса, следовательно, можем заменить цикл на один INSERT-SELECT запрос;

2. В цикле есть другая логика, следовательно, можем создать новый INSERT-SELECT-запрос и разместить его перед циклом, предварительно удалив INSERT-запрос в теле цикла;

Исходные параметры: Коллекция циклов из метода
Результат: Советы по улучшению и новый программный код

```
1  до тех пор, пока есть цикл в методе выполнять
2      до тех пор, пока есть SQL-запрос в цикле выполнять
3          если это INSERT-запрос тогда
4              InsertQueryFields ← БЕРЁМ список полей из запроса;
5              ИТЕРИРУЕМ вверх по общему списку SQL-запросов в
               методе;
6              если это SELECT-запрос тогда
7                  SelectQueryFields ← БЕРЁМ список полей из запроса;
                   если InsertQueryFields ⊂ SelectQueryFields тогда
8                      ЛОГИРУЕМ информацию о потенциальном
                       INSERT-SELECT запросе;
9                      ТРАНСФОРМАЦИЯ;
10                     ВЫХОД;
11                 конец
12             конец
13         конец
14     конец
15 конец
```

Алгоритм 1: Поиск потенциального INSERT-SELECT запроса

Процесс рефакторинга был реализован следующим образом:

Исходные параметры: Метод, цикл, INSERT-запрос, SELECT-запрос

Результат: Новый программный код

1 $NewQuery \leftarrow$ ФОРМИРУЕМ новый SQL-запрос;
2 **если** *цикл не содержит других* `Statement`*'ов, кроме INSERT-запроса* **тогда**
3 | УДАЛЯЕМ цикл;
4 | СОХРАНЯЕМ $NewQuery$ на место цикла;
5 **иначе**
6 | УДАЛЯЕМ из цикла INSERT-запрос;
7 | СОХРАНЯЕМ $NewQuery$ перед циклом;
8 **конец**
9 ПРОВЕРЯЕМ используются ли результаты искомого SELECT-запроса далее в методе;
10 **если** *не используются* **тогда**
11 | УДАЛЯЕМ искомый SELECT-запрос;
12 **конец**

Алгоритм 2: Трансформация к INSERT-SELECT запросу

На рисунке 4 изображён случай, когда можно заменить цикл одним SQL-запросом. Процесс формирования SQL-запроса для данного случая был реализован следующим образом:

Исходные параметры: SELECT-запрос, INSERT-запрос

Результат: Новый INSERT-SELECT запрос

1 $NewQueryString \leftarrow$ БЕРЁМ начало INSERT-запроса до ключевого слова VALUES;
2 ИТЕРИРУЕМ по полям из INSERT-запроса и соответствующим образом СОРТИРУЕМ поля в SELECT-запросе;
3 $NewQueryString \leftarrow NewQueryString \cup$ SELECT-запрос;
4 ВОЗВРАЩАЕМ $NewQueryString$

Алгоритм 3: Формирование INSERT-SELECT запроса

Рассмотрим случай однотипных SELECT-запросов. Для анализа были выбрали два случая:

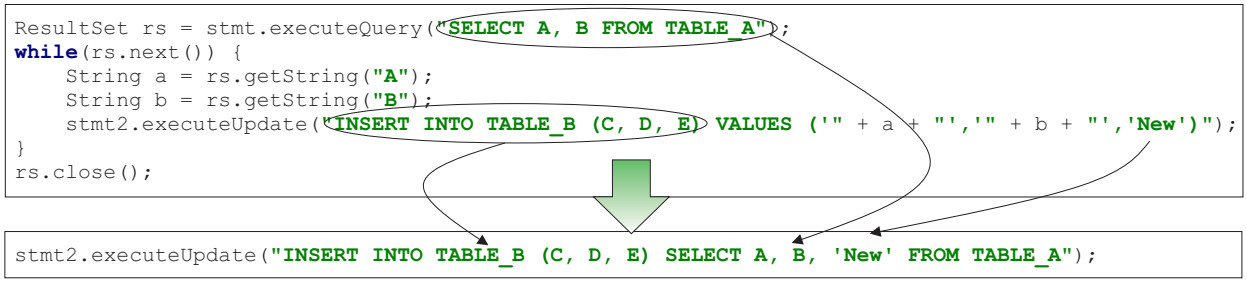

Рис. 4: Фрагменты кода до и после трансформации к INSERT-SELECT запросу

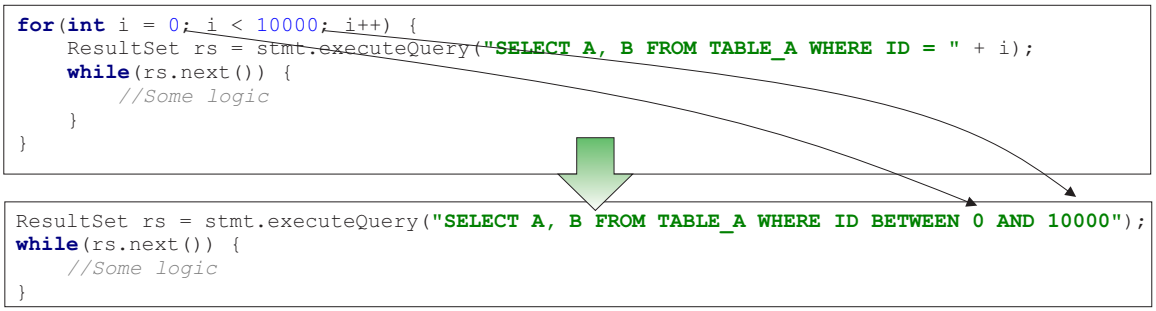

Рис. 5: Фрагменты кода до и после рефакторинга однотипного SELECT-запроса. Случай 1.

1. Производится итерация по FOR-циклу и его условия статичны. Если условия цикла (переменная) не используется нигде, кроме SELECT-запроса, то мы можем избавиться от внешнего цикла и переписать искомый запрос с использованием оператора BETWEEN. Данный случай наглядно изображён на рисунке 5;

2. Производится итерация по FOR-циклу и в его условиях используется размер переменной типа List<String>. Если условия цикла (переменная) не используется нигде, кроме SELECT-запроса, то мы снова можем удалить внешний цикл, предварительно создав переменную, в которой все элементы искомой коллекции соединяются в одну строку и запрос переписывается с использованием оператора IN. Этот случай изображён на рисунке 6;

В обоих случаях, мы анализируем лишь те запросы, в которых в WHERE-операторе участвует один атрибут.

Формирование строки для IN-оператора может быть выполнено посредством метода `String.join` (JDK 1.8) либо `StringUtils.join` (тре-

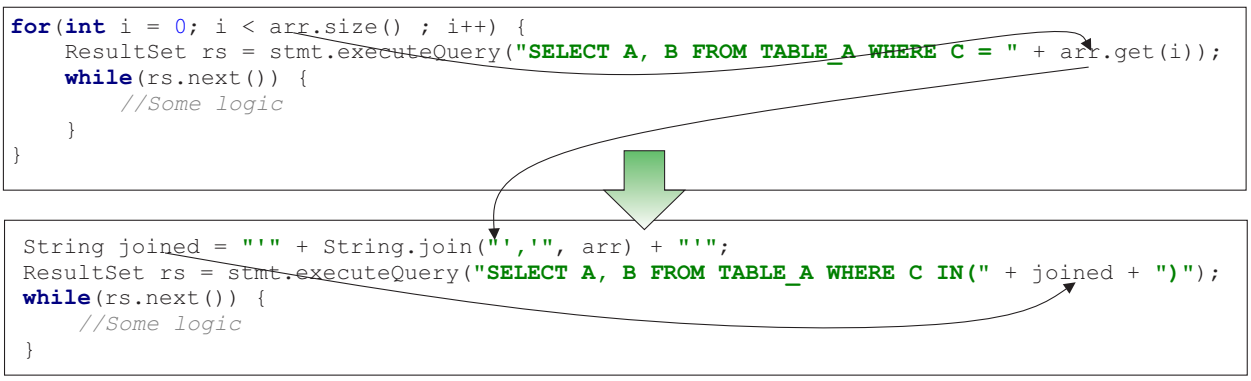

Рис. 6: Фрагменты кода до и после рефакторинга однотипного SELECT-запроса. Случай 2.

буется добавление соответствующей записи в секцию import).

Рассмотрим случай, когда в цикле производятся однотипные DELETE-запросы. Был имплементирован поиск следующих двух случаев:

1. В FOR-цикле производится итерация по массиву. В SQL-запросах используются элемент данного массива;

2. В WHILE-цикле производится итерация по результату (объект типа `ResultSet`) внешнего SELECT-запроса. В теле цикла инициализируется переменная, которые далее используются в SQL-запросе;

Отметим, что мы рассматривали запросы со следующим ограничением: допускается только одна переменная WHERE-операторе.

Если подобные однотипные запросы были найдены, то информация о них сохраняется в лог-файл и производится рефакторинг. Суть его заключается в следующем: оставляем цикл на месте (если в нём есть другая логика, иначе – удаляем цикл) и формируем новый SQL-запрос с оператором IN. Если в DELETE-запросе используется один атрибут внешнего SELECT-запроса, то трансформация такого участка кода заключается в формировани вложенного SQL-запроса (в оперторе WHERE).

2.1.2. Отсутствие агрегирующих функций

Рассмотрим случаи, когда производятся агрегирующие операции, такие как суммирование или конкатенация строк, без помощи стан-

дартных SQL-функций. Примерами могут служить такие процессы, как вычисление суммы долга и «склейка» какой-либо информации в одну строку с предопределённым разделительным символом. Был реализован следующий алгоритм поиска случаев потенциального использования функций `LISTAGG` и `SUM`:

Исходные параметры: Коллекция циклов из метода
Результат: Советы по улучшению и новый программный код

```
1  до тех пор, пока есть цикл в методе выполнять
2      ИТЕРИРУЕМ по Statement'ам цикла;
3      если Statement является Expression ∧ к переменной типа
       "строка" конкатенируется одна и та же переменная тогда
4          ПРОВЕРЯЕМ переменную на связь с SELECT-запросами в
           общем списке;
5          если SQL-запрос найден тогда
6              ЛОГИРУЕМ информацию о потенциальном
               использовании функции LISTAGG;
7              ТРАНСФОРМАЦИЯ;
8          конец
9      конец
10     если Statement является Expression ∧ целочисленная
       переменная инкрементируется на одну и ту же переменную
       тогда
11         ПРОВЕРЯЕМ переменную на связь с SELECT-запросами в
           общем списке;
12         если SQL-запрос найден тогда
13             ЛОГИРУЕМ информацию о потенциальном
               использовании функции SUM;
14             ТРАНСФОРМАЦИЯ;
15         конец
16     конец
17 конец
```

Алгоритм 4: Поиск потенциального использования функций SUM/LISTAGG

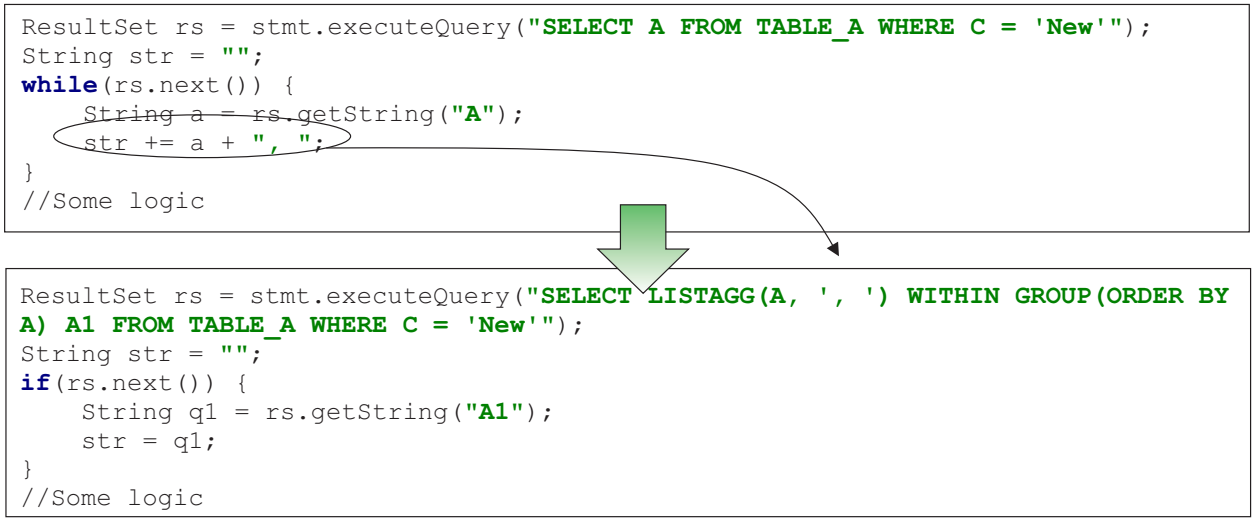

Рис. 7: Фрагменты кода до и после рефакторинга LISTAGG-запроса.

Процесс трансформации приведённых выше случаев отличается лишь в синтаксисе соответствующих SQL-операторов. Также мы анализировали лишь те случаи, в которых циклы не содержали другой логики, кроме как конкатенации или суммирования соответствующих переменных. Был имплементирован следующий алгоритм:

1. Проверяем, есть ли другая логика в цикле, связанная с искомым SELECT-запросом;

2. Если другой логики нет, то модифицируем SQL-запрос, заменив выбор соответствующих атрибутов на агрегирующую функцию с уникальным алиасом, и корректируем связь переменных в цикле;

3. Если есть другая логика, то создаём новый SQL-запрос с использованием агрегирующей функции, размещаем его до инициализации цикла и корректируем связь переменных в цикле;

2.1.3. Вызов в цикле методов, содержащих SQL-запрос

В статье [2] производится вручную поиск методов, которые выполнили тот или иной SQL-запрос. Мы же предлагаем делать это в автоматизированном режиме, и конкретно рассматривать вызовы методов в

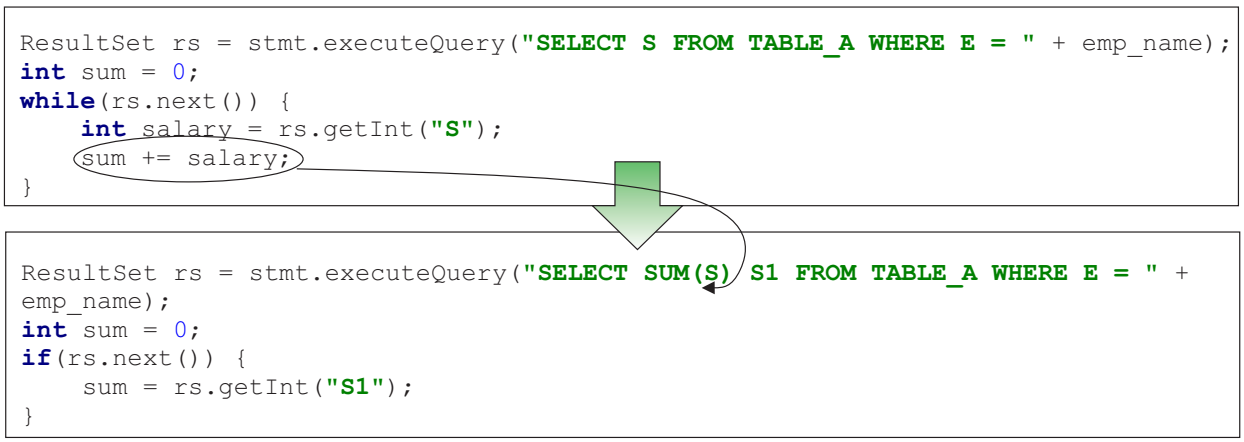

Рис. 8: Фрагменты кода до и после рефакторинга SUM-запроса.

циклах. Если вызываемый метод содержит один лишь SELECT-запрос, то формируем JOIN-запрос, а в цикле меняем вызов метода на результат выборки. Таким образом, был реализован следующий алгоритм (рисунок 9):

1. Проверяем вызываемые методы в цикле, смотрим их список параметров;

2. Если метод содержит один SQL-запрос и одно возвращаемое значение простого типа – атрибут выполненного запроса – то логируем информацию о потенциальной Push Up Method-трансформации;

3. Формируем и сохраняем новый JOIN-запрос, используя информацию из оператора WHERE искомого запроса и передаваемых в метод параметров;

4. Удаляем вызов метода из цикла и корректируем связи переменных в нём;

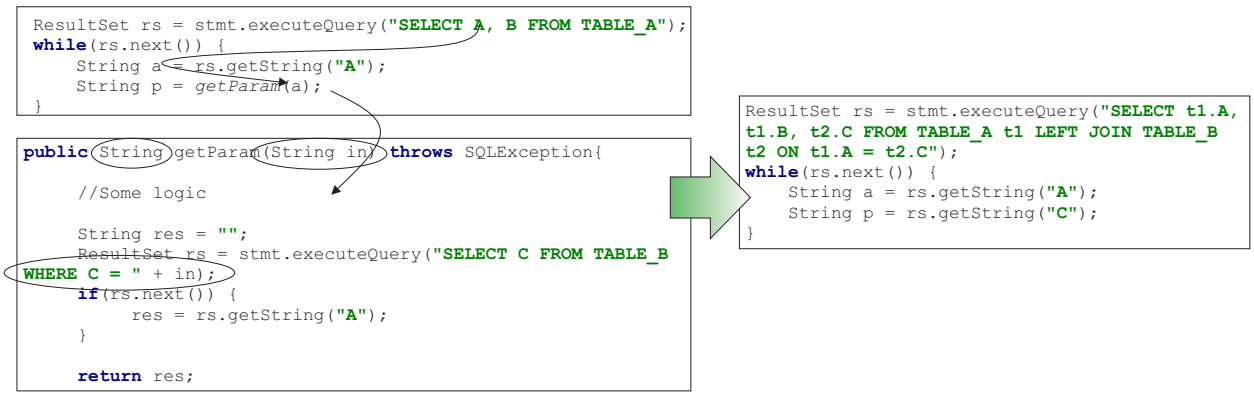

Рис. 9: Фрагменты кода до и после рекфакторинга

Приведём чуть более формальное описание алогритма:

Исходные параметры: Коллекция циклов из метода
Результат: Советы по улучшению и новый программный код

до тех пор, пока *есть цикл в методе* **выполнять**
  если *циклу предшествует SELECT-запрос* **тогда**
    ИТЕРИРУЕМ по `Statement`'ам цикла;
    если *Statement является* `Expression` $\wedge$ *в нём вызывается метод, который мы можем отследить* **тогда**
      если *в качестве параметра передаётся один атрибут, который является результатом предшествующего циклу SELECT-запроса* **тогда**
        если *вызываемый метод содержит лишь один SELECT-запрос и возваращает в качестве результата атрибут из этого запроса* **тогда**
          ЛОГИРОВАНИЕ информации о потенциальном JOIN-запросе;
          ТРАНСФОРМАЦИЯ;
        конец
      конец
    конец
  конец
конец

Алгоритм 5: Поиск потенциального использования JOIN-запроса

Алгоритм трансформации к JOIN-запросу был имплементирован

следующим образом:

Исходные параметры: метод, цикл, предшествующий SELECT-запрос, вызываемый метод

Результат: Новый программный код

1 $NewJoinQuery \leftarrow$ ФОРМИРУЕМ новый SQL-запрос путём добавления уникального алиаса к атрибутам первого SELECT запроса и формирования LEFT JOIN оператора на основе передаваемого в метода параметра;

2 ЗАМЕНЯЕМ предшествующий циклу SELECT-запрос на $NewJoinQuery$;

3 КОРРЕКТИРУЕМ в цикле связь переменной с возваращаемым из метода значением путём замены на возваращаемое значение из $NewJoinQuery$;

Алгоритм 6: Трансформация к JOIN-запросу

Ещё раз отметим, что подразумевается, что в вызываемом методе нет других действий, кроме как выполнения SQL-запроса и возвращения атрибута в качестве результата.

3. Эксперименты

Для замеров времени выполнения фрагментов кода до и после рефакторинга были сгенерированы тестовые данные. В качестве СУБД была использована Oracle 11G Release 2 Express Edition. Данная бесплатная редакция подразумевает следующие ограничения: 1 гигабайт оперативной памяти, 1 процессор и 11 гигайбайт – максимальный размер базы данных. Тесты были проведены на машине с процессором Intel Core i5 @ 2.4 GHz и операционной системой Microsoft Windows 10.

На рисунке 10 показана производительность фрагмента программного кода, в котором производится выборка и вставка данных, до и после процесса трансформации. Здесь и далее, по оси ординат – затраченное время в секундах, по оси абсцисс – количество строк, участвующих в процессе. Очевидно, что при возрастании объёмов данных (и количества запросов, соответственно) время обработки вырастает до недопустимых 14 минут для одного миллиона строк в первоначальной выборке, в то время как трансформация этого процесса к одному SQL-запросу позволяет повысить производительность в 140 раз (6 секунд).

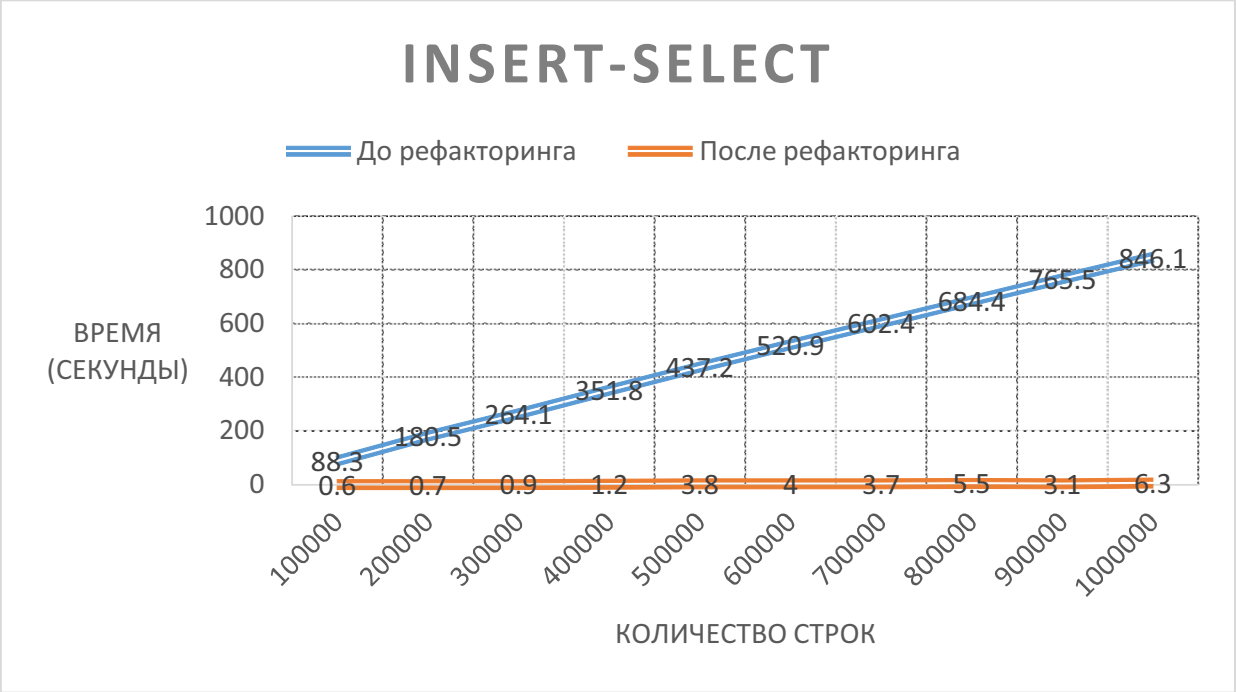

Рис. 10: Эффективность трансформации INSERT-SELECT фрагмента

На следующем графике (рисунок 11) приведено время выполнения фрагмента кода, который был сведён к одному JOIN-запросу. Как и следовало ожидать, производительность даже на отосительно маленькой выборке в тысячу строк увеличилась в сотни раз: 215 секунд против 1.123 секунды.

Рис. 11: Эффективность трансформации JOIN фрагмента

Эффективность рефакторинга фрагмента кода, содержащего однотипый SELECT/DELETE запросы, можно увидеть на следующих графиках:

Наиболее важным выводом, следующим из приведённых графиков, является тот факт, что линейный рост времени выполнения удалось заменить практически на постоянную. Данную проблему невозможно увидеть на малых объёмах данных. Успешные результаты трансформации напрямую влияют на масштабируемость приложения.

Что касается замеров времени производительности для LISTAGG функции, то оказалось, что есть зависимость от размера возвращаемого значения, и производительность до и после рефакторинга остаётся приблизительно на том же уровне (из-за малых объёмов данных). Использование функции SUM (Рисунок 14) было эффективным решением.

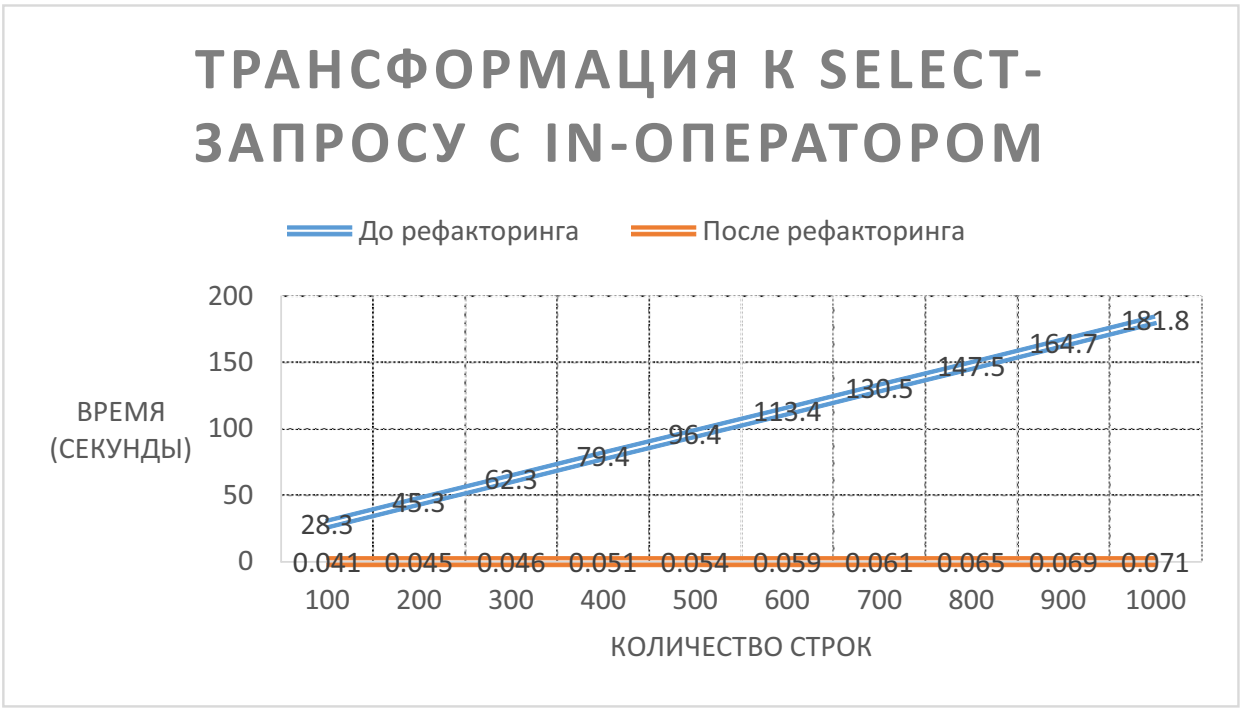

Рис. 12: Эффективность трансформации SELECT фрагмента

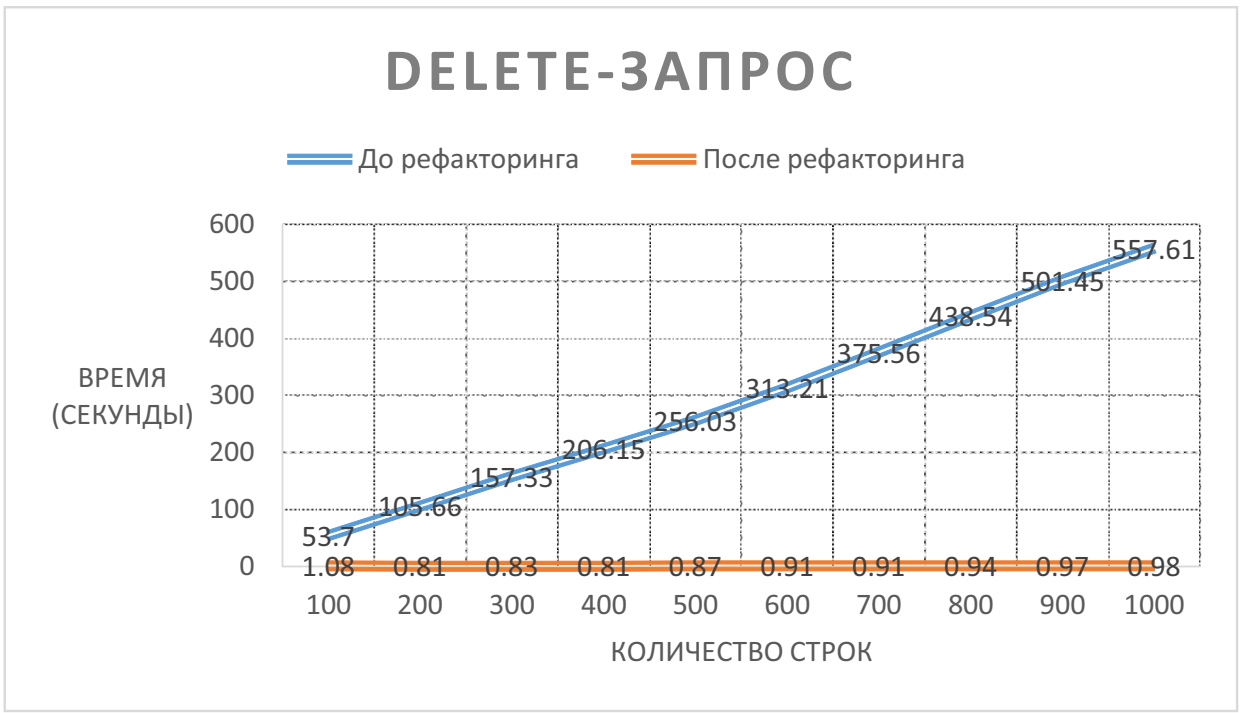

Рис. 13: Эффективность трансформации DELETE фрагмента

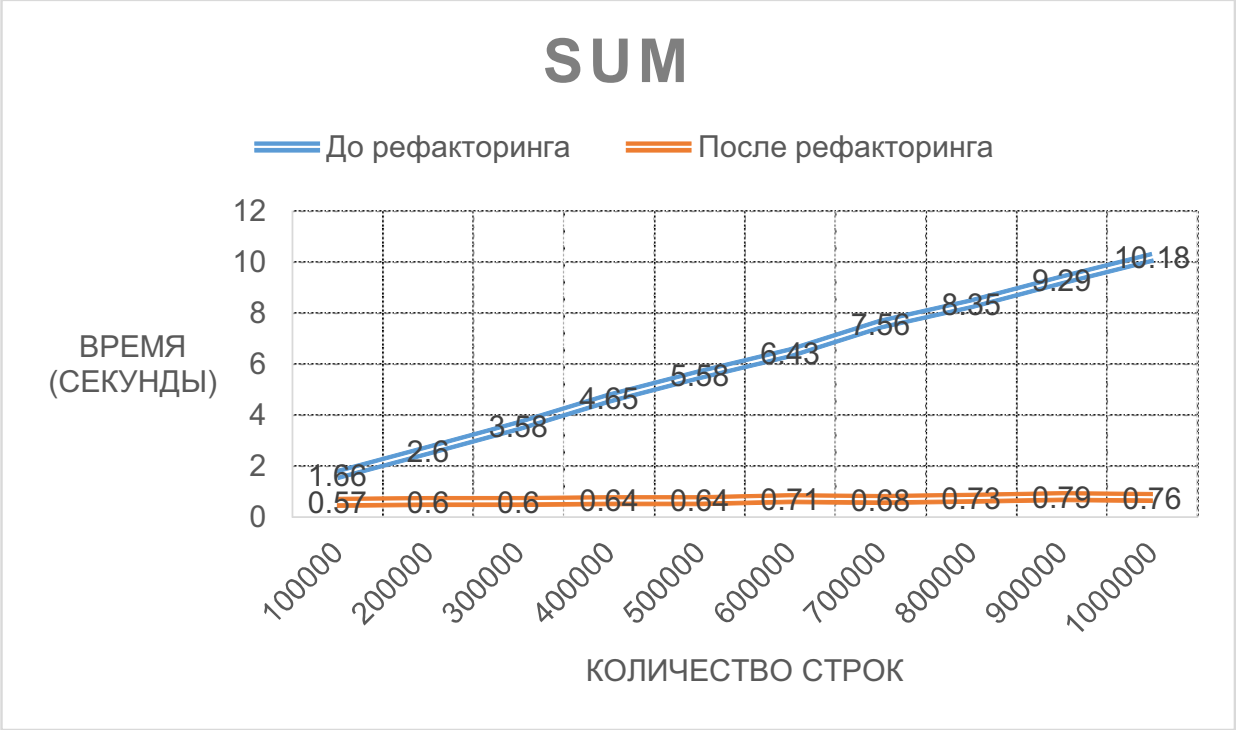

Рис. 14: Эффективность трансформации SUM фрагмента

4. Оптимизация взаимодействия с базами данных

Отправной точкой для нашего исследования является статья [1], которая является первой попыткой оптимизации SQL-приложений на уровне их исходного кода. Авторы использовали статический подход, заключающийся в синтаксическом анализе исходного кода `PL/SQL` и `T-SQL` программ, совместно с динамическим анализом, в котором они использовали логи СУБД `Oracle` для построения графа вызовов методов и SQL-запросов. Этот граф являлся результатом работы утилиты `AppSleuth`. В нашем исследовании мы не только анализируем приложение на наличие потенциально неэффективных фрагментов, но и производим автоматичекую трансформацию для ряда простых шаблонов.

Авторы статьи [2] говорят о так называемой проблеме «object-relational impedance mismatch» и приводят пример, когда для отрисовки всего лишь одной веб-страницы их приложение (на языке `Ruby on Rails`) делало тысячи запросов в базу данных (`PostgreSQL`). И хотя каждый из

запросов выполнялся считанные миллисекунды, то общее время выполнения выливалось в солидные секунды для одной страницы и в целые часы нагрузки на базу данных в течении суток. В статье была предложена авторская методика для ручного рефакторинга, основанная на анализе связей между таблицами в базе данных и методов, вызываемых в коде приложения. В нашей работе мы проводили анлогичную трансформацию в автоматическом режиме в случае формирования JOIN-запроса при вызове из цикла метода, содержащего один SQL-запрос.

В статье [3] приводится описание системы для автоматического рефакторинга Java-кода на основе фактов и правил вывода на языке `Prolog`. Мы использовали близкий подход для разделения логики поиска неэффективных фрагментов кода и непосредственно сам рефакторинг, создав промежуточный этап – выдача совета (логирование) по улучшению. Статья [4] также интересна своим подходом, в котором использовались open-source библиотеки для составления UML-диаграмм и рефакторинга посредством выделения различий между желаемым и фактическим дизайном приложения.

Также существует класс методов для рефакторинга Java-приложений в определённый объектно-ориентированный шаблон проектирования, например, в паттерн Стратегия [5]. Что касается анализа исходного кода, то в статье [8] производится сравнение трёх техник: синтаксический анализ исходного кода, анализ байт-кода (дизассемблирование) и профайлинг. Каждая из этих техник хороша для своего круга задач, при этом анализ исходного кода даёт наибольшее количество информации и в то же время является наиболее трудоёмкой задачей.

В большинстве своём статьи по рефакторингу не затрагивают целостной оптимизации приложения, а именно, оптимизацию работы с СУБД. Поэтому было решено выбрать данное направление для исследования.

Заключение

На основе анализа проблемы для ряда простых шаблонов намеченные задачи были решены. Благодаря open-source инструментам, мы смогли анализировать исходный код Java-приложений как абстрактное синтаксическое дерево, а также вносить необходимые изменения в него. Была исследована и реализована возможность выделения предопределённых вычислительно неэффективых фрагментов программного кода, касающиеся выполнения SQL-запросов в цикле. Исходя из выделенных «проблемных» шаблонов, мы смогли выдавать советы по улучшению программного кода (запись в лог-файл). Были приведены и имплементированы алгоритмы по поиску и автоматической трансформации найденных неэффективынх фрагментов. Для оценки полезности и эффективности выполненных посредством трансформации изменений мы использовали комплекс из различных оценок, таких как цикломатическая сложность и замеры времени выполнения фрагмента кода (приложения). Для проверки эквивалентности произведённых изменений – использовали JUnit-тесты.

Как показали эксперименты, после произведённых трансформаций ранее наблюдаемый линейный рост времени выполнения «проблемных» фрагментов кода практически сводится к постоянной.

В будущем целесообразно рассмотреть более общие и сложные шаблоны, зависящие от конкретных требований к коду анализируемого приложения.

Список литературы

[1] Cao W., Shasha D. AppSleuth: a tool for database tuning at the application level // EDBT '13 Proceedings of the 16th International Conference on Extending Database Technology. Pages 589–600.

[2] Dombrovskaya H., Lee R. Talking to the Database in a Semantically Rich Way // Proc. of EDBT/ICDT'17 Joint Conference, March 24-28, 2014, Athens, Greece.

[3] Jeon S.-U., Lee J.-S., Bae D.-H. An Automated Refactoring Approach to Design Pattern-Based Program Transformations in Java Programs // APSEC '02 Proceedings of the Ninth Asia-Pacific Software Engineering Conference. Pages 337–345.

[4] Moghadam I. H., Ó Cinnéide M. Automated Refactoring Using Design Differencing // 2012 16th European Conference on Software Maintenance and Reengineering (CSMR). Pages 43–52.

[5] Christopoulou A., Giakoumakis E. A., Zafeiris E. V., Soukara V. Automated refactoring to the Strategy design pattern // Journal Information and Software Technology. Volume 54, Issue 11, November 2012. Pages 1202–1214.

[6] McCabe T. J. A Complexity Measure // IEEE Transactions on Software Engineering (Volume:SE-2, Issue: 4). Pages 308–320.

[7] Myers J. G. An extension to the cyclomatic measure of program complexity // ACM SIGPLAN Notices. Volume 12, Issue 10, October 1977. Pages 61–64.

[8] Bowman T.I., Godfrey W. M., Holt C. R. Extracting Source Models from Java Programs: Parse, Disassemble, or Profile? // The 1999 ACM SIGPLAN Workshop on ProgramAnalysis for Software Tools and Engineering

[9] Java Parser with AST generation and visitor support. https://github.com/javaparser/javaparser

[10] Java Parser Roaster. https://github.com/forge/roaster

[11] Software Complexity Visualizer CyVis. http://cyvis.sourceforge.net/

[12] Eclipse Metrics plugin. https://sourceforge.net/projects/metrics/

[13] CheckStyle tool. http://checkstyle.sourceforge.net/

[14] WebGraph http://webgraph.di.unimi.it/