Оглавление

Введение

Скопировать текст в цифровом формате очень легко. Это же относится и к программному коду. Однако даже если переименовать переменные, заменить *for*-циклы на *while*-циклы, изменить систему исчисления констант — списанный код по сути останется тем же. Поэтому при обучении студентов необходимо оценивать добросовестность выполнения задания в том числе с точки зрения уникальности работы. Возникает задача: для двух файлов дать численную оценку их схожести, а так же при получении нового файла найти среди полученных ранее программ источник, с которого мог быть списан данный код (имеющий с ним определенную меру схожести).

Система обнаружения некорректных заимствований (плагиата), используемая на кафедре ИУ9 МГТУ им. Н. Э. Баумана, работает по принципу сравнения k-грам системы MOSS (Measure Of Software Similarity, мера схожести программ) [1] (описание см. в разделе 2.1). Полученные от студентов файлы и результаты сравнения необходимо хранить. Также требуется возможность просмотреть исходные коды и наглядно представить совпадающие участки, чтобы можно было оценить, действительно ли программа списана или совпадение обусловлено самой задачей.

Поэтому необходимо создать базу данных[1] для хранения файлов исходных кодов программ с информацией об авторе, номере задачи, времени отправки и пр., а также промежуточной информацией, позволяющей быстро искать файлы с совпадающими участками. Также требуется реализовать приложение, позволяющее добавлять новые файлы в базу данных, получать информацию о полученных сравнениях и визуализировать их.

[1] *База данных (БД)* — именованная совокупность данных, отражающая состояние объектов и их отношений в рассматриваемойпредметной области. [2]

1 Постановка задачи

Имеется система распознавания некорректных заимствований, работающая следующим образом: программный код разбивается на лексемы, в полученном массиве выделяются т.н. k-грамы (k-gram, здесь — k подряд идущих лексем), для каждой из них вычисляется ее хэш-сумма (hash-sum) и полученные массивы хэш-сумм для каждого файла сравниваются на предмет совпадения элементов. Равенство хэш-сумм означает совпадение соответствующих им участков кода (подробнее см. в разделе 2.1).

Требуется создать базу данных для хранения информации о присланных студентами файлов исходных кодов и совпадений в них. Для каждого файла база данных должна хранить:

– исходный код программы;

– данные об отправителе;

– время отправки файла;

– массив лексем;

– массив хэш-сумм;

– данные о задаче (номер задачи и версия набора тестов);

– файлы, с которыми имеется совпадение, и координаты идентичных участков.

База данных должна обладать следующими характеристиками:

– надежность (возможность восстановления в случае чрезвычайной ситуации);

– высокая скорость добавления новых файлов;

– высокая скорость получения для данного файла списка файлов, имеющих совпадающие участки;

– возможность моментально получить информацию о проценте совпадения двух файлов и координаты совпадающих участков;

– хранение минимально возможного количества информации;

– работа на ОС GNU/Linux;

– СУБД с открытым исходным кодом;

Также требуется разработать клиентское приложение, которое должно предоставлять следующие возможности:

– добавление новых файлов в базу данных;

– получение статистики о совпадении файлов в базе данных;
– визуализация идентичных участков кода для двух выбранных файлов;

2 Обзор алгоритмов и технологий

2.1 Система обнаружения некорректных заимствований MOSS

MOSS (Measure Of Software Similarity, мера схожести программ) — автоматическая система обнаружения некорректных заимствований в программном обеспечении, впервые внедренная Alex Aiken в 1997 г. и доработанная в 2003 г. в соавторстве с Saul Schleimer и Daniel S. Wilkerson [1].

В общем виде алгоритм работает следующим образом:

1. Из текста удаляются незначимые символы (пробелы, переносы строк и т.п.).

2. Полученный текст разбивается на k-грамы — непрерывные подстроки длины k (параметр k определяется пользователем). Причем каждая позиция в тексте является началом новой k-грамы.

3. Каждая полученная k-грама хэшируется.

4. Из полученного массива k-грам выбираются «отпечатки» (fingerprint) — набор хэш-сумм, характеризующий документ. Каждый отпечаток хранит информацию о файле и участке текста, из которого он получен.

5. Если два файла имеют одинаковые отпечатки — значит, имеет место плагиат текста.

Пример из статьи [1], иллюстрирующий работу алгоритма:

1. «`A do run run run, a do run run`» — Исходный текст.

2. «`adorunrunrunadorunrun`» — Текст после удаления незначимых символов (а также приведения всех символов к одному регистру).

3. «`adoru dorun orunr runru unrun nrunr runru unrun nruna runad unado nador adoru dorun orunr runru unrun`» — Последовательность 5-грам, полученных из текста.

4. «`77 72 42 17 98 50 17 98 8 88 67 39 77 72 42 17 98`» — Гипотетическая последовательность хэш-сумм для выделенных 5-грам.

В системе обнаружения некорректных заимствований кафедры ИУ9 для исключения пропуска совпадающих участков сравниваются все полученные хэш-коды, поэтому необходимо эффективно реализовать их хранение, а так же найти способ быстрого поиска файла, содержащего данный набор хэш-кодов.

В дальнейшем также будем называть k-граму *чанк* (англ. chunk — кусок, порция), а значения хэш-суммы от нее — *дайджест* (англ. digest — отпечаток).

2.2 Методология проектирования баз данных

Введем некоторые определения.

***Определение* 1.** *Сущность есть объект любой природы, данные о котором хранятся в базе данных. [3]*

***Определение* 2.** *Атрибуты представляют собой свойства, характеризующие сущность.*

***Определение* 3.** *Домен — множество всех возможных значений определенного атрибута отношения.*

***Определение* 4.** *Ключом (потенциальным ключом, идентификатором сущности) называется атрибут (или набор атрибутов), однозначно идентифицирующий экземпляр сущности. [2, с. 35]. Один из потенциальных ключей может быть выбран в качестве первичного ключа (предпочтительно числового типа). Остальные потенциальные ключи называются альтернативными. Идентификатор, состоящий из нескольких атрибутов, называется составным.*

Проектирование базы данных включает в себя несколько этапов:

– сбор информации о предметной области и концептуальное проектирование;

– логическое проектирование;

– физическое проектирование базы данных;

Концептуальное проектирование базы данных представляет собой процесс создания информационной модели работы организации, не зависящей от любых физических параметров реализации [4, с. 520].

Концептуальная модель включает в себя [4, сс. 503–504]:

– типы сущностей (самые важные объекты и концепции предметной области, существующие независимо от других);

– типы связей;

– атрибуты и домены атрибутов;

– ключи (первичные и альтернативные);

– ограничения целостности;

Логическое проектирование заключается в определении числа и структуры таблиц, формировании запросов к БД, определении типов отчетных документов, разработке алгоритмов обработки информации, создании форм для ввода и редактирования данных в базе и решении ряда других задач. [3, с. 149] На этом этапе производится конструирование информационной модели предприятия на основе существующих конкретных моделей данных, но без учета используемой СУБД и прочих физических условий реализации. [4, с. 521]

Для реляционных баз данных (см. раздел 2.4) на этапе логического проектирования также производится нормализация отношений и удаление избыточного дублирования данных, многозначных атрибутов, связей типа «многие-ко-многим».

Физическое проектирование базы данных представляет собой процесс подготовки описания того, каким образом база данных будет представлена во внешней памяти. Этот процесс во многом зависит от выбранной СУБД и других применяемых технологий. Для реляционных баз данных физическое проектирование заключается в следующих действиях [2, с. 30]:

– создание описания набора реляционных таблиц и ограничений для них на основе информации, представленной в глобальной логической модели данных;

– определение конкретных структур хранения данных и методов доступа к ним, обеспечивающих оптимальную производительность системы с базой данных;

– разработка средств защиты создаваемой системы.

2.3 ER-диаграммы

Для визуального представления таблиц и связей между ними используются *модель «сущность-связь»*, где отношения представлены как сущности, имеющие набор атрибутов, ключи для идентификации с связи между сущностями. Графически это представляется *ER-диаграммами* (entyty-relationship, сущность-связь), где сущность отображается блоком, с вписанным в него атрибутами и подписанным ключом, а связи обозначаются стрелками с пометками о кардинальном числе (степени связи, которая указывает число экземпляров

сущностей, которые участвуют в связи) и классе принадлежности (обязательном или необязательном).

Введем определения, которые будут использоваться в дальнейшем при проектировании базы данных. ER-диаграммы в дальнейшем будут рисоваться в нотации Дж. Мартина (crow's foot notation).

***Определение* 5.** *Сильная сущность (strong entity) — сущность, существование которой не зависит от какого-то иного типа сущности. На ER-диаграммах обозначается как прямоугольник.*

***Определение* 6.** *Слабая сущность (weak entity) — сущность, существование которой зависит от какого-то другого типа сущности. На ER-диаграммах обозначается как прямоугольник со скругленными краями.*

Сущности слабого типа иногда называют *дочерними*, *зависимыми* или *подчиненными*, а сущности сильного типа — *родительскими*, *сущностями-владельцами* или *доминантными* [4, с. 410].

***Определение* 7.** *Идентификационно-зависимой (ID-dependent) называется такая дочерняя сущность, идентификатор которой содержит идентификатор родительской сущности.*

Идентифицирующие связи (связь между идентификационно-зависимой сущностью и ее родительской сущностью) ER-диаграммах будем изображать сплошными линиями, а неидентифицирующие — пунктирными.

Первичный ключ на ER-диаграмме выделяется жирным шрифтом, также может быть помечен меткой PK (primary key). Для составных ключей поля атрибуты, составляющие ключ, помечаются нумерованными метками PK1, PK2 и т.д.

Альтернативные ключи на ER-диаграмме помечаются меткой AK (alternative key). Если альтернативных ключей несколько либо они составные, каждый входящий в них атрибут помечается нумерованной меткой AKn.m, где n - номер составного ключа, m - номер атрибута в данной ключе.

***Определение* 8.** *Кардинальное число (maximum cardinality) указывает число экземпляров сущностей, которые участвуют в связи.*

***Определение* 9.** *Максимальное кардинальное число (maximum cardinality) определяет максимальное число экземпляров сущностей, которые могут участвовать в связи.*

***Определение* 10.** *Минимальное кардинальное число (minimum cardinality) — минимальное количество экземпляров сущностей, которые должны участвовать в связи.*

Для двусторонней связи разделяют связи типа «один-к-одному» (1:1), «один-ко-многим» (1:N) и «многие-ко-многим» (M:N) [4, с. 418].

***Определение* 11.** *Степень участия определяет, участвуют ли в связи все или только некоторые экземпляры сущности [4, с. 418].*

Участие может быть обязательным (M, mandatory), когда минимальное кардинальное не меньше единицы или необязательным (O, optional), в этом случае минимальное кардинальное число равно нулю.

2.4 Реляционные СУБД

Реляционная модель данных (РМД) [1] некоторой предметной области представляет собой набор отношений, изменяющихся во времени. [3].

***Определение* 12.** *Отношение — двумерная таблица, содержащая некоторые данные.*

Математически отношение можно описать следующим образом. Пусть даны n множеств $D_l, D_2, D_3, \ldots, D_n$, тогда отношение R есть множество упорядоченных кортежей $< d_1, d_2, d_3, \ldots, d_n >$, где $dk \in D_k$, d_k — атрибут, а Dk — домен отношения R.

Отношение хранит в себе данные о сущности в виде строк (записей) таблицы. Атрибутам соответствуют столбцы таблицы (каждый атрибут именуется и ему соответствует заголовок некоторого столбца таблицы). Связи между таблицами указываются с помощью внешних ключей.

***Определение* 13.** *Пусть в отношении R_1 имеется не ключевой атрибут A, значения которого являются значениями ключевого атрибута B другого отношения R_2. Тогда говорят, что атрибут A отношения R_1 есть внешний ключ. [3]*

[1] Структуры данных и способы доступа к ним, обеспечиваемы конкретной СУБД, называют ее *моделью данных*. [5]

Отношение обладает следующими характеристиками [4]:

– отношение имеет имя, которое отличается от имен всех других отношений в реляционной схеме;

– каждая ячейка отношения содержит только одно элементарное (неделимое) значение;

– каждый атрибут имеет уникальное имя;

– значения атрибута берутся из одного и того же домена;

– каждый кортеж является уникальным, т.е. дубликатов кортежей быть не может;

– порядок следования атрибутов не имеет значения;

– теоретически порядок следования кортежей в отношении не имеет значения. (Но практически этот порядок может существенно повлиять на эффективность доступа к ним.)

Над отношениями производятся операции реляционной алгебры:

– унарные операции: выборка и проекция;

– операции с множествами: объединение, разность, пересечение, декартово произведение;

– различные операции соединения и пр.;

Результатом этих операций снова становится отношение. Благодаря этому исходные отношения можно трансформировать в новые, создавая большую гибкость управления данными и их обработки.

Взаимодействия с РСУБД (реляционной СУБД) осуществляется с помощью запросов на языке SQL (Structured Query Language). Значения данных типизированы (числа, строки, даты и пр.). Тип данных контролируется системой [6, с. 24].

Важной особенностью реляционных баз данных являются *представления*, дающие возможность представлять результаты сложных запросов как одну новую таблицу [6, с. 54].

***Определение* 14.** *Представление (view) — это именованное виртуальное производное отношение, представленное в системе исключительно через определение в терминах других именованных отношений [2].*

По сути представление есть псевдоним запроса. Однако с его помощью можно не только упростить логику работы с данными, но и ограничить пользователю доступ к данным (напр. разрешить пользователю видеть только часть

строк и столбцов таблицы). Представление позволяет пользователю увидеть результаты сохраненного запроса, а SQL обеспечивает доступ к этим результатам таким образом, как если бы они были реальной таблицей базы данных. [5, с. 413]

С момента своего появления в 1980-х годах реляционные СУБД практически стали стандартом баз данных. Информация в реляционной базе данных хранится в простом табличном виде, что дает реляционным базам данных много преимуществ по сравнению с базами данных более ранних разработок (иерархических и сетевых, имеющих сложную внутреннюю структуру) [5, с. 50]. На самом деле реляционная модель данных — единственная, имеющая формальную спецификацию. [7, с. 816] Это дает такие преимущества реляционных баз данных перед остальными, как: а) возможность возложить поддержку целостности данных на СУБД и б) независимость логической структуры данных от физической [8, с. 7].

Базами данных такого типа с открытым исходным кодом являются, например, MySQL, H2, HSQLDB, SQLite, PostgreSQL.

Для увеличения производительности, уменьшения избыточности хранения данных, избежания ошибок, порожденных удалением или изменением каких-либо данных требуется *нормализация* — приведение данных к одной из *нормальных форм*, то есть обеспечение выполнения определенного набора требований (в основном касательно уменьшения упрощения функциональных зависимостей между атрибутами в каждом конкретном отношении) [3, с. 159] [4, с. 448].

Для увеличения скорости выполнения запросов системы управления базами данных являются *индексы* — средства ускорения операции поиска записей в таблице. Таблицу, для которой используется индекс, называют индексированной. [3, с. 54]

Главная причина повышения скорости выполнения различных операций в индексированных таблицах состоит в том, что основная часть работы производится с небольшими индексными файлами, а не с самими таблицами. [3, сс. 56–59]

Для решения задач, в которых сложно представить данные в виде таблицы (реляционная модель предполагает неделимость данных) или такой тип хранения данных оказывается неэффективен, появились новые подходы — *по-*

стреляционные базы данных или NoSQL (Not Only SQL — не только SQL). Описанию некоторых из них, используемых наиболее часто, посвящены следующие пункты.

2.5 Документо-ориентированные базы данных

В отличие от реляционных баз данных, где данные представлены в виде таблиц, в документо-ориентированных базах данных хранятся структуры данных (документы), которые могут содержать вложенные структуры, а потому обладают высокой гибкостью и не налагает на входные данные много ограничений, кроме того, что они должны быть представимы в виде документа. У документа есть некоторый ключ, по которому к нему можно быстро обратиться. Кроме того возможно создавать и удалять пользовательские индексы по ключевым словам внутри документа [7, с. 880].

В документных базах данных так же, как и в реляционных база данных, возможно формулирование произвольных запросов, репликация, обеспечение согласованности и др. [6, с. 27]

Основные представители документо-ориентированных СУБД — MongoDB и CouchDB. Языком запросов этих баз данных является JavaScript.

Наиболее эффективно использование документо-ориентированных баз данных тогда, когда предметная область имеет иерархическую структуру, то есть данные, представленные документами, могут быть полностью вложены друг в друга без необходимости дублирования данных в других документах или указания ссылок на внешние документы. Такие базы данных могут эффективно использоваться, например, в больших он-лайн каталогах. [7, с. 880]

2.6 Столбцовые базы данных

Столбцовые базы данных очень похожи на реляционные, но с существенным отличием: в реляционных базах данных в памяти рядом хранятся данные, принадлежащие одной строке таблицы, а в столбцовых базах данных рядом хранятся данные, принадлежащие одному столбцу. Благодаря этому дешевой становится операция добавление нового столбца (которое производится построчно). Более того, это значит, что становится возможным хранение строк с разным количеством столбцов (даже строк, в которых вообще нет столбцов) и

не нужны дополнительные расходы памяти на хранение null-значений. Однако из-за этого столбцовые базы данных не поддерживают операцию соединения [7, с. 881].

К хранилищам такого типа относятся базы данных HBase, Cassandra и Hypertable.

HBase спроектирована по образцу Google BigTable и предназначена для обработки больших объемов данных (big data). Система ориентирована на распределение данных (горизонтальное масштабирование) на кластерах, составленных из стандартного оборудования на базе Hadoop. [6, с. 26]

2.7 Хранилища ключей и значений

Хранилище ключей и значений (key-value, КЗ-хранилище) практически представляет собой словарь, который сопоставляет ключам значения. Примером КЗ-хранилища можно считать файловую систему, где путь к файлу — это ключ, а его содержимое — это значение.

Среди баз данных с открытым исходным кодом, реализующих данный подход —memcached (и родственные ему memcachedb и membase), Voldemort, Redis и Riak. [6, сс. 25–26]

Хранилище ключей и значений может очень быстро выполнить поиск по ключу, но показывает невысокую производительность при необходимости произвести поиск по данным, хранящимся внутри «значения», так как эти данные не имеют определенной структуры. А значит, базы данных такого типа не подходят для систем, в которых нужны сложные запросы и агрегирование данных. [7, с. 880]

2.8 Графовые базы данных

Графовые базы данных используются только для обработки данных с большим количеством связей. Пары ключ-значение в них можно ассоциировать как с узлами (вершинами), так и связями (ребрами). Графовые базы данных (например, Neo4J) работают лучше других систем при обходе данных с ссылками на себя или со сложно устроенными связями, так как позволяют быстро обходить узлы и просматривать информацию в узлах и связях. Такие базы данных достаточно гибкие и часто используются в социальных сетях. Однако на

небольших наборах данных они будут показывать невысокую производительность в связи со сложной внутренней архитектурой.[6, с. 28]

2.9 Объектно-ориентированные баз данных

В объектно-ориентированной модели предоставляются возможности, аналогичные объектно-ориентированным языкам программирования:

– возможность идентифицировать отдельные записи базы;

– между записями функциями их обработки устанавливаются взаимосвязи;

– *инкапсуляция* — ограничение области видимости имени свойства пределами того объекта, в котором оно определено;

– *наследование* — распространение области видимости свойства на всех потомков объекта;

– *полиморфизм* — возможность объектам одного класса иметь разный набор свойств, полученный от разных родителей;

Поиск в такой базе данных представляет собой сравнение объекта, хранящегося в базу, и *объекта-цели* (goal), сформированного пользователем. Результат поиска может представлять собой подмножество всей хранимой в базе данных иерархии объектов. [3, сс. 41–43]. Иногда к таким базам данных относят PostgreSQL, которая является, по сути, объектно-реляционной.

3 Проектирование базы данных

Проектирование базы данных производится по схеме, описанной в разделе 2.2.

3.1 Концептуальная модель

В соответствии с требованиями к базе данных, данными в разделе «Поставновка задачи», были выделены следующие сущности (расшифровку обозначений и определения см. в разделах 2.2 и 2.3):

Студент — информация о студенте, отправившем данный файл исходного текста. Сильная сущность. Атрибуты:

- логин (PK);
- фамилия;
- имя;
- отчество;
- факультет;
- группа.

Имеет неидентифицирующую связь с сущностью «Файл исходного текста» вида 1:N, M-O;

Задача — описание задачи. Сильная сущность. Атрибуты:

- номер задачи (PK);
- условие;
- баллы.

Имеет идентифицирующую связь с дочерней сущностью «Версия задачи» вида 1:N, M-M;

Файл исходного текста (или «Файл» для краткости) — данные, получаемые системой от студентов. Слабая сущность (не может существовать в отсутствии сущностей «Задача» и «Студент»). Атрибуты:

- идентификатор файла (имя, штамп либо числовой идентификатор, PK);
- текст программы;
- язык;
- дата отправки.

Имеет следующие связи:

– рекурсивная связь сущности с самой собой, помеченная процентом совпадения между двумя файлами (вида O-O, M:N, неидентифицирующая);

– идентифицирующая связь с дочерней сущностью «Лексема» вида M-O, 1:N;

– неидентифицирующая связь с сущностью «Хэш-сумма» вида M-O, M:N.

Версия задачи — набор тестов. Так как набор тестов для каждой задачи может изменяться со временем (тесты могут быть изменены, исключены, добавлены), следует хранить версию набора тестов для каждой задачи, а в таблице, описывающей файл исходного текста, указатель на версию задачи, на которой он был протестирован. Идентификационно-зависимая сущность (зависит от сущности «Задача»). Имеет атрибуты:

– номер задачи (PK1);

– версия задачи (PK2);

– дата создания.

Участвует в следующих связях:

– неидентифицирующая связь с сущностью «Файл исходного текста» вида M-O, 1:N;

– идентифицирующая связь с сущностью «Тест» вида M-O, 1:N.

Тест — данные теста (исходный текст) и правильный ответ. Идентификационно-зависимая сущность (зависит от сущности «Версия задачи»). Атрибуты:

– номер задачи (PK1);

– версия задачи (PK2);

– номер теста (PK3);

– исходный текст;

– ответ.

Хэш-сумма — банк хэш-сумм, встречающихся в файлах. Эта сущность необходима для поиска файлов, в которых встречаются одинаковые хэш-суммы. Поэтому эта сущность имеет связь типа «многие-ко-многим» с сущностью «Файл исходного текста». Слабая сущность, зависит от сущности «Файл» (экземпляр должен быть удален при удалении последнего файла, содержащего данную хэш-сумму). Имеет единственный атрибут «значение» (PK).

Лексема — лексемы, входящие в хэш-суммы из файлов исходного текста. Данная сущность должна хранить в себе информацию о файле, из которого она получена, координаты лексемы в этом файле, образ лексемы, а так же указатели на хэш-суммы, в которые она входит. Идентификационно-зависимая сущность (зависит от сущности «Файл исходного текста»). Атрибуты:

– имя файла (PK1);
– позиция в файле (PK2);
– тип (PK3);
– образ;
– вердикт об успешности прохождения тестов.

Имеет связь с сущностью «Хэш-сумма» вида M-M, M:N.

Представление описанной выше концептуальной схемы базы данных в виде ER-диаграммы дано на рисунке 1.

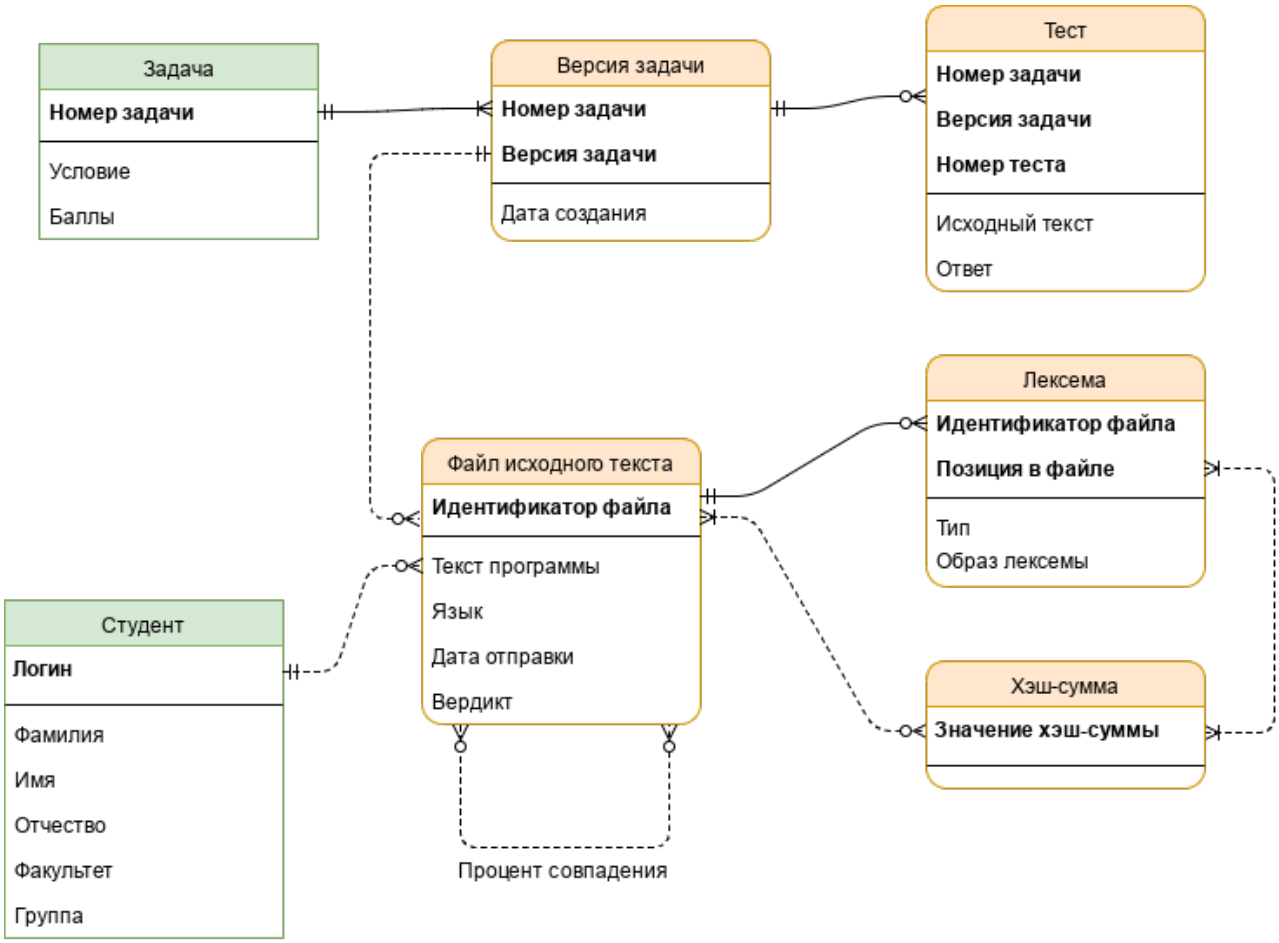


Рисунок 1 — Концептуальная схема базы данных.

3.2 Логическая модель

На данном этапе проектирования необходимо выбрать модель данных, определить число и структуру таблиц, произвести нормализацию отношений (для реляционной модели), удаление избыточной информации и связей вида «многие-ко многим», разработать алгоритмы обработки информации.

В качестве модели данных была выбрана реляционная. Это обусловлено главными задачами, которые должна выполнять база данных — это получение списка файлов, которые имеют общие хэш-суммы с данным и получение совпадающих хэш-сумм и их координат. Для решения второй задачи необходимо хранить связь каждой хэш-суммы с лексемами, которые в нее входят, для получения их координат. Из концептуальной схемы базы данных видно, что данные не имеют иерархической структуры, то есть использование документо-ориентированной модели или объектно-ориентированной не целесообразно. Так как наиболее часто будут выполняться операции соединения и поиска по данным, а данные имеют четкую структуру, то использование какой-либо постреляционной модели, направленной на возможность хранения неструктурированных данных, не целесообразно.

Поэтому будем проводить преобразования, требующиеся для реляционной модели данных.

Преобразуя сущности в таблицы, расставим ссылки — внешние ключи (FK, foreign key), отображающие связи между таблицами. Для связей типа 1:N внешний ключ добавляется в таблицу, сущность которой имеет множественное вхождение в эту связь. В связях типа M:N добавляется новая идентификационно-зависимая сущность, первичным ключом которой является объединение первичных ключей входящих в связь сущностей. Атрибутом новой связи становится атрибут связи.

На данном этапе можно заметить, что связь «Хэш-сумма»–«Лексема» вида «многие-ко-многим» на самом деле не обязательна и даже избыточна, ведь хэш-сумма включает в себя k *подряд идущих* лексем (напомним, что k есть количество лексем, для которых вычисляется хэш-сумма). Это значит, что в таблице «Лексема» можно держать запись только о той хэш-сумме, которая начинается с данной лексемы. Чтобы получить все лексемы, входящие в данную хэш-сумму, достаточно первую входящую в нее лексему и запросить следующие за ней подряд $k-1$ лексему. Чтобы получить все хэш-суммы, в которые входит некоторая

лексема, достаточно запросить значения хэш-сумм из $k-1$ лексемы, предшествующих данной. Эти операции не выполняются часто в работе базы, поэтому скорость их выполнения не столь значима, по сравнению с накладными расходами на хранение промежуточной таблицы.

Если на таблицу есть внешние ссылки, а ее первичный ключ является составным, добавляется суррогатный ключ, а атрибуты, составлявшие первичный ключ, становятся альтернативным ключом. Так как файлы, подающиеся на вход базы данных, именуются штампом, включающим логин студента, время отправки, вердикт о прохождении тестов, эти атрибуты должны стать альтернативным ключом. Однако такой длинный ключ избыточен, так как на практике невозможно отправить два файла одновременно, и в качестве альтернативного ключа достаточно взять два атрибута «студент» и «время отправки».

Таким образом проводим следующие преобразования концептуальной схемы:

1. Преобразование рекурсивной связи «Файл»–«Файл» в новую идентификационно-зависимую сущность (таблицу) «Результат сравнений», имеющую атрибут «процент совпадения» и две идентифицирующие связи «Файл»–«Результат сравнений» типа 1:N, M-O.

2. Преобразование связи «Файл»–«Хэш-сумма» вида M:N в новую идентификационно-зависимую сущность «Связь файл-хэш», не имеющую атрибутов, кроме первичных ключей родительских таблиц. Имеет идентифицирующие связи с родительскими таблицами: связь «Файл»–«Связь файл-хэш» типа 1:N, M-O и связь «Хэш-сумма»–«Связь файл-хэш» типа 1:N, M-M.

3. Замена составных первичных ключей таблиц «Задача», «Версия задачи», «Тест», «Студент» на суррогатные числовые идентификаторы.

После описанных выше преобразований имеем предварительный вариант логической схемы данных, представленной на рисунке 2.

После произведенных преобразований, в частности, после оптимизации связи «Хэш-сумма»–«Лексема» из типа «многие-ко многим» к типу «один-ко-многим», становится видно, что таблица «Хэш-сумма» с единственным атрибутом, который является и внешним ключом в других таблицах, избыточна. Однако если ее удалить, то новая таблица «Связь файл-хэш» становится избыточной, ведь пара атрибутов «файл» и «значение хэш-суммы» этой таблицы есть подмножество атрибутов таблицы «Лексема» и их семантическое значение сов-

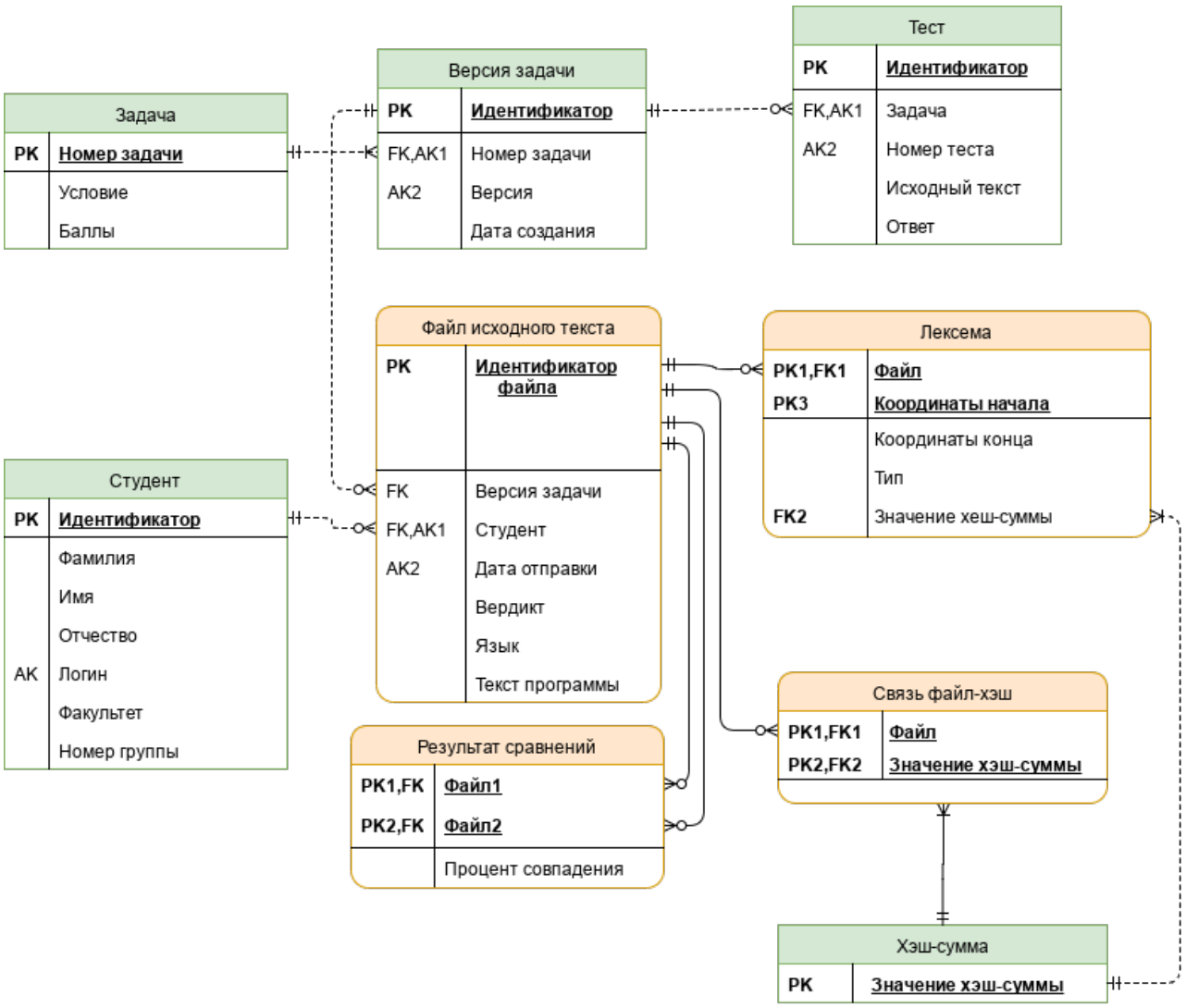


Рисунок 2 — Предварительный вариант логической схемы базы данных.

падает: они обозначают позицию лексемы (а значит, хэш-суммы) в конкретном файле.

Окончательный вариант логической схемы базы данных представлен на рисунке 3.

При такой схеме для поиска файлов, которые имеют общие хэш-суммы, необходимо из таблицы «Лексема» выбрать столбцы «файл» и «значение хэш-суммы» и соединить таблицу «Файл исходного текста» с собой по признаку «файл_1 и файл_2 имеют одинаковое значение хэш-суммы».

Для поиска совпадающих участков кода для каждого из файлов из таблицы «Лексема» для данных значений столбца «файл» выбираются соответствующие им «значение хэш-суммы» и координаты, после чего берется соединение полученных выборок по столбцу «значение хэш-суммы». Координаты начала участка, входящего в хэш-сумму (чанка), находятся в той же строке, что и ее

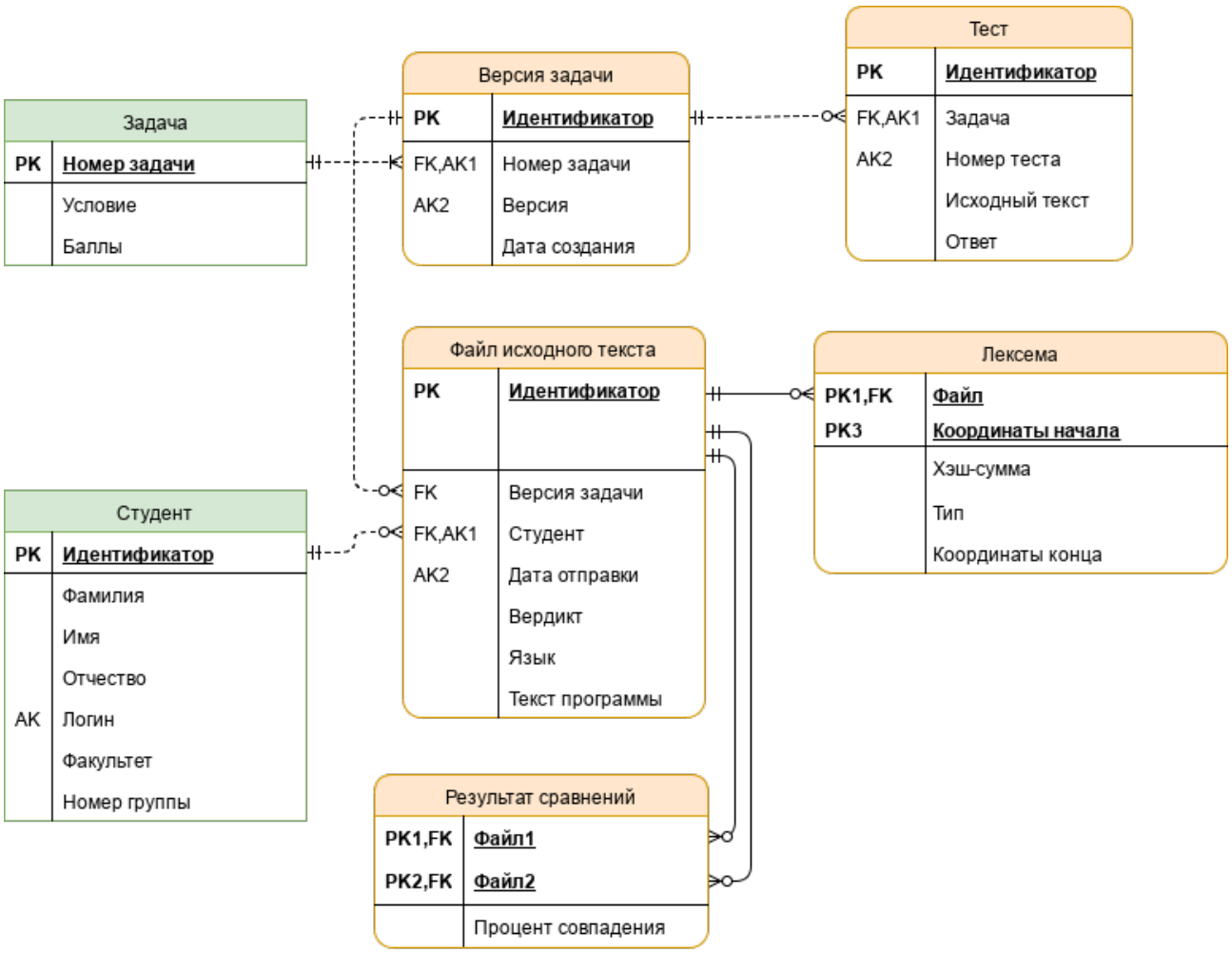


Рисунок 3 — Логическая схема базы данных.

значение. Для получения координат конца можно поступить двумя способами: а) выбрать все строки из таблицы «Лексема» и отсортировать их по стартовой позиции, после чего найти $(k-1)$-ю подряд идущую запись за данной и получить из нее координаты конца последней входящей в данную хэш-сумму лексему; б) добавить в таблицу «Лексема» поле «индекс» для обозначения индекса лексемы в массиве лексем, принадлежащих данному файлу, тогда для получения конечных координат достаточно выбрать из таблицы одну лексему с индексом, больше данного на $k-1$.

В качестве процента совпадения берется процентное соотношение длины массива общих дайджестов к длине массива всех дайджестов из тестируемого файла (последнюю величину можно получить с помощью функции, приведенной в листинге 9 в приложении 2).

3.3 Физическое проектирование базы данных

Для реализации базы данных была выбрана СУБД PostgreSQL [9], так как она удовлетворяет требованиям открытости исходного кода, возможности работы на операционной системе GNU/Linux, имеет постоянную поддержку со стороны разработчиков, удовлетворяет требованиям надежности, имеет подробную документацию и описание в большом количестве источников. Помимо стандартных средств реляционной базы данных (индексы, представления, ограничения целостности данных и др.) предоставляет дополнительные возможности, такие как хранение составных типов данных (массивов) и возможность создания пользовательских типов данных.

Опишем физическую схему базы данных SQL-скриптом для ее создания.

Создадим пользовательские типы данных для описания структур данных, используемых в полученном исходном коде системы обнаружения некорректных заимствований. Скрипт добавления данных приведен в листинге 1.

Листинг 1 — Создание пользовательских типов данных.

```
1 CREATE TYPE proglang      -- язык программирования
2     AS ENUM ('c', 'cpp', 'go', 'java', 'unknown');
3 CREATE TYPE verdict       -- вердикт о прохождении тестов
4     AS ENUM ('Passed tests', 'Failed tests', 'Draft', 'Not tested');
5 CREATE TYPE lexem_type    -- тип лексемы
6     AS ENUM ('keyword', 'special', 'ident', 'skippable', 'literal',
7              'pragma', 'other');
8 CREATE TYPE coord         -- координаты начала или конца лексемы
9     AS (pos int, line int, col int, tab int);
```

Скрипт создания наиболее важных таблиц для сущностей «Файл исходного текста», «Лексема», «Результат сравнений» представлен в листинге 2. Скрипт для создания таблиц для оставшихся сущностей «Задача», «Версия задачи», «Тест», «Студент» (в запросах упоминаются лишь ключи для данных сущностей) приведены в листинге 7 в приложении 2.

Этими скриптами описывается физическая схема база данных. Параметр k задается статически в исходном коде приложения и остается неизменным. В соответствии с исходным кодом, который был получен от кафедры для ознакомления с механизмом работы системы обнаружения некорректных заимствований, параметр k равен 20.

Листинг 2 — Создание таблиц для сущностей «Файл исходного текста», «Лексема», «Результат сравнений».

```
-- Файл исходного текста
CREATE TABLE SourceFile(
file_id       SERIAL PRIMARY KEY, -- идентификатор
task_fk       int    NOT NULL     -- версия задачи
    REFERENCES ProblemVersion(problem_ver_id) ON DELETE CASCADE,
student_fk    int    NOT NULL    -- студент
    REFERENCES Student(student_id) ON DELETE CASCADE,
code          text      NOT NULL, -- текст программы
code_language proglang  NOT NULL, -- язык
send_time     timestamp NOT NULL, -- дата отправки
tests_verdict verdict   NOT NULL, -- вердикт о прохождении теста
CONSTRAINT ak_sourcefile --задание альтернативного ключа
UNIQUE (student_fk, send_time);
);

-- Лексема
CREATE TABLE Lexem(
from_file_fk int    -- файл
    REFERENCES SourceFile(file_id) ON DELETE CASCADE,
array_index  int,          -- индекс в массиве лексем данного файла
digest       numeric(39),  -- хэшсумма
lexem_type   lexem_type,   -- тип лексемы
start_coord   coord NOT NULL, -- координаты начала
follow_coord  coord NOT NULL, -- координаты конца
CONSTRAINT PK_Lexem PRIMARY KEY(from_file_fk, pos)
);

-- Результат сравнений
CREATE TABLE FilesComparison(
file_to_test int NOT NULL   -- файл_1
    REFERENCES SourceFile(file_id) ON DELETE CASCADE,
file_base    int NOT NULL   -- файл_2
    REFERENCES SourceFile(file_id) ON DELETE CASCADE,
match_rating int NOT NULL,  -- процент совпадения
CONSTRAINT PK_FilesComparison -- описание составного первичного ключа
    PRIMARY KEY(file_base, file_to_test)
);
```

3.4 Основные запросы к базе данных

Первый запрос, который необходимо произвести к базе данных — получение всех пар файлов, имеющих общие хэш-суммы. Скрипт запроса представлен на листинге 3. Знаком вопроса помечено значение, которое будет вставлено при запросе (номер задачи, в пределах которой должен выполняться поиск). Кроме того получаемые пары файлов упорядочены по времени добавления, первым идет файл, который «проверяется на списывание» (*f1*), вторым — файл, «с которого может быть списан первый» (*f2*).

Листинг 3 — Запрос всех пар файлов, имеющих общие хэш-суммы.

```
-- Выбрать все пары с совпадающими хешами в( пределах задачи)
SELECT f1.file_id AS to_test_file, f2.file_id AS base_file
FROM SourceFile AS f1 JOIN SourceFile AS f2
ON  f1.task_fk = ?
AND f2.task_fk = ? -- Выбираем в пределах одной задачи
-- Игнорировать файлы того же студента
AND f1.student_fk != f2.student_fk
-- Выбирать только пары текущий" – добавленный ранее"
AND f1.send_time > f2.send_time
WHERE EXISTS(SELECT l1.digest FROM Lexem l1 JOIN Lexem l2
             ON  l1.digest = l2.digest
             AND l1.digest IS NOT NULL
             WHERE l1.from_file_fk = f1.file_id
             AND   l2.from_file_fk = f2.file_id);
```

Второй запрос, который, возможно, потребуется пользователю — получение всех файлов, имеющих общие хэш-суммы с данным. Скрипт данного запроса представлен в листинге 4 (для наглядности в запрос подставлены гипотетические возможные значения). Для упрощения запроса было введено представление *LexemSourceFileStudentName*, являющееся соединением таблиц *Lexem* и *SourceFile* по идентификатору файла, позволяющее получить полную информацию о файле, в который входит данная лексема: идентификатор студента, время отправки и пр. Код для создания этого представления приведен в листинге 8 в приложении 2.

Листинг 4 — Запрос всех файлов, имеющих общие хэш-суммы с данным.

```
-- Поиск файлов, из которых потенциально может быть списано решение 8684
-- Для запроса нужно знать: идентификатор файла.
SELECT task_fk, student_fk, send_time FROM SourceFile
WHERE from_file_fk = 8684;

-- Значения file_id, task_fk, student_fk, send_time получены в
   предыдущем запросе
SELECT * FROM LexemSourceFile
WHERE task_fk = 9 -- Выбрать решения той же задачи
AND   student_fk != 77 -- Игнорировать другие файлы того же студента
      -- Искать только среди файлов, отправленных раньше данного
AND   send_time <= '2013-02-18 21:34:47'
    -- Искать файлы с совпадающими массивами хэшсумм-
AND digest IN (SELECT l.digest FROM Lexem l WHERE l.from_file_fk=8684);
```

Главная задача, которую должна решать база данных — это нахождение всех совпадающих участков кода между двумя файлами. Для этого необходимо получить список общих дайджестов (хэш-сумм) для этих файлов, и координат чанков, входящих в эти хэш-суммы. Для этого вначале создадим представление,

которое для каждого дайджеста из файла хранит координаты его начала и конца (скрипт для его создания приведен в листинге 5).

Листинг 5 — Создание представления для получения координат участка, составляющего данную хэш-сумму.

```
1 -- Представление для получения координат участка, составляющего данный
      дайджест
2 CREATE VIEW Digests AS
3 SELECT l1.from_file_fk, l1.array_index, l1.digest,
4        l1.start_coord, l2.follow_coord
5 FROM Lexem l1 JOIN Lexem l2
6 ON  l1.from_file_fk = l2.from_file_fk
7 AND l1.digest       IS NOT NULL
8 AND l2.array_index  = l1.array_index + 19;
9 -- Нас интересуют координаты конца последней лексемы в чанке => к
      текущему индексу прибавить k - 1, где k = 20 количество лексем в
      чанке.
```

Теперь можно получить пересекающиеся координаты схожих участков файлов, соединив таблицу *Digests* с собой, выбрав записи с заданными идентификаторами файлов и совпадающим значением хэш-сумм. Соответствующий код запроса приведен в листинге 6.

Листинг 6 — Запрос для выбора совпадающих хэш-сумм для двух файлов.

```
1  -- Выбрать совпадающие дайджесты двух файлов
2  -- В примере файлы с гипотетическими идентификаторами 10664 и 10841
3  SELECT
4  d1.array_index   AS "d1.array_index",
5  d1.start_coord   AS "d1.start_coord",
6  d1.follow_coord  AS "d1.follow_coord",
7  d1.digest,
8  d2.array_index   AS "d2.array_index",
9  d2.start_coord   AS "d2.start_coord",
10 d2.follow_coord  AS "d2.follow_coord"
11 FROM  Digests d1, Digests d2
12 WHERE d1.from_file_fk = 10664 AND d2.from_file_fk = 10841
13 AND   d1.digest = d2.digest;
```

4 Реализация базы данных

Целью курсового проекта было создание базы данных файлов исходного кода и написание пользовательского приложения для добавления файлов в базу данных, получения статистики о схожих файлах и наглядного представления совпадающих участков для двух выбранных файлов. Эта задача была выполнена.

База данных была реализована с использованием СУБД PostgreSQL-9.5 [9].

Размер реализованной базы данных, хранящей 1928 реальных файлов: 172 МБ.

Размер клиентского приложения, упакованного как jar-файл: 4,2 МБ.

4.1 Повышение производительности базы данных

В книгах, посвященных работе с SQL и повышению производительсноти PostgreSQl ([10, сс. 143, 321], [8, сс. 24–28]) в качестве средств повышения производительности указываются:

– использование индексов;

– перенос логики на сторону сервера (пересылка промежуточных запросов на сервер, получение промежуточных результатов на клиент и их обработка);

– оптимизация и рекомпиляция запросов.

Использование индексов может как ускорить выполнение запросов, так и увеличить время модификации записей (так как эти операции требуют обновления индексов). Однако модификация записей не предусмотрена логикой работы базы данных, каждый файл добавляется единожды, а время выполнения запросов критично, то помимо первичных индексов, созданных базой данных для первичных ключей было решено добавить индексацию по следующим полям:

– столбец *task_fk* таблицы *SourceFile* (для оптимизации запроса пар файлов в пределах задачи);

– столбец *digest* таблицы *Lexem*;

В PostgreSQL по умолчанию индексы организованы как B-деревья (B-tree) [11, с. 1263]. Влияние добавленных индексов на производительность см.

в разделе «Тестирование». Добавление указанных индексов увеличило размер базы данных на 40% (со 172 МБ до 242 МБ).

Перенос логики на сторону сервера был произведен на этапе проектирования базы данных: все задачи, которые должно решать приложение, выполняются запросами к базе данных.

Все часто повторяющиеся запросы были выполнены в виде процедур для возможности предварительной компиляции и подготовленных запросов (prepared). В JDBC такие объекты представляет класс *PreparedStatement*).

Подготовленный запрос (параметризованный запрос, в котором опущены фактические параметры, устанавливаемые непосредственно перед каждым выполнением) позволяет базе данных единожды построить план выполнения, проверить его корректность и оптимизировать, после чего многократно будет использовать его, подставляя новые параметры, если сохраняется критерий фильтрации. [10, с. 287]. Более того, использование таких запросов увеличивает надежность базы данных, так как не позволяет пользователю внедрить SQL-инъекцию (SQL Injection) при формировании запроса через пользовательское приложение. Например, при построении запроса:

```
1 String query = ”delete from users where username=” + username;‘
```

передать вместе с данными SQL-код, выполнить следующий запрос:

```
1 delete from users where username=’smith\’ or \’a\’=\’a’
```

и удалить тем самым все данные из таблицы (пример взят из источника [12]).

В реализованном приложении используются подготовленные запросы для добавления файлов и лексем (так как они требуют пользовательского ввода и не имеют параметров фильтрации).

4.2 Основные классы и структура приложения

Так как для ознакомления с работой системы обнаружения некорректных заимствований кафедрой был предоставлен исходный код на языке Java, было принято решение использовать этот код в клиентском приложении. Поэтому исходный код клиентского приложения был так же написан на языке Java. Для

взаимодействия клиентского приложения с базой данных использовался драйвер PostgreSQL JDBC Driver 42.0.0 [13].

Для каждой из таблиц, построенных на этапе физического проектирования, был создан одноименный класс («SourceFile», «Lexem» и т.д.). Каждый такой класс содержит в себе неизменяемые статические строки с именами столбцов соответствующей таблицы и подготовленными запросами, а также статические методы для выполнения запросов и обработки их результатов (статические методы использовались для того, чтобы не создавать объект класса для выполнения запроса, который обычно подразумевает только передачу параметров).

Для выполнения запросов доступ к подключению методы получают через класс «ConnectionHolder», который содержит в себе само подключение, все экземпляры подготовленных запросов и закрывает их перед завершением приложения. Также в него может быть записан результат выполнения предыдущей операции. Это удобно в случае, если для выполнения запроса нужно знать об успешности выполнения предыдущего запроса. Так как этот класс управляет подключением, то вызовом соответствующих методов можно открывать и закрывать транзакции.

Класс «SourceFile» инкапсулирует работу с файлами. Основные методы:

– «addFileToDB» — добавляет файлы в базу данных;

– «clear» — удаляет все файлы из базы данных;

– «filePairsOrderedBySendTime» — возвращает список пар файлов <добавленный ранее, добавленный позднее> имеющих общие дайджесты, выполняет запрос из листинга 3;

– «filePairsOrderedBySendTimeNotChecked» — аналогичен предыдущему методу за тем исключением, что озвращает только пары файлов, которые еще не сравнивались между собой и не записаны в таблицу «FilesComparison»;

– «findMatches» — позволяет найти файлы, имеющие общие дайджесты с данным (запрос из листинга 4);

Класс «Lexem» выполняет работу над лексемами. Основной метод — «addLexemArrayToDB» (добавляет массив лексем данного файла в базу данных).

Класс «FilesComparison» представляет сущность «Результат сравнений». Основные функции класса:

– «findAllUncheckedMatches» — находит в базе данных все файлы, имеющие общие дайджесты, но не занесенные в таблицу «FilesComparison», выполняет их сравнение и запись (вызывает «SourceFile.filePairsOrderedBySendTimeNotChecked»);

– «add2fileIntersectionRate» — находит процентное соотношение схожести двух файлов;

– «findAllMatches» — находит в базе данных все файлы, имеющие общие дайджесты (вызывает «SourceFile.filePairsOrderedBySendTime»);

– «intersect2FilesChunks» — получает общий список дайджестов с координатами (запрос из листинга 6);

– «optimizeOutput» — объединяет подряд идущие чанки в непрерывные участки для визуального представления схожести;

Добавление файлов из директории происходит в три этапа:

– производится лексический и синтаксический анализ каждого файла и вычисление массива хэш-кодов из директории с помощью модуля *ossmoss-hasher*, предоставленного Скоробогатовым Сергеем Юрьевичем;

– каждый файл добавляется в базу данных: в функции «SourceFile.addFileToDB» открывается транзакция, добавляется сам файл, выполняется функция «Lexem.addLexemArrayToDB», закрывается транзакция;

– после добавления всех файлов выполняется функция «FilesComparison.findAllUncheckedMatches», сравнивающая новых файлов на предмет совпадений кода и добавляющая результат в таблицу «FilesComparison».

Добавление файлов, получение статистики из таблицы «FilesComparison» и визуальное представление совпадающих участков кода производятся в графическом интерфейсе приложения, реализованном на базе стандартной библиотеки Java Swing.

4.3 Установка и запуск приложения

Установка СУБД производится в соответствии с документацией [11]. Архив с пользовательским приложением содержит следующие файлы:

– файл *CREATE_DATABASE_OBJECTS.sql*, содержащий скрипт для создания всех типов данных, таблиц, представлений и процедур, необходимых для работы базы данных;

– jar-архив *ossmoss-hasher-database-1.0.jar*, являющийся собственно пользовательским приложением;

– папка *test-data* с реальными файлами исходного кода, предоставленными кафедрой для тестирования;

– скрипты для создания базы данных и запуска приложения (должны быть испольняемыми): *init_database.sh*, *create_database.sh*, *init_test_db.sh* (для инициализации тестового примера), *start.sh*;

– папку *source* с исходным текстом приложения;

– файл *README.md*.

После создания базы данных и пользователя в консоли или с помощью графических приложений для работы с базами данных выполнить скрипт в файле (исполняет *CREATE_DATABASE_OBJECTS.sql*). Этот скрипт является компиляцией листингов, приведенных в разделах «Физическое проектирование» и «Приложение 2» и вспомогательного кода, не описанного в данной расчетно-пояснительной записке.

Также можно создать пользователя и базу данных с помощью приведенных в архиве скриптов:

```
# Создание пользователя и базы данных
./create_db.sh <user> <database name>
# Инициализация базы данных – создание таблиц и процедур
# выполнение( CREATE_DATABASE_OBJECTS.sql)
./init_db.sh <user> <database name>
```

Приложение из jar-архива запускается из консоли. Перед запуском приложения сервер `postgresql` должен быть запущен, иначе приложение выдаст сообщение об ошибке и завершится. Команда для запуска зависит от дистрибутива операционной системы. Например, для Gentoo-4.9.5: `'sudo systemctl start postgresql-9.5'`.

Аргументы при запуске приложения:

```
-d <URL базы данных> -u <имя пользователя>
```

либо

```
-f <файл настроек>.json
```

либо без аргументов.

Аргумент <URL базы данных> отличается от использованного ранее имени базы данных (<database name>). Например, для локальной базы данных с именем db_name URL будет иметь вид jdbc:postgresql://localhost/db_name.

При запуске приложения появляется окно с приглашением к вводу URL базы данных, имени пользователя и пароля (поле остается пустым, если для базы данных не установлен пароль). (Рисунок 4.)

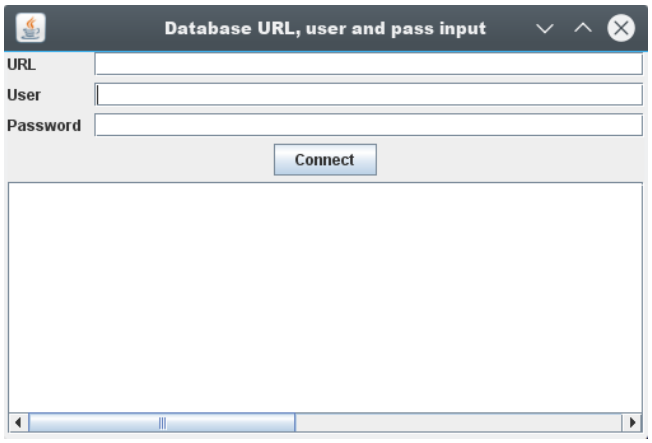


Рисунок 4 — Окно с приглашением к вводу URL, имени пользователя и пароля для подключения к базе данных.

Если при запуске были указаны аргументы, они подставляются в соответствующие поля, и их можно отредактировать. Также данные о URL базы данных и имени пользователя можно хранить в JSON-файле вида:

```
1 {
2   ”saved_db_url”: ” < URL базы данных > ”,
3   ”saved_db_user”: ” < имя пользователя > ”
4 }
```

и при запуске передавать его с ключом -f. Если не указано ни одного аргумента командной строки, производится попытка прочитать файл saved_properties.json, если он пуст либо отсутствует, поля остаются пустыми, иначе заполняются данными из файла. При выходе из приложения данные о последней подключенной БД сохраняются в файл saved_properties.json и подставятся автоматически при запуске без аргументов.

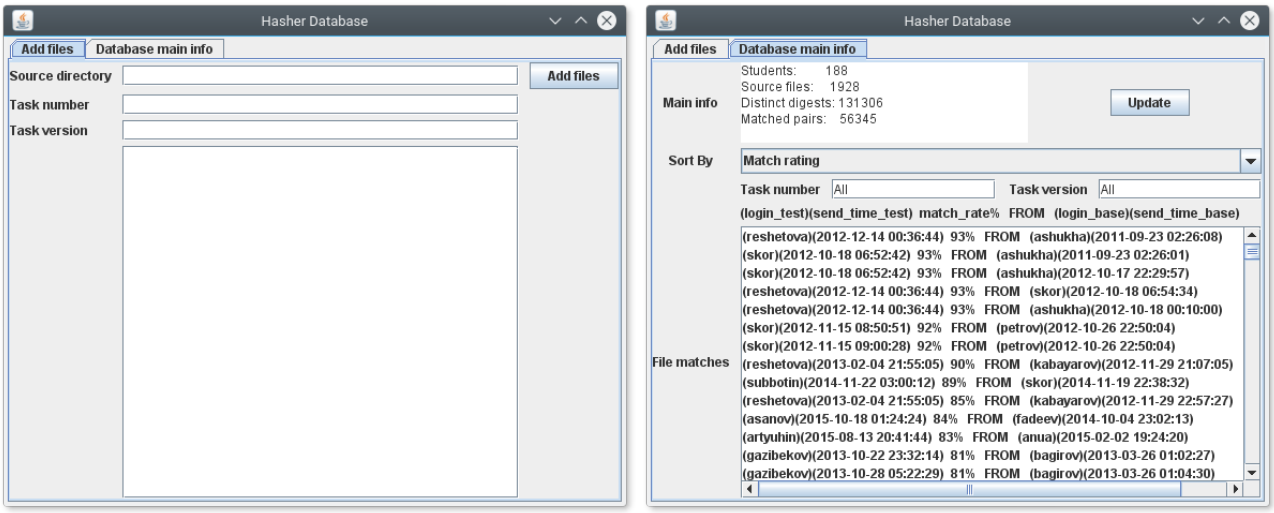
Пример запуска программы из консоли:

```
java -jar hasher_db.jar -d jdbc:postgresql://localhost/db_name -u
    user_name
```

или

```
./start.sh -d jdbc:postgresql://localhost/db_name -u user_name
```

После введения пароля появится диалоговое окно с пользовательским приложением. Первая вкладка «Add files» предназначена для добавления файлов из директории (рисунок 5.а)). После заполнения всех полей и нажатия кнопки Add files файлы будут закружены в базу данных и в нижнем поле появится отчет о завершении операции (количество добавленных файлов, количество пар файлов с общими дайджестами, время, затраченное на каждый этап загрузки файлов либо сообщение об ошибке).



а) б)

Рисунок 5 — Окно пользовательского приложения.
Вкладка добавления файлов (а) и получения статистики (б).

Во второй вкладке «Database main info» (рисунок 5.б)) отображается основная информация о базе данных: количестве студентов в базе («Students»), количестве файлов («Source files»), количестве всех возможных дайджестов, встречающихся в файлах во всей базе данных («Distinct digests»), количестве пар файлов с совпадениями («Matched pairs»). В нижней области приведены списки файлов и процент совпадения между ними. Слева указаны «списанные» файлы, справа — «с которых списали». Можно ограничить выбор файлов

по номеру задачи («Task version») и ее версии («Task version») или отсортировать список по любому из столбцов (по умолчанию сортировка производится по проценту совпадения). Если база данных была изменена вне приложения, для обновления информации необходимо нажать кнопку «Update».

Дважды кликнув на любой строке можно получить получить подробную информацию о сравнении двух файлов (рисунок 6). Слева показана информация о «списанном» файле (формально — том, который был добавлен позже), справа — о файле, «с которого списан» первый. Одинаковые участки кода выделяются цветом (желтым для первого файла и зеленым для второго). Если совпадающих участков несколько, переключаться между ними можно кнопками «Previous» и «Next».

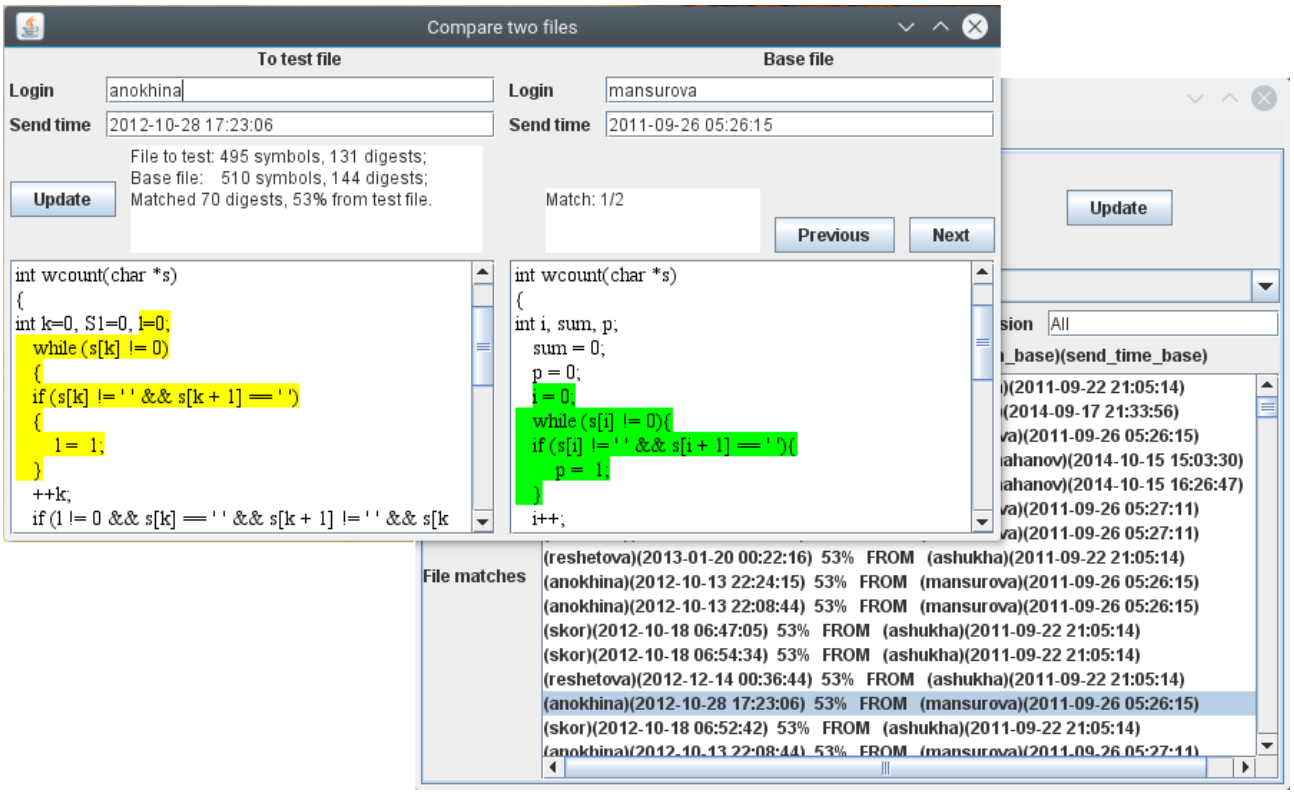


Рисунок 6 — Сравнение двух файлов.

5 Тестирование

Тестирование направлено на установку работоспособности приложения и скорости выполнения запросов. Характеристики тестирующего устройства:

- версия ядра Linux kernel 4.9.5;
- процессор Intel(R) Core(TM) i5-3317U CPU @ 1.70GHz;
- Java 1.8.0;
- JVM 25;
- ОЗУ 10 ГБ.

5.1 Скорость выполнения запросов

В таблице 1 представлены данные о скорости выполнения основных запросов.

Тесты проводились для базы данных, содержащей 989 файлов исходных текстов программ. Приведены средние результаты пяти запусков. Время выполнения вычислялось с помощью директивы *EXPLAIN (ANALYZE)*. Листинги запросов приведены в разделе «Физическое проектирование».

Таблица 1 — Результат добавления большого числа файлов.

Запрос	Время выполнения запроса
Получение всех пар файлов в базе данных, имеющих общие дайджесты (листинг 3, с. 26).	34 сек
Получение всех всех файлов, имеющих общие дайджесты с данным (листинг 4, с. 26).	21 мс
Получение координат всех совпадающих участков для двух данных файлов (листинг 6, с. 27).	1,2 мс

5.2 Добавление большого числа файлов

Для примера добавим в базу данных содержимое папки «data/011 Подсчет слов в строке».

Номер задачи: 11.

Версия задачи: 1.

Усредненные показатели пяти запусков приведены в таблице 2. Замеры производились в миллисекундах. Снимок экрана с работающим приложением приведен на рисунке 7.

Таблица 2 — Результат добавления большого числа файлов.

Параметр	Значение без индексации	Значение с индексацией
Количество файлов	989	
Количество пар файлов для сравнения	10808	
Количество уникальных дайджестов	37919	
Общее затраченное время	104 сек	105 сек
Время, затраченное на:		
– лексический анализ	1 сек	1 сек
– добавление файлов	60 сек	64 сек
– сравнение файлов	43 сек	43 сек

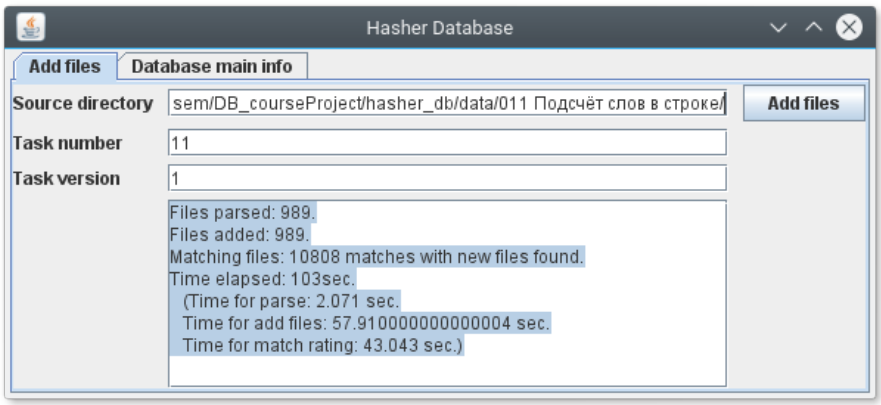


Рисунок 7 — Добавление файлов в базу данных с помощью пользовательского приложения.

Изображения остальных окон приведены в приложении 2 на рисунках 9 и 10.

5.3 Добавление одного файла

Установим, как изменяется время добавления файла при увеличении числа файлов, хранящихся в базе данных. Результаты добавления одного файла в базу данных, в которой не хранится ни одного файла, представлены в таблице 3.

Таблица 3 — Результат добавления одного файла в пустую базу данных.

Параметр	Значение без индексации	Значение с индексацией
Количество файлов	1	
Количество пар файлов для сравнения	0	
Количество уникальных дайджестов	112	
Общее затраченное время	0 сек	0 сек
Время, затраченное на:		
– лексический анализ	0,005 сек	0,005 сек
– добавление файлов	0,006 сек	0,06 сек
– сравнение файлов	0,003 сек	0,003 сек

Добавим в базу данных файлы из первого теста, за исключением двадцати двух файлов, отправленных одним человеком, чтобы впоследствии добавить в базу данных без коллизий один из его файлов. Результаты теста представлены в таблице 4. Видно, что при увеличении числа файлов в базе данных на тысячу, скорость работы падает в пятьсот раз.

Таблица 4 — Результат добавления одного файла в базу данных с большим числом файлов.

Параметр	Значение без индексации	Значение с индексацией
Количество файлов	968	
Количество пар файлов для сравнения	80	
Количество уникальных дайджестов	37674	
Общее затраченное время	1 сек	1 сек
Время, затраченное на:		
– лексический анализ	0,005 сек	0,005 сек
– добавление файлов	0,06 сек	0,07 сек
– сравнение файлов	1,7 сек	1,7 сек

Из таблиц 2–4 можно сделать вывод о том, что использование дополнительных индексов не влияет на скорость добавления тысячи файлов. Вероятнее всего это связано с тем, что в запросах фильтрация идет в основном по столбцам, являющимися первичными ключами, а для них по умолчанию создается индекс, поэтому увеличения количества проиндексированных полей приводит

к тому, что при выполнении запроса производятся избыточные обращения к индексам таблиц, не приводящие к заметному увеличению скорости работы.

5.4 Чрезвычайная ситуация

В приложении предусмотрено выведение сообщений об ошибках в случае неудачного окончания операции. Пример корректной обработки ошибки и выведения сообщения (при попытке добавить в базу данных повторно те же файлы) приведен на рисунке 8.

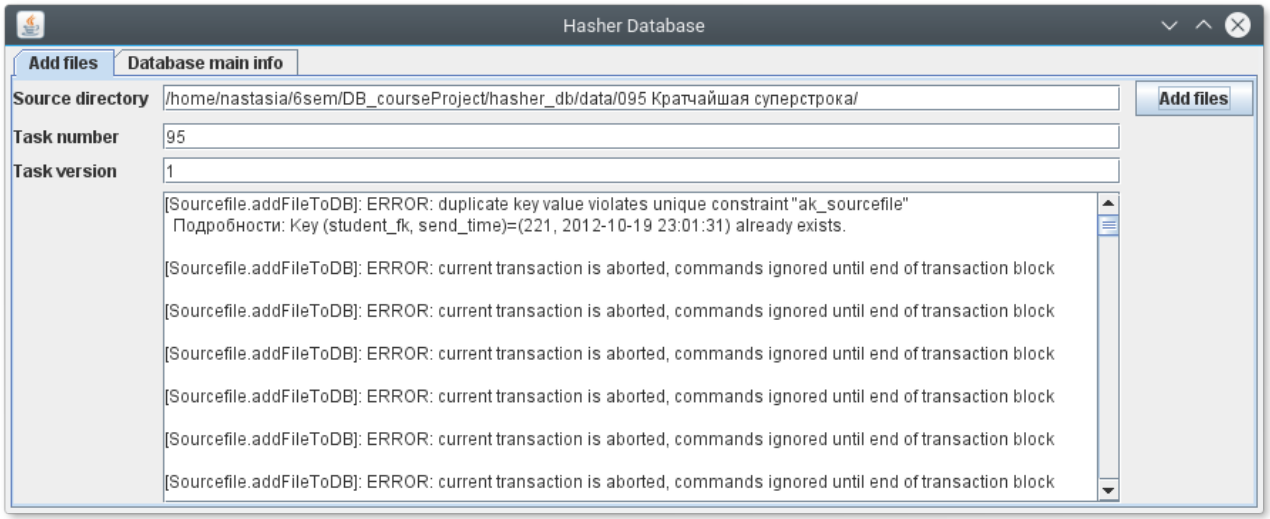


Рисунок 8 — Вывод сообщения об ошибке при попытке повторно добавить файлы в базу данных.

Заключение

Целью данной работы было создание базы данных системы обнаружения некорректных заимствований в исходных текстах программ и написание пользовательского приложения для добавления исходных текстов в базу. Одной из главных задач была реализация возможности наглядного для человека представления совпадающих участков исходных текстов программ. Эти задачи были выполнены.

Была реализована база данных на основе PostgreSQL и пользовательское приложение, позволяющее добавлять файлы в базу данных, просматривать статистику о некорректных заимствованиях, в графическом интерфейсе выбирать из списка файлов с некорректными зависимостями пары документов и просматривать их в окне с интерактивной подсветкой совпадающих участков.

Приложение позволяет за миллисекунды находить файлы с некорректными зависимостями и их источники, а также координаты совпадающих участков в базе, хранящей в себе порядка тысячи файлов.

Из недостатков реализованной системы нужно отметить заметный рост времени, требующийся на выполнение запросов с увеличением количества файлов, хранимых в базе данных. Этот недостаток связан с большим количеством соединений таблиц, которые необходимо произвести реляционной СУБД для выполнения запроса. Однако реляционная модель данных гарантирует также целостность данных и надежность их хранения средствами СУБД, поэтому выбор СУБД можно считать оправданным.

Систему можно продолжать совершенствовать в нескольких направлениях:

– использовать другую модель данных, например, графовую;

– увеличить функциональность приложения, например, добавить возможность выполнения пользовательского SQL-запроса из графического интерфейса приложения;

– денормализовать данные и сравнить производительность.

Список литературы

[1] Schleimer S., Wilkerson D., Aiken A. Winnowing: Local Algorithms for Document Fingerprinting // SIGMOD. 2003. 10 p.

[2] Бураков П.В., Петров В.Ю. Введение в системы баз данных. Учебное пособие / ИТМО. 2010.

[3] Хомоненко А. Д., Цыганков В.М., Мальцев М. Г. Базы данных: учебник для высших учебных заведений. Корона-век, 2009. 736 с.

[4] Коннолли Т., Бегг К. Базы Данных. Проектирование, реализация и сопровождение. Теория и практика. 3-е издание. Издательский дом Вильямс, 2003. 1440 с.

[5] Грофф Дж. Р., Вайнберг П. Н., Оппель Э. Дж. SQL: полное руководсво. 3-е издание. Издательский дом Вильямс, 2015. 960 с.

[6] Редмонд Э., Уилсон Дж. Р. Введение в современные базы данных и идеологию NoSQL. ДМК Пресс, 2013. 384 с.

[7] Harrington J. L. Relational Database Design and Implementation. 4th Edition. Morgan Kaufman, 2016. 1051 p.

[8] Васильев А. Ю. Работа с PostgreSQL: настройка и масштабирование. Creative Commons Attribution-Noncommercial 4.0 International, 2014. 231 с.

[9] PostgreSQL: The world's most advanced open source database [Электронный ресурс]. URL: https://www.postgresql.org/ (дата обращения: 13.03.2017).

[10] Fritchey G. SQL Server Query Performance Tuning. 4th Edition. Apress, 2014. 565 p.

[11] The PostgreSQL Global Development Group. Документация к PostgreSQL 9.5.6. 2016. 2138 с.

[12] JavaTalks Articles. PreparedStatement (подготовленные запросы) [Электронный ресурс]. URL: https://articles.javatalks.ru/articles/34 (дата обращения: 25.05.2017).

[13] PostgreSQL JDBC Driver [Электронный ресурс]. URL: https://jdbc.postgresql.org/ (дата обращения: 13.03.2017).

Приложение 1: Дополнительные листинги

Листинг 7 — Создание таблиц для сущностей «Задача», «Версия задачи», «Тест», «Студент».

```
-- Задача
CREATE TABLE Problem(
problem_id int  PRIMARY KEY, -- номер задачи
content    text, -- условие
points     int   -- баллы
);

-- Версия задачи
CREATE TABLE ProblemVersion(
problem_ver_id SERIAL    PRIMARY KEY, -- идентификатор
problem_fk        int   -- номер задачи
    REFERENCES Problem(problem_id) NOT NULL ON DELETE CASCADE,
ver            int       NOT NULL, --версия
creation_time  timestamp NOT NULL  --дата создания
    DEFAULT localtimestamp,
--задание альтернативного ключа
CONSTRAINT AK_ProblemVersion UNIQUE(problem_fk, ver)
);

--Тест
CREATE TABLE Test(
test_id     SERIAL PRIMARY KEY, -- идентификатор
problem_fk  int    NOT NULL     -- задача
    REFERENCES ProblemVersion(problem_ver_id) ON DELETE CASCADE,
test_number int    NOT NULL,   -- номер теста
content     text   NOT NULL,   -- исходный текст
answer      text   NOT NULL,   -- ответ
--задание альтернативного ключа
CONSTRAINT AK_Test UNIQUE(problem_fk, test_number)
);

-- Студент
CREATE TABLE Student(
student_id   SERIAL      PRIMARY KEY, -- идентификатор
login        varchar(20) UNIQUE NOT NULL, -- логин
surname      varchar(50) NULL, -- фамилия
student_name varchar(50) NULL, -- имя
patronymic   varchar(50) NULL, -- отчество
department   varchar(5)  NULL, -- факультет
group_num    int         NULL  -- номер группы
);
```

Листинг 8 — Создание представления «LexemSourceFile».

```
-- Представление для сопоставления информации о лексеме и полной информации о файле, которой она принадлежит.
CREATE VIEW LexemSourceFile AS SELECT
SourceFile.file_id, SourceFile.student_fk, SourceFile.task_fk, SourceFile.code, SourceFile.code_language, SourceFile.send_time,
Lexem.array_index, Lexem.digest, Lexem.lexem_type, Lexem.start_coord, Lexem.follow_coord
FROM SourceFile JOIN Lexem ON SourceFile.file_id = Lexem.from_file_fk;
```

Листинг 9 — Вычисление длины массива дайджестов для данного файла.

```
-- Создание функции, возвращающей количество дайджестов для файла с данным id
CREATE OR REPLACE FUNCTION countDigestArrayLength(IN id int)
RETURNS int AS
DECLARE counter int := (SELECT COUNT(digest) FROM Lexem WHERE from_file_fk = id);
BEGIN
RETURN counter;
END;
 LANGUAGE plpgsql;

-- Пример использования функции в запросе запрос( информации о файле с номером 345):
SELECT *, countDigestArrayLength(345) AS DIGEST_NUM
FROM SourceFile WHERE file_id = 345;
```

Приложение 2: Снимки экрана

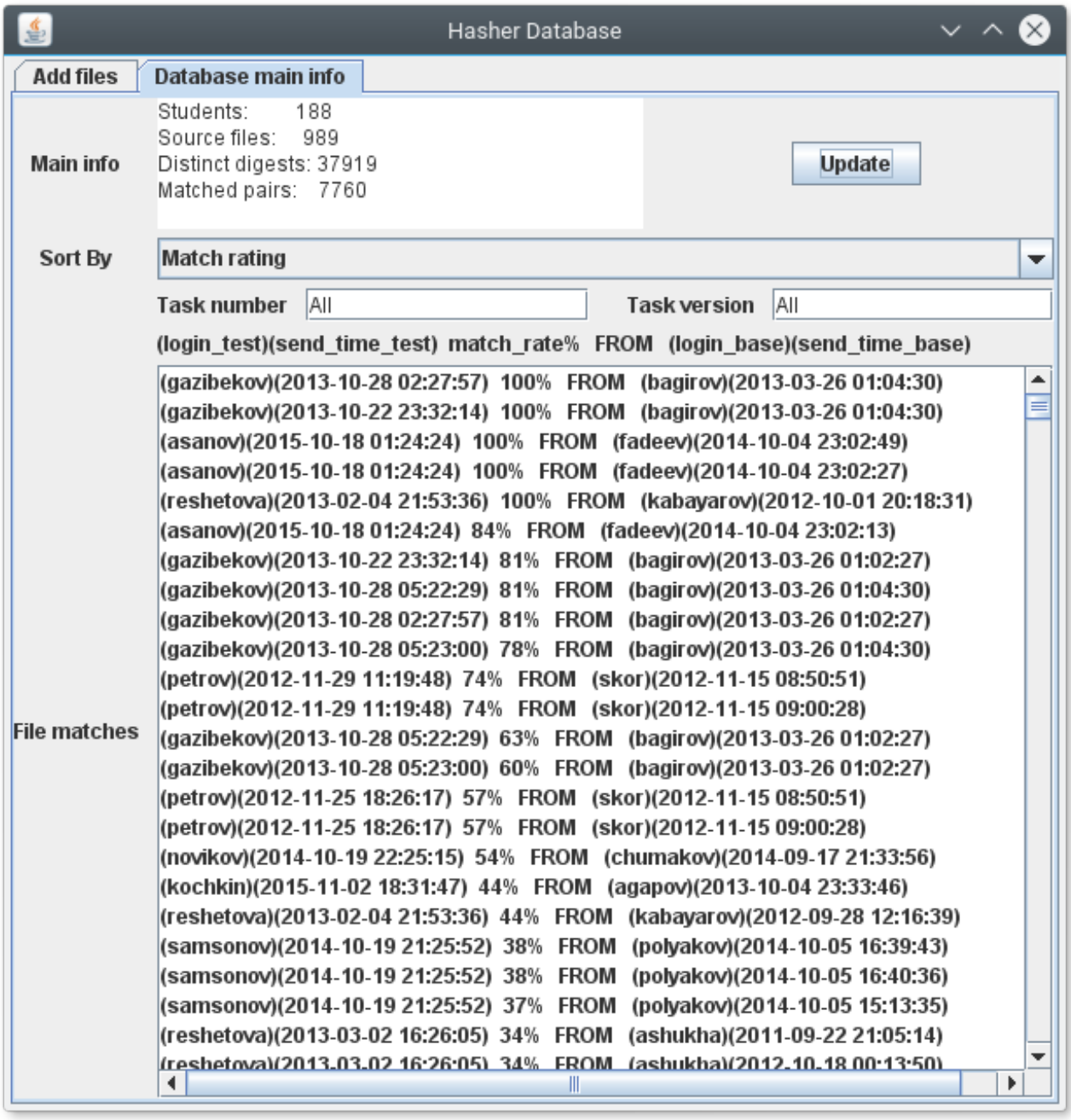
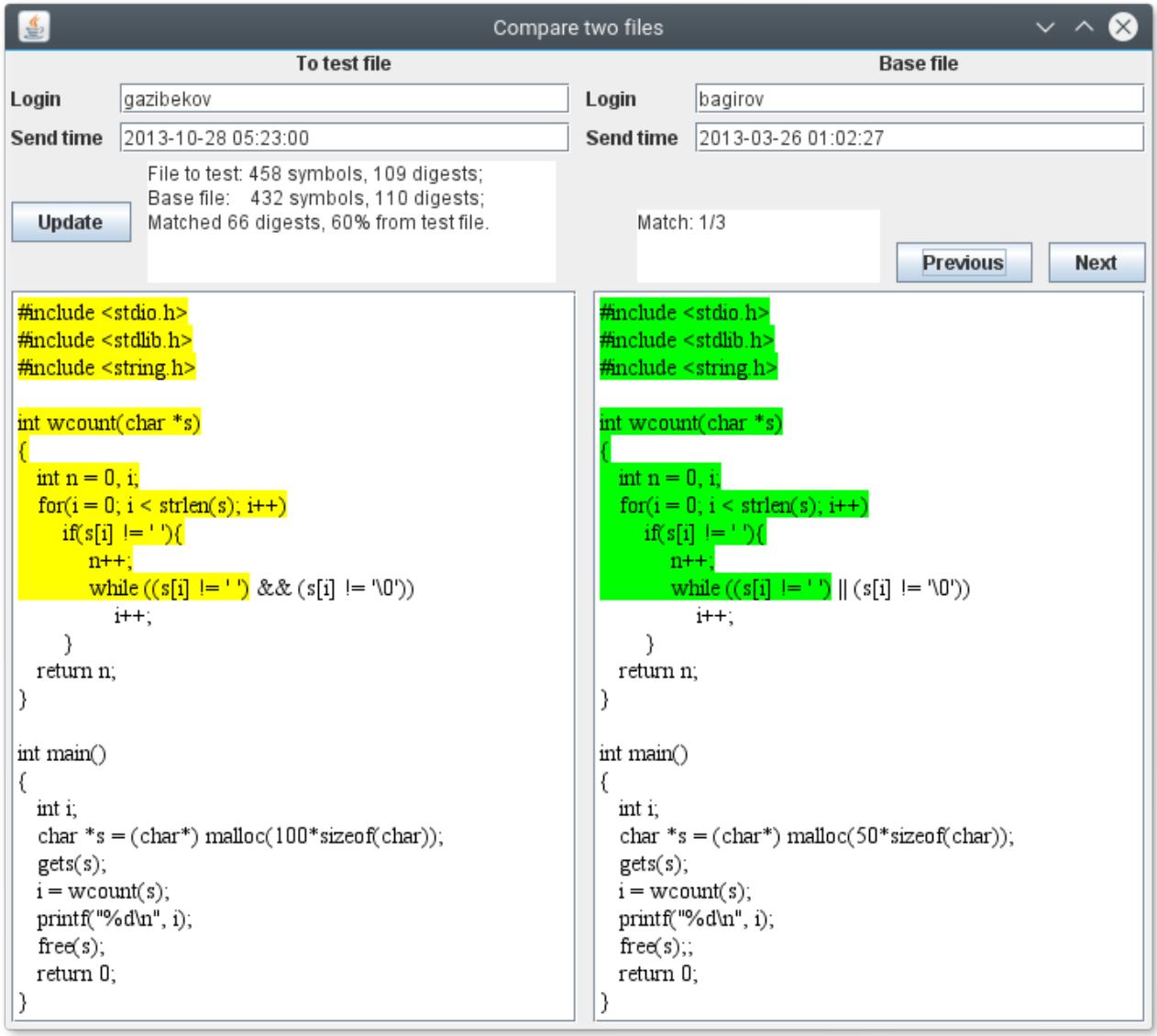
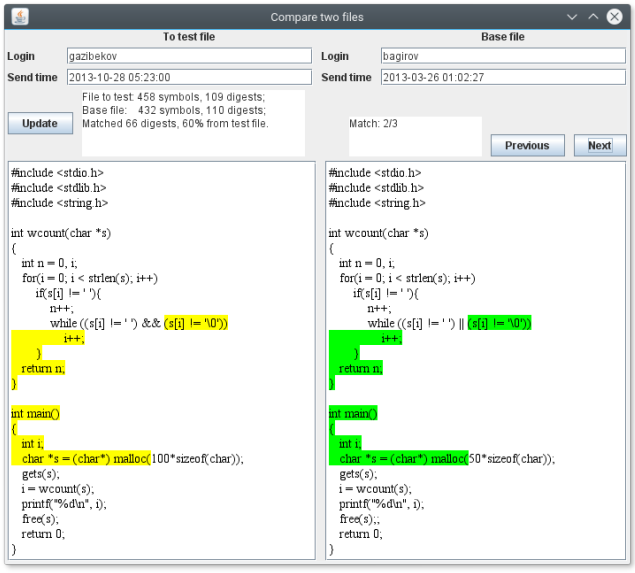
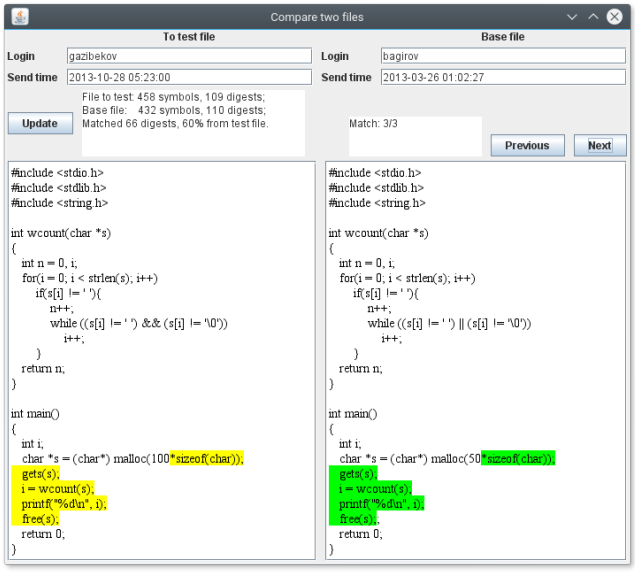


Рисунок 9 — Информация о базе данных после добавления 989 файлов.



а)





б) в)

Рисунок 10 — Сравнение двух файлов.

Переключение подсветки между схожими участками кода (а), (б) и (в).