Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

Московский государственный технический университет имени Н.Э. Баумана
(МГТУ им. Н.Э. Баумана)

Факультет: «Информатика и системы управления»
Кафедра: «Теоретическая информатика и компьютерные технологии»

Расчетно-пояснительная записка к курсовому проекту по дисциплине «Базы данных»

Конвертация базы данных T-BMSTU в формат MySQL

Руководитель курсового проекта: _____ (С.Ю. Скоробогатов)
(подпись, дата)

Исполнитель курсового проекта,
студент группы ИУ9-62: _____ (А. В. Беляев)
(подпись, дата)

Москва, 2016

Содержание

1 ВВЕДЕНИЕ

Основная цель данной работы — конвертация базы данных автоматизированной системы тестирования T-BMSTU, используемой на кафедре ИУ9 для проведения лабораторных работ по курсам программирования, в формат MySQL. Сравнение производительности новой реализации базы и, при необходимости, оптимизация запроса к базе, либо оптимизация имеющейся базы данных в новом формате.

В ходе работы будет исследована имеющаяся исходная реализация базы в формате SQLite. Затем, данные из исходной базы будут извлечены и перенесены в новую, целевую базу данных с помощью конвертера. Затем в новый формат будет конвертирован модельный SQL запрос и будет исследовано его выполнение на новой базе данных. Этот запрос будет оптимизирован под целевую реализацию базы данных.

Путем сравнения производительности реализаций «новой» и «старой» баз данных и запросов будет принято решение о целесообразности перехода на новую реализацию.

2 ОБЗОР СИСТЕМ

Прежде всего стоит рассмотреть отличительные черты исходной и целевой СУБД. После этого мы сможем заключить, возможна ли в теории выгода от перехода от одной СУБД к другой.

2.1 SQLite. Преимущества и недостатки

SQLite является популярной встраиваемой реляционной базой данных. Релиз последней на данный момент версии одноименной СУБД (SQLite 3.14.1) состоялся в августе 2016 года. СУБД выпускается под общественной (public domain) лицензией, не накладывающей никаких ограничений на использование.

Далее будут рассмотрены особенности базы данных. [1]

Главным отличием SQLite от других баз данных является парадигма, в которой создана база. В отличие от большинства остальных баз данных, использующих клиент-серверную архитектуру, само приложение SQLite по сути является сервером. SQLite не является отдельным процессом. Вместо этого она предоставляет библиотеку для создания подключений к единственному файлу, в виде которого она находится в конечной системе.

Относительная простота реализации такого подхода является, пожалуй, главной отличительной чертой этой СУБД. Вся база хранится в одном файле. Это позволяет существенно экономить ресурсы системы, сокращает время отклика и существенно упрощает логику работы программ, использующих эту БД. База данных является единственным файлом в кросплатформенном формате, что обеспечивает бОльшую мобильность по сравнению с другими СУБД, так как для работы с БД в новой системе, развертывание базы не требуется.

Однако, у такой реализации есть и обратная сторона. Простота реализации достигается за счет того, что во время записи весь файл блокируется одним процессом. А значит, несколько процессов, одновременно подключенных к базе могут лишь считывать данные, в то время, как только один из них может изменять эти данные. Этот принцип, «читают многие – пишет один», является одним из недостатков этой СУБД.

Еще одним недостатком является отсутствие полной поддержки SQL-92: Не поддерживается, например, удаление или изменение столбца в таблице с помощью команд: `ALTER TABLE DROP COLUMN ...` `ALTER TABLE ALTER COLUMN ...` отсутствуют в SQLite; Опущены `RIGHT OUTER JOIN` и `FOR EACH STATEMENT` По умолчанию отключена поддержка foreign key; Недоступны хранимые процедуры; Триггеры SQLite намного менее функциональны, нежели триггеры других СУБД.

Для взаимодействия с SQLite из приложений отсутствуют официальные драйвера. Таковых нет ни под JDBC, ни под ADO.Net, ни под ODBC. Отсутствие этого «из коробки» является существенным минусом SQLite.

Еще одной особенностью SQLite является «слабая типизация», или концепция «близости типов» (type affinity). Так, тип столбца не определяет тип хранимого в этом столбце значения. В любой столбец может быть записано любое значение, а сам тип столбца используется для приведения значений к одному типу при сравнении значений. Все это позволяет создавать таблицу «простым» `CREATE TABLE sampleTable`

`(col1, col2, col3)` без указания чего либо еще, что является недопустимым и недоступным в других СУБД.

Однако, в SQLite доступны лишь 5 типов данных: NULL; INTEGER (знаковое целое число до 8 байт); REAL (число с плавающей точкой, 8 байт в формате IEEE); TEXT (строка в кодировке UTF-8 или UTF-16); BLOB (входное значение, «как есть»).

Согласно руководству, для хранения типа Boolean рекомендуется использовать INTEGER 0 или 1, а Date и Time типы хранить в виде строк. Вышесказанное является одновременно как недостатком, так и достоинством и не может быть однозначно интерпретировано в рамках «общих» задачах.

В SQLite также отсутствуют какие-либо механизмы репликации.

Отсутствует система пользователей.

Отсутствует возможность увеличения производительности.

Несмотря на все это, SQLite является отличным кандидатом на использование в качестве встраиваемой системы и пользуется большой популярностью в этой сфере. [5]

2.2 MySQL. Преимущества и недостатки

Теперь необходимо рассмотреть целевую базу данных, MySQL.

MySQL является самой распространенной СУБД. Разрабатывается корпорацией Oracle, как доступная под универсальной общественной лицензией GNU (GNU General Public License) замена промышленной БД Oracle.

Далее рассматриваются особенности MySQL. [2]

MySQL разработана в соответствии с «классической» клиент-серверной архитектурой и поддерживает все основные ОС. Для работы с базой из приложений имеются официальные драйвера ADO.Net, JDBC и ODBC. Хотя количество языков, поддерживаемых API MySQL и меньше, чем у SQLite, недостатком это не является, так как упущены не самые популярные в настоящее время языки, такие, как Basic, Forth и Fortran.

MySQL поддерживает большое количество типов данных: TINYINT (BOOL), SMALLINT, MEDIUMINT, INTEGER, BIGINT для целочисленных значений; FLOAT, DOUBLE, NUMERIC, REAL для значений с плавающей точкой; DATE, TIME, DATETIME, YEAR, TIMESTAMP для значений даты и времени; CHAR, VARCHAR для строковых значений фиксированной / переменной длины; TINYTEXT, TEXT, MEDIUMTEXT, LONGTEXT для текстовых значений длины $2^8 - 1$ / $2^{16} - 1$ / $2^{24} - 1$ / $2^{32} - 1$ соответственно; TINYBLOB, BLOB, MEDIUMBLOB, LONGBLOB для значений «как есть»; ENUM, SET для значений типа перечисление / множество.

Это является существенным преимуществом MySQL перед SQLite, так как таблицы могут быть построены более оптимально в соответствии с бизнес логикой.

Поддерживается почти полный стандарт SQL-92 (DML, DDL, DCL), хотя и присутствует проприетарное расширение синтаксиса. В частности, в этой БД используется свой синтаксис для написания триггеров, а также хранимых процедур, что не поддерживаются в SQLite. Однако, стоит отметить, что в MySQL упущены, в частности, INSTEAD OF триггеры, хотя они и могут быть самостоятельно реализованы

отдельно.

В Mysql присутствуют механизмы репликации, так как разработчики создавали функциональность по заказу лицензионных пользователей и это (репликация) является важным фактором при выборе БД для коммерческого использования. Так, поддерживается Многомастерная (Multi-master replication) репликация. Данные в данном случае хранятся группой устройств и могут быть изменены любым устройством из этой группы»мастеров», так и Система с ведущими и ведомыми устройствами (Master-slave), где ведущая база данных рассматривается, как «авторитетный» источник данных, а подчиненные синхронизируются с ней.

Все это существенно повышает надежность работы этой БД и отличает ее от SQLite, где механизмы репликации впринципе отсутствуют.

Mysql поддерживает многопоточную работу с помощью механизма блокировки таблиц или строк, что является более вариативной функциональностью, нежели захват всего файла базой SQLite.

Присутствуют механизмы управления уровнем доступа пользователей, хотя отсутствуют механизмы деления пользователей на роли и группы.

Система Mysql масштабируема.

Также, среди преимуществ можно отметить наличие множества «движков» (database engine), каждый из которых обладает своими преимуществами и может быть выбран согласно решаемой задаче. В ходе работы некоторые из движков будут рассмотрены чуть подробнее.

Благодаря свободному доступу к исходному коду, эта СУБД может быть подстроена под индивидуальное решение. А вывсокая популярность системы обеспечивает наличие поддержки по многим возможным проблемам.

Исходя из всего вышеперечисленного, переход с SQLite на MySQL видится разумным в рамках большей части задач общего назначения. MySQL объективно обладает большим числом преимуществ перед SQLite.

Однако, в рамках курсовой работы рассматривается конкретная база данных автоматизированной системы тестирования T-BMSTU. И для оценки реальной выгоды от перехода с одной базы на другую необходимо прежде всего оценить уместность преимуществ MySQL перед SQLite в рамках конкретной задачи.

Для этого рассмотрим теперь исходную базу данных T-BMSTU.

3 ИЗУЧЕНИЕ ВХОДНЫХ ДАННЫХ

Исходными данными курсовой работы являются SQL скрипт создания базы данных, непосредственно база данных в виде файла *tbmstu.db* и запрос, формирующий выборку данных из базы для отображения на Web-сервере тестирования.

3.1 Исходная база данных T-BMSTU

Взглянем на устройство базы данных. Исходная база данных содержит в себе 22 таблицы:

```
Institutions
Subjects
```

```
Modules
Groups
RelGroupsModules
Persons
Sessions
CurrentAdmins
CurrentTaskAuthors
Students
ModuleAuthors
Approvers
Languages
Tasks
RelTasksLanguages
RelLanguagesModules
Submissions
FailedTests
Comments
Approvements
RelTasksModules
```

Также имеются 6 представлений:

```
TimeDesc
RelModulesInstitutions
RelSubmissionsApprovers
FinalApprovements
PersonRoles
RelStudentsTasks
```

Присутствуют 7 триггеров, выводящих сообщение об ошибке, в случае нарушений работы с внешними ключами (вставки и обновления):

```
fki_RelGroupsModules
fku_RelGroupsModules
fki_Tasks_PersonId
fku_Tasks_PersonId
fki_Approvements_PersonId
fku_Approvements_PersonId
fku_Submissions_PassedTests
```

Назначение таблиц, представлений и триггеров понятно из их названий.

В большинстве таблиц в базе хранится по несколько десятков записей – это таблицы со списками студентов групп, предметов и модулей, таблица-временная шкала, таблица авторов заданий. Есть таблицы на одну или несколько сотен записей: аккаунты в системе, задания, проверяющие и вспомогательные таблицы. Есть таблица университетов с одной записью, так как на данный момент система применяется только в МГТУ. И есть 4 таблицы с десятками тысяч записей. В порядке убывания по количеству записей (по данным на февраль 2016):

Sessions, ≈ 130 тыс. записей; Submissions, ≈ 80 тыс. записей; Comments, ≈ 70 тыс. записей; Approvements, ≈ 70 тыс. записей.

Очевидно, что работа с этими таблицами представляет наибольшую сложность не только по причине частой записи в них, но и по причине того, что эти таблицы имеют дочерние записи и ссылаются на большое число других таблиц. Рассмотрим, например, таблицу решений, отправленных на сервер тестирования – таблицу *Submissions*:

Листинг 1: Скрипт создания таблицы Submissions

```
CREATE TABLE Submissions (
    SubmissionID    INTEGER NOT NULL PRIMARY KEY,

    PersonID    INTEGER NOT NULL,
    GroupID     INTEGER NOT NULL,
    TaskID      INTEGER NOT NULL,
    ModuleID    INTEGER NOT NULL,
    SubmissionTime  TEXT NOT NULL CHECK (length(SubmissionTime) < 32),

    LanguageID  INTEGER NOT NULL,
    SourceCode  TEXT NOT NULL CHECK (length(SourceCode) < 50*1024),
    Draft       INTEGER CHECK (Draft = 0 OR Draft = 1),

    SentToTestTime  TEXT,
    PassedTests     INTEGER,
    TestingServerId INTEGER,

    UNIQUE (PersonID, GroupID, TaskID, ModuleID, SubmissionTime),
    FOREIGN KEY (PersonID, GroupID) REFERENCES Students
                                    ON DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (TaskID, ModuleID)  REFERENCES RelTasksModules
                                    ON DELETE RESTRICT ON UPDATE CASCADE,
    FOREIGN KEY (GroupID, ModuleID) REFERENCES RelGroupsModules
                                    ON DELETE RESTRICT ON UPDATE CASCADE,
    FOREIGN KEY (TaskID, LanguageID) REFERENCES RelTasksLanguages
                                    ON DELETE RESTRICT ON UPDATE CASCADE,
    FOREIGN KEY (LanguageID, ModuleID) REFERENCES RelLanguagesModules
                                    ON DELETE RESTRICT ON UPDATE CASCADE
);
```

Как видно из листинга 1, эта таблица ссылается сразу на 5 других таблиц. А значит, при добавлении записи в нее необходимо обратиться сразу к 5 таблицам. Также с этой таблицей работает триггер *fku_Submissions_PassedTests*, проверяющий наличие валидного сервера тестирования, назначенного по решению.

Тут же становятся видны ограничения, озвученные ранее при анализе SQLite «из коробки», такие, как ограниченное число типов данных: *SubmissionTime* хранится в виде строки TEXT, *Dratf* хранится в виде INTEGER'а. Из-за этого приходится осуществлять валидацию входных данных проверяя длину или значние.

Также необходимо соблюдение уникальности группы полей *PersonID, GroupID, TaskID, ModuleID, SubmissionTime* и PRIMARY KEY.

Работа с этой таблицей весьма труднозатратна относительно работы с остальными таблицами в рамках базы данных.

У этой таблицы имеются сразу 4 вручную созданных индекса. Учтем это в дальнейшем при возможной оптимизации этой таблицы.

Таблиц, подобных этой, в базе еще 3. Нужно отметить, что таблица *Sessions* выделяется среди «крупных таблиц» своими размерами, а также относительной простотой работы: в ней есть только валидация ip-адреса по длине и ссылка на таблицу пользователей сервера для прикрепления его к сеансу. Поиск по этой таблице не нужен, поэтому есть только стандартный PRIMARY KEY

Эти 4 таблицы постоянно используются в ходе работы сервера. Данные в них обновляются постоянно.

Большинство данных остальных таблиц редактируется либо с началом модуля (например, таблицы *Modules, Tasks*), либо семестра (такие, как *Subjects, Groups, TimeScale*), либо с началом учебного года (*Persons, Groups, TimeScale*). Таблицы *Languages, CurrentAdmins, CurrentTaskAuthors, Institutions* редактируются еще реже. То есть, данные большей части таблиц редактируются совсем не часто. Но, бОльшая часть данных всей базы редактируется постоянно. Учтем это в дальнейшем.

3.2 Исходный запрос к базе

Во входных данных также присутствует запрос к описанной выше базе. Этот запрос формирует выборку данных для отображения на странце Web-сервера. Взглянем на запрос в его первоначальном варианте (см. Листинг 1 Приложения 1)

В запросе из Листинга 1 Приложения 1 присутствуют 11 выборок с помощью SELECT из большей части таблиц базы. В том числе есть обращения к «большим» таблицам – *Submissions* и *Approvements*. Присутствует множество объединений таблиц, несколько группировок по учебным модулям и группам а также с использованием агрегирующих функций, несколько DISTINCT выборок и сортировок.

Выполнять оптимизации согласно заданию необходимо «с оглядкой» на этот запрос.

4 ПОРТИРОВАНИЕ БАЗЫ ДАННЫХ

Теперь, когда есть представление об устройстве исходной базы данных SQLite, необходимо портировать ее на целевую платформу MySQL.

Стандартные реверс-инжиниринговые утилиты для портирования могут некорректно взаимодействовать со схемой базы данных и в результате некоторые таблицы могут не создаться, а внутреннее устройство других будет отличаться. К тому же нам необходимо произвести некоторые изменения типов согласно best practices, описанных в руководстве пользователя MySQL. [3]

В случае данной курсовой работы, имеется преимущество перед такими утилитами в виде еще одного входного элемента – скрипта создания базы. Все что необходимо

сделать в данном случае – перенести непосредственно данные в подготовленную на новом месте схему.

Так как обе СУБД не полностью поддерживают формат SQL-92, простым запуском скрипта создания базы обойтись не удастся. Необходимо преобразовать исходный скрипт с учетом отличий диалекта SQL, используемого в SQLite от диалекта, используемого в MySQL. Также выполнить преобразования типов данных, заменив строки, содержащие даты и время на тип DATETIME. Другие строковые константы заменить на тип VARCHAR, более предпочтительный в рамках MySQL. Своеобразный Boolean в SQLite, реализованный через INTEGER и проверку значений с помощью CHECK() заменить на TINYINT, являющийся, аналогом BOOLEAN'a в MySQL. Проведем еще некоторые преобразования и портируем триггеры, переведя их на проприетарный синтаксис триггеров и хранимых процедур MySQL.

Подразумевается, что схема базы данных уже подготовлена. Разработанная небольшая утилита на *C#*, которая с помощью технологии ADO.Net доступа приложений на платформе .Net к данным выберет все данные из базы SQLite и вставит в подготовленную схему базы MySQL. Единственной возможной проблемой является отсутствие официальных драйверов SQLite для ADO.Net, однако это решается загрузкой аналога через встроенный менеджер пакетов NuGet. Запускаем утилиту и получаем на выходе наполненную базу данных MySQL с описанными выше изменениями. Триггеры необходимо добавить вручную.

Поскольку никаких других существенных изменений сделано не было, схема базы данных и внутреннее устройство таблиц с точностью до типов некоторых столбцов остались без изменений и повторно приводить их не имеет смысла. Портирование на этом завершено. На данный момент имеется полный MySQL аналог исходной SQLite базы.

5 ОПТИМИЗАЦИИ В MYSQL

На данный момент имеется база данных MySQL. Можно считать, что все дальнейшие операции, если не оговорено иное, происходят над ней.

Запрос, рассмотренный ранее в Листинге 1 Приложения 1 не сработает в своем исходном виде на новой базе из-за отличия используемых диалектов. В MySQL у каждой выборки должен быть свой алиас (alias, псевдоним). Проименуем все таблицы в запросе, добавив к ним соответствующие технические наименования.

Теперь, выполнив запрос на новой базе, фиксируем результат запроса, чтобы при дальнейшем его изменении их можно было сравнить и проверить, выдается одинаковая выборка или нет.

Перейдем теперь к главной части этой работы — исследованию и оптимизации запроса и/или базы.

5.1 Исследование выполнения запроса на базе MySQL

Как уже отмечалось ранее, у MySQL имеются некоторые преимущества перед SQLite. Среди них – наличие команды EXPLAIN. При выполнении любого запроса, оптимизатор запросов MySQL создает наиболее оптимальный план его выполнения.

Этот план можно посмотреть с помощью команды EXPLAIN. Эта команда является одним из самых мощных инструментов разработчика, доступных в MySQL. [4]

Выполним эту команду вместе с запросом из Листинга 1 Приложения 1.

Рассмотрим результат выполнения запроса в таблице Приложения 3. Результат выполнения состоит из 11 столбцов:

- Id – порядковый номер SELECT'a внутри запроса
- Select_type - тип запроса SELECT. Среди возможных значений:
  - PRIMARY – самый внешний запрос в JOIN'e
  - DERIVED – данный запрос является подзапросом
  - SUBQUERY – первый SELECT в подзапросе
  - UNION – второй или последующий SELECT в UNION'e
  - UNION RESULT – результат UNION'a
- Table – таблица, к которой относится строка результата
- Type – тип связывания таблиц. Среди возможных значений:
  - Const – таблица имеет только одну соответствующую строку, которая проиндексирована. Таблица в данном случае читается лишь раз и в дальнейшем значение строки воспринимается, как константа. Это наиболее быстрый тип связывания
  - Eq_ref – все части PRIMARY KEY или UNIQUE NOT NULL индекса используются для связывания. Еще один наилучший тип связывания
  - Ref – прямая ссылка. Все строки индексного столбца противопоставляются строкам предыдущей таблицы. Неплохой вариант
  - Index – сканирование всего индексного дерева для поиска строк
  - All – худший тип связи. Для нахождения строк используется полнотекстовое сканирование таблицы. Указывает на отсутствие подходящих индексов в таблице
- Possible_keys – возможные индексы. Возможно, они не используются. Значение NULL указывает на отсутствие подходящих индексов в таблице.
- Key – фактически использованный ключ. Может отличаться от указанных в *Possible_keys* значений
- Key_len – длина используемого ключа
- Ref – столбцы или константы, которые сравниваются с индексом
- Rows – число обработанных записей
- Filtered – процент отфильтрованных записей

- Extra – дополнительная информация об обработке запроса. Среди возможных значений:
  - Using index – информация получена с применением индексного дерева без доп. поиска для чтения строки. Возможно при всех проиндексированных столбцах.
  - Using temporary – создание временной таблицы. Например, при ORDER BY на наборе столбцов, отличном от набора в GROUP BY
  - Using filesort – Дополнительный проход с сохранением ключей строк, которые попали под условие WHERE и последующей сортировкой самих ключей.
  - Using join buffer (Block Nested Loop) – использование буфера для сохранения таблиц с последующим их объединением путем выборки подходящих строк из буфера

Выше представлена интерпретация значений результата EXPLAIN'a. Оценим результат нашего запроса:

- В процессе выполнения запроса полнотекстово сканируются сразу 6 таблиц, которые впоследствие хранятся в виде временных таблиц (*type: all*)
- В первой выборке присутствует полное сканирование индексного дерева первой просматриваемой таблицы (*type: index*), несмотря на то, что в таблице содержится одна запись
- Большая длина используемого ключа таблицы первой выборки
- Использование индексного дерева (*extra: using index*) для создания временной таблицы (*extra: using temporary*) и дополнительный проход по временной таблице для сортировки (*extra: using filesort*) при том, что в таблице 1 запись
- Большое количество вложенных (*Select_ type: DERIVED*) запросов, 10 подзапросов
- Присутствуют 2 таблицы (*id: 5,6*) с большим количеством просмотренных записей (относительно результата запроса) (*rows: 2247*)
- Отсутствуют данные о ключах в шести запросах (*id: 1, 4, 5, null, 2, null*)
- Отсутствуют данные о количестве/проценте обработанных строк (*rows/filtered: null*) в процессе объединения (*select_ type: union reslut*) выборок *id:5 + id:10* и *id:3 + id:4*

5.2 Оптимизация запроса

«Узкие места» запроса на данный момент ясны. Приступим к оптимизации запроса. Для этого вновь обратимся к best practices по оптимизации запросов из руководства к MySQL и выполним некоторые преобразования.[3]

5.2.1 Оптимизация условия WHERE

Оптимизатор MySQL умеет удалять ненужные скобки, сворачивать константы, удалять ненужные условия. Однако, эти действия выполняются практически без затрат и выполнение этих преобразований вручную можно опустить, чтобы оставить запрос в более понятном и удобном для чтения виде.

В некоторых случаях, если все столбцы в индексе числовые, MySQL может читать строки из индекса, совсем не обращаясь непосредственно к данным.

Среди возможных доступных, но еще не примененных оптимизаций выделяется «углубление» условия WHERE в запросе. Так, при JOIN'е выборка строк, подходящих под условие, будет осуществляться до объединения, а значит при выполнении запроса не придется просматривать конечную «большую» выборку.

Эффективность данной оптимизации будет протестирована позднее.

Имеется как минимум одна возможность применить эту оптимизацию к основному запросу. Это будет сделано в случае, если тестирование оптимизации покажет приемлемый результат.

Листинг 2: основной запрос до I оптимизации WHERE

```
FROM Students as a3, RelGroupsModules as a4
LEFT OUTER JOIN RelTasksModules USING(ModuleId)
LEFT OUTER JOIN Tasks USING(TaskId)
LEFT OUTER JOIN Submissions
        ON Submissions.PersonId = 1 AND ...
        AND (Submissions.Draft IS NULL OR Submissions.Draft = 0)
WHERE a3.PersonId = 1
        AND a3.GroupId = a4.GroupId
```

Листинг 3: основной запрос после I оптимизации WHERE

```
FROM (
        SELECT PersonID, a3.GroupID, ModuleID, ExpireTime
                FROM Students as a3,
                relgroupsmodules as a4
                WHERE a3.PersonId = 1
                        AND a3.GroupID = a4.GroupId
) as pg
```

На листингах 2 и 3 представлены запрос до и после оптимизации, озвученной выше. Существует еще одна возможность применить оптимизацию к основному запросу:

Листинг 4: основной запрос до II оптимизации WHERE

```
SELECT * FROM ( ... )
UNION SELECT ... FROM ...
LEFT OUTER JOIN Persons
        ON Persons.PersonId = Submissions.PersonId
WHERE a1.PersonId = 1
```

Листинг 5: основной запрос после II оптимизации WHERE

```
UNION SELECT ... FROM (
        SELECT * FROM Approvers
                WHERE PersonID = 1
) AS a1
```

На листингах 4 и 5 применена еще одна аналогичная оптимизация WHERE.

5.2.2 Оптимизация ORDER BY / GROUP BY

При использовании GROUP BY в общем случае при выполнении запроса будет просканирована вся таблица, затем будет создана временная таблица для распределения записей по группам и применения к ним агрегирующих функций. Но, в некоторых случаях MySQL может поступить иначе, если имеет дело с индексами.

Самым важным предусловием в данном случае является наличие индекса по всем столбцам, фигурирующим в GROUP BY. В таком случае создания дополнительной таблицы может не потребоваться и она будет заменена работой с индексным деревом.

В MySQL заложены 2 способа выполнения GROUP BY запроса с использованием индексов. Рассмотрим их по отдельности.

Loose Index Scan — «свободное» сканирование индекса. Самый эффективный способ обработки GROUP BY - когда индекс используется чтобы выбрать столбцы группировки. MySQL может в данном случае выгодно использовать свойство хранения ключей, подразумевающее их сортировку. Это свойство позволяет выбирать группы из индекса без необходимости рассматривать все ключи, удовлетворяющие условию WHERE. Сканирование в таком случае принято называть «свободным». Столбцы, фигурирующие в GROUP BY при этом обязательно должны составлять префикс в каком-либо из индексов. Есть и еще одно условие - допустимы только агрегирующие функции MIN() и MAX() и ссылаются они при этом на один и тот же столбец, присутствующий в индексе и следующий непосредственно за столбцами GROUP BY. Для столбцов должны быть созданы полноценные индексы, индексирующие значения соответствующих столбцов полностью. В случае, если все это будет выполнено, в столбце *Extra* соответствующей выборки будет значение *Using index for group-by*. Однако, добиться выполнения всех этих условий довольно сложно. В таком случае применяется Tight Index Scan.

Tight Index Scan — «плотное» сканирование индекса. В случае, если условия для свободного сканирования не могут быть выполнены, все еще можно добиться результата без создания дополнительной таблицы. Если в условии WHERE присутствует проверка диапазона значений, данный метод читает лишь индексы, удовлетворяющие условиям. Иначе он запускает сканирование индекса. Лишь после этих операций начинается группировка значений.

Для начала необходимо проверить оптимизацию на примере, в котором применяется форсированное использование индекса. Проверка будет осуществлена в рамках секции тестирование.

Для оптимизации «боевого» запроса мы можем добавить индесы на сориуемые /группируемые поля, либо добиться использования левых префиксов уже имеющихся индексов. В данном случае необходимо избавиться от повторной выборки (таблицы

a7 и *a8*):

Листинг 6: основной запрос до I оптимизации GROUP BY

```
SELECT ...
        FROM (
        SELECT DISTINCT ... FROM ...
        GROUP BY GroupId, ModuleId, TaskId
) as a8
```

Листинг 7: основной запрос после I оптимизации GROUP BY

```
SELECT ...
        FROM ( ... ) AS a7
GROUP BY GroupId, ModuleId, TaskId
```

Итак, дублирующей выборки больше нет, как следует из Листингов 6 и 7. Применим эту же оптимизацию еще раз:

Листинг 8: основной запрос до II оптимизации GROUP BY

```
(SELECT GroupId, ModuleId, max(IsStudent) AS IsStudent, max(IsApprover) AS IsApprover
        FROM (
                ...
        ) as a10
        GROUP BY GroupId, ModuleId
) as a12
```

Листинг 9: основной запрос после II оптимизации GROUP BY

```
(SELECT GroupId, ModuleId, IsStudent, IsApprover
        FROM WorkRole
                ...
        GROUP BY GroupId, ModuleId
) as a12
```

Согласно Листингам 8 и 9 было сокращено количество используемых таблиц путем ввода дополнительной таблицы *WorkRole*, речь о которой подробнее будет немного позднее. Сейчас стоит лишь отметить, что в запросе задействован алгоритм группировки по индексам, имеющимся в *WorkRole*.

Оптимизируем теперь и ORDER BY с одним допущением. Поскольку система T-BMSTU на данный момент используется только в МГТУ, обращение к таблице *Institutions* можно заменить выборкой из этой таблицы константы (самого быстрого типа связи):

Листинг 10: основной запрос до оптимизации ORDER BY

```
SELECT ... FROM ...
JOIN ... ON ...
...
ORDER BY
```

```
        Institutions.InstitutionId ASC, ...
```

Листинг 11: основной запрос после оптимизации ORDER BY

```
SELECT ... FROM ...
JOIN ... ON ...
JOIN (SELECT institutions.InstitutionID, institutions.InstitutionName FROM ...
        WHERE InstitutionID = 1) AS inst
ORDER BY ...
```

Итак, мы сократили количество полей (см Листинги 10 и 11), по которым происходит сортировка и теперь в одной из таблиц первичной выборки присутствует константная запись из таблицы, что немного ускоряет выполнение запроса.

5.2.3 Оптимизация LIMIT X

В некоторых случаях оптимизатор MySQL оптимизирует запрос, который имеет в своем составе LIMIT и не имеет при этом HAVING. Если в качестве лимита указано небольшое значение, MySQL вероятно предпочтет выполнить проход по индексам, в то время, как в обычном случае началось бы сканирование таблицы.

Если LIMIT используется совместно с ORDER BY, MySQL закончит сортировку, как только наберется достаточное для LIMIT'а количество строк. В случае с DISTINCT MySQL поступит аналогичным образом.

Однако, комбинация ORDER BY вместе с LIMIT 1 может быть оптимизирована. Так, запрос вида `SELECT col FROM table ORDER BY col LIMIT 1` может быть заменен на `SELECT MIN(col) FROM table`. В случае, если столбец проиндексирован, MySQL просто вернет минимальное значение столбца из индекса, в то время, как LIMIT+ORDER BY должны упорядоченно обойти индекс.

Для начала необходимо проверить возможность оптимизации. См. для этого секцию Тестирование.

Эта оптимизация дает незначительные преимущества в случае индексированных столбцов и небольшой выигрыш в случае отсутствия индексов. Оптимизация присутствует на Листингах 25 и 26.

Применение этой оптимизации к основному запросу:

Листинг 12: основной запрос до оптимизации LIMIT

```
SELECT Accepted FROM approvements a6
        JOIN submissions
        JOIN relgroupsmodules a4
        WHERE a6.SubmissionId = Submissions.SubmissionId
                AND (a4.ExpireTime IS NULL
                OR Submissions.SubmissionTime < a4.ExpireTime)
        ORDER BY ApprovementTime DESC
        LIMIT 1;
```

Листинг 13: основной запрос после оптимизации LIMIT

```
SELECT MAX(Accepted) FROM approvements a6
        JOIN submissions
        JOIN relgroupsmodules a4
        WHERE a6.SubmissionId = Submissions.SubmissionId
                AND (a4.ExpireTime IS NULL
                OR Submissions.SubmissionTime < a4.ExpireTime)
                AND ApprovementTime = (SELECT MAX(ApprovementTime)
                                       FROM approvements);
```

Доступна еще одна оптимизация в запросе, подобная изложенной на Листингах 12 и 13 . Опустим ее, так как их механики в запросе идентичны с точностью до знаков.

5.2.4 Оптимизация UNION и DISTINCT

Согласно best practices, преобразование UNION в UNION ALL в запросе дает большую выгоду. Первым предположением является то, что это достигается за счет того, что UNION ALL'у не нужна дополнительная таблица для хранения результата, однако это не совсем верно. Обе формы объединения используют временную таблицу для генерации результата.

Интересен тот факт, что создание этой временной таблицы можно посмотреть с помощью SHOW STATUS. В обычном EXPLAIN'е этой дейстмие по-умолчанию скрыто.

Отличием же в выполнении этих запросов является то, что обычный UNION создает промежуточную таблицу, накладывает на нее индекс и лишь после этого приступает к выборке, удаляя дубликаты. В то же время UNION ALL пропускает эти действия, за счет чего и достигается повышение производительности.

Проверим вышесказанное в рамках Тестирования.

Основным моментом здесь является то, что в обоих запросах присутствуют ключевые слова DISTINCT. А значит, по схеме выполнения MySQL сначала сделает выборку из одной таблицы, отфильтровав дубликаты, затем поступит по аналогии со второй таблицей. Затем в запросе с обычным UNION, после объединения выборок во временную таблицу, MySQL добавит индексы и вновь отфильтрует результаты, пройдя по таблице. Однако в таблице к тому моменту уже не будет дубликатов и этот проход будет лишним.

Избавившись от него с помощью использования UNION ALL, получили выигрыш в производительности.

Стоит также заметить, что как и в случае с GROUP BY / ORDER BY, MySQL может использовать лишь левые префиксы индексов для работы с ключами, и в целом DISTINCT иногда рассматривается оптимизатором, как частный случай GROUP BY.

Выполним описанные выше преобразования, так как аналогичные условия имеются в рассматриваемом нами запросе и запишем новые запросе в Листинги 14 и 15:

Листинг 14: основной запрос до оптимизации UNION

```
SELECT DISTINCT ... FROM ... AS a8
UNION
SELECT DISTINCT ... FROM ... AS a1
```

Листинг 15: основной запрос после оптимизации UNION

```
SELECT DISTINCT ... FROM ... AS a7
UNION ALL
SELECT DISTINCT ... FROM ... AS a1
```

5.2.5 Оптимизация LEFT и RIGHT JOIN

MySQL выполняет объединение таблиц, например `A LEFT JOIN B` подобным образом:

1. Таблица B устанавливается зависимой от A и от всех таблиц, от которых зависит A. Таблица A в свою очередь устанавливается зависимой от всех таблиц, кроме B, которые используются в условии LEFT JOIN

2. Используется условие LEFT JOIN для того, чтобы решить, как выбрать записи из таблицы B

3. Выполняются все стандартные оптимизации входящих внутрь запроса условий

4. В случае, если в таблице B запись, соответствующая условию ON, и для которой в таблице A имеется запись, отсутствует, то в таблицу B дописывается запись со всеми столбцами, равными NULL

5. Новая запись в таблице со всеми параметрами, равными NULL, добавляется в результирующей выборке в соответствие непустой записи из таблицы A

Реализация RIGHT JOIN аналогична с точностью до порядка таблиц. Оптимизацией JOIN запросов является перестановка таблиц в запросах. Однако LEFT JOIN и STRAIGHT JOIN практически не оптимизируются.

Еще одна форма объединения – STRAIGHT JOIN. Эта команда не дает оптимизатору выбора, кроме как принять порядок, заданный пользователем, вследствие чего никакие дополнительные преобразования не производятся и запрос за счет этого ускоряется. Как и в остальных случаях упор делается на наличие индексов. Достигается это улучшение только в том, случае, если известно, что «навязанный» оптимизатору порядок соединения однозначно лучше чем тот, который может выбрать он сам. В большинстве случаев «навязывание» подобных условий оптимизатору может привести к обратному результату.

Навязывание использования индексов совместно с «неперестaвляемым» LEFT JOIN'ом уже рассматривалось ранее. Чтобы избежать дублирования, опустим этот момент здесь.

5.2.6 Оптимизация IS (NOT) NULL

MySQL оптимизатор умеет удалять ненужны проверки на наличие /отсутствие NULL значений. Так, если условие WHERE содержит проверку столбца *a* IS NULL при том, что сам столбец объявлен как NOT NULL, проверка будет удалена.

Также, согласно best practices, не рекомендуется использовать значения типа NULL на столбцах типов DATE, TIME и DATETIME. Убрав поддержку NULL значений и заменив ее сравнением с «новым отсутствующим» значением, например стандартной NULL-датой 0000-00-00 00:00:00, обеспечим выполнение best practices в рамках таблицы. Оптимизация представлена на Листингах 16 и 17:

Листинг 16: основной запрос до оптимизации IS (NOT) NULL

```
(SELECT Accepted
        FROM Approvements as a6
        WHERE a6.SubmissionId = Submissions.SubmissionId
                AND (a4.ExpireTime IS NULL
                        OR Submissions.SubmissionTime < a4.ExpireTime)
```

Листинг 17: основной запрос после оптимизации IS (NOT) NULL

```
(SELECT Accepted
        FROM Approvements as a6
        WHERE a6.SubmissionId = Submissions.SubmissionId
                AND (a4.ExpireTime > '0000-00-00 00:00:00'
                        OR Submissions.SubmissionTime < a4.ExpireTime)
```

5.3 MySQL и индексы

Лучший способ улучшить производительность операции SELECT это создать индексы на одном или нескольких столбцах, используемых в запросе. Индексы выступают в качестве указателей на записи, позволяя быстро определить какие записи подходят под условие WHERE и выбрать остальные поля записи.

Все типы данных в MySQL могут быть проиндексированы. Все индексы в MySQL в рамках движка InnoDB, будь то PRIMARY, UNIQUE и INDEX, сохранены в B-деревьях. Индексные страницы при этом хранятся вместе с данными. MyISAM же использует для этих целей хеш-таблицы.

И, хотя может возникнуть желание создать индексы по всем возможным столбцам, неиспользуемые и ненужные ндексы занимают место и отнимают у оптимизатора MySQL время на поиск необходимого ему оптимального индекса.

Индексы также увеличивают «стоимость» операций вставки, удаления и обновления, так как каждый индекс должен быть обновлен.

MySQL поддерживает до 16 ключей на одной таблице.

Столбцы типов BLOB и TEXT поддерживают неполное индексирование. На столбцах типов CHAR и VARCHAR разрешено создавать частичные индексы, которые при этом могут составлять часть многостолбцового индекса.

Как уже оговаривалось ранее, только крайние левые префиксы индекса могут быть использованы большинством операций. Однако, иногда выборка может быть

произведена совсем без обращения к данным. Это происходит в том случае, если выбирается часть индекса по условию другой части индекса. Так, трехстолбцовый индекс на столбцах *(a, b, c, d)* дает поисковые преимущества в таких сочетаниях: *(a), (a, b), (a, b, c)*. При выборке же можно использовать столбец *c* в то время, как условие наложено на столбец *a* или *b*.

Ранее было отмечено, что данные большинства таблиц в исходной базе данных редактируются нечасто. Поэтому, на них можно без ограничений накладывать индексы. В базе присутствуют лишь 4 постоянно используемых таблицы, которые были отмечены ранее. На этих таблицах уже присутствуют индексы помимо PRIMARY и UNIQUE. Эти индексы, действительно, используются оптимизатором и «утяжелять» таблицы дополнительными индексами не следует.

Вместо этого добавим несколько индексов, возможное отсутствие некоторых из которых приводило к полному сканированию таблиц согласно первому результату EXPLAIN'a:

```
CREATE INDEX InstitutionIDInstitutionNameIndex
             ON Institutions(InstitutionID, InstitutionName);
CREATE INDEX GroupInstitutionTimeNameIndex
             ON Groups (GroupID, InstitutionID, TimeID, GroupName);
CREATE INDEX GroupIndex ON Approvers (GroupID);
CREATE UNIQUE INDEX RelGroupsModulesExpireTimeGroupIDModuleIDUniqueIndex
             ON RelGroupsModules(ExpireTime, GroupID, ModuleID);
```

5.4 Другие оптимизации

5.4.1 Выбор движка базы данных

Одной из особенностей MySQL является наличие большого числа движков, практически каждый из которых специализирован под конкретную задачу. Предполагается, что выбор движка происходит на этапе проектирования. Перечислим основные движки и озвучим их особенности:

- MyISAM – не поддерживает транзакции, но поддерживает полнотекстовый поиск. Внешние ключи недоступны. Данные и индексы хранятся отдельно. Сравнительно невысокая надежность хранения данных

- Memory (ранее, HEAP) – отличается несравнимо быстрой работой с небольшими таблицами, так как хранит все временные таблицы в оперативной памяти. Практически не имеет конкурентов по скорости работы

- Federated- - федерация серверов, обеспечивающая высокую масштабируемость и высокую отказоустойчивость

- CSV – хранит таблицы в CSV формате и позволяет редактировать их внешними приложениями. Отличается также невысокой стабильностью работы

- InnoDB – движок для таблиц «общего» назначения, тем не менее поддерживающий большие таблицы. Полная поддержка транзакций (ACID), внешних ключей. Максимальный объем - 64ТБ. По заявлениям разработчиков InnoDB

- самый быстрый основанный «на диске» движок. Однако, сильно зависит от надлежащей индексации данных

- Blackhole – движок, созданный для задач репликации.Не умеет самостоятельно хранить данные. Может выступать «мастером» в схеме репликации master-slave

- Example – экспериментальный движок для разработчиков. Таблицы, основанные на нем не могут хранить данные и нужны в первую очередь для создания новых типов таблиц.

Как было замечено ранее, в работе был выбран движок InnoDB по причине поддержки внешних ключей, транзакций и других функций «из коробки».

5.4.2 Оптимизация некоторых типов данных

Одним из способов измерения производительности запросов MySQL называет измерение количества дисковых операций. Для небольших таблиц обычно можно найти строку одним обращением. Для больших таблиц, использующих дерево индексов, количество дисковых операции можно оценить формулой

$$C = \frac{\ln row\_count}{\ln \frac{index\_block\_length \times 2}{3 \times (index\_length + data\_pointer\_length)}}$$

Индексный блок обычно составляет 1024 байта, указатель – 4 байта.
Можем оценить количество дисковых операций для таблицы *Submissions*:

$$C = \frac{\ln 130'000}{\ln \frac{1'024 \times 2}{3 \times (16+4)}} = \frac{11.775}{3.530} \approx 3$$

Итого, потребуется в среднем 3 дисковых операции для того, чтобы найти конкретную строку в таблице *Submissions*, вмещающей 130'000 записей при наличии на ней индекса длины 16.

Логарифмическая зависимость объема таблицы позволяет оценить незначительность количества дисковых операций для большинства таблиц базы. Взяв абстрактную таблицу, содержащую, например, 300 записей и имеющую индекс длины 4 на столбце типа INTEGER выясним, что записи из подобных таблиц выбираются за одно обращение. Это еще раз подтверждает возможность наложения дополнительных индексов на подобные таблицы:

$$C = \frac{\ln 300}{\ln \frac{1'024 \times 2}{3 \times (4+4)}} \approx 1$$

И хотя в наше время скорость работы важнее объема затраченной памяти, в MySQL рекомендуется содержать данные компактно. А значит, следуют и небольшие оптимизации некоторых таблиц базы.

TINYINT или MEDIUMINT предпочтительнее «обычного» типа INT, если это не противоречит логике работы;

NULL требует дополнительного места, а значит, ограничения в виде NOT NULL значений немного уменьшат место. Это преимущество в рассматриваемой базе незначительно ввиду небольшого общего объема данных;

Вынос логики работы с величинами времени и даты во вне. В базе при этом можно оставить значения типа TIMESTAMP или INT. Эта оптимизация значительнее предыдущих и может дать преимущество в несколько раз.

5.4.3 Оптимизация базы данных

В ходе рассмотрения плана выполнения запроса было выявлено большое количество подзапросов, в том числе с полным сканированием таблиц. Один из таких запросов, с *id=1* отмечен как *derived 2*, ссылающийся на подзапрос с *id=2*. Тот в свою очередь отмечен, как *derived 3*. Подзапрос с *id=3* представляет из себя 2 подзапроса – *a11* и *RelGroupsModules*. После этого выполняется выборка из таблицы *a9*. И лишь после этого происходит объединение этих подзапросов в результирующую выборку. Все это сочетается с неопределенностью ключей для всех этих выборок а также постоянным сканированием таблиц.

Это можно исправить путем создания дополнительной таблицы. Назовем ее WorkRole. В нее перенесем функциональность под разделению ролей пользователей на студентов и принимающую их сторону:

Листинг 18: Скрипт создания таблицы WorkRole

```
CREATE TABLE WorkRole (
    PersonID        INTEGER NOT NULL REFERENCES Persons
                            ON DELETE CASCADE ON UPDATE CASCADE,
    GroupID         INTEGER NOT NULL REFERENCES Groups
                            ON DELETE CASCADE ON UPDATE CASCADE,
    ModuleID        INTEGER NOT NULL REFERENCES Modules
                            ON DELETE CASCADE ON UPDATE CASCADE,

    IsStudent       TINYINT NOT NULL,
    IsApprover      TINYINT NOT NULL,

    PRIMARY KEY (PersonID, GroupID, ModuleID, IsStudent)
);
```

Наличие ключа, состоящего из 4 столбцов объясняется спецификой выборки данных в запросе. Таблица носит технический характер и создана с целью сбора данных для запроса. При этом необходимо наличие PRIMARY индекса в подобном порядке, чтобы впоследствие использовать его также как и в изначальной версии запроса.

Создадим дополнительные индексы для выборки непосредственно ролей (так как в PRIMARY индексе при отсутствии выборки по префиксу *PersonID, GroupID, ModuleID*, исполнитель не сможет оптимально работать со столбцами *IsStudent* и *IsApprover*:

```
CREATE INDEX WorkRoleIsStudentIndex ON WorkRole(PersonID, IsStudent);
CREATE INDEX WorkRoleIsApproverIndex ON WorkRole(PersonID, IsApprover);
```

Чтобы не повлиять на текущую функциональность базы, все изначально имею-

щиеся в ней таблице редактировать не будем. Для работы же с этой вновь созданной таблицей создадим необходимые триггеры, которые изменяют данные внутри *WorkRole* основываясь на изменении данных в «базовых» для нее таблицах *Students* и *Approvers*.

Перепишем ту часть запроса, которая некогда выполняла описанные выше действия, изменив порядок выполнения на работу с новой таблицей WorkRole, согласно Листингам 19 и 20:

Листинг 19: основной запрос до внедрения таблицы WorkRole

```
SELECT GroupId, ModuleId, max(IsStudent)  AS IsStudent,
                          max(IsApprover) AS IsApprover
        FROM (
                SELECT a11.GroupId, ModuleId, 1 AS IsStudent, 0 AS IsApprover
                        FROM Students as a11
                        JOIN RelGroupsModules USING(GroupId)
                        WHERE PersonId = 1
                UNION
                SELECT GroupId, ModuleId, 0 AS IsStudent, 1 AS IsApprover
                        FROM Approvers as a9
                        WHERE PersonId = 1
                ) as a10
        GROUP BY GroupId, ModuleId
```

Листинг 20: основной запрос после внедрения таблицы WorkRole

```
SELECT GroupId, ModuleId, max(IsStudent)  AS IsStudent,
                          max(IsApprover) AS IsApprover
        FROM WorkRole
        WHERE PersonID = 1
        GROUP BY GroupId, ModuleId
```

6 ТЕСТИРОВАНИЕ

Прежде всего, необходимо определить, являются ли описанные выше отпимизации эффективными. Для этого протестируем каждую из них на специально подобранном примере, затрагивающем оптимизируемый участок запроса.

Конфигурация системы, на которой проводилось тестирование, следующая: Intel ®Core ™i5-3230M @2.60 ГГц; NVIDIA ®GeForce ™GT740M 1ГБ; 8 ГБ ОЗУ.

При тестировании во все запросы дополнительно был добавлен параметр `SQL_NO_CACHE` для запрета кэширования запроса, либо чтения из кэша уже закэшированных данных.

6.1 Оптимизация WHERE

Приступим к тестированию озвученной ранее отпимизации WHERE. Для этого

выполним запрос, задействующий именно ее и сравним с «наивной» реализаций такого же запроса.

Листинг 21: тестовый запрос до оптимизации WHERE

```
SELECT SQL_NO_CACHE p.personID,  m.moduleID, s.taskID
        FROM submissions s
        JOIN persons p ON s.PersonID = p.PersonID
        JOIN modules m ON s.ModuleID = m.ModuleID
        WHERE p.PersonID BETWEEN 92 AND 109 AND m.ModuleName LIKE '%C%';
```

Листинг 22: тестовый запрос после оптимизации WHERE

```
SELECT SQL_NO_CACHE p.personID, m.moduleID, s.taskID
        FROM submissions s
        JOIN (SELECT personID FROM persons
                WHERE PersonID BETWEEN 92 AND 109) p
        ON p.PersonID = s.PersonID
        JOIN (SELECT moduleID FROM modules WHERE ModuleName LIKE '%C%') m
        ON m.ModuleID = s.ModuleID;
```

Таблица 1: Сравнение времени выполнения запросов

Запрос	Время выполнения, клиент, с	Время выполнения, сервер, с
До оптимизации	0.0471	0.0463
После оптимизации	0.0313	0.0317

Оптимизация имеет место быть. Она также может быть применена в основном запросе.

6.2 Оптимизация ORDER BY / GROUP BY

Тестирование оптимизации ORDER BY / GROUP BY:

Листинг 23: тестовый запрос до оптимизации ORDER BY

```
SELECT SQL_NO_CACHE * FROM persons
        HAVING FirstName LIKE '%a%'
        ORDER BY FirstName;
```

Листинг 24: тестовый запрос после оптимизации ORDER BY

```
CREATE INDEX PersonsFirstNameIndex ON Persons(FirstName);
SELECT SQL_NO_CACHE * FROM persons FORCE INDEX (PersonsFirstNameIndex)
        HAVING FirstName LIKE '%a%'
        ORDER BY FirstName;
```

Таблица 2: Сравнение времени выполнения запросов

Запрос	Время выполнения, клиент, с	Время выполнения, сервер, с
До оптимизации	0.0160	0.0159
После оптимизации	0.0	0.0018

Выполним оба запроса. Сравним результаты выполнения:

Скорость выполеенения существенно изменилась в лучшую сторону. Значительная часть работы InnoDB завязана на работе с индексами. Результат закономерен. Оптимизация может быть применена.

6.3 Оптимизация LIMIT X

Тестирование оптимизации LIMIT X:

Листинг 25: тестовый запрос до оптимизации LIMIT

```
SELECT SQL_NO_CACHE LoginTime FROM Sessions
        WHERE PersonID BETWEEN 92 AND 109
        ORDER BY LoginTime DESC
        LIMIT 1;
```

Листинг 26: тестовый запрос после оптимизации LIMIT

```
SELECT SQL_NO_CACHE MAX(LoginTime) FROM Sessions
        WHERE PersonID BETWEEN 92 AND 109;
```

Тестовые запросы выполнены. Далее – сравнение результатов выполнения:

Таблица 3: Сравнение времени выполнения запросов

Запрос	Время выполнения, клиент, с	Время выполнения, сервер, с
До оптимизации	0.0780	0.0913
После оптимизации	0.0470	0.0501

Оптимизация также имеет место быть. Однако, следует быть аккуртанее при работе с большей выборкой.

6.4 Оптимизация UNION и DISTINCT

Проверим оптимизацию, выполнив запрос один раз с объединением UNION, другой – с объединением UNION ALL, как показано на Листингах 27 и 28:

Листинг 27: тестовый запрос до оптимизации UNION

```
SELECT DISTINCT personID, groupID, moduleID, taskID, SubmissionTime
        FROM Submissions WHERE PassedTests IS NOT NULL
UNION
```

```
SELECT DISTINCT personID, groupID, moduleID, taskID, SubmissionTime
        FROM Submissions WHERE PassedTests IS NULL;
```

Листинг 28: тестовый запрос после оптимизации UNION

```
SELECT DISTINCT personID, groupID, moduleID, taskID
        FROM Submissions WHERE PassedTests IS NOT NULL
UNION ALL
SELECT DISTINCT personID, groupID, moduleID, taskID
        FROM Submissions WHERE PassedTests IS NOT NULL;
```

Выполним оба тестовых запроса. Сравним результаты выполнения:

Таблица 4: Сравнение времени выполнения запросов

Запрос	Время выполнения, клиент, с	Время выполнения, сервер, с
До оптимизации	0.1400	0.1337
После оптимизации	0.0780	0.0657

Прирост производительности составил почти два раза. Используем эту оптимизацию.

6.5 Сравнение производительности

На данный момент имеется оптимизированный по описанным выше пунктам запрос. Некоторые изменения в ходе работы коснулись и самой базы данных. Она также была оптимизирована.

Все проводимые оптимизации не затрагивали результат выборки основного запроса. Так что исходный запрос и получившийся в итоге, оптимизированный, можно считать идентичными по своей функциональности.

Взглянем на план выполнения нового запроса – см.Приложение 4.

Вот что изменилось по сравнению с предыдущим выполнением EXPLAIN'a:

- Остались 2 полнотекстовых сканирования, вместо 6 первоначальных
- Убрана неиспользуемая временная выборка, дублирующая подзапрос ($id = 4$)
- Остались лишь 7 подзапросов вместо 10 первоначальных
- Первая выборка имеет максимально быстрый (после ($type{=}system$) тип объединения, а длина используемого ключа сокращена
- Уменьшено общее количество участвующих в запросе строк

Согласно результатам EXPLAIN'a, запрос, действительно, стал работать оптимальнее. В качестве результата представим таблицу профилирования запросов инструментом MySQL Profiler, который не учитывает время подключения к базе данных, время ожидания открытия таблиц, завершения других процессов и т.д., а показывает идеализированное «чистое» время выполнения.

Сгруппированный по типам действия результат выполнения профилирования представлен на таблице 5. Как следует из результата, время подготовки к запуску запроса осталось прежним. Очистка временных файлов и закрытие таблиц занимают чуть большее время в новом запросе. Фактически, эти значения можно приравнять, списав их разности в $\approx 1 \times 10^{-6}$ секунды на погрешность. Небольшие улучшения заметны в финальной части запроса – его завршении и освобождении памяти. В них наблюдается прирост скорости выполнения примерно на один порядок выше.

Наилучший результат относительно результата исходного запроса показывает отправка данных. В ходе оптимизации удалось существенно уменьшить количество переносов данных по временным файлам, сократив при этом и объем этих данных. Вследствие чего разность результатов составляет ≈ 0.01388 секунды. Однако, время выполнения, согласно таблице, фактически определяется временем пересылки данных по ходу запроса и временем создания индекса сортировки. Поэтому, ускорение времени отправки на 38% даст общий прирост производительности схожего характера.

Таблица 5: Сгруппированный по действиям результат профилирования, с

Тип действия	Время вып. исх. запр, с	Время вып. нов. запр, с
Подготовка	0.000045	0.000045
Отправка данных	0.037241	0.023355
Создание индекса сортировки	0.014082	0.012395
Удаление временных таблиц	0.000556	0.000483
Закрытие таблиц	0.000028	0.000029
Очистка временных файлов	0.000017	0.000019
Завершение запроса	0.000136	0.000082
Освобождение памяти	0.000137	0.000129

Просуммируем результат предыдущей таблицы, чтобы получить итоговое значение времени выполнения запроса. Запишем это время в таблицу 6.

Общее время выполнения нового запроса сократилось на $\approx 37\%$. Подобный результат можно считать успешным, так как входная база данных была достаточно «оптимизирована» изначально и повторное применение оптимизации не дало бы результата. При сравнении результатов запросов в SQLite и MySQL ситуация аналогичная с точностью до затрат по подключению непосредственно к базам данных (из утилиты, основанной на ADO.Net). Это объясняется нивелированием преимуществ одной СУБД над другой за счет оптимизации общей их части – механики выполнения запросов, которая по большей части идентична. Результат закономерен.

Выполним «проверку статусов» SELECT запросом `SESSION STATUS LIKE 'Select\%'`.

Результаты можно наблюдать в 7. Общее количество сканирований сократилось. Аналогичный результат показывает и EXPLAIN. Выполним проверку создания вре-

Таблица 6: Итоговый результат профилирования, с

Тип действия	Время вып. исходного запроса, с	Время вып. нового запроса, с
Выполнение запроса	0.052253	0.036553

Таблица 7: Количество выполненных запросов, ед

Тип запроса	Исходный запрос	Новый запрос
Select_full_join	1	0
Select_range_check	0	0
Select_range	0	0
Select_scan	6	2

менных таблиц `SESSION STATUS LIKE 'Created_tmp\%'`:

Таблица 8: Количество выполненных созданий врем. объектов, ед

Тип произведенного действия	Исходный запрос	Новый запрос
Created_tmp_files	4	4
Created_tmp_tables	12	8

Из результата этой выборки, согласно таблице 8 также видно уменьшение количества временно созданных таблиц с 12 до 8.

6.6 Оценка оптимизации

Оптимизация, выполненная над базой не включала в себя масштабные изменения. Структура базы практически не затронута. Все изначально существовавшие таблицы сохранены, представления сохранены. Триггеры адаптированы и сохранены. Рефакторинг структуры базы не был произведен ввиду необходимости сохранить ее функционирующее состояние.

Изначально спроектированная база данных достаточно нормализована, не содержит избыточных данных, многозначных и многоцелевых столбцов. На данный момент в базе данных присутствует сравнительно небольшой объем данных. Основываясь на всем этом можно сделать вывод, что имеющаяся база данных в рефакторинге не нуждается, по крайней мере на данный момент.

Переход с SQLite на MySQL является спорным моментом, так как одним из плюсов СУБД SQLite является ее простота. Большинство преимуществ MySQL над SQLite в ходе данной работы задействованы не были и в целом являются преимуществами в специфических условиях.

7 ЗАКЛЮЧЕНИЕ

В ходе работы над курсовым проектом было изучено поведение MySQL при обработке SQL запросов. Протестирована механика многих оптимизаций, которые могут быть применены ко входной базе данных. Большинство этих оптимизаций было осуществлено. Были оценены улучшившиеся показатели работы, были произведены тесты производительности полученной системы.

7.1 Возможные дальнейшие улучшения

Данная работы не может быть названа полностью законченной, так как абсолютно любую систему можно так или иначе улучшить. Вопрос состоит лишь в том, насколько необходимо «улучшение», и не приведет ли оно к противоположному результату в другом месте. Так, оптимизация конкретной базы данных в рамках работы шла «с оглядкой» на выполнение конкретного моделируемого запроса.

При возникновении подобной необходимости в будущем, работа может быть продолжена и доведена до необходимого состояния с учетом будущих потребностей и возможностей.

Список литературы

[1] SQLite Complete Documentation [Электронный ресурс] — Режим доступа: https://www.sqlite.org/docs.html

[2] System Properties Comparison MySQL vs. SQLite [Электронный ресурс] — Режим доступа: http://db-engines.com/en/system/MySQL%3BSQLite

[3] MySQL 5.7 Reference Manual [Электронный ресурс] — Режим доступа: http://dev.mysql.com/doc/refman/5.7/en/

[4] Бэрон Шварц, Петр Зайцев, Вадим Ткаченко, Джереми Д. Зооднай, Дерек Дж. Баллинг, Арьен Ленц. MySQL. Оптимизация производительности, 2-е издание. Перевод с англ. А. Слинкина.— СПб: Символ-Плюс, 2010. — 206 с.

[5] A Comparison Of Relational Database Management Systems [Электронный ресурс] — Режим доступа: https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems