Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Оптимизирующий компилятор запросов подмножества SQL на основе LLVM»

Студент ИУ9-71Б (Группа) _____ (Подпись, дата) Старовойтов А.И. (И.О. Фамилия)

Руководитель _____ (Подпись, дата) Непейвода А.Н. (И.О. Фамилия)

Консультант _____ (Подпись, дата) _____ (И.О. Фамилия)

2026 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

С развитием технологий обработки больших данных и появлением все большего числа распределенных высоконагруженных систем, как никогда актуальным становится вопрос эффективного хранения данных. Системы управления базами данных решают эти задачи, упрощая разработку и экономя вычислительные ресурсы, а также место на дисках.

Реляционные системы управления базами данных занимают более половины рынка [1] и остаются стандартом для хранения данных. Основным языком запросов к таким системам является SQL.

Разработчики СУБД постоянно борются за производительность исполнения SQL запросов, применяя различные техники, например: машинное обучение, аппаратное ускорение, векторизированное исполнение и т.д. Одним из способов оптимизации, позволяющим сэкономить ресурсы процессора при выполнении сложных запросов, может быть JIT-компиляция некоторых частей запроса. Такая техника реализована в самых популярных СУБД, таких как PostgreSQL [2] и является довольно актуальной для автоматически сгенерированных запросов, например в системах визуализации данных в виде графиков и диаграмм, ввиду сложности выражений для фильтрации и их неоптимальности.

Целью данной работы является реализация модельной реляционной СУБД с поддержкой опциональной JIT-компиляции выражений в SELECT запросах на основе LLVM-JIT, а также сравнение производительности исполнения с компиляцией и без.

1 Обзор предметной области

Обычно, СУБД верхнеуровнево разделяют на следующие модули [3] (рисунок 1):

— модуль хранения данных;

— модуль обработки запросов;

— модуль управления транзакциями.

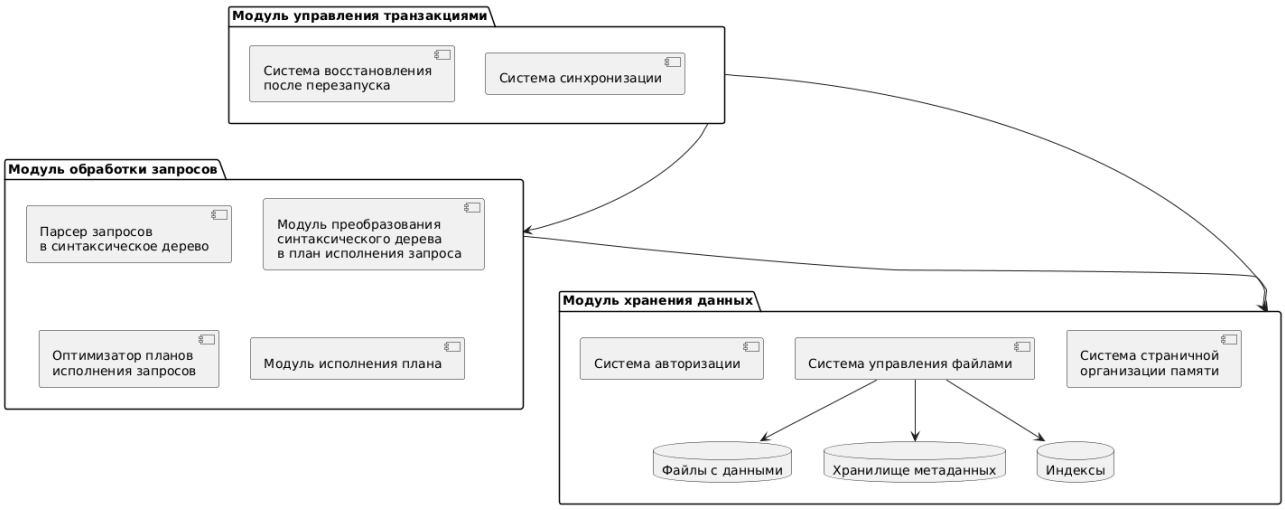

Рисунок 1 — Схема верхнеуровнего устройства СУБД.

1.1 Модуль хранения данных

Модуль хранения данных реализует удобный интерфейс для физического хранения данных на диске, предоставляя возможность сохранять, читать и обновлять информацию в базе данных.

Подсистемами модуля хранения данных являются:

— система авторизации;

— система управления файлами;

— система страничной организации памяти.

Данные на диске хранятся в нескольких формах: файлы с даннми, хранилище метаданных и индексы.

1.2 Модуль обработки запросов

Модуль обработки запросов отвечает за исполнение запросов пользователя. Его подсистемами являются:

— парсер запросов в синтаксическое дерево;
— модуль преобразования синтаксического дерева в план исполнения запроса;
— оптимизатор планов исполнения запросов;
— модуль исполнения плана.

1.3 Модуль управления транзакциями

Задача этого модуля — обеспечить атомарное выполнение нескольких запросов, чтобы сохранить определенные инварианты относительно хранимых данных. Его подсистемами могут являться:

— система восстановления после перезапуска;
— система синхронизации.

1.4 Реляционная модель и реляционная алгебра

Реляционная модель объединяет несколько концептов:

— структура: определения отношений и их содержимого не зависит от физического представления. Каждое отношение — множество атрибутов, каждый атрибут имеет множество значений;
— целостность: соблюдение определенных инвариантов относительно хранимых данных;
— интерфейс: декларативный интерфейс для доступа и изменения данных при помощи отношений. Разработчик задает только желаемый результат, а СУБД определяет наиболее оптимальный способ его достижения.

Все это образует фреймворк для работы с данными, который не отличается от одной СУБД к другой и избавляет от заботы о низкоуровневых деталях хранения.

Определение 1 (Отношение)**.** *Неупорядоченное множество, элементами которого являются кортежи из значений атрибутов, задающих сущности. Отношения иногда называют таблицами.*

Определение 2 (Кортеж)**.** *Множество значений атрибутов в отношении. Значениями могут быть скаляры или более сложные структуры данных. Каждый атрибут, если не задано специальное ограничение, может принимать значение* `NULL`*, означающее, что для данного кортежа значение данного атрибута не определено.*

Определение 3 (Реляционная алгебра)**.** *Множество фундаментальных оперторов для получения и изменения кортежей в отношении. Каждый оператор принимает одно или несколько отношений в качестве входа и возвращает новое отношение в качестве выхода.*

Чтобы представлить SQL запрос в терминах реляционной алгебры можно скомбинировать соответствующие операторы. Перечислим основные:

— $\sigma_{\text{predicate}}(R)$ — позволяет фильтровать кортежи по предикату;

— $\pi_{A_1,\ldots,A\_n}(R)$ — позволяет выбрать нужные атрибуты из кортежа;

— $\cap, \cup, -$ — стандартные операции над множествами;

— $\times$ — декартово произведение, соответствует `INNER JOIN`;

— $\rtimes, \ltimes, \rtimes\ltimes$ — реализуют `OUTER JOIN`.

1.5 Язык SQL

С помощью языка запросов SQL, “Structured Query Language”, происходит все взаимодействие клиентов с СУБД: определение схем данных, изменение содержимого таблиц и т.п. Этот язык появился в начале 1970-х и продолжает развиваться по сей день. Язык разделяют на две части:

— “Data-definition language” — язык для определения схем таблиц;

— “Data-manipulation language” — язык для получения и изменения данных в таблицах.

В этой работе мы фокусируемся на DML, а именно на части для получения данных, где наиболее актуальна JIT-компиляция. Этот функционал реализован с помощью ключевого слова “SELECT”.

Грамматика “SELECT” (листинг 1) позволяет задать:

Листинг 1: Грамматика “SELECT” [4].

```
SELECT [ DISTINCT | ALL ] select_list
    [ FROM table_expression ]
    [ WHERE condition ]
    [ GROUP BY grouping_element [, ...] ]
    [ HAVING condition ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ ORDER BY sort_expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT | SKIP
    ↪  LOCKED ] [, ...] ] ;
```

Листинг 2: Пример SQL запроса.

```
SELECT name, phone FROM users WHERE age > 22;
```

— “FROM” — из каких таблиц брать данные;

— “WHERE” — как их отфильтровать;

— “GROUP BY” — какие функции агрегации использовать;

— “ORDER BY” — порядок сортировки;

— “LIMIT” — ограничение на количество кортежей;

— “select_list” — какие атрибуты возвращать.

Пример “SELECT” запроса представлен в листинге 2.

1.6 Парсинг SQL запросов

Парсер отвечает за преобразование SQL-запросов в абстрактное синтаксическое дерево.

Помимо ANSI стандартов язка практически каждая СУБД реализует свой набор расширений. Самым популярным из них считается синтаксис “PostgreSQL”.

Синтаксис SQL довольно объемный, поэтому обычно используют генераторы лексических и синтаксических анализаторов, такие как Bison и ANTLR. Например, Postgres использует Bison [5].

1.7 Исполнение SQL запросов

После преобразования запроса в абстрактное синтаксическое дерево, СУБД транслирует его во внутреннее представление. Обычно, для этого используют представления на основе расширенной реляционной алгебры. Дополнительно, на этом этапе проверяется наличие таблиц и атрибутов, к которым обращается запрос.

Есть несколько вариантов, как исполнить один конкретный запрос: в каком порядке применять фильтрацию и сортировку, когда делать проекцию? За нахождение оптимального порядка отвечает оптимизатор. Он получает на вход запрос в форме реляционной алгебры, а на выходе строит физический план выполнения.

Физический план принципиально отличается от запроса в формате реляционной алгебры тем, что содержит все детали выполнения: конкретные алгоритмы исполнения каждого этапа, используемые индексы и т.п.

Физический план является деревом, в котором данные перемещаются от листьев к корню (рисунок 2). Листья являются таблицами, а остальные вершины — операторами. Стрелочки показывают направление движения данных. Операторы обычно являются одноместными или двуместными и соответствуют операторам расширенной реляционной алгебры.

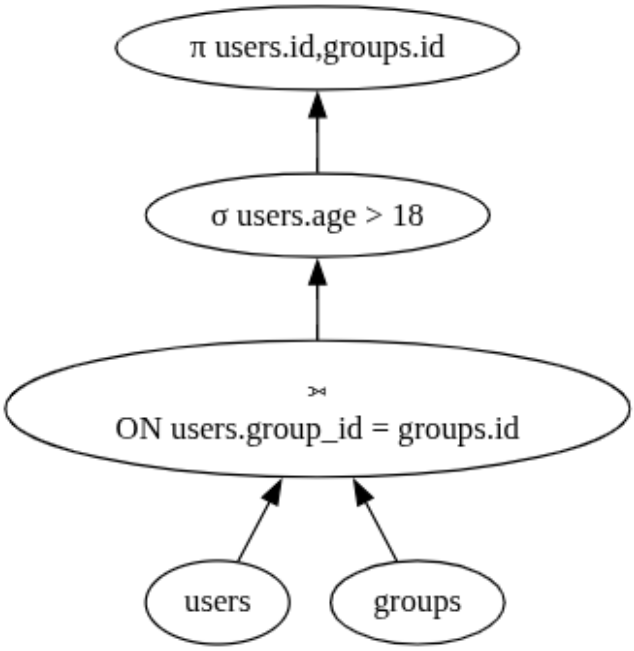

Рисунок 2 — План запроса

Существует три основные модели исполнения физических планов:

— модель итераторов;

— материализационная модель;

— векторизированная модель.

1.7.1 Модель итераторов

Самая распространенная модель исполнения — модель итераторов [6]. Каждый оператор является итератором — реализует функцию `Next()`, которая возвращает следующий кортеж из отношения. Простота данной идеи выгодно отличает ее от других вариантов и позволяет легко комбинировать операторы, и, что важнее, свободно рассуждать о них, потому что каждый оператор независим от другого.

В этой модели естественным образом получается конвейеризированное исполнение: когда кортеж проходит все возможные операторы до чтения следующего. Операторы, которые исполняются вместе, называют конвейером.

Некоторые операторы блокируют конвейверное исполнение. Например, для сортировки требуется иметь в доступе все кортежи.

Также, в модели итераторов легко реализуется ограничение количества кортежей. Это достигается путем прекращения вызовов `Next()`, когда нужное количество данных уже получено.

Минусом данной модели является большое количество накладных расходов на вызов функций. Применение каждого оператора к каждому кортежу является вызовом функции `Next()`.

1.7.2 Материализационная модель

В материализационной модели каждый оператор обрабатывает все данные за один вызов. Вместо `Next()` реализуют функцию `Output()`, которая в качестве результата возвращается весь набор кортежей.

Зачастую, отношения настолько большие, что их нельзя держать целиком в памяти, поэтому в результате каждой операции создается временная таблица на диске для хранения промежуточных данных.

Эта модель подходит для баз данных для обработки транзакций, где запросы затрагивают небольшое количество кортежей, но плохо работает в колоночных

СУБД, потому что там обрабатывается большое количество данных и постоянно приходится сохранять промежуточные результаты на диск.

1.7.3 Векторизированная модель

Векторизованная модель схожа с моделью итераторов в том, что в ней обычно реализуют функцию `Next()`. Отличием является то, что эта функция возвращает не один кортеж, а сразу массив из них.

Эта модель может показывать большую эффективность на современных процессорах, потому что позволяет обрабатывать массивы данных без зависимости от управляющих конструкций, таких как вызовы функций в итераторной модели.

1.7.4 Порядок обработки

Реализуют две модели:

— сверху-вниз;

— снизу-вверх.

В модели сверху-вниз операторы запрашивают данные у своих потомков, что предполагает вызовы функций. Здесь сложнее реализовать конкурентность в рамках одного запроса, потому что оператор должен блокироваться на вызове `Next()`.

В модели снизу-вверх данные передаются от потомков к родителям. Эта модель позволяет добиться прироста к производительности засчет более эффективного использование кэша процессора в конвейерном исполнении запроса. Минусами можно назвать более сложность ограничения количества обрабатываемых кортежей и сложность реализации некоторых операторов в целом.

1.7.5 Алгоритмы для реализации операторов реляционной алгебры

Для доступа к данным на диске реализуют следующие основные алгоритмы:

— последовательное сканирование;

— параллельное сканирование;

— сканирование с использованием индекса.

Листинг 3: Пример запроса с выражением.

```
SELECT * FROM users WHERE users.age > 33 + 1;
```

Последовательное сканирование итерируется по всем кортежам в одном потоке исполнения.

Параллельное сканирование использует несколько потоков, которые читают непересекающиеся блоки таблиц.

Сканирование с использованием индекса объединяет чтение таблицы и ее фильтрацию, и использует для это индекс. Выбор подходящего индекса зависит от содержимого таблицы и запроса, и может сильно ускорить чтение относительно последовательного сканирования.

Операторы проекции и фильтрации реализуются очевидным образом и не ограничивают конвейеризированное выполнение.

Выражения в SQL используются в качестве условий для фильтрации, проекции, определения операции соединения и т.д. Они представляются в виде дерева, где узлами являются операции и значения:

1. операции сравнения: `=`, `<`, `>`, `!=`;
2. логические операции: `AND`, `OR` и `NOT`;
3. арифметические операции: `+`, `-`, `*`, `/`, `%`;
4. константы;
5. атрибуты отношений.

Например, для выражения в запросе на листинге 3, дерево выглядит как на рисунке 3.

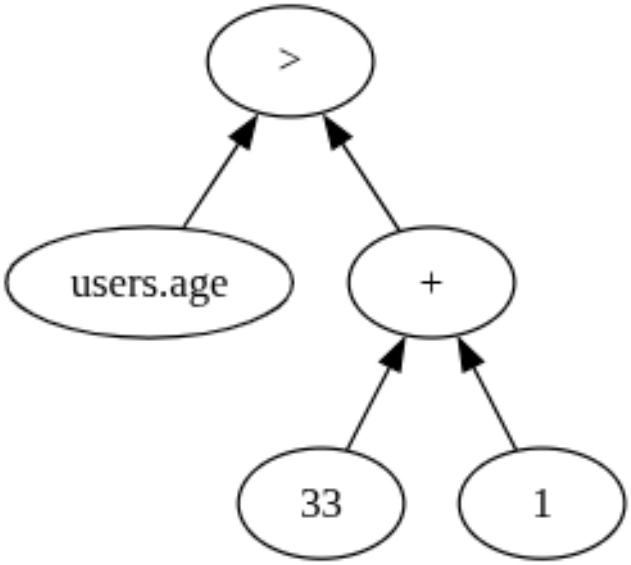

Рисунок 3 — Дерево для вычисления выражения из примера на листинге 3.

Чтобы вычислить значение такого выражения, СУБД хранит информацию о текущем кортеже и схемах используемых таблиц. Затем производится обход дерева с выполнением операторов снизу-вверх. Результатом вычислений считается результат вычисления оператора в корневой вершине.

Вычисление относительно сложных выражений происходит медленно, потому что каждый оператор требует вызова функции, поэтому появляется множество вохможностей для оптимазиции. Например:

— JIT-компиляция;
— векторизация;
— сворачивание выражений с константами;
— дедупликация.

1.8 JIT-компиляция

JIT-компиляция означает динамическую генерацию машинного кода под текущую аппаратную платформу во время исполнения запроса. Это позволяет значительно упростить само выражение, применяя различные оптимизации, от сворачивания констант, до исключения повторяющихся или невозможных частей. Особенная эффективность данного подхода наблюдается в аналитических запросах, которые работают с большими объемами данных и зачастую вычисляют запросы со сложными условиями фильтрации, агрегации и большим количеством произ-

водных данных. Также в числе возможных оптимизаций вычисление сдвигов для доступа к атрибутам внутри кортежей.

Эту технику реализует большинство современных баз данных. Например, Postgres и Clickhouse.

Основным преимуществом является возможность использования для оптимизаций данных, доступных во время выполнения запроса, таких как схемы таблиц и типы атрибутов. Недостатком можно считать большие затраты по времени на компиляцию, которые делают JIT-компиляцию нецелесообразной для несложных или затрагивающих небольшое количество кортежей запросов.

2 Разработка модельной СУБД

В рамках данной работы была разработана модельная реляционная СУБД, поддерживающая опциональную JIT-компиляцию выражений.

2.1 Парсер

В качестве синтаксиса SQL было выбрано подмножество синтаксиса PostgreSQL, как наиболее популярного на данный момент.

Из грамматики оставлены только `SELECT` запросы, потому что в них зачастую встречаются выражения и они представляют наибольший интерес для оптимизаций. Также, именно запросы на чтение обычно автогенерируются системами визуализации данных.

На вход парсер получает запрос в виде строки. На выходе в случае успешного разбора получается синтаксическое дерево, а в случае ошибки или неподдерживаемого запроса об этом сообщается пользователю.

Так как полная грамматика SQL очень большая, было решено воспользоваться одним из генераторов синтаксических и лексических анализаторов и взять для него готовое определение грамматики PostgreSQL.

2.2 Преобразование запроса в релиционную алгебру

После этапа лексического и синтаксического анализа, полученное дерево необходимо преобразовать в представление в формате реляционной алгебры. Это достигается засчет обхода дерева в глубину и возврата соответствущих операторов реляционной алгебры вместо поддеревьев синтаксического дерева.

В результате получается дерево, состоящее из операторов реляционной алгебры и отношений в качестве листьев. Для отладки было решено реализовать его визуализацию.

Дальнейших преобразований запроса в формате реляционной алгебры предусмотрено не было, так как это работа оптимизатора, что выходит за рамки работы и является отдельной сложной темой.

2.3 Исполнение запросов

Далее, запрос необходимо исполнить. Для того, чтобы не увеличивать количество внутренних представлений, запрос принято решение интерпретировать запрос напрямую в форме реляционной алгебры.

Модель исполнения была выбрана векторизированная. Это сделано для того, чтобы не органичивать возможности параллельного исполнения в рамках одного запроса и использовать потенциал современных процессоров по-максимуму.

Порядок обработки был выбран снизу-вверх, а коммуникация между операторами реализована с помощью потокобезопасных каналов, как в языке программирования Go. Это упрощает рассуждения и уменьшает возможную площать для возникновения ошибок в реализации.

Таким образом, каждый оператор создает по отдельному каналу для кортежей и типов атрибутов своих потомков. Затем вызывает их в цикле читает из канала кортежи (рисунок 4). Такая схема легко реализуется и предполагает одинаковый интерфейс у всех операторов, что позволяет с легкостью их комбинировать.

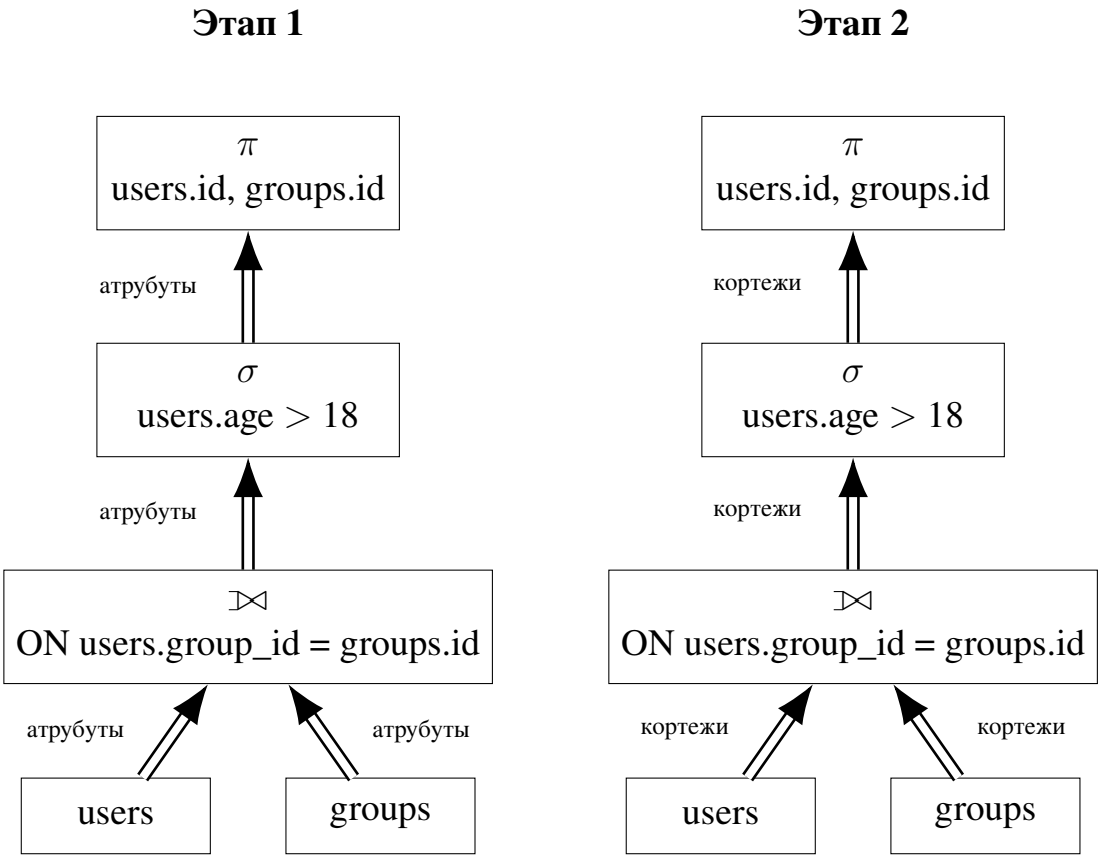

Рисунок 4 — Двухэтапное исполнение запроса.

2.4 Модуль хранения данных

В целях простоты, а также ввиду реализации только запросов чтения, было решено хранить данные в формате CSV (Comma-separated values) таблиц. Данный формат легок в поддержке, и иделально подходит для реализации модельной СУБД в условиях неизменяемости хранимых таблиц.

Каждая таблица хранится в отдельном CSV файле, который начинается с заголовка, определяющего названия и типы атрибутов. В следующих строках содержатся кортежи. Для простоты решено поддержать только два типа атрибутов: целочисленный и логический. Пример отношения `users`, состоящего из двух атрибутов: `id` и `age`, можно увидеть на листинге 4.

Листинг 4: Пример CSV файла, где хранится отношение users.

```
id:int,age:int
1,33
2,64
3,22
4,NULL
5,15
```

Для чтения данных из CSV файла применяется алгоритм последовательного сканирования, ввиду удобства реализации и очевидной оценки сложности.

2.5 Алгоритмы для операторов реляционной алгебры

Первый и простейший оператор, проекция, реализуется с помощью цикла, отражающего каждый кортеж в новый с нужными атрибутами (листинг 5). Это работает, т.к. $A_1, \ldots, A_n$ — выражения, которые могут включать атрибуты, константы и операции над ними, что позволяет добавлять значения, производные от существующих.

Листинг 5: Алгоритм выполнения проекции.

```
1: procedure Projection(π_{A_1,...,A_n}(R), out_attr_chan, out_tuples_chan)
2:     (attrs, tuples) ← Compute(R)
3:     out_attr_chan ← map(GetExpressionType, [A_1, ..., A_n])
4:     for tuple ∈ tuples do
5:         for expr ∈ [A_1, ..., A_n] do
6:             out_tuples_chan ← CalcExpression(expr, tuple)
```

Следующий оператор, фильтрация, реализуется аналогично (листинг 6). Включать ли в результат текущий кортеж определяет значение выражения над ним.

Листинг 6: Алгоритм выполнения фильтрации.

```
1: procedure Filter(σ_predicate(R), out_attr_chan, out_tuples_chan)
2:     (attrs, tuples) ← Compute(R)
3:     out_attr_chan ← attrs
4:     for tuple ∈ tuples do
5:         if CalcExpression(predicate, tuple) = true then
6:             out_tuples_chan ← tuple
```

Оператор соединения реализуется с помощью материализации одного из отношений на диск и последующего многократного чтения и объединения с кортежами из второго отношения (листинг 7).

Листинг 7: Алгоритм выполнения декартова произведения.

```
1: procedure CrossJoin(R_1 × R_2, out_attr_chan, out_tuples_chan)
2:     out_attr_chan ← Concat(attrs_1, attrs_2)
3:     materialized_1 ← Materialize(Compute(R_1))
4:     (attrs_2, tuples_2) ← Compute(R_2)
5:     for tuple_2 ∈ tuples_2 do
6:         for tuple_1 ∈ materialized_1 do
7:             joined ← ConcatTuples(tuple_1, tuple_2)
8:             out_tuples_chan ← joined
```

Алгоритмы для `OUTER` соединений аналоичны реализации `INNER JOIN`, но добавляют фильтрацию после генерации соединенных кортежей.

Наибольший интерес представляет вычисление выражений, которое, как можно заметить, используется в реализации операторов реляционной алгебры повсеместно.

В случае вычисления выражения путем прямой интерпретации, необходимо рекурсией обойти дерево, которым это выражение представлено, и вычислить каждую операцию. Каждая вершина такого дерева требует вызов функции, что для больших и сложных выражений, умножить на количество кортежей в отношении, выливается в большие накладные расходы на рекурсию (листинг 8).

Листинг 8: Алгоритм интерпретации выражения.

```
1:  function CalcExpression(expr, tuple, attrs)
2:      if expr — бинарная операция (lhs, op, rhs) then
3:          val_1 ← CalcExpression(lhs, tuple, attrs)
4:          val_2 ← CalcExpression(rhs, tuple, attrs)
5:          return ApplyOperator(op, val_1, val_2)
6:      else if expr — унарная операция (op, child) then
7:          val ← CalcExpression(child, tuple, attrs)
8:          return ApplyUnaryOperator(op, val)
9:      else if expr — атрибут A then
10:         return Lookup(A, tuple, attrs)
11:     else if expr — константа c then
12:         return c
13:     else if expr — литерал l then
14:         return l
```

2.6 JIT-компиляция

Проблема высоких накладных расходов при рекурсивной интерпретации выражений решается с помощью JIT-компиляции. Во время исполнения запроса, каждое выражение перед исполнением компилируется в машинный код, что позволяет напрямую выполнить набор инструкций для вычисления каждого выражения.

Скомпилированный код кэшируется в словарь, а затем переиспользуется для вычисления каждого выражения.

В качестве фреймворка для JIT-компиляции был выбран LLVM, а в частности современный интерфейс `ORC v2`, ввиду наличия широкого набора оптимизационных проходов, хорошей документации и понятного API для JIT.

Скомпилированный код представляет собой функцию, которая принимает на вход указатель на результат, указатель на кортеж и указатель на информацию о типах и названиях атрибутов. Результат вычисления выражения записывается по соответствующему указателю.

Выражение вычисляется как комбинация операций эквивалентных исходному дереву. Ветвления не используются в целях оптимизации.

Выбраны следующие оптимизационные проходы, актуальные для данной задачи:

— `EarlyCSEPass` — раннее устранение общих подвыражений;

— `SROAPass` — скалярная замена агрегатов;

— `InstCombinePass` — комбинирование инструкций, включая сворачивание констант;

— `SimplifyCFGPass` — упрощение графа потока управления;

— `ReassociatePass` — переассоциация выражений;

— `GVNPass` — глобальная нумерация значений;

— `MemCpyOptPass` — оптимизация копирований памяти;

— `SimplifyCFGPass` — повторное упрощение графа потока управления;

— `InstCombinePass` — повторное комбинирование инструкций;

— `ADCEPass` — агрессивное удаление мёртвого кода.

`EarlyCSEPass` — легковесный проход, который исключает дублирующие вычисления, путем обхода дерева доминаторов.

`SROAPass` — заменяет выделения памяти под структуры на скаляры, что упрощает IR.

`InstCombinePass` — кобинирует инструкции, делая алгебраические преобразования.

`SimplifyCFGPass` — убирает мертвый код.

`ReassociatePass` — меняет местами операнды и делает некоторые алгебраические преобразования, чтобы позволить другим проходам сделать дополнительные оптимизации.

`GVNPass` — убирает лишние, дублирующиеся, вычисления, где это возможно.

`MemCpyOptPass` — оптимизирует некоторые копирования и инициализации.

`ADCEPass` — удаляет мертвый код после всех других проходов.

Вместе все эти проходы делают все возможные оптимизации для ускорения вычисления выражений.

3 Реализация модельной СУБД

Для выполнения поставленной задачи необходимо реализовать парсер, преобразование в реляционную алгебру, выполнение соответствующих операторов и JIT-компиляцию выражений, а также модульные тесты и бенчмарки производительности.

В качестве языка программирования использовался C++, ввиду следующих его особенностей:

— Высокая производительность. Современные СУБД обрабатывают огромные потоки данных от миллионов клиентов, поэтому важны минимальное потребление ресурсов и задержки.

— Возможность работать на низком уровне. Ручное управление памятью и прямая работа с системными вызовами открывает возможности для оптимизаций и позволяет сделать время обработки запросов предсказуемым.

— Поддержка ООП и RAII. Позволяет разрабатывать системный код читаемым и поддерживаемым.

— Широкий выбор библиотек для производительного сетевого взаимодействия и бинарной сериализации данных.

Все эти причины позволили C++ стать де-факто стандартным языком для реализации СУБД.

Разработка проекта осуществлялась в редакторе кода Emacs. Данный редактор может быть использван в качестве IDE благодаря обширной библиотеке плагинов.

Для сборки использовалось окружение NixOS. Этот дистрибутив Linux построен на пакетном менеджере `nix` и позволяет декларативно и воспроизводимо задавать зависимости для каждого проекта, используя функциональный доменно-специфичный язык.

В качестве системы сборки использован CMake, а компилятором выступает clang с поддержкой стандарта C++23.

Для реализации конкурентной обработки запросов, были использованы C++20 корутины в связке с boost::asio, так как это становится стандартным спо-

собом реализации асинхронной модели в C++, и позволяет писать код, который выглядит линейно, но исполняется конкурентно.

3.1 Особенности реализации парсера

Для реализации парсера был выбран ANTLR4, ввиду его удобства и возможности генерировать лексические и синтаксические анализаторы под практически любой язык программирования.

Грамматика PostgreSQL была взята из официального репозитория проекта ANTLR4 [7]. Она была портирована автоматически из грамматики для Bison [5] из официального репозитория Postgres, поэтому является наиболее полной из существующих.

ANTLR4 генерирует несколько файлов на C++, в том числе лексический и синтаксический анализаторы, а также класс `Visitor` для обхода абстрактного синтаксического дерева.

Основная функция определяющая интерфейс парсера представлена в листинге 9. Она принимает поток, откуда читает запрос, а возвращает `Operator` (листинг 10), который является деревом из операторов реляционной алгебры.

Листинг 9: Интерфейс парсера.

```
Result<Operator> GetAST(std::istream& in);
```

Листинг 10: Типы Operator, Table и Projection.

```
using Operator = std::variant<Table, Projection, Filter, CrossJoin,
↪  Join>;

struct Table {
    std::string name;
    auto operator<=>(const Table& other) const = default;
};

struct Projection {
    std::vector<Expression> expressions;
    std::shared_ptr<Operator> source;
    bool operator==(const Projection& other) const;
};
```

Функция GetAST внутри строит абстрактное синтаксическое дерево и обходит его с помощью Visitor, который и генерирует представление запроса в виде дерева из Operator.

Дополнительно реализована визуализация дерева из операторов реляционной алгебры в формате Graphviz с помощью функции GetDotRepresentation.

3.2 Особенности реализации хранения данных

Данные хранятся в одной директории, где каждому отношению соответствует файл с CSV таблицей. Название файла совпадает с названием отношения.

Для чтения таблиц реализован класс CsvDirSequentialScanner (листинг 11).

Листинг 11: Объявление класса CsvDirSequentialScanner.

```
struct CsvDirSequentialScanner {
    std::string dir;

    boost::asio::awaitable<Result<>> operator()(const std::string&
    ↪  table_name,
                                                AttributesInfoChannel&
                                                ↪  attrs_chan,
                                                TuplesChannel&
                                                ↪  tuples_chan) const;
};
```

3.3 Особенности реализации исполнения запроса

За исполнение запросов отвечает класс Executor (листинг 12). Он реализует исполнение всего дерева операторов, и каждого оператора отдельно, с помощью функций Execute.

Листинг 12: Объявление класса Executor.

```
template <typename ExpressionExecutor = InterpretedExpressionExecutor>
class Executor {
public:
  using SequentialScan = std::function<boost::asio::awaitable<Result<>>(
      const std::string& table_name, AttributesInfoChannel& attr_chan,
      ↪  TuplesChannel& tuples_chan)>;
  Executor(SequentialScan seq_scan, boost::asio::any_io_executor
  ↪  executor);
  boost::asio::awaitable<Result<Relation>> Execute(const Operator& op);
private:
  boost::asio::awaitable<void> Execute(const Operator& op,
  ↪  AttributesInfoChannel& attr_chan,
                                       TuplesChannel& tuples_chan);
  boost::asio::awaitable<void> ExecuteProjection(const Projection& proj,
  ↪  AttributesInfoChannel& attr_chan,
                                                 TuplesChannel&
                                                 ↪  tuples_chan);
...
  boost::asio::awaitable<void> SpawnExecutor(const Operator& op,
  ↪  AttributesInfoChannel& attr_chan, TuplesChannel& tuple_chan);
private:
  SequentialScan sequential_scan_;
  ExpressionExecutor expression_executor_;
};
```

Этот класс параметризуется интерфейсами `SequentialScan` и `ExpressionExecutor`.

Первый служит для того, чтобы выбирать различные реализации чтения таблиц с диска.

`ExpressionExecutor` является шаблонным параметром и нужен для выбора реализации исполнения выражений. Реализовано три таких класса:

— `InterpretedExpressionExecutor` — реализует обыкновенную интерпретацию выражений с помощью рекурсии;

— `JitCompiledExpressionExecutor` — реализует JIT-компиляцию выражения;

— `CachedJitCompiledExpressionExecutor` (листинг 13) — кэширует результаты JIT-компиляции для переиспользования.

Листинг 13: Объявление класса CachedJitCompiledExpressionExecutor.

```
using ExecExpression = std::function<Value(const Tuple& source, const
↪  AttributesInfo& source_attrs)>;
class CachedJitCompiledExpressionExecutor {
  public:
    explicit
    ↪  CachedJitCompiledExpressionExecutor(boost::asio::any_io_executor
    ↪  executor);
    boost::asio::awaitable<ExecExpression> GetExpressionExecutor(const
    ↪  Expression& expr, const AttributesInfo& attrs);
  private:
    JITCompiler compiler_;
    std::unordered_map<std::string, ExecExpression> cache_;
};
```

Значения внутри выражений передаются в виде объектов класса `Value` (листинг 14). Этот класс состоит из флага `is_null`, означающего определено ли данное значение и объекта `NonNullValue`, где `NonNullValue` — объединение `int64_t` и `bool`.

Листинг 14: Объявление класса Value.

```
union NonNullValue {
    int64_t int_value;
    bool bool_value;
};

struct Value {
    bool is_null;
    NonNullValue value;

    bool operator==(const Value& other) const;
};
```

Отношения представлены в виде списка атрибутов и списка кортежей. Атрибуты представлены в виде структуры с полями: имя таблицы, название атрибута и тип атрибута. Кортеж — вектор объектов класса `Value`.

Коммуникация между операторами осуществляется при помощи класса `boost::asio::concurrent_channel`. Он предоставляет интерфейс, схожий с каналами в языке Go. Его особенности:

— потокобезопасность;

— буферизированность;

— асинхронность при использовании `async_read` и `async_write`;

— совместимость с корутинами.

В модельной СУБД используется два типа каналов: `TuplesChannel` для передачи кортежей и `AttributesChannel` для передачи атрибутов отношений (листинг 15).

Листинг 15: Объявление алиасов для каналов.

```
using TuplesChannel = boost::asio::experimental::concurrent_channel<
void(boost::system::error_code, Tuples)>;
using AttributesInfoChannel =
↪  boost::asio::experimental::concurrent_channel<
void(boost::system::error_code, AttributesInfo)>;
```

3.4 Особенности реализации JIT-компиляции

JIT-компиляция происходит в классе `JITCompiler` (листинг 16). Метод `CompileExpression` принимает поддерево в реляционной алгебре и список доступных атрибутов, а возвращает `CompiledExpression` — указатель на скомпилированную функцию, которую можно по нему вызвать.

Листинг 16: Объявление класса JITCompiler.

```
class JITCompiler {
    public:
        using CompiledExpression = void (*)(Value*, const Value*, const
        ↪  AttributeInfo*);
        explicit JITCompiler(boost::asio::any_io_executor executor);
        boost::asio::awaitable<std::pair<CompiledExpression,
        ↪  llvm::orc::ResourceTrackerSP>>
        CompileExpression(const Expression& expr, const AttributesInfo&
        ↪  attrs);
    private:
      llvm::Function* GenerateIR(llvm::Module& llvm_module, const
      ↪  Expression& expr,
                                 const AttributesInfo& attrs);
    private:
        std::unique_ptr<llvm::orc::LLJIT> jit_;
        std::atomic<uint64_t> id_;
        boost::asio::strand<boost::asio::any_io_executor> jit_strand_;
};
```

В конструкторе (листинг 21) производится настройка оптимизирующих проходов и остальной инфраструктуры llvm.

На примере класса `GenerateIRVisitor` (листинг 22) можно увидеть, как производится генерация LLVM IR. Здесь нет ветвлений и вызовов функций, что позволяет генерировать оптимальный код, и делать это быстро. Результат генерации после оптимизации для выражения на листинге 17 можно увидеть на листинге 18.

Листинг 17: Пример выражения.

```
users.age > 30 AND users.age < 60 OR users.age = 10;
```

Листинг 18: LLVM IR для примера на листинге 17.

```
; ModuleID = 'expr_module_0'
source_filename = "expr_module_0"
target datalayout = "e-m:e-p270:32:32-p271:32\
:32-p272:64:64-i64:64-i128:128-f80:128-n8:16:32:64-S128"

%Value = type { i8, i64 }

define void @eval_expr_1(ptr noalias %0, ptr %1, ptr %2) {
entry:
  %struct_ptr = getelementptr inbounds nuw i8, ptr %1, i64 16
  %loaded_struct = load %Value, ptr %struct_ptr, align 8
  %value.is_null = extractvalue %Value %loaded_struct, 0
  %is_null = icmp ne i8 %value.is_null, 0
  %value.value = extractvalue %Value %loaded_struct, 1
  %is_null_i8 = zext i1 %is_null to i8
  %result.with_is_null = insertvalue %Value undef, i8 %is_null_i8, 0
  %3 = add i64 %value.value, -31
  %4 = icmp ult i64 %3, 29
  %5 = icmp eq i64 %value.value, 10
  %6 = or i1 %5, %4
  %i1_to_i64_zext38 = zext i1 %6 to i64
  %result41 = insertvalue %Value %result.with_is_null, i64
   ↪  %i1_to_i64_zext38, 1
  store %Value %result41, ptr %0, align 8
  ret void
}
```

4 Тестирование

Тестирование разработанной модельной СУБД осуществлялось с помощью модульных тестов. Также для замеров производительности были реализованы бенчмарки.

4.1 Модульные тесты

Модульные тесты реализованы с использованием библиотеки googletests и проверяют корректность реализации основных классов в парсере и исполнителе выражений.

Пример такого теста приведен в листинге 19.

Листинг 19: Пример модульного теста.

```
TEST(ParserTest, SelectSingleColumnFromSingleTable) {
  std::stringstream s{"SELECT users.id FROM users;"};

  Operator got = GetAST(s).value();

  ASSERT_THAT(got, VariantWith<Projection>(
      Projection{std::vector<Expression>{
      {Attribute{"users", "id"}}},
      std::make_shared<Operator>(Table{"users"})}));
}
```

4.2 Бенчмарки

Тесты производительности реализованы с помощью библиотеки google benchmarks. Они служат для измерения производительности SELECT запросов с JIT-компиляцией и без в различных профилях нагрузки и с различными настройками. Пример такого бенчмарка можно увидеть в листинге 20.

Листинг 20: Пример бенчмарка.

```
template <typename ExprExecutor, const char* Query>
void BM_SQL(benchmark::State& state) {
  std::ofstream nullstream("/dev/null");
  std::clog.rdbuf(nullstream.rdbuf());
  boost::asio::io_context ctx;
  boost::asio::co_spawn(
      ctx,
      [&state]() -> boost::asio::awaitable<void> {
        std::stringstream s{Query};
        Operator op = GetAST(s).value();
        CsvDirSequentialScanner seq_scan{kProjectDir +
        ↪  "/test/static/executor/test_data"};
        Executor<ExprExecutor> executor(
          std::move(seq_scan),
          co_await boost::asio::this_coro::executor
        );
        benchmark::DoNotOptimize(co_await executor.Execute(op));
        for (auto _ : state) {
          benchmark::DoNotOptimize(co_await executor.Execute(op));
        }
      }(),
      [](std::exception_ptr p) {});
  ctx.run();
}
```

При дебаге проблем с производительностью также было удобно пользоваться утилитой `perf`, с помощью которой были построены и проанализированы “флеймграфы”.

Далее, с помощью `Python` и библиотеки `pandas` были проанализированы результаты выполнения бенчмарков (рисунок 5). На рисунке по оси абсцисс изображен условный размер используемых отображений в количестве кортежей. Как можно увидеть на рисунке 6, на среднем количестве кортежей, где не так сильно влияет время на чтение и запись на диск, получается около 40% прироста к производительности.

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы была разработана модельная реляционная СУБД с поддержкой опциональной JIT-компиляции выражений, реализация проверена с помощью модульных тестов, а также проведены измерения производительности.

Цель курсовой работы была достигнута. Получена стабильная система, демонстрирующая возможности JIT-компиляции для оптимизации запросов на чтение.

В заключение, несмотря на достигнутые положительные результаты, существует потенциал для дальнейшего улучшения реализации модельной СУБД. Это включает в себя поддержку большего объема синтаксиса языка SQL, а также реализация более продвинутых вариаций алгоритмов для основных операций реляционной алгебры.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Исследование рынка СУБД. — URL: https://www.mordorintelligence.com/industry-reports/database-market ; (дата обращения: 17.01.2026).
2. Machine code caching in postgresql query jit-compiler / M. Pantilimonov [и др.] // 2019 Ivannikov Memorial Workshop (IVMEM). — IEEE. 2019. — С. 18—25. — (дата обращения: 17.01.2026).
3. Database system concepts / A. Silberschatz, H. F. Korth, S. Sudarshan [и др.]. — Mcgraw-hill New York, 2020. — (дата обращения: 17.01.2026).
4. Документация PostgreSQL. — URL: https://www.postgresql.org/docs/current/sql-select.html ; (дата обращения: 17.01.2026).
5. Грамматика PostgreSQL. — URL: https://github.com/postgres/postgres/blob/master/src/backend/parser/gram.y ; (дата обращения: 17.01.2026).
6. Graefe G. Volcano - an extensible and parallel query evaluation system // IEEE Transactions on Knowledge and Data Engineering. — 2002. — Т. 6, № 1. — С. 120—135. — (дата обращения: 17.01.2026).
7. Грамматика PostgreSQL для ANTLR4. — URL: https://github.com/antlr/grammars-v4/blob/master/sql/postgresql/PostgreSQLParser.g4 ; (дата обращения: 17.01.2026).

ПРИЛОЖЕНИЕ А

Листинг 21: Реализация конструктора класса JITCompiler.

```
JITCompiler::JITCompiler(boost::asio::any_io_executor executor) :
↪  jit_strand_(executor) {
  llvm::InitializeNativeTarget();
  llvm::InitializeNativeTargetAsmPrinter();
  llvm::InitializeNativeTargetAsmParser();
  auto jit_or_error = llvm::orc::LLJITBuilder().create();
  if (!jit_or_error) {
    throw std::runtime_error("failed to create llvm::LLJIT");
  }
  jit_ = std::move(*jit_or_error);
  jit_->getIRTransformLayer().setTransform(
      [](llvm::orc::ThreadSafeModule tsm,
      ↪  llvm::orc::MaterializationResponsibility& r)
          -> llvm::Expected<llvm::orc::ThreadSafeModule> {
        tsm.withModuleDo([](llvm::Module& m) {
          llvm::LoopAnalysisManager lam;
          llvm::FunctionAnalysisManager fam;
          llvm::CGSCCAnalysisManager cgam;
          llvm::ModuleAnalysisManager mam;
          llvm::PassBuilder pb;
          pb.registerModuleAnalyses(mam);
          pb.registerCGSCCAnalyses(cgam);
          pb.registerFunctionAnalyses(fam);
          pb.registerLoopAnalyses(lam);
          pb.crossRegisterProxies(lam, fam, cgam, mam);
          llvm::ModulePassManager mpm;
          llvm::FunctionPassManager fpm;
          fpm.addPass(llvm::EarlyCSEPass(true));
          fpm.addPass(llvm::SROAPass(llvm::SROAOptions::ModifyCFG));
          fpm.addPass(llvm::InstCombinePass());
          fpm.addPass(llvm::SimplifyCFGPass());
          fpm.addPass(llvm::ReassociatePass());
          fpm.addPass(llvm::GVNPass());
          fpm.addPass(llvm::MemCpyOptPass());
          fpm.addPass(llvm::SimplifyCFGPass());
          fpm.addPass(llvm::InstCombinePass());
          fpm.addPass(llvm::DCEPass());
          fpm.addPass(llvm::ADCEPass());
          mpm.addPass(llvm::createModuleToFunctionPassAdaptor(
                          std::move(fpm)));
          mpm.run(m, mam);
        });
...
```

Листинг 22: Класс GenerateIRVisitor

```
  struct GenerateIRVisitor {
      llvm::Value* operator()(const BinaryExpression& expr) {
          auto* lhs = std::visit(*this, *expr.lhs);
          auto* rhs = std::visit(*this, *expr.rhs);

          auto* is_null_lhs = CheckNull(lhs);
          auto* is_null_rhs = CheckNull(rhs);
          auto* value_lhs = LoadValue(lhs);
          auto* value_rhs = LoadValue(rhs);

          auto* is_null = builder.CreateOr(is_null_lhs, is_null_rhs);
          llvm::Value* res_value;

          switch (expr.binop) {
              case BinaryOp::kGt:
              {
                  auto* tmp = builder.CreateICmpSGT(value_lhs,
                  ↪  value_rhs);
                  res_value = builder.CreateZExt(tmp,
                  ↪  builder.getInt64Ty(), "i1_to_i64_zext");
                  break;
              }
            case BinaryOp::kLt:
              {
                  auto* tmp = builder.CreateICmpSLT(value_lhs,
                  ↪  value_rhs);
                  res_value = builder.CreateZExt(tmp,
                  ↪  builder.getInt64Ty(), "i1_to_i64_zext");
                  break;
              }
            case BinaryOp::kLe:
              {
                  auto* tmp = builder.CreateICmpSLE(value_lhs,
                  ↪  value_rhs);
                  res_value = builder.CreateZExt(tmp,
                  ↪  builder.getInt64Ty(), "i1_to_i64_zext");
                  break;
              }
            case BinaryOp::kGe:
              {
                  auto* tmp = builder.CreateICmpSGE(value_lhs,
                  ↪  value_rhs);
                  res_value = builder.CreateZExt(tmp,
                  ↪  builder.getInt64Ty(), "i1_to_i64_zext");
                  break;
              }
...
```

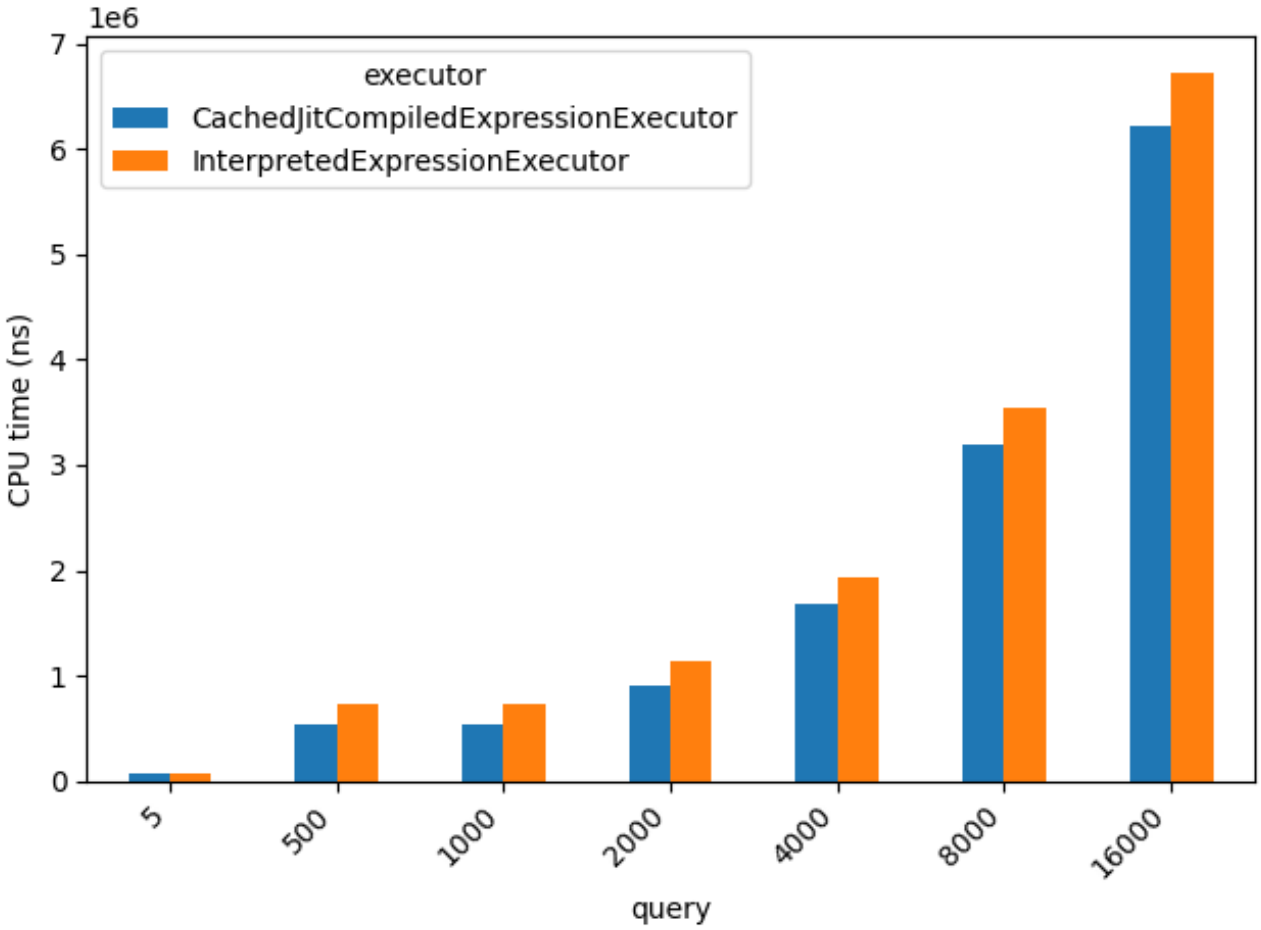

Рисунок 5 — Результаты бенчмарков.

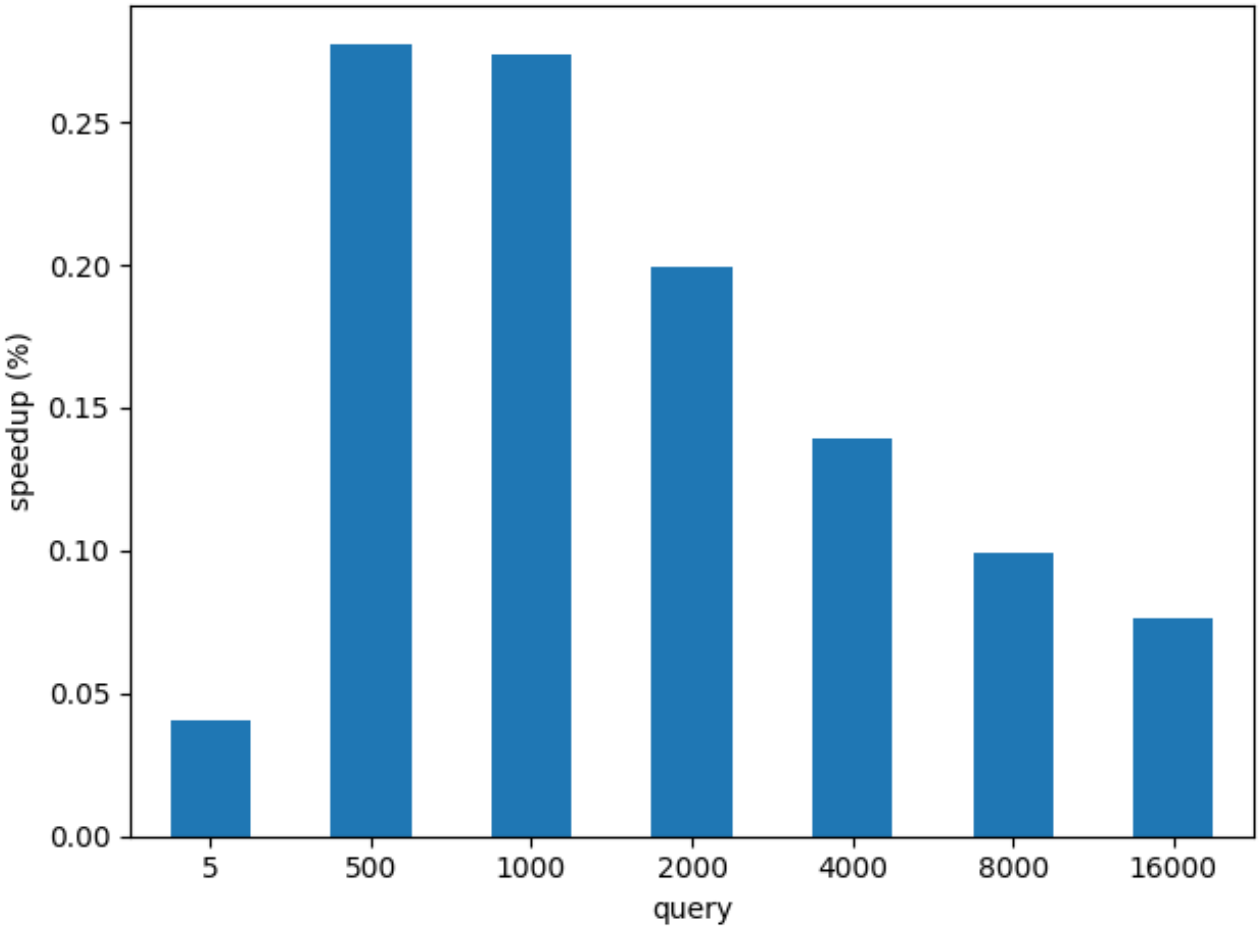

Рисунок 6 — Относительный прирост производительности.