Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем

Смирнов Александр Андреевич

Оптимизация алгоритма DFD поиска функциональных зависимостей в рамках платфомы Desbordante

Отчет по учебной практике

Научный руководитель:
ассистент кафедры ИАС Чернышев Г. А.

Санкт-Петербург
2021

Оглавление

Введение

Функциональная зависимость — это бинарное отношение на множестве подмножеств столбцов некоторой таблицы. Такие зависимости позволяют понять, есть ли некоторая связь (зависимость) между данными из двух наборов столбцов. Функциональные зависимости широко используются при работе с базами данных — например, при поиске и удалении дубликатов, оптимизации, нормализации.

Для поиска функциональных зависимостей был разработан ряд алгоритмов. Каждый алгоритм реализует собственный подход к поиску, из-за чего на различных наборах данных работает по-своему, потребляя разное количество ресурсов. Поэтому, в зависимости от свойств набора, может возникнуть потребность запустить на этом наборе какой-то конкретный алгоритм, чтобы добиться максимальной производительности.

Для этих целей на языке Java была реализована платформа Metanome [4, 5]. Она предоставляет возможность удобно запускать различные алгоритмы поиска функциональных зависимостей на некотором наборе данных. Однако, производительности Java не всегда достаточно для требовательных к ресурсам вычислений с большим объемом входной информации, поэтому Metanome может быть мало эффективным для промышленного использования. Чтобы устранить этот недостаток, была создана альтернативная платформа Desbordante [6]. При этом, чтобы достичь максимальной производительности, было решено использовать язык C++.

Desbordante — молодой проект, и на данный момент реализация алгоритмов в нём опирается на их реализацию из Metanome. Таким образом, используются не все возможности по оптимизации кода — незатронутой осталась как логика алгоритмов, так и возможные оптимизации, которые предоставляет более низкоуровневый C++. Поэтому возникла необходимость исследовать алгоритмы и структуры данных, используемые в Desbordante, на предмет возможных оптимизаций, и благодаря этому ускорить работу инструмента, насколько это возможно. Ускорению одного из таких алгоритмов и посвящена эта работа.

Постановка задачи

Целью работы является исследование возможности оптимизации алгоритма поиска функциональных зависимостей DFD в рамках платформы Desbordante.

Для достижения цели были поставлены следующие задачи:

- провести обзор и сравнение инструментов Metanome и Desbordante, а также обзор алгоритма DFD;
- выполнить предварительное исследование алгоритма на предмет возможных оптимизаций;
- реализовать обнаруженные оптимизации;
- сравнить производительность базовой и оптимизированной версий алгоритма.

1. Обзор

1.1. Основные определения

Definition 1 *Пусть A — множество атрибутов некоторой таблицы, B — какой-то атрибут. Говорят [2], что между A и B существует функциональная зависимость (Functional Dependency, FD) $A \to B$, если любые два кортежа, совпадающие на атрибутах A, совпадают на атрибуте B. В этом случае, A называется левой частью функциональной зависимости (Left Hand Side, LHS), а B — правой (Right Hand Side, RHS).*

Definition 2 *Если B не зависит функционально ни от одного подмножества A, функциональную зависимость $A \to B$ называют [2] минимальной.*

Definition 3 *Пусть A — множество атрибутов некоторой таблицы, B — какой-то атрибут. Говорят [2], что между A и B не существует функциональной зависимости, если найдутся два кортежа, совпадающие на атрибутах A, но не совпадающие на атрибуте B. В этом случае, отношение $A \not\to B$ называется нефункциональной зависимостью (Non-Functional Dependency, non-FD), A называется левой частью зависимости (Left Hand Side, LHS), а B — правой (Right Hand Side, RHS).*

Definition 4 *Если B зависит функционально от любого надмножества A, нефункциональную зависимость $A \not\to B$ называют [2] максимальной.*

Definition 5 *Пусть A — множество атрибутов некоторой таблицы, r — множество её кортежей. Партицией P набора столбцов A называют [2] множество классов эквивалентности кортежей, причём два кортежа входят в один класс, если их значения совпадают на атрибутах A.*

1.2. Существующие решения

1.2.1. Metanome

Платформа Metanome[1] — инструмент для получения метаданных (профилирования) наборов данных с открытым исходным кодом, написанный на язке Java. Он включает в себя множество алгоритмов профилирования, таких как алгоритмы поиска уникальных наборов столбцов, зависимостей включения, и в том числе алгоритмы поиска функциональных зависимостей. Задача Metanome — обеспечить удобную для пользователя работу с этими алгоритмами, объединив их в один инструмент [5]. Это позволяет быстро получить информацию из нужного набора данных — достаточно загрузить его в платформу, и становится возможным запускать на нём различные алгоритмы с помощью единого элемента управления. Чтобы сделать этот процесс более удобным, Metanome оснащён frontend-клиентом, который позволяет совершать эти операции в графическом интерфейсе. Также фреймворк предоставляет единый интерфейс для разработки новых алгоритмов на базе платформы, тем самым позволяя сразу интегрировать их в свою экосистему.

Благодаря тому, что реализованные алгоритмы собраны в одном инструменте и используют одни и те же вспомогательные структуры для обработки таблиц, становится возможным довольно точное сравнение производительности этих алгоритмов между собой. Этим преимуществом воспользовались авторы статьи [7].

1.2.2. Desbordante

Платформа Desbordante[2] — новый инструмент для поиска функциональных зависимостей с открытым исходным кодом, реализованный на языке C++. Его назначение отчасти совпадает с назначением Metanome: обе фреймворка объединяют существующие алгоритмы поиска ФЗ в один инструмент, а также предоставляют интерфейс для

[1] hpi.de/naumann/projects/data-profiling-and-analytics/metanome-data-profiling.html
[2] github.com/Mstrutov/Desbordante

разработки новых алгоритмов на базе платформы. Также инструмент оснащён frontend-клиентом, который обеспечивает удобную работу с исследуемым набором данных, в том числе и визуализацию полученных результатов.

Существенное отличие состоит в том, Metanome может не подойти для промышленного использования, поскольку производительности Java недостаточно [6] для работы с большими наборами данных. По этой причине Desbordante был реализован с целью достижения максимально возможной производительности алгоритмов поиска ФЗ. Этого планировалось достичь за счёт использования языка C++, а также возможности низкоуровневых оптимизаций, которые он предоставляет (например, реализация собственных аллокаторов, использование векторных SIMD-инструкций).

В Desbordante, как и в Metanome, реализованы собственные стркутуры, которые отвечают за обработку таблицы и представляют её данные в едином для всех алгоритмов виде.

1.3. Используемые инструменты

Backend часть Desbordante полностью реализована на языке C++. В процессе разработки активно используется стандартная библиотека шаблонов (STL). Кроме того, используются библиотеки boost [3] — это набор библиотек с открытым исходным кодом для языка C++. В частности, в процессе выполнения работы были использованы библиотеки dynamic_bitset и boost_asio.

Анализ производительности алгоритма производился при помощи профилировщика perf и графического интерфейса для него, который поставляется вместе со средой разработки CLion. Perf [14] — консольное приложение, инструмент для анализа производительности программ, являющийся частью ядра Linux. Графический интерфейс наглядно отображает полученные от perf данные в виде флейм-графа (flame graph).

1.4. Алгоритм DFD

DFD — один из алгоритмов поиска функциональных зависимостей. Он, как и ряд других алгоритмов, представляет [2, 7] пространство поиска функциональных зависимостей как *алгебраическую решетку* (рисунок 1), вершинами которой являются подмножества атрибутов таблицы.

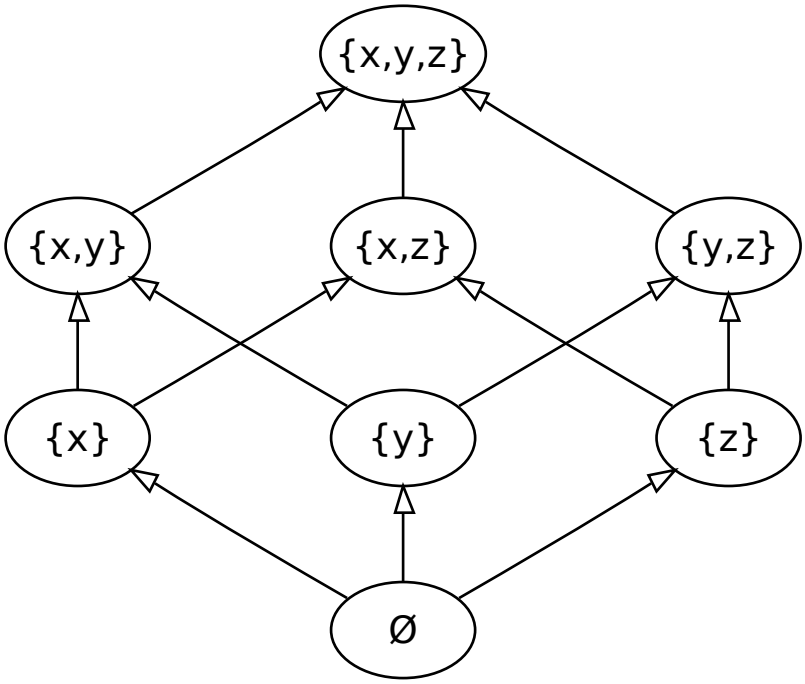

Рис. 1: **Пример алгебраической решетки.** Источник: [1]

В основном цикле алгоритм последовательно фиксирует каждый атрибут как RHS, и строит для него отдельную алгебраическую решетку из всех оставшихся атрибутов. С её помощью алгоритм находит LHS для зафиксированного RHS. Таким образом, алгоритм последовательно обратабывает набор алгебраичеких решеток. Отличительной особенностью данного алгоритма является то, что он исследует решетку методом, аналогичным поиску «в глубину» (depth-first manner), причем каждая следующая из доступных для обхода вершин выбирается случайным образом (random-walk). При проходе решетки алгоритм ищет минимальные FD, максимальные non-FD и отсеивает вершины, основываясь на информации об уже найденных зависимостях. Однако, при таком отсеивании возникают «островки» — части решетки, до которых

поиск в глубину дойти не сможет, потому что они окружены отброшенными вершинами. Такие места алгоритм обнаруживает благодаря найденным ранее максимальным non-FD, после чего запускает поиск в глубину уже внутри них. Для обнаружения FD/non-FD алгоритм либо применяет определение, исследуя подмножества/надмножества какой-то вершины, либо вычисляет и пересекает партиции.

В рамках платформы Desbordante данный алгоритм был ранее реализован автором отчёта.

Производительность алгоритма DFD на наборах данных с различными характеристиками была исследована в работе [7]. Исследователи выяснили, что при увеличении количества строк в таблице время работы алгоритма возрастает линейно, в то время как в ряде других алгоритмов наблюдается квадратичный рост. Другое преимущество алгоритма заключается в том, что он потребляет меньше памяти, чем некоторые другие алгоритмы с похожим способом обработки данных. Однако, если при обходе решётки образуется много «островков», алгоритму потребуется ощутимое количество времени, чтобы их обнаружить. Эта проблема зависит от свойств набора данных и, как правило, заментно усиливается с ростом числа колонок. По этой причине алгоритм медленно обрабатывает таблицы, в которых количество столбцов превышает 30. Настоящая работа пытается решить именно эту проблему.

За обнаружение таких «островков» и, соответственно, производительность этого процесса отвечает метод generateNextSeeds(), который описан в статье [2]. Метод запускается тогда, когда завершился предыдущий обход решётки. Если непосещённые вершины ещё остались, то, на основе найденных ранее non-FD, метод генерирует список новых вершин, каждая из которых лежит в своём «островке». В противном случае метод выдаёт пустой список, и работа всего алгоритма завершается. От каждой из полученных вершин в последующих методах запускается обход «островка», в котором эта вершина содержится. Основной цикл метода проходит по списку всех найденных non-FD и с его помощью генерирует список потенциальных искомых вершин. Генерация новых

вершин происходит с помощью операций над битсетами (bitset), такими как пересечение и инверсия. После этого на каждой итерации список нужно минимизировать, то есть удалить все надмножества лежащих в нём множеств атрибутов. В реализации из Desbordante за это отвечает метод minimize().

Методы, представляющие операции над битсетами, реализованы при помощи логических операторов. В случае с Desbordante, внутренняя реализация этих методов находится в классе dynamic_bitset. Метод minimize() отбрасывает вершины, подмножества которых уже лежат в списке. Для этого он, попарно в цикле сравнивая элементы списка, множество раз проверяет, является ли данная вершина (набор атрибутов) подмножеством каких-то других. Эта проверка осуществляется с помощью метода contains().

1.5. dynamic_bitset

boost::dynamic_bitset — шаблонный класс для работы с битсетами, который является частью набора библиотек boost. В отличие от класса std::bitset из стандартной библиотеки, этот класс не требует задания размера битсета на этапе компиляции, а позволяет задавать его во время исполнения программы. По этой причине биты в нём хранятся внутри динамического массива (std::vector), заполненного целочисленными значениями. Ячейки такого массива называются блоками.

В реализации алгоритма DFD часто используется метод contains(), который реализован на основе метода is_subset_of(). Метод is_subset_of() является членом класса dynamic_bitset. Он позволяет проверить, является ли один битсет подмножеством другого, при условии, что они имеют одинаковый размер. Поскольку в этом случае количество блоков одинаковое, алгоритм попарно сравнивает сначала первые блоки, затем вторые, и так далее. Проверка на то, содержится ли один блок в другом, происходит с помощью логических операций следующим образом:

1. применяется побитовое НЕ к первому блоку, который по предпо-

ложению содержит второй блок;

2. выполняется побитовое И между результатом предыдущего пункта и вторым блоком;

3. если в результате получился ноль, то предположение верно, и первый блок содержит второй. В противном случае — не содержит.

1.6. Возможные оптимизации

Существует различные способы оптимизации алгоритмов для работы с большими даными:

- распараллеливание;
- использование векторных SIMD-инструкций;
- реализация собственных аллокаторов для структур;
- использование вычислений на графических ускорителях.

В рамках данной работы было решено исследовать возможность распараллеливания алгоритма и использования SIMD-инстукций.

Как в оригинальной статье [2], представляющей алгоритм, так и в его релизации из Metanome не была предложена возможность распараллеливания. Далее, был изучен ряд статей [12, 13], в которых предлаются способы распараллеливания алгоритов поиска ФЗ. В работе [13] вводятся несколько структур, позволяющие распараллелить часть алгоритмов, но, по словам авторов, в случае DFD они не приведут к заметному увеличению производительности.

Pyro [11] — ещё один алгоритм поиска функциональных зависимостей. Его сходство с DFD заключается в том, что он так же последовательно строит и обрабатывает набор алгебраических решёток. Авторы статьи предложили параллельную версию алгоритма, в которой несколько таких решёток обрабатываются одновременно. Это привело автора настоящей работы к мысли о реализации похожего подхода у алгоритма DFD. Кроме того, авторами также была предложена структура

данных для хранения партиций. Точно такая же по своему назначению структура была описана в статье [2]. Поэтому для алгоритма DFD была использована реализация этой структуры из алгоритма Pyro.

1.7. Векторные инструкции

SIMD (Single-Instruction-Multiple-Data) — способ машинных вычислений, при котором достигается параллелизм на уровне данных. Со стороны процессора это значит, что за одну инструкцию можно провести некоторую операцию над массивом данных, размер которого в несколько раз превышает размер регистра общего назначения. Регистры, которые хранят такие массивы, и инструкции для работы с этими регистрами назвают векторными. Например, при помощи одной стандартной инструкции процессор может сложить между собой только одну пару чисел, а при помощи одной векторной — четыре пары, причём результат суммы каждой такой пары будет получен независимо от остальных. Такой подход позволяет существенно ускорить обработку больших массивов.

В процессорах с архитектурами IA-32 и IA-64 SIMD реализуется [8] при помощи двух типов расширений:

- Intel® Streaming SIMD Extension (SSE, SSE2, SSE3, SSE4);
- Intel® Advanced Vector Extensions (AVX, AVX2, AVX-512).

Каждое последующее расширение одного типа является улучшенной версией предыдущего, при этом сохраняя обратную совместимость. Расширения предоставляют набор дополнительных векторных регистров, размер которых составляет 128 бит у типе SSE, 256 бит у типа AVX, и, в особенности, 512 бит у AVX-512. Для работы с этими регистрами расширения содержат набор специальных векторных инструкций. Эти инструкции по своему назначению похожи на обычные инструкции процессора — например, они реализуют логические, арифметические и другие операции. Различие состоит в том, что стандартные инструкции

работают с регистрами общего назначения, в то время как векторные инструкции работают с векторными регистрами.

Расширение AVX-512 — новейшее расширение, позволяющее реализовать SIMD вычисления на процессоре. Однако, на данный момент его поддерживают лишь серверные процессоры и лишь немногие настольные процессоры. Поэтому на данном этапе работы было принято решение использовать расширения предыдущего поколения — AVX2. Сейчас оно более распространено, и, в том числе, поддерживается процессором тестового стенда, на котором прооводились эксперименты.

Поддержка векторных инструкций в языке C++ реализизуется с помощью «интринсиков» (intrinsics) [9, 10]. Это набор функций, написанных на языке ассемблера. Они позволяют использовать векторные инструкции в коде программы в виде привычных для программиста функций и переменных. Для работы с этими функциями предоставляются особые векторные типы данных. Такой подход позволяет компилятору встраивать векторные инструкции в код с учётом применяемых оптимизаций времени компиляции. В случае компилятора gcc, объявление всех необходимых функций и типов данных для работы с векторными инструкциями находится в заголовочном файле “immintrin.h”.

2. Описание решения

2.1. Распараллеливание

При обходе каждой из решёток создаётся новый набор структур, который не зависит от результаов обхода предыдущей решётки. Далее будем называть их структурами, представляющими решётку. Общими для всех решёток являются всего две структуры — список ранее найденных алгоритмом функциональных зависимостей, а также структура, которая хранит вычисленные партиции. Более того, список ранее найденных зависимостей не используется непосредственно во аремя обхода решётки, а пополняется уже по его завершении. Поэтому, хранилище партиций — оставшаяся структура, которая будут активно использоваться при одновременном обходе нескольких решёток.

В исходной версии обход решётки запускался с помощью метода findLHSs(). На вход ему подавался текущий RHS, для которого строилась решётка. Структуры, представляющие решётку, являлись полями основного класса алгоритма — класса DFD.

В распараллеленой версии алгоритма структуры, представляющие решётку, были вынесены в отдельный класс LatticeTraversal. Теперь для каждой решётки создаётся свой объект этого класса. Его конструктор в качестве параметра принимает RHS, для которого строится решётка. В классе реализован метод findLHS(), который запускает обход текущей решётки и возвращает список найденных зависимостей.

Параллельное выполнение реализовано при помощи пула потоков (thread pool) из библиотеки boost_asio. В DFD::execute_internal() — основном методе выполнения алгоритма — создаётся пул потоков, размер которого по умолчанию совпадает с количеством логических ядер на машине. Также размер можно задать и с помощью аргумента командной строки при запуске программы. Затем, в цикле по всем столбцам набора данных, в пул потоков добавляются задачи, каждая из которых запускает обход решётки для текущего столбца.

Ранне упоминалось, что реализация структуры, которая хранит пар-

тиции, была взята из алгоритма Pyro. Поскольку Pyro изначально был реализован с поддержкой распараллеливания, эта структура уже обладала необходимым свойством потокобезопасности.

2.2. Векторизация

В разделе 1.4 «Алгоритм DFD» было высказано предположение, что при запуске алгоритма на наборах данных с большим количеством колонок появляется много «островков». Из-за этого на таблицах с таким свойством производительность алгоритма падает. За процедуру поиска островков отвечает метод generateNextSeeds().

Было решено провести профилирование алгоритма и выявить наиболее медленные части кода, запустив его на нескольких таблицах с большим количеством колонок (от 19 до 30). Похожая проверка была выполнена в работах [2, 7], однако, там было приведено лишь общее время работы алгоритма.

Действительно, выяснилось, что при количестве колонок, превышающем 19, значительную часть времени работы алгоритма занимает метод generateNextSeeds(). В свою очередь, значительную часть времени работы этого метода составляет метод minimize(). А большую часть времени выполнения последнего работает метод contains(), который проверяет, содержится ли одно множество атрибутов в другом. Метод contains(), как уже известно, является обёрткой для метода is_subset_of().

Размер алгебраическиой решётки экспоненциально [2] растёт с увеличением числа колонок, поэтому рассматривать таблицы с большим набором столбцов в случае алгоритма DFD не имеет смысла. Для удобства векторзации и работы со внутренним представлением битсета было принято решение ограничить количество колонок входной таблицы до 64, и таким же числом задать размер блока. Таким образом, экземпляр класса dynaimc_bitset будет содержать всего лишь один блок размером 64 бита.

В разделе 1.5 «dynaimc_bitset» был описан упрощённый принцип

работы метода проверки пары блоков двух битсетов на включение. Поскольку в нашем случае битсет содержит только один блок, то эта операция будет проведена единожды. Получаем, что проверка на то, содержится ли один битсет длиной 64 бита в в другом, проводится за 3 машинных инструкции: побитовое отрицание, побитовое И, сравнение с нулём.

Таким образом, за три машинные инструкции мы можем проверить, содержится ли один битсет длиной 64 бита в в другом. Здесь мы учитываем то, что регистр общего назначения имеет длину 64 бита, как на тестовом стенде.

Теперь мы можем применить векторные регистры и векторные инструкции. В векторном регистре длиной 256 бит мы сможем поместить сразу 4 битсета, и за то же самое количество инструкций проделать предыдущие операции сразу для четырёх пар битсетов, а не для одной.

Класс dynamic_bitset не предоставляет прямого доступа к своему массиву блоков, который является его приватным членом. Для работы с содержимым массива определена дружественная шаблонная функция to_block_range(). Она принимает на вход битсет и итератор некоторого контейнера, после чего **копирует** содержимое массива битсета по этому итератору. Таким образом, возникает проблема излишнего копирования даже при простом чтении блоков. Она была решена следующим образом. Один из шаблонных параметров функции — тип принимаемого итератора. Была реализована особая структура, которая заполняла один из векторных регистров значением поданного на вход блока. Затем шаблон функции to_block_range() был специализирован таким образом, что в качестве типа итератора был подставлен тип введённой структуры. Получили, что специализированная шаблонная функция смогла принимать на вход эту структуру. Внутри этой функции все структуры имеют доступ к приватным полям класса dynamic_bitset, что и использует новая структура. Это позволило считывать блоки напрямую из битсета (который, напомним, содержит всего один блок) в векторный регистр. При этом не производилось никаких дополнительных копирований. Таким образом, был получен модифицированный с

использованием SIMD-инструкций метод minimize().

3. Эксперименты

3.1. Параметры тестового стенда и угрозы валидности

Тестирование проводилось на тестовом стенде со следующими характеристиками:

- операционная система: Manjaro Linux 20.2.1 (Linux kernel 5.13.19-2-MANJARO), gcc version 11.1.0;
- процессор: x86_64 Intel Core i5-8300H 4 cores 8 threads, 4.00 GHz 8 MiB L3 cache;
- оперативная память: 16 GiB DDR4.

Алгоритм DFD относится к рандомизированным алгоритмам, поскольку путь обхода решётки выбирается случайным образом. Это может влиять на общее время работы алгоритма, поэтому в серии его замеров может возникнуть ощутимый разброс. Чтобы увеличить точность замеров времени работы алгоритма, рандомизация была отключена. Таким образом, при каждом запуске алгоритма на фиксированном наборе данных обход решётки производился по одному и тому же пути.

3.2. Результаты

Сначала было произведено сравнение первоначальной версии алгоритма и его распараллеленой версии. Распараллеленая версия запускалась на 4 потоках. По результатам десяти измерений были построены доверительные интервалы. Они представлены в таблице 1, а также нанесены на рисунке 2.

Видно, что на некоторых наборах данных ускорения почти нет, либо оно довольно незначительное. На таблицах breast_cancer и hepatitis скорость работы алгоритма увеличилась почти в 3 раза.

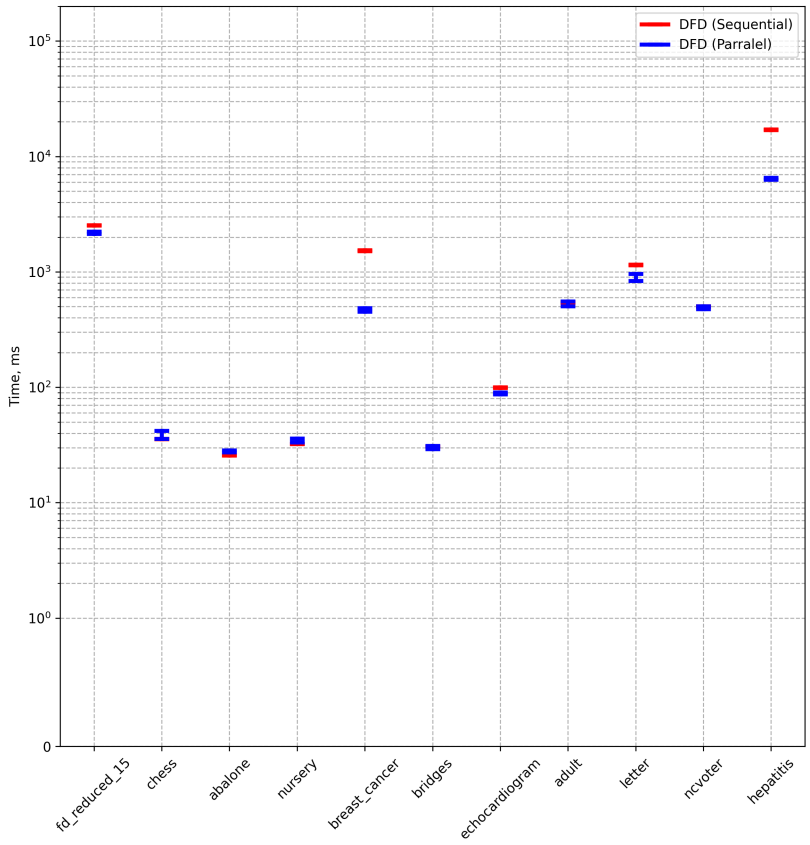

Рис. 2: **Результаты замеров**

	abalone	adult	breast_cancer	bridges	chess	cardiogram	fd_reduced	hepatitis	letter	ncvoter	nursery
Sequential	25 ± 1	546 ± 1	1535 ± 8	30 ± 0	35 ± 0	99 ± 0	2542 ± 14	17101 ± 97	1150 ± 5	503 ± 3	31 ± 0
Parallel	30 ± 1	546 ± 26	448 ± 17	32 ± 1	39 ± 2	93 ± 3	2219 ± 83	6236 ± 23	851 ± 13	502 ± 5	40 ± 7

Таблица 1: Время работы распараллеленой и последовательной версий DFD в миллисекундах

Затем сравнили изначальную (последовательную) версию алгоритма с версией, в которой использовались SIMD-инструкции. Замеры векторизованной версии были проведены на одном потоке, результаты приведены в таблице 2. Оказалось, что использование SIMD-инструкций почти не повлияло на скорость работы алгоритма — небольшой прирост производительности наблюдается лишь на таблицах breast_cancer и hepatitis.

	abalone	adult	breast_cancer	bridges	chess	cardiogram	fd_reduced	hepatitis	letter	ncvoter	nursery
Sequential	25 ± 1	547 ± 7	1533 ± 10	30 ± 0	35 ± 0	99 ± 1	2548 ± 1	17029 ± 95	1145 ± 1	503 ± 4	31 ± 0
Vectorized	25 ± 1	545 ± 1	1483 ± 7	30 ± 0	35 ± 0	99 ± 0	2543 ± 7	16445 ± 93	1152 ± 10	506 ± 5	31 ± 0

Таблица 2: Время работы последовательной и векторизованной версий DFD в миллисекундах

3.3. Анализ полученных результатов

В случае распараллеливания на большинстве таблиц производительность алгоритма почти не увеличилась, а на таблицах breast_cancer и hepatitits увеличилась в несколько раз. Это связано с тем, что эти наборы данных имеют большое число колонок (30 и 20 соответственно), в то время как остальные наборы содержат менее 19 колонок. Последовательная версия алгоритма вынуждена обойти столько решёток, сколько колонок содержит набор данных. Параллельная же обходит сразу несколько, за счёт чего, как и ожидалось, выросла производительность.

Ожидалось, что в случае векторизованой версии также увеличится производительность на таблицах с большим количеством колонок. Однако, этого не произошло, и максимальный прирост составил около 3% (на таблице hepatitis). Причина этому — то, что структуры данных, которые изначально используются в Desbordante, не были ориентированы на использоварние SIMD-инструкций, поэтому данные в них хранятся разрозненно. Это, во-первых, не позволяет использовать эффективные инструкции загрузки упакованных данных в векторные регистры, а во-вторых, затрудняет кеширование данных процессором.

Заключение

В ходе работы над оптимизацией алгоритма были выполнены следующие задачи:

- проведён обзор и сравнение инструментов Metanome и Desbordante, а также алгоритма DFD;
- выполнено предварительное исследование возможностей оптимизации алгоритма;
- реализованы выбранные оптимизации;
- произведено сравнение производительности базовой и оптимизированной версии алгоритма.

Исходный код алгоритма доступен на Github [15]. Он находится в папке src/algorithms/DFD.

Список литературы

[1] By I, KSmrq, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2118211.

[2] Abedjan Ziawasch, Schulze Patrick, and Naumann Felix. DFD: Efficient Functional Dependency Discovery // Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management. — New York, NY, USA : Association for Computing Machinery. — 2014. — CIKM '14. — P. 949–958. — Access mode: https://doi.org/10.1145/2661829.2661884.

[3] Boost C++ libraries. — Access mode: https://www.boost.org/ (online; accessed: 2021-12-14).

[4] Papenbrock Thorsten, Bergmann Tanja, Finke Moritz, Zwiener Jakob, and Naumann Felix. Data Profiling with Metanome // Proc. VLDB Endow. — 2015. — Aug. — Vol. 8, no. 12. — P. 1860–1863. — Access mode: http://dx.doi.org/10.14778/2824032.2824086.

[5] Papenbrock Thorsten, Bergmann Tanja, Finke Moritz, Zwiener Jakob, and Naumann Felix. Data Profiling with Metanome // Proc. VLDB Endow. — 2015. — aug. — Vol. 8, no. 12. — P. 1860–1863. — Access mode: https://doi.org/10.14778/2824032.2824086.

[6] Strutovskiy Maxim, Bobrov Nikita, Smirnov Kirill, and Chernishev George. Desbordante: a Framework for Exploring Limits of Dependency Discovery Algorithms // 2021 29th Conference of Open Innovations Association (FRUCT). — 2021. — P. 344–354.

[7] Papenbrock Thorsten, Ehrlich Jens, Marten Jannik, Neubert Tommy, Rudolph Jan-Peer, Schönberg Martin, Zwiener Jakob, and Naumann Felix. Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms // Proc. VLDB Endow. — 2015. — June. — Vol. 8, no. 10. — P. 1082–1093. — Access mode: https://doi.org/10.14778/2794367.2794377.

[8] Intel SIMD extensions. — Access mode: https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html (online; accessed: 2021-12-14).

[9] Intel® Intrinsics Guide. — Access mode: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html (online; accessed: 2021-12-14).

[10] Intel® Intrinsics Reference. — Access mode: https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/details-about-intrinsics.html (online; accessed: 2021-12-14).

[11] Kruse Sebastian and Naumann Felix. Efficient Discovery of Approximate Dependencies // Proc. VLDB Endow. — 2018. — Mar. — Vol. 11, no. 7. — P. 759–772. — Access mode: https://doi.org/10.14778/3192965.3192968.

[12] Saxena Hemant, Golab Lukasz, and Ilyas Ihab F. Distributed Discovery of Functional Dependencies // 2019 IEEE 35th International Conference on Data Engineering (ICDE). — 2019. — P. 1590–1593.

[13] Saxena Hemant, Golab Lukasz, and Ilyas Ihab F. Distributed Implementations of Dependency Discovery Algorithms. — 2019. — jul. — Vol. 12, no. 11. — P. 1624–1636. — Access mode: https://doi.org/10.14778/3342263.3342638.

[14] perf: Linux profiling with performance counters. — Access mode: https://perf.wiki.kernel.org/index.php/Main_Page (online; accessed: 2021-12-14).

[15] Исходный код. — Access mode: https://github.com/Mstrutov/Desbordante (online; accessed: 2021-12-14).