Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Группа 23.Б08-мм

Обзор алгоритма поиска функциональных зависимотей FDHits

Бокай Иван Андреевич

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
ассистент кафедры ИАС Чернышев Г. А.

Санкт-Петербург
2025

Оглавление

Введение

В современном мире объём данных неумолимо растёт, и вместе с этим увеличивается потребность в их анализе. Профилирование данных — это набор действий и процессов, направленных на извлечение метаданных о данном наборе данных [1]. Метаданные — это некоторые данные о данных. Они могут включать в себя как их размер, количество строк или столбцов, так и более скрытые свойства.

У людей, работающих, с данными часто возникает необходимость профилирования данных, с которыми они работают. Метаданные могут служить основой для вынесения каких-либо суждений о данных, либо могут использоваться для очистки данных и тем самым повышения их качества [8] или служить для оптимизации запросов к данным [2]. В связи с потребностью, разработаны и продолжают разрабатываться алгоритмы поиска различных видов метаданных.

Функциональные зависимости [4] — один из самых распространённых и важных видов метаданных. В платформе Desbordante [3] реализовано большое количество алгоритмов поиска функциональных зависимостей. Однако существуют массивы данных, для которых поиск функциональных зависимостей даже с помощью новейших алгоритмов либо занимает огромное количество времени, либо вовсе не возможен. Для решения данной проблемы, авторами статьи [4] был предложен новый алгоритм поиска функциональных зависимостей FDHits. Он включает в себя оптимизации из всех ранее предложенных алгоритмов и, как утверждают авторы, значительно превосходит их как по производительности, так и по количеству потребляемой памяти.

В данной работе будет проведён обзор алгоритма FDHits для поиска функциональных зависимостей с целью его дальнейшей реализации.

1. Постановка задачи

Целью данной работы является обзор статьи “Discovering Functional Dependencies through Hitting Set Enumeration” [4], предложенного в ней алгоритма FDHits и замеров производительности. Для её выполнения были поставлены следующие задачи:

1. Провести обзор предметной области;
2. Разобрать статью [4] и выполнить обзор алгоритма FDHits;
3. Выполнить обзор замеров производительности алгоритма, предложенных авторами статьи [4].

2. Обзор

2.1. Обзор предметной области

2.1.1. Функциональные зависимости

Начнём с нескольких важных определений (взяты из статьи [4]).

Пусть R — множество отношений(реляционная схема), r — какое-то отношение из этого множества.

Определение 1. Пусть $S \subseteq R$ — множество атрибутов, $A \in R$ — конкретный атрибут.

Зависимость $S \to A$ является функциональной зависимостью тогда и только тогда, когда не существует таких $t_1, t_2 \in r$, что $t_1[S] = t_2[S]$, но $t_1[A] \neq t_2[A]$.

Определение 2. Функциональная зависимость $S \to A$ называется минимальной, если не существует таких функциональных зависимостей $S' \to A$, что $S' \subsetneq S$.

Определение 3. Функциональная зависимость $S \to A$ называется нетривиальной, если $A \notin S$.

Под задачей поиска функциональных зависимостей понимается нахождение всех минимальных, нетривиальных функциональных зависимостей в некотором наборе данных [4].

2.1.2. Основные алгоритмы

На данный момент существует большое количество алгоритмов поиска функциональных зависимостей. Первыми основными алгоритмами являлись Tane, FUN, FD_Mine, DFD, DEP-Miner, FastFDs и FDEP, экспериментальное исследование которых представлено в статье [6]. Все эти алгоритмы были крайне медленными для исследования наборов данных реальных размеров. Чтобы справиться с этой проблемой был представлен более эффективный алгоритм — HyFD [12]. Так же стоит отметить алгоритм DynFD [5], позволяющий осуществлять поиск

функциональных зависимостей на динамически меняющихся наборах данных. Однако даже HyFD по разным причинам перестал справляться с некоторыми наборами данных: либо из-за серьёзных требований по памяти, либо из-за недостаточной производительности. С целью решения данной проблемы был представлен алгоритм FDHits [4], который по словам авторов в разы эффективнее предыдущих решений как по производительности, так и по количеству потребляемой памяти, что влечёт за собой способность обрабатывать большие наборы данных за разумное время.

2.1.3. Назначение функциональных зависимостей

Функциональные зависимости являются одними из самых важных ограничений целостности в базах данных. Они служат для оптимизации запросов к данным [9], интеграции [10] и преобразования данных [14]. Также функциональные зависимости позволяют улучшить очистку данных для обнаружения и устранения несоответствий [8].

2.2. Определения

В данном подразделе приводятся определения из работ [4], [7] и [11].

Определение 4. Для двух $t_1, t_2 \in r$ их множеством расхождений (Difference Set) называется множество всех атрибутов $A \in R$ таких, что $t_1[A] \neq t_2[A]$.

Определение 5. Гиперграф $\mathcal{F}$ — это множество подмножеств $F_1, \ldots, F_m$ множества вершин V, при этом, каждое подмножество $F_i \subseteq V$ называется гиперребром.

Определение 6. Вершинное покрытие (Hitting Set) гиперграфа $\mathcal{F}$ — это подмножество V, имеющее непустое пересечение с каждым ребром $\mathcal{F}$.

Определение 7. Вершинное покрытие S называется минимальным (Minimal Hitting Set), если не существует такого вершинного покрытия $S_1 : S_1 \subseteq S$.

Определение 8. Пусть $S \subseteq V$, $v \in S$. Ребро F гиперграфа $\mathcal{F}$ называется критическим для v, если $S \cap F = v$.

Обозначение 1. Обозначим $\mathcal{F}(v)$ — множество рёбер гиперграфа $\mathcal{F}$, включающих в себя вершину $v \in V$.

Обозначение 2. Пусть $S \subseteq V$. Обозначим $uncov(S)$ — множество рёбер гиперграфа $\mathcal{F}$, не пересекающихся с S. S — вершинное покрытие тогда и только тогда, когда $uncov(S) = \emptyset$.

Обозначение 3. Пусть $S \subseteq V$. Обозначим $crit(v, S)$ — множество всех критических рёбер для v. S — минимальное вершинное покрытие тогда и только тогда, когда $uncov(S) = \emptyset$ и $\forall v \in Scrit(v, S) \neq \emptyset$.

2.3. Обзор алгоритма

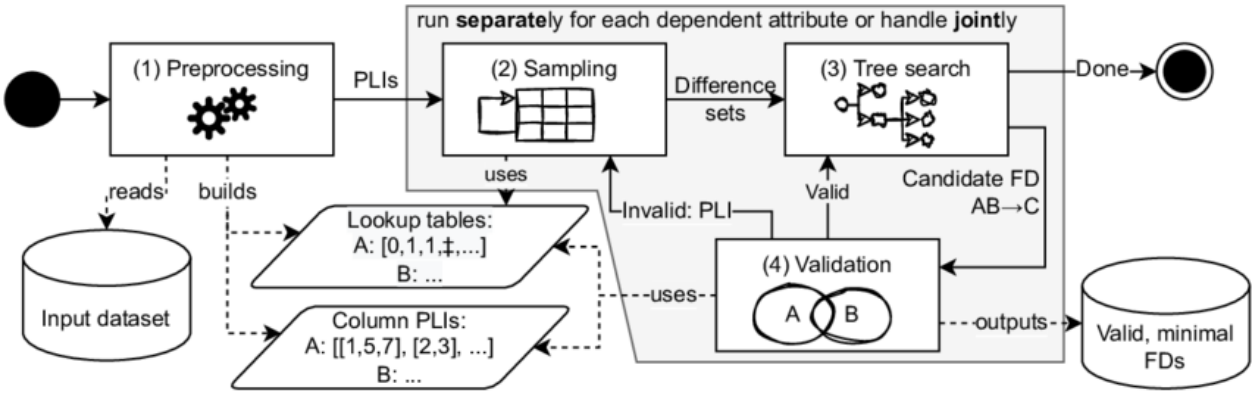

Рис. 1: Общая структура алгоритма FDHits (изображение взято из работы [4])

Структура алгоритма 1 повторяет структуру алгоритма HPIValid [7] для поиска уникальных комбинаций колонок, однако в связи с их отличием от функциональных зависимостей отдельные компоненты требуют изменения. На рисунке изображена схема работы алгоритма FDHits. В этой главе будет представлен разбор каждого этапа работы алгоритма FDHits.

2.3.1. Общая идея

Идея алгоритма заключается в нахождении всех минимальных покрытий множества наборов расхождений. Если взять множество наборов расхождений содержащих атрибут A, то отношения $S \to A$ являются функциональными зависимостями, где S — одно из найденных минимальных вершинных покрытий. Авторы предлагают две вариации алгоритма поиска минимальных вершинных покрытий. Первая — $\text{FDHits}_{\text{sep}}$ рассматривает каждый зависимый атрибут функциональной зависимости отдельно. Вторая — $\text{FDHits}_{\text{joint}}$ рассматривает зависимые атрибуты вместе. Полный алгоритм является гибридным и зависимости от некоторых условий выбирает версию алгоритма, которая будет работать лучше.

2.3.2. Предварительная обработка данных (Preprocessing)

На данной стадии алгоритм для каждого подмножества столбцов таблицы рассчитывает позиционные индексы списка (Position List Indices, PLIs) и таблицы просмотра для них (Lookup Table).

PLI для подмножества атрибутов $S \subseteq R$ делит базу данных на кластеры. Две записи, представляемые их номером, попадают в один и тот же кластер, если они совпадают по значению.

Таблица просмотра генерируется одновременно с PLI для каждого столбца. Она содержит в себе номер кластера PLI, в котором лежит атрибут, если атрибут принадлежит единичному кластеру, то он маркируется по особому. Таблица просмотра используется для эффективной генерации PLI для множества столбцов [13].

2.3.3. Выборка множеств расхождений (Sampling)

На данном этапе происходит выборка множеств расхождений. Выборка должна производиться для всех пар строк, однако сложность такого подхода $O(n^2)$. В качестве оптимизации, авторами, было предложено делать выборку не по всем парам, а по какому-то случайному количеству. Авторы заявляют, что оптимальным является количество

выборок c_p^{ε} при $\varepsilon = 0.3$, где c_p — количество всевозможных пар строк. Затем на основе такой случайной выборки формируется частичный гиперграф.

Пример построения частичного гиперграфа

	Room_Nr	Time	Course	Lecturer
t_1	101	Wed 10:00 am	Programming	Miako
t_2	101	Wed 02:00 pm	Databases	Daniel
t_3	102	Fri 02:00 pm	Programming	Miako
t_4	101	Fri 02:00 pm	Databases	Saurabh

Рис. 2: Пример таблицы с данными (изображение взято из работы [4])

В качестве примера возьмём таблицу 2. Количество всевозможных пар строк равно 6. Следовательно количество рассмотренных пар при округлении будет равно $6^{0,3} = 2$. Допустим были рассмотрены строки t_1, t_2 и t_1, t_3. Формируем гиперграф, состоящий из наборов расхождений (difference sets). В нашем случае получится $\mathcal{F} = \{\{time, course, lecturer\}, \{room\_Nr, time\}\}$.

2.3.4. Алгоритм MMCS

Алгоритм MMCS — это алгоритм для поиска минимальных вершинных покрытий гиперграфа.

Описание алгоритма взято из работы [15] Михаила Синельникова.

Алгоритм MMCS [11] совершает обход дерева, в котором узлы представляются как наборы атрибутов. В корне дерева узел с пустым набором, на каждом последующем уровне — число атрибутов в наборе увеличивается на один.

Обозначения переменных:

- S — текущий набор атрибутов
- *uncov* — множество рёбер, не пересекающихся с S
- *crit* — множество множеств критических рёбер для атрибутов из S (*crit[u]*)

Algorithm 1 MMCS

```
1: Input: S (initially ∅)
2: Output: Set of vertices S (minimal hitting set)
3: Global variables:
4: crit[u] = ∅ for each u
5: uncov = F
6: CAND = V
7: function MMCS(S)
8:     if uncov = ∅ then
9:         output S
10:        return
11:    end if
12:    Choose a hyperedge F from uncov
13:    C := CAND ∩ F
14:    CAND := CAND \ C
15:    for each v ∈ C do
16:        UPDATE_CRIT_UNCOV(v, crit[], uncov)
17:        if crit(f, S ∪ {v}) ≠ ∅ for each f ∈ S then
18:            MMCS(S ∪ {v})
19:        end if
20:        CAND := CAND ∪ {v}
21:        Recover changes to crit[] and uncov done in 14
22:    end for
23: end function
```

Algorithm 2 Update_crit_uncov

```
1: function UPDATE_CRIT_UNCOV(v, crit[ ], uncov)
2:     for each F ∈ F(v) do
3:         if F ∈ crit[u] for a vertex u ∈ S then
4:             remove F from crit[u]
5:         end if
6:         if F ∈ uncov then
7:             uncov := uncov \ {F}
8:             crit[v] := crit[v] ∪ {F}
9:         end if
10:    end for
11: end function
```

- *CAND* — множество кандидатов на добавление в S

Как было написано в разделе 2.2, вершинное покрытие является минимальным, если $uncov = \emptyset$ и $\forall u \in S$ $crit[u] \neq \emptyset$ Алгоритм MMCS начинает работу с $S = \emptyset$ и заканчивается, когда кандидаты перестают удовлетворять второму условию минимального вершинного покрытия (условию минимальности). При каждом вызове к текущему набору S добавляются вершины v из множества *CAND*. Затем алгоритм вызывается для $S \cup v$.

Поскольку S удовлетворяет условию минимальности, то если $uncov \neq \emptyset$ S является минимальным вершинным покрытием и поиск в текущей ветви дерева прекращается.

Далее алгоритм выбирает ребро F (строка 10), минимизирующее $|F \cap CAND|$. Для минимальности вершинного покрытия S необходимо, чтобы $S \cap F \neq \emptyset$. Заметим, что $F \in uncov \Rightarrow S \cap F = \emptyset$. Поиск продолжается для $C = CAND \cap F$ для того, чтобы найти такое $S' \supseteq S$, что оно содержит хотя бы одну вершину $v \in F$. Как было сказано выбирается минимизирующее F. Авторы утверждают, что такой выбор ускоряет работу алгоритма.

На следующей стадии (строки 13–20) происходит перебор всех вершин $v \in C$. Для каждой вершины формируется новый кандидат $S \cup v$ и происходит проверка его минимальности. Если проверка успешна, то производится рекурсивный вызов алгоритма. Происходит обновление *uncov*, *crit* и *CAND* с помощью алгоритма 2.

В платформе Desbordante уже есть реализация алгоритма MMCS, которая присутствует в коде алгоритма HPIValid[1].

2.3.5. Поиск минимальных покрытий

Если обобщать идеи HPIValid напрямую на поиск функциональных зависимостей, то нужно рассматривать каждый подгиперграф $\mathcal{H}_A = (R, \mathcal{D}_A)$ отдельно для каждого атрибута $A \in R$. Данная идея используется авторами статьи в вариации алгоритма $\text{FDHits}_{\text{sep}}$. Он запускает

[1]https://github.com/Desbordante/desbordante-core/tree/main/src/core/algorithms/ucc/hpivalid

поиск минимальных покрытий (MMCS) для каждого атрибута.

Однако, авторы утверждают, что рассмотрение зависимых частей функциональной зависимости совместно может значительно ускорить работу алгоритма. В качестве иллюстрации ими приводится идеальный случай, при котором каждый набор расхождений, содержащий A также содержит B, это образует действительную функциональную зависимость $B \to A$. Таким образом, если будут найдены функциональные зависимости для атрибута A, учитывающие наборы расхождений для A, они также будут актуальны для атрибута B. С этого момента требуется покрыть только наборы расхождений, актуальные для B и найти недостающие функциональные зависимости.

Для этого авторы предлагают алгоритм $\text{FDHits}_{\text{joint}}$.

Псевдокод 3 описывает алгоритм поиска минимальных покрытий в $\text{FDHits}_{\text{joint}}$.

Обозначения переменных:

- S — текущий набор вершин;
- V — набор кандидатов на добавление в S;
- W — множество возможных зависимых частей функциональной зависимости;
- *uncov(S, W)* — множество рёбер, содержащих вершину из W, но ещё не покрытых S;
- $\mu = (S, V, W)$ — узел дерева с выбранными вершинами S, кандидатами V и возможными зависимыми атрибутами W.

Ветвление (branching)

Поиск покрытий в $\text{FDHits}_{\text{joint}}$ так же базируется на алгоритме MMCS и использует такое же ветвление. Для того, чтобы рассматривать несколько зависимых атрибутов одновременно поддерживается множество W. Для поддерева под узлом μ целью является нахождение всех функциональных зависимостей $T \to A$, где $A \in W$, $S \subseteq T \subseteq S \cup V$. Это эквивалентно нахождению для каждого $A \in W$ всех минимальных

Algorithm 3 The tree search of $\text{FDHits}_{\text{joint}}$

```
1: function treeSearch(S, V, W)                                   ▷ Pruning
2:     for (C, A) ∈ S × W do
3:         if critical_S(C, A) = ∅ then
4:             W ← W \ {A}
5:         end if
6:     end for
7:     for B ∈ V \ S do
8:         if ∀A ∈ W ∃C ∈ S ∀E ∈ critical_S(C, A) : B ∈ E then
9:             V ← V \ {B}
10:        end if
11:    end for
12:    if W = ∅ then
13:        return
14:    end if
                                          ▷ Validation at the leaves
15:    if uncov(S, W) = ∅ ∧ validate(S → W) then
16:        output S → W
17:        return
18:    end if
                                                    ▷ Branching
19:    E ← edge in uncov(S, W) minimizing |E ∩ V| + |W \ E|
20:    {B_1, ..., B_k} ← E ∩ V
21:    treeSearch(S, V, W \ E)
22:    for i ∈ {1, ..., k} do
23:        treeSearch(S ∪ {B_i}, V \ {B_1, ..., B_i}, W ∩ E)
24:    end for
25: end function
```

покрытий T гиперграфа $\mathcal{H}_A$, где $S \subseteq T \subseteq S \cup V$. Алгоритм начинает свою работу в корне $(S = \emptyset, V = R, W = R)$.

Если $uncov(S, W) = \emptyset$ для некоторого узла $\mu = (S, V, W)$, то μ — это лист, для которого $S \to W$ — это действительная функциональная зависимость. Далее она попадает на стадию верификации (о ней дальше).

Если же $uncov(S, W) \neq \emptyset$, алгоритм берёт непокрытое ребро. Выбирается такая $E \in uncov(S, W)$, что $|E \cap V| + |W \backslash E|$ минимально. Положим $E \cap V = \{B_1, \ldots, B_k\}$ — множество вершин-кандидатов из E. Выбирая ребро E создаётся k+1 дочерних узлов $\mu_0, \ldots, \mu_k$. Узел μ_0 соответствует факту, что покрытие E имеет значение только для зависимых атрибутов, которые так же содержаться в E. Следовательно, в узле μ_0 перебираются атрибуты, не находящиеся в E.

Для зависимых атрибутов из $W \cap E$, выполняется ветвление по каждой вершине из E. Это порождает потомков $\mu_1, \ldots, \mu_k$, где $\mu_i = (S \cup \{B_i\}, V \setminus \{B_1, \ldots, B_i\}, W \cap E)$.

Корректность такого ветвления авторы объясняют леммой.

Лемма 1. В поддереве узла $\mu = (S, V, W)$ поиск покрытий находит каждую минимальную действительную функциональную зависимость $T \to A$ ровно один раз, где $A \in W$, $S \subseteq T \subseteq S \cup V$.

Доказательство леммы приведено в статье [4].

Пример работы

Возьмём в качестве примера таблицу изображённую на рисунке 2.

Допустим, что изначально алгоритм выбрал ребро $E \in uncov(S, W)$, $E = [time, course, lecturer]$ (получено из сравнения строк t_1, t_2).

$E \cap V = [time, course, lecturer]$. Соответственно $B_1 = time, B_2 = course, B_3 = lecturer$.

Рекурсивно запускается TreeSearch(S, V, $W \setminus E = [room]$).

Далее перебираются $i \in 1, \ldots, k$ (количество атрибутов B, в нашем случае 3). На каждой итерации рекурсивно запускается TreeSearch($S \cup \{B_i\}$, $V \setminus \{B_1, \ldots, B_i\}$, $W \cap E$).

В нашем случае:

1. TreeSearch($S \cup \{time\} = [time]$, $V \setminus \{time\} = [room, course, lecturer]$, $W \cap E = [time, course, lecturer]$)

2. TreeSearch($S \cup \{course\} = [course]$, $V \setminus \{time, course\} = [room, lecturer]$, $W \cap E = [time, course, lecturer]$)

3. TreeSearch($S \cup \{lecturer\} = [lecturer]$, $V \setminus \{time, course, lecturer\} = [room]$, $W \cap E = [time, course, lecturer]$)

Далее в каждом таком узле для уже новых S, W проверяется, пусто ли множество $uncov(S, W)$. Если оно оказалось пусто и $S \to W$ — это ФЗ, то выдаём как результат. Если оказалось пусто, но $S \to W$ — это не ФЗ, то гиперграф дополняется новым ребром.

Если не выдался никакой результат, выполняются аналогичные шаги.

Отсечение тривиальных и не минимальных функциональных зависимостей (pruning)

Положим $\mu = (S, V, W)$ — текущий узел, $A \in W$ — один из возможных зависимых атрибутов. Если вершина из S не имеет ребра в подгиперграфе $\mathcal{H}_A$, тогда не существует минимальной функциональной зависимости $T \to A$, где $S \subseteq T$. В таком случае убираем вершину A из множества W. Если оказалось, что $W = \emptyset$, в текущем узле дерево обрезается.

Для реализации для каждой вершины $C \in S$ и $A \in W$ поддерживается множество $critical_{\mathrm{S}}$(C, A), которое содержит критические рёбра для C в подгиперграфе $\mathcal{H}_A$. Если $critical_{\mathrm{S}}$(C, A), то A удаляется из множества W. Далее если добавление вершины B в множество S влечёт удаление каждого зависимого атрибута $A \in W$, то B удаляется из V. В обозначения $critical_{\mathrm{S}}$(C, A) — это случай, когда для каждого $A \in W$ существует такая выбранная вершина $C \in S$, что все рёбра $critical_{\mathrm{S}}$(C, A) также содержат B.

2.3.6. Верификация

Так как поиск минимальных покрытий запускается для частичного гиперграфа, полученные минимальные покрытия могут не является таковыми для полного гиперграфа. Следовательно необходимо проверить, является ли полученное минимальное покрытие функциональной зависимостью. Если да, то в результат записывается новая функциональная зависимость, если нет, то частичный гиперграф дополняется наборами расхождений для тех пар, для которых не была пройдена проверка и поиск минимального покрытия запускается снова. Зависимость $S \to A$ является функциональной тогда и только тогда, когда значения каждого кластера из PLI для S совпадают на A. Для проверки функциональных зависимостей кластеры единичного размера могут быть удалены. В качестве оптимизации авторы предлагают при проверке функциональной зависимости удалять целые кластеры из PLI, которые не являются релевантными для рассматриваемого зависимого атрибута A в случае $\text{FDHits}_{\text{sep}}$ или для одного из атрибутов из набора W в случае $\text{FDHits}_{\text{joint}}$. Каждый PLI, рассчитанный для правых частей W должен содержать хотя бы одну пару строк, которая нарушает функциональную зависимость $S \to A$ хотя бы для одного атрибута $A \in W$. Для реализации используется лемма из статьи [13], согласно которой для верификации функциональной зависимости можно сравнивать мощность PLI, полученных путем пересечения партиций (Partition Refinement).

2.3.7. Выбор стратегии алгоритма

Как было сказано ранее, алгоритм является гибридным. Выбор стратегии происходит после изначальной выборки. Для этого по формуле $\frac{differencesets}{recordpairs}$, где difference sets — это число полученных на стадии изначальной выборки наборов расхождений, a record pairs — это количество рассмотренных пар, вычисляется некоторая граница. Авторы заявляют, что взятие в качестве границы числа 0.5 является оптимальным решением. Таким образом, если полученный гиперграф окажется достаточно большим (граница > 0.5), то запускается $\text{FDHits}_{\text{sep}}$, иначе $\text{FDHits}_{\text{joint}}$.

2.3.8. Параллельность

В статье [4] авторы упоминают возможность распараллеливания алгоритма лишь в разделе с экспериментами. Как было написано в разделе с описанием поиска минимальных покрытий, вариация алгоритма $\text{FDHits}_{\text{sep}}$ запускает алгоритм MMCS для подгиперграфа $\mathcal{H}_A$ для каждого атрибута A. Это процесс авторы предлагают выполнять параллельно.

3. Эксперимент

3.1. Условия эксперимента

Авторы статьи указали, что эксперименты проводились на сервере Dell® R620 со следующими техническими характеристиками:

1. Два процессора Xeon E5-2650
2. 256 гигабайт оперативной памяти DDR-3, работающей на частоте 1600 мегагерц
3. Операционная система: Ubuntu 20.04.4 LTS

Алгоритм FDHits был реализован на языке программирования Rust. Сравнивался он с алгоритмами HyFD, FDEP и TANE, для которых была выбрана реализация на языке Java из проекта Metanome[2]. Для минимальной разницы между Rust и Java авторы запускали алгоритмы с серверным флагом и использовали значение времени работы, полученное самим алгоритмом во время его работы, чтобы исключить время, требуемое для запуска виртуальной машины Java. Все алгоритмы были запущены с использованием семантики “NULL-equals-NULL”.

3.2. Выбор параметров алгоритма

Алгоритм FDHits принимает только один параметр ε на вход, который определяет количество выборок на стадии выборки. Чем больше ε, тем больше времени у алгоритма уходит на стадию выборки. На графике, изображённом на рисунке 3 показана зависимость времени выполнения $\text{FDHits}_{\text{sep}}$ и $\text{FDHits}_{\text{joint}}$ в зависимости от фактора выборки ε. Минимальное время работы достигается при $\varepsilon = 0.3$.

3.3. Результаты эксперимента

Таблица, изображённая на рисунке 4 представляет результаты замеров производительности, проведённых авторами статьи [4]. Для каж-

[2] https://github.com/HPI-Information-Systems/Metanome

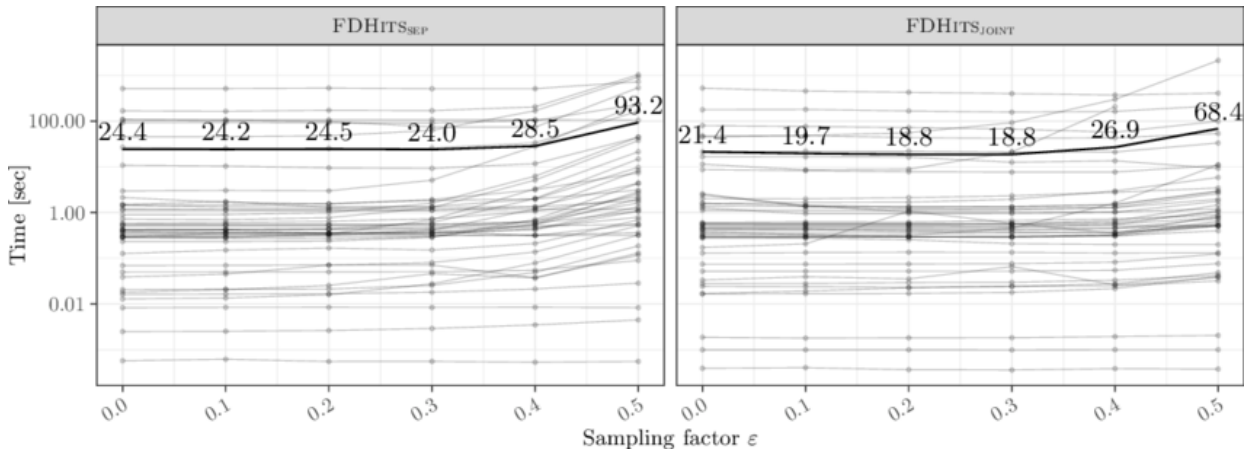

Рис. 3: Влияние фактора выборки ε на время работы (изображение взято из работы [4])

дого набора данных каждый алгоритм был запущен пять раз и было взято среднее значение полученных результатов. Сравнивая варианты алгоритма, $\text{FDHits}_{\text{joint}}$ быстрее, чем $\text{FDHits}_{\text{sep}}$ практически на всех представленных наборах данных. Авторы утверждают, что такая разница достигается на этапе пересечения позиционных индексов списка (PLIs).

Количество потребляемой памяти примерно равно у обоих вариаций алгоритма и значительно меньше, чем у HyFD и других алгоритмов.

Сравнивая FDHits с алгоритмами TANE, FDEP и HyFD, видно, что он не только превосходит их в производительности практически на всех наборах данных, но и в принципе способен обработать их всех, в то время как другие алгоритмы не справляются с большими наборами данных.

Также авторы провели сравнение сравнение производительности параллельных версий алгоритма FDHits и HyFD. Оно представлено на Рис. 5. $\text{FDHits}_{\text{parallel}}$ отработал быстрее, чем $\text{HyFD}_{\text{parallel}}$. А так же $\text{FDHits}_{\text{parallel}}$ смог обработать наборы данных ncvoter_allc и pdb-atom-site, для которых полный набор функциональных зависимостей был неизвестен.

Dataset	|R|	|r|	#FDs	TANE		FDEP		HyFD		FDHits$_{sep}$		FDHits$_{joint}$		Speedup
				Time [s]	Mem.	Time [s]	Mem.	Time [s]	Mem.	Time [s]	Mem.	Time [s]	Mem.	
Iris	5	147	4	0.144	91.5 MB	0.040	71.9 MB	0.052	61.8 MB	0.000	27.2 MB	**0.000**	27.3 MB	163.6
T-Bioc-Metadata	56	4	2575	0.161	99.5 MB	0.041	68.2 MB	0.064	62.5 MB	0.007	27.3 MB	**0.001**	27.3 MB	56.6
Echocardiogram	13	132	527	0.209	100 MB	0.069	80.4 MB	0.095	68.2 MB	0.003	27.3 MB	**0.002**	27.3 MB	55.7
T-Bioc-Measurementsorfacts	24	3.11 k	449	0.488	198 MB	3.655	566 MB	0.267	128 MB	0.023	27.3 MB	**0.012**	27.3 MB	21.5
T-Bioc-Specimenunit-Mark	12	8.98 k	84	0.565	191 MB	16.901	642 MB	0.360	147 MB	0.024	27.3 MB	**0.021**	27.3 MB	17.0
Hepatitis	20	155	8250	6.145	859 MB	0.295	121 MB	0.384	235 MB	**0.017**	27.3 MB	0.026	27.3 MB	22.9
Nursery	9	13 k	1	1.452	426 MB	26.578	640 MB	0.357	124 MB	0.036	27.3 MB	**0.032**	27.3 MB	11.1
Sg-Taxon-Name	3	106 k	2	0.577	258 MB	738.985	2.36 GB	0.592	309 MB	**0.050**	27.3 MB	**0.050**	27.3 MB	11.8
T-Bioc-Multimediaobject	15	18.8 k	133	0.941	338 MB	135.130	694 MB	0.628	255 MB	0.077	27.2 MB	**0.070**	27.3 MB	8.9
Chess	7	28.1 k	1	0.999	334 MB	84.967	660 MB	0.366	158 MB	**0.054**	27.3 MB	0.071	27.3 MB	5.1
Amalgam1-Denormalized	87	50	450,020	32.719	2.17 GB	1.605	335 MB	0.950	436 MB	0.982	27.3 MB	**0.172**	27.2 MB	5.5
Spstock	7	122 k	56	1.486	669 MB	1871.975	2.5 GB	1.405	607 MB	0.299	27.3 MB	**0.221**	27.3 MB	6.4
T-Bioc-Gath-Agent	18	72.7 k	186	4.662	1.57 GB	2604.715	885 MB	1.479	613 MB	0.271	34.9 MB	**0.244**	34.8 MB	6.1
Sg-Bioentry-Ref-Assoc	5	358 k	5	2.141	809 MB	TL	-	2.526	647 MB	0.328	46.1 MB	**0.269**	46 MB	9.4
T-Bioc-Unit	14	91.3 k	69	4.922	1.51 GB	3412.231	878 MB	1.478	612 MB	**0.284**	46 MB	**0.273**	46 MB	5.4
T-Bioc-Id-Highertaxon	3	563 k	1	1.903	726 MB	TL	-	2.366	627 MB	0.356	43.8 MB	**0.289**	45.8 MB	8.2
T-Bioc-Preparation	21	81.8 k	363	2.645	924 MB	3066.389	1.51 GB	1.523	628 MB	0.376	38.2 MB	**0.300**	38.2 MB	5.1
Hospital	15	115 k	83	54.188	21.6 GB	TL	-	2.759	652 MB	**0.339**	27.3 MB	**0.340**	27.8 MB	8.1
Sg-Bioentry	9	184 k	19	1.562	570 MB	TL	-	1.720	613 MB	**0.341**	68.2 MB	**0.343**	68.2 MB	5.0
T-Bioc-Gath-Namedareas	11	138 k	59	3.485	1.43 GB	TL	-	2.738	649 MB	0.441	46.3 MB	**0.384**	46.3 MB	7.1
Entytysrcgen	46	26.1 k	1454	-	ML	1448.888	2.07 GB	25.923	1.24 GB	0.620	27.3 MB	**0.401**	27.3 MB	64.7
T-Bioc-Gath-Sitecoordinates	25	91.3 k	467	6.226	1.98 GB	TL	-	2.072	648 MB	0.556	43.9 MB	**0.402**	43.9 MB	5.2
Sg-Biosequence	6	184 k	9	2.235	783 MB	TL	-	2.179	757 MB	**0.421**	114 MB	**0.428**	114 MB	5.1
Sg-Reference	6	129 k	13	1.576	496 MB	2237.348	2.38 GB	1.380	585 MB	**0.500**	105 MB	**0.501**	105 MB	2.8
Sg-Dbxref	4	618 k	4	1.746	823 MB	TL	-	1.842	795 MB	**0.530**	135 MB	**0.526**	135 MB	3.5
Sg-Seqfeature-Qual-Assoc	4	825 k	3	2.222	1.12 GB	TL	-	2.600	1.04 GB	**0.533**	97.8 MB	**0.548**	97.8 MB	4.7
Letter	17	18.7 k	61	234.215	55.8 GB	138.529	761 MB	2.315	593 MB	**0.133**	27.3 MB	0.618	27.3 MB	17.4
T-Bioc-Gath	35	91 k	925	17.464	5.65 GB	TL	-	4.413	1.15 GB	1.154	47.2 MB	**0.643**	47.1 MB	6.9
Horse	29	300	128,727	TL	-	6.659	394 MB	4.888	1.52 GB	**0.290**	27.3 MB	0.657	27.3 MB	16.8
T-Bioc-Id	38	91.8 k	972	43.765	13.6 GB	TL	-	4.934	1.21 GB	1.479	68 MB	**0.850**	68.1 MB	5.8
Sg-Seqfeature	6	1.02 M	7	3.383	1.38 GB	TL	-	4.636	1.36 GB	1.063	182 MB	**0.917**	182 MB	5.1
Sg-Bioentry-Dbxref-Assoc	3	1.85 M	2	4.026	1.55 GB	TL	-	6.333	1.41 GB	1.227	125 MB	**0.971**	138 MB	6.5
Sg-Bioentry-Qual-Assoc	4	1.82 M	2	8.268	2.01 GB	TL	-	7.460	1.32 GB	**1.090**	137 MB	1.145	167 MB	6.5
Sg-Location	8	1.02 M	11	4.918	1.53 GB	TL	-	4.955	1.64 GB	**1.302**	281 MB	**1.274**	281 MB	3.9
Plista	63	996	178,152	TL	-	27.742	1.79 GB	18.721	4.46 GB	**0.652**	27.2 MB	2.062	27.3 MB	28.7
Flight	109	1 k	982,631	-	ML	210.770	3.2 GB	44.781	13.3 GB	**1.732**	27.3 MB	2.508	27.3 MB	17.9
Tax	15	1 M	263	1006.375	79.9 GB	TL	-	58.741	2.54 GB	8.931	190 MB	**7.374**	218 MB	8.0
Ditag-Feature	13	3.96 M	58	1979.883	125 GB	TL	-	623.954	8.81 GB	25.270	1.19 GB	**19.294**	1.19 GB	32.3
Fd-Reduced-30	30	250 k	89,571	34.923	4.24 GB	TL	-	289.790	8.63 GB	104.308	139 MB	**19.889**	139 MB	14.6
Census	42	196 k	41,861	-	ML	TL	-	TL	-	**4.511**	95.7 MB	23.183	163 MB	>798
Struct-Sheet-Range	32	664 k	9150	-	ML	TL	-	407.227	20.9 GB	91.496	404 MB	**41.729**	404 MB	9.8
Pdbx-Poly-Seq-Scheme	13	17.3 M	68	TL	-	TL	-	262.810	21.4 GB	74.853	3.44 GB	**58.350**	3.44 GB	4.5
Musicbrainz-Denormalized	100	79.6 k	1,678,277	TL	-	TL	-	TL	-	**51.690**	85.1 MB	82.428	260 MB	>69
Ncvoter	19	8.06 M	822	TL	-	TL	-	2459.640	24.5 GB	**159.354**	3.56 GB	**156.788**	3.56 GB	15.7
Lineitem	16	6 M	3984	TL	-	TL	-	1965.967	25.1 GB	499.043	1.7 GB	**416.412**	1.72 GB	4.7

Рис. 4: Показатели производительности и потребляемой памяти алгоритма FDHits по сравнению с другими алгоритмами поиска функциональных зависимостей (изображение взято из работы [4])

Dataset	|R|	|r|	#FDs	HyFD$_{parallel}$ Time [s]	HyFD$_{parallel}$ Mem.	FDHits$_{parallel}$ Time [s]	FDHits$_{parallel}$ Mem.	Speedup
Musicbrainz-Denormalized	100	79.6 k	1,678,277	2423.902	40.2 GB	5.790	491 MB	418.6
Ditag-Feature	13	3.96 M	58	608.025	9.71 GB	12.861	1.78 GB	47.3
Struct-Sheet-Range	32	664 k	9150	65.869	44.6 GB	14.427	1.08 GB	4.6
Fd-Reduced-30	30	250 k	89,571	28.495	34.3 GB	21.635	299 MB	1.3
Pdbx-Poly-Seq-Scheme	13	17.3 M	68	214.921	24.3 GB	44.848	7.81 GB	4.8
Ncvoter	19	8.06 M	822	1516.107	51.7 GB	51.960	7.67 GB	29.2
Lineitem	16	6 M	3984	512.368	41.1 GB	98.327	6.57 GB	5.2
ncvoter_allc	94	7.50 M	1,197,767,282	-	-	11,971.066	204.45 GB	-
pdb-atom-site	31	219 M	9052	-	-	2214.245	253.71 GB	-

Рис. 5: Результаты производительности параллельных версий алгоритмов (изображение взято из работы [4])

3.4. Поведение алгоритма при увеличении набора данных

Рис. 6 показывает линейный рост времени работы всех трёх алгоритмов в зависимости от количества строк. Но в сравнении с HyFD, время работы обоих вариаций алгоритма FDHits возрастает медленнее.

Теоретически, время работы алгоритмов в зависимости от количества столбцов может расти экспоненциально. Это объясняется тем, что число результатов также растёт экспоненциально вместе с экспоненциальным ростом количества атрибутов, что видно из графика, изображённого на рисунке 7. Открытый вопрос — существует ли полиномиальный алгоритм поиска минимальных покрытий. Для MMCS, и следовательно для FDHits тоже, неизвестна верхняя граница по времени выполнения, которая является субэкспоненциальной.

3.5. Детальный анализ

Авторы представили график, изображённый на рисунке 8, показывающий, почему при выборе варианта алгоритма была выбрана именно такая граница. График показывает оставшийся потенциал экономии времени при разных границах для выбора версии алгоритма. Минимальная оставшаяся экономия достигается при выборе границы 0.5.

Для исследования влияния размера графа, авторы отключили ми-

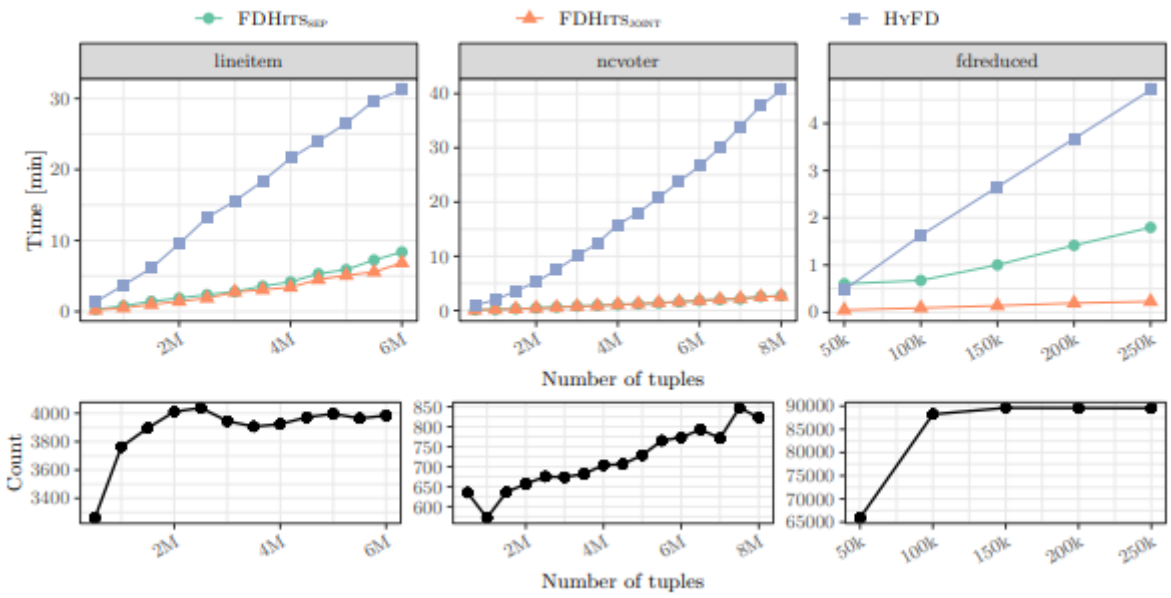

Рис. 6: Результаты производительности алгоритмов и количества полученных функциональных зависимостей в зависимости от количества строк в наборе данных (изображение взято из работы [4])

нимизацию в $\text{FDHits}_{\text{sep}}$. Самая большая разница проявилась на наборах данных *Census* и *Musicbrainz-Denormalized*. Как итог, время работы алгоритма увеличилось вдвое для обоих версий.

Для лучшего понимания, в каких условия $\text{FDHits}_{\text{joint}}$ работает быстрее, авторы рассмотрели размеры деревьев, количество операций с PLI и другие показатели для того, чтобы посмотреть, как они влияют на время работы. Для большинства наборов данных, размер дерева $\text{FDHits}_{\text{joint}}$ получился меньше, чем сумма размеров в $\text{FDHits}_{\text{sep}}$. Это обуславливается тем, что $\text{FDHits}_{\text{joint}}$ рассматривает все зависимые атрибуты одновременно. Это позволяет сэкономить время работы на пересечении PLI, потому что одинаковые операции используются для верификации многих функциональных зависимостей.

Несмотря на то, что рассчитываются только необходимые PLI, пересечение всё ещё занимает довольно большую часть времени работы, что видно из графика, изображённого на рисунке 9. Авторы также полагают, что предел по оптимизации алгоритма уже достигнут, так как большую часть времени занимает чтение набора данных и вывод результатов и дальнейшие ускорения могут быть достигнуты только на

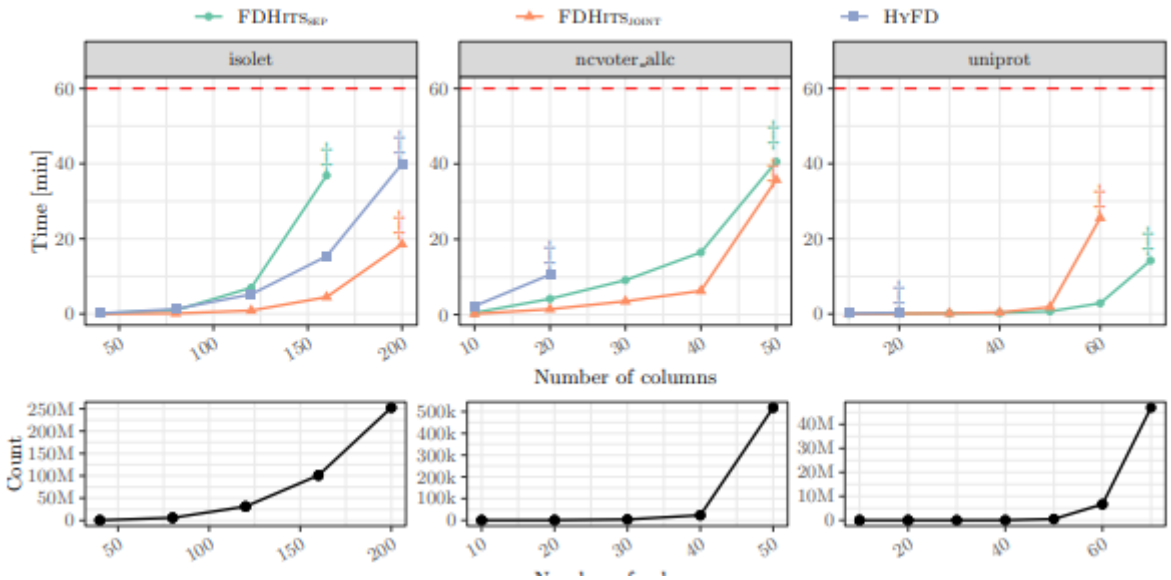

Рис. 7: Результаты производительности алгоритмов и количество полученных функциональных зависимостей в зависимости от количества атрибутов в наборе данных (изображение взято из работы [4])

этапе верификации.

Также авторы отмечают, что на этапе верификации крайне важен процесс удаления не релевантных кластеров. Они провели эксперимент, в котором убрали данную оптимизацию. Это повлекло увеличение времени работы алгоритма, зависящее от набора данных. К примеру на наборе *Census* время работы $\text{FDHits}_{\text{sep}}$ и $\text{FDHits}_{\text{joint}}$ составило 65 и 72 секунд соответственно. Это замедление в 15 и в 3 раза. Ещё большая разница наблюдается на наборе *Musicbrainz-Denormalized* — время работы $\text{FDHits}_{\text{sep}}$ возросло с 52 секунд до 38 минут (в 44 раза), а $\text{FDHits}_{\text{joint}}$ с 82 секунд до 30 минут (в 22 раза). На других наборах данных так же прослеживается замедление в 3–5 раз.

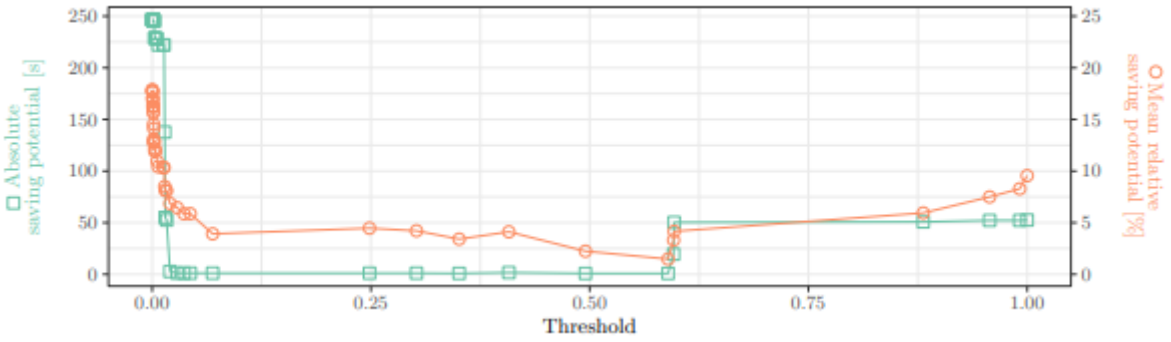

Рис. 8: Оставшийся потенциал экономии времени работы в зависимости от выбранной границы (взято из работы [4])

4. Заключение

В данной работе был произведён обзор алгоритма FDHits для поиска функциональных зависимостей с целью его дальнейшей интеграции в платформу Desbordante. Были выполнены следующие задачи:

- Проведён обзор предметной области
- Выполнен обзор алгоритма FDHits
- Выполнен обзор замеров производительности алгоритма, предложенных авторами статьи, а также проведён их анализ.

Дальнейшей целью является интеграция алгоритма в платформу Desbordante. Также, так как реализация алгоритма FDHits на языке Rust сравнивалась с реализациями других алгоритмов на языке Java, возникают вопросы к достоверности результатов замеров, полученных авторами. Данная проблема может быть решена реализацией алгоритма FDHits на языке C++ и сравнением производительности этой реализации с уже реализованными на этом же языке алгоритмами в платформе Desbordante.

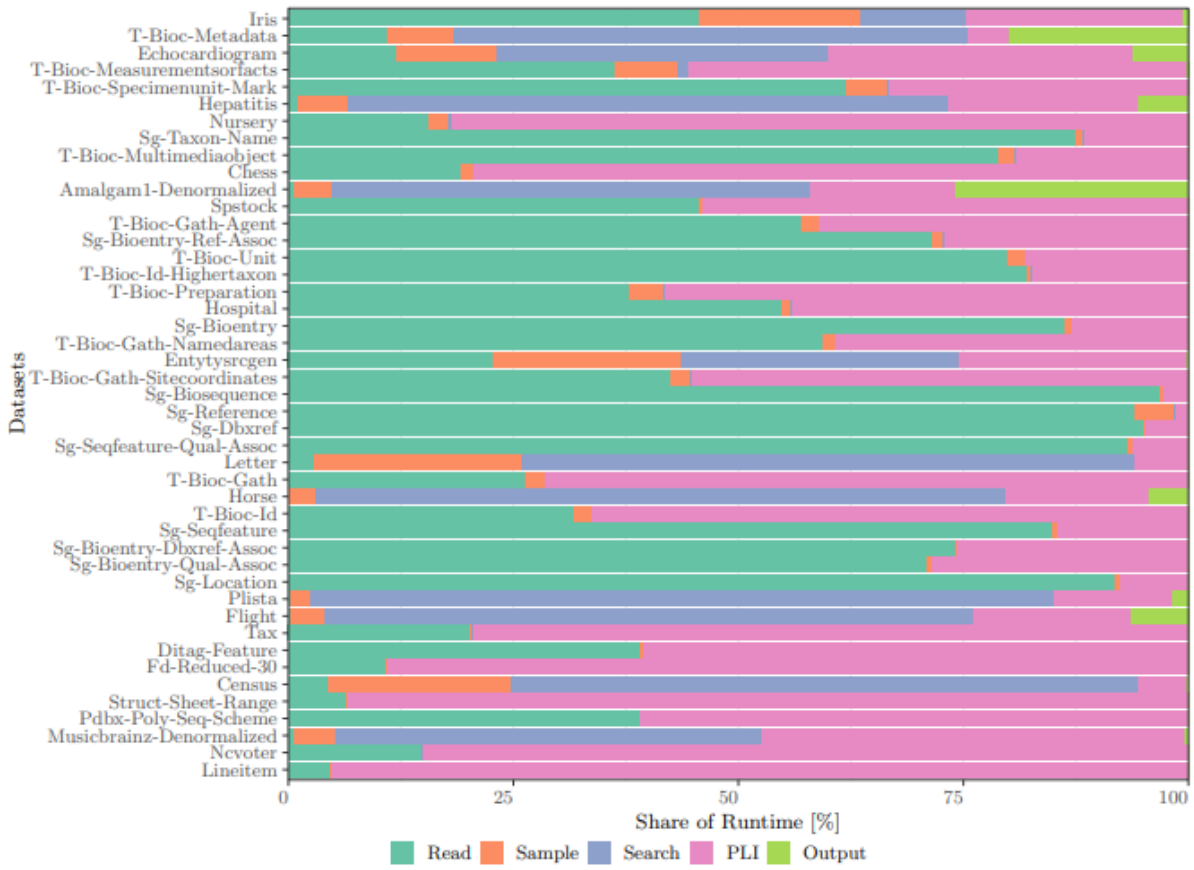

Рис. 9: Время работы каждого отдельного компонента алгоритма FDHits (Изображение взято из работы [4])

Список литературы

[1] Abedjan Ziawasch, Golab Lukasz, Naumann Felix. Profiling relational data: a survey // The VLDB Journal. — 2015. — Aug.. — Vol. 24, no. 4. — P. 557–581. — URL: https://doi.org/10.1007/s00778-015-0389-y.

[2] Data Profiling / Ziawasch Abedjan, Lukasz Golab, Felix Naumann, Thorsten Papenbrock // Synthesis Lectures on Data Management. — 2018. — 11. — Vol. 10. — P. 1–154.

[3] Chernishev George, Polyntsov Michael, Chizhov Anton et al. Desbordante: from benchmarking suite to high-performance science-intensive data profiler (preprint). — 2023. — 2301.05965.

[4] Discovering Functional Dependencies through Hitting Set Enumera-

tion / Tobias Bleifuß, Thorsten Papenbrock, Thomas Bläsius et al. // Proc. ACM Manag. Data. — 2024. — Mar.. — Vol. 2, no. 1. — 24 p. — URL: https://doi.org/10.1145/3639298.

[5] DynFD: Functional Dependency Discovery in Dynamic Datasets. / Philipp Schirmer, Thorsten Papenbrock, Sebastian Kruse et al. // EDBT. — 2019. — P. 253–264.

[6] Functional dependency discovery: an experimental evaluation of seven algorithms / Thorsten Papenbrock, Jens Ehrlich, Jannik Marten et al. // Proc. VLDB Endow. — 2015. — Jun.. — Vol. 8, no. 10. — P. 1082–1093. — URL: https://doi.org/10.14778/2794367.2794377.

[7] Hitting set enumeration with partial information for unique column combination discovery / Johann Birnick, Thomas Bläsius, Tobias Friedrich et al. // Proc. VLDB Endow. — 2020. — Jul.. — Vol. 13, no. 12. — P. 2270–2283. — URL: https://doi.org/10.14778/3407790.3407824.

[8] Ilyas Ihab F., Chu Xu. Data Cleaning. — New York, NY, USA : Association for Computing Machinery, 2019. — ISBN: 9781450371520.

[9] Kossmann Jan, Papenbrock Thorsten, Naumann Felix. Data dependencies for query optimization: a survey // The VLDB Journal. — 2021. — Jun.. — Vol. 31, no. 1. — P. 1–22. — URL: https://doi.org/10.1007/s00778-021-00676-3.

[10] Lehmberg Oliver, Bizer Christian. Stitching web tables for improving matching quality // Proc. VLDB Endow. — 2017. — Aug.. — Vol. 10, no. 11. — P. 1502–1513. — URL: https://doi.org/10.14778/3137628.3137657.

[11] Murakami Keisuke, Uno Takeaki. Efficient algorithms for dualizing large-scale hypergraphs // Discrete Appl. Math. — 2014. — Jun.. — Vol. 170. — P. 83–94. — URL: https://doi.org/10.1016/j.dam.2014.01.012.

[12] Papenbrock Thorsten, Naumann Felix. A Hybrid Approach to Functional Dependency Discovery // Proceedings of the 2016 International Conference on Management of Data. — SIGMOD '16. — New York, NY, USA : Association for Computing Machinery, 2016. — P. 821–833. — URL: https://doi.org/10.1145/2882903.2915203.

[13] Tane: An Efficient Algorithm for Discovering Functional and Approximate Dependencies / Yká Huhtala, Juha Kärkkäinen, Pasi Porkka, Hannu Toivonen // The Computer Journal. — 1999. — Vol. 42, no. 2. — P. 100–111.

[14] Towards Automatic Data Format Transformations: Data Wrangling at Scale / Alex Bogatu, Norman Paton, Alvaro Fernandes, Martin Koehler // The Computer Journal. — 2019. — 07. — Vol. 62. — P. 1044–1060.

[15] Интеграция алгоритма поиска UCC в рамках платформы Desbordante : Rep. / Санкт-Петербургский государственный университет ; Executor: Михаил Синельников : 2024. — https://github.com/Desbordante/desbordante-core/blob/main/docs/papers/HPIValid%20integration%20-%20Michael%20Sinelnikov%20-%202024%20spring.pdf.