# НОВІ ЗАСОБИ КІБЕРНЕТИКИ, ІНФОРМАТИКИ, ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ ТА СИСТЕМНОГО АНАЛІЗУ

**Д.А. РАЧКОВСКИЙ**

УДК 004.22+004.93'11

## ИНДЕКСНЫЕ СТРУКТУРЫ ДЛЯ БЫСТРОГО ПОИСКА СХОДНЫХ СИМВОЛЬНЫХ СТРОК

**Аннотация.** Дан обзор индексных структур для быстрого поиска по сходству объектов, представленных символьными строками. Рассмотрены индексные структуры как для точного, так и для приближенного поиска по расстоянию редактирования. Представлены индексные структуры на основе обратного индексирования, сохраняющего сходство хэширования, древовидных структур. Изложены идеи известных и предложенных в последнее время алгоритмов.

**Ключевые слова:** поиск по сходству, расстояние редактирования, ближайший сосед, ближний сосед, индексные структуры, обратное индексирование, *n*-граммы, локально-чувствительное хэширование, древовидные структуры.

### ВВЕДЕНИЕ

Поиск объектов базы, сходных с объектом-запросом по некоторой мере расстояния/сходства, называют поиском по сходству. Объекты-запросы могут не принадлежать базе. Меру расстояния/сходства между представлениями объектов, по которой выполняется поиск, будем считать заданной.

Выполнение запросов поиска по сходству (типы запросов см. в подразд. 1.2, 1.3) линейным (или последовательным) поиском заключается в вычислении расстояний/сходств объекта-запроса до всех объектов базы и возвращении объектов, удовлетворяющих условиям запроса. Поэтому сложность (время) выполнения запроса пропорциональна $N$ и времени вычисления расстояния/сходства между двумя объектами. Такой поиск часто оказывается недопустимо медленным для больших баз и объектов со сложными многокомпонентными представлениями, особенно для вычислительно сложной меры расстояния/сходства.

Один подход к ускорению линейного поиска по сходству состоит в том, чтобы быстро (хотя и приближенно) оценить величины всех расстояний/сходств. Другой подход заключается в построении по базе такой структуры данных (индексной структуры, ИС), использование которой при выполнении запроса поиска по сходству позволило бы сократить количество вычислений сходства объекта-запроса с другими объектами по сравнению с линейным поиском (т.е. обеспечить поиск, сублинейный относительно $N$). Отметим, что алгоритмы обоих подходов зачастую ускоряют поиск ценой получения результатов, не полностью совпадающих с результатами (точного) линейного поиска по сходству (т.е. ценой превращения поиска по сходству из точного в приближенный).

Обзоры быстрой оценки сходства объектов по векторам, в которые они преобразуются, приведены в [1, 2]. Обзор ИС для быстрого поиска по сходству объектов, для которых известны только значения сходств или расстояний до других объектов,

представлен в [3]. ИС для векторов с компонентами — бинарными числами — рассмотрены в [4], а с компонентами — вещественными числами — в [5, 6].

В настоящей работе приведен обзор ИС для поиска по сходству строк, т.е. последовательностей символов из некоторого алфавита. В качестве меры несходства применяется расстояние редактирования строк (подразд. 1.1). В отличие от [7–9] в данной статье представлены недавно появившиеся ИС для поиска строк, а также ИС с теоретическими гарантиями скорости поиска и ошибки результатов по сравнению с точным поиском. Основные понятия рассмотрены в разд. 1. Структура обзора приведена в подразд. 1.8.

## 1. ОСНОВНЫЕ ПОНЯТИЯ

**1.1. Расстояния между строками и векторами.** Для оценки сходства между объектами часто используют расстояние, т.е. несходство. Большим значениям сходства соответствуют малые значения расстояния. Многие расстояния являются метриками, т.е. подчиняются метрическим аксиомам, таким как неравенство треугольника и др. [3].

Широко распространено расстояние редактирования между строками (обозначим его $\mathrm{dist}_{\mathrm{edit}}$), которое определяется как минимальная суммарная стоимость последовательности элементарных операций редактирования символов строк, преобразующей одну строку в другую. Элементарными являются операции удаления, вставки и замены символа. Часто применяют вариант $\mathrm{dist}_{\mathrm{edit}}$ с одинаковой стоимостью элементарных операций (расстояние Левенштейна), тогда их суммарная стоимость эквивалентна общему количеству операций редактирования.

К строкам одинаковой длины $D$ (которые можно рассматривать в качестве векторов) применимо расстояние Хэмминга ($\mathrm{dist}_{\mathrm{Ham}}$), вычисляемое как количество несовпадающих символов строк (компонентов векторов) с одинаковым порядковым номером. В настоящем обзоре также используется манхэттенское расстояние ($\mathrm{dist}_{\mathrm{Man}}$) между векторами $\mathbf{a}$, $\mathbf{b}$ размерности $D$ с компонентами $a_i$, $b_i$: $||\mathbf{a}-\mathbf{b}||_1 = \sum_{i=1,\,D} |a_i - b_i|$.

Сложность вычисления $\mathrm{dist}_{\mathrm{edit}}$ (с применением динамического программирования) между строками длины $D_1$, $D_2$ квадратичная, т.е. составляет $O(D_1 D_2)$, в отличие от линейной сложности $O(D)$ вычисления многих типов расстояний между векторами ($\mathrm{dist}_{\mathrm{Ham}}$, $\mathrm{dist}_{\mathrm{Man}}$ и др.). Более того, в [10] показано, что если справедлива strong exponential time hypothesis (SETH), то $\mathrm{dist}_{\mathrm{edit}}$ невозможно вычислить за время $O(D^{2-\gamma})$ для любого фиксированного $\gamma > 0$ (т.е. за «существенно субквадратичное» время). Поэтому для строк с $\mathrm{dist}_{\mathrm{edit}}$ особенно актуально ускорение поиска по сходству за счет обеспечения сублинейности поиска с помощью ИС.

**1.2. Запросы точного поиска по сходству.** Диапазонный запрос (обозначим его rNN) возвращает объекты базы, расстояния которых от объекта-запроса (по мере расстояния, заданной в запросе) не превышают радиуса запроса $r$. При некоторых $r$ результатом диапазонного запроса может быть пустое множество объектов или все объекты конкретной базы (количество осс объектов базы, которые являются ответом на запрос, соответственно равно 0 или $N$). В последнем случае ускорение выполнения запроса rNN по сравнению с линейным поиском невозможно в принципе. Любой объект, являющийся ответом на диапазонный запрос, называют $r$-ближним соседом. Запрос rNN1 возвращает $r$-ближнего соседа, если он есть в базе.

Запрос ближайшего соседа (обозначим его NN) возвращает объект базы, ближайший к объекту-запросу. Обозначим kNN запрос $k$ ближайших соседей. Запрос kNN (как точного, так и приближенного поиска, см. подразд. 1.3) можно выполнить последовательностью запросов rNN1 с различными радиусами, что требует построения ИС для этих радиусов [11]. Чтобы избежать увеличения за-

трат памяти, для поиска ближайшего соседа часто используют другие ИС без теоретических гарантий времени поиска.

Результаты выполнения запроса all-pair similarity search (APSS), т.е. поиск всех пар объектов базы с $\text{dist} \le r$ (также называемый задачей similarity join), можно получить, используя в качестве объектов-запросов для rNN все объекты базы.

**1.3. Приближенный поиск по сходству и его запросы.** К сожалению, быстрое выполнение запросов точного поиска не гарантируется и не наблюдается на практике. Анализ всех известных алгоритмов точного выполнения запроса NN с сублинейным от $N$ временем для объектов с векторными представлениями показывает, что затраты времени или памяти растут экспоненциально от размерности векторов $D$ [11]. Согласно предположению «проклятия размерности» [11] такая зависимость неизбежна при точном поиске по сходству для данных худшего случая (т.е. для объекта-запроса и объектов базы, которые дают наибольшее время выполнения запроса). В случае невекторных данных, у которых $D$ — внутренняя размерность [3], также наблюдается проклятие размерности.

Запросы приближенного поиска по сходству возвращают результаты, которые могут отличаться от результатов запросов точного поиска по сходству. Приближенный поиск по сходству востребован на практике, поскольку для многих применений достаточно получать приближенные результаты, но быстро.

Приведем распространенные типы запросов приближенного поиска по сходству [12, 11].

Обозначим $(c, \delta)$-rNN1 запрос вероятностного $c$-приближенного $r$-ближнего соседа (randomized $c$-approximate $r$-near neighbor). Он с вероятностью, не меньшей $1-\delta$, возвращает любой объект базы с расстоянием от объекта-запроса, не превышающим $cr$ ($c>1$), если существует объект базы с расстоянием от объекта-запроса, не бóльшим $r$. Если такого объекта не существует, то возвращается объект с расстоянием, не превышающим $cr$, или ответ «нет».

Запрос $c$-приближенного ближайшего соседа (approximate nearest neighbor, обозначим его $c$-NN) возвращает объект базы, расстояние которого до объекта-запроса не более чем в $c>1$ раз превышает расстояние до точного ближайшего соседа. Вероятностный вариант такого запроса обозначим $(c, \delta)$-NN.

**1.4. Меры качества поиска по сходству.** Для алгоритмов приближенного поиска с количественными гарантиями качества результатов рассматривают два типа отличий их результатов от результатов точного поиска. В случае детерминированных приближенных алгоритмов расстояние до найденных объектов не более чем на заданный множитель превышает расстояние до объектов, которые являются точным ответом на запрос. В случае рандомизированных алгоритмов (точных или приближенных) для поиска по сходству допускаются false negatives (т.е. алгоритм может не возвратить объекты, которые являются ответом на запрос). Вероятностные приближенные запросы (см. подразд. 1.3) могут иметь оба отличия. Рандомизированные алгоритмы Las Vegas и Monte Carlo рассмотрены в обзоре [5].

На практике пользователей часто интересуют не асимптотическая вычислительная сложность алгоритмов (или ИС) и их теоретические гарантии качества, а время выполнения запросов и качество результатов в экспериментах на конкретных базах. Для точного поиска лучшей является ИС, обеспечивающая наименьшие затраты времени. Для приближенного поиска выбор параметров ИС обычно позволяет достичь некоторого компромисса между качеством результатов поиска (степенью их отличия от результатов точного поиска) и временем поиска. Поэтому ИС для приближенного поиска сравнивают по (усредненному) времени выполнения запроса при фиксированном качестве поиска или по значению меры качества поиска при одинаковом времени поиска. Важной характеристикой также являются затраты памяти ИС (затраты, квадратичные от $N$, неприемлемы).

Часто применяют меры качества выполнения конкретного запроса, известные (см. ссылки в [3–5]) как полнота (recall), равная $n_1 / n_2$, и точность (precision), равная $n_1 / n_3$, где $n_1$ — количество возвращенных объектов, совпавших с релевантными запросу объектами в базе, $n_2$ — количество релевантных запросу объектов в базе, $n_3$ — количество возвращенных объектов базы. Для запроса kNN $n_2 = n_3 = k$, поэтому точность равна полноте.

В качестве релевантных объектов для приближенного поиска можно использовать объекты, возвращаемые запросами точного поиска. Результаты выполнения запросов точного или приближенного поиска можно также сравнивать с результатами, указанными экспертом в качестве эталонных.

При выполнении множества запросов полученные для индивидуальных запросов значения мер качества усредняют. Так, для запроса NN полноту (равную точности) измеряют как процент запросов, для которых был возвращен эталонный ближайший сосед [3–5].

**1.5. Стратегия фильтрации и уточнения.** Для ускорения поиска по базе строят ИС и затем используют ее при выполнении запросов. Во многих ИС при выполнении запроса применяют процедуры, которые можно считать вариантами двухэтапной стратегии фильтрации и уточнения F&R. На первом этапе осуществляется быстрый отбор объектов-кандидатов. Результаты первого этапа уточняются на втором этапе (обычно с использованием линейного поиска среди объектов-кандидатов по мере расстояния/сходства, заданной в запросе). Стратегию F&R иногда применяют многократно при выполнении одного запроса для различных подмножеств объектов базы и/или типов фильтров.

Отметим, что для запроса rNN второй этап устраняет false positives (объекты-кандидаты, не являющиеся ответом на запрос). Если суммарное количество объектов-кандидатов меньше $N$ и их отбор выполняется достаточно быстро, можно получить ускорение выполнения запроса относительно линейного поиска. Для точного поиска при выполнении запроса кандидаты выбираются так, что они гарантированно содержат ответ на запрос.

**1.6. Извлечение признаков и вложения в векторные пространства.** На этапе фильтрации эффективное исключение (или отбор) кандидатов часто выполняют на основе признаков, выделенных из исходных представлений. Так, представления объектов делят на пересекающиеся или непересекающиеся части, а признаком является индикатор наличия и отсутствия в данной части определенных элементов исходного представления.

Наличие у двух объектов некоторого количества совпадающих признаков связывают с величиной некоторой меры сходства между исходными представлениями этих объектов, по которой выполняется поиск по сходству. При выполнении запроса быстро находят объекты-кандидаты из базы, содержащие требуемое количество признаков, совпадающих с признаками объекта-запроса. Для этого используют ИС на основе обратного индексирования (подразд. 1.7).

Извлечение признаков из исходных представлений объектов можно рассматривать как формирование векторов с компонентами, соответствующими признакам. Меры расстояния/сходства между полученными векторами признаков обычно вычисляются значительно проще, чем меры сходства исходных представлений. Поэтому применение векторов признаков позволяет в ряде случаев быстро получать границы исходных значений расстояний и выбирать по ним объекты-кандидаты на первом этапе фильтрации в стратегии F&R. Другой связанный подход — вложение в векторное пространство, т.е. преобразование исходных представлений объектов в такие векторы, расстояние между которыми аппроксимирует исходное с некоторым искажением [1, 2]. Для поиска по сходству востребованы забывчивые вложения, т.е. позволяющие преобразовывать объекты-запросы не из базы без изменения представлений других объектов.

Для объектов с заданным типом представления и мерой сходства применяют специализации этого подхода — в данном обзоре для исходных строк с $\mathrm{dist}_{\mathrm{edit}}$ и для получаемых вещественных и бинарных векторов в основном с $\mathrm{dist}_{\mathrm{Man}}$. Для строк в качестве признаков используют, например $n$-граммы, т.е. последовательности $n$ смежных символов, а в качестве компонентов вектора вложения — частоту встречаемости $n$-грамм.

**1.7. Индексные структуры и обратное индексирование.** Многие ИС при конструировании разбивают множество объектов базы на подмножества так, чтобы при выполнении запроса рассматривать не все, а только часть подмножеств, и таким образом ускоряют время выполнения запроса относительно линейного поиска [3–6]. ИС на основе хэширования, деревьев, графов соседства и др. рассмотрены в [3–6] в основном для векторных представлений объектов [4–6], а также для объектов, у которых известны величины расстояний между ними, но не их представления [6].

Для быстрой фильтрации на основе совпадения признаков объектов (см. подразд. 1.6) применяют ИС для поиска по мерам сходства множеств [13, 4, 14]. Распространены ИС на основе обратных списков и обратного индексирования [4]. В них для каждого признака запоминают объекты базы, в которых имеется этот признак, и частоту его встречаемости в каждом объекте. При выполнении запроса обрабатывают только признаки, имеющиеся в объекте-запросе. Если количество посещенных объектов меньше $N$, получают сублинейное время поиска (лишь эти объекты подлежат ранжированию). Поэтому обратный индекс особенно эффективен для векторов с малым количеством ненулевых компонентов при малом количестве посещенных объектов. Однако в худшем случае сублинейное время поиска не гарантируется.

Часто в ИС последовательно используют несколько фильтров с различной вычислительной сложностью и эффективностью исключения, что сокращает количество кандидатов.

Как упоминалось в подразд. 1.3, ИС для точного поиска с гарантиями быстрого поиска в худшем случае требуют очень больших затрат памяти. Поэтому их можно практически реализовать только для частных случаев объектов с малым количеством элементов представления и/или с малым радиусом запроса. Практические ИС для точного поиска используют эвристики и не гарантируют сублинейного времени поиска в худшем случае.

Для приближенного поиска строк по $\mathrm{dist}_{\mathrm{edit}}$ с сублинейным временем в худшем случае предложены ИС на основе вложений (подразд. 3.2, 5.1). Они используют вложения строк в векторные пространства с $\mathrm{dist}_{\mathrm{Man}}$, $\mathrm{dist}_{\mathrm{Ham}}$ и метриками произведения (product metrics). Затем применяют ИС, специализированные для поиска по полученным векторам. Результат поиска объектов-векторов дает соответствующие им объекты-строки в качестве кандидатов. Это в основном «теоретические» ИС (т.е. без известных практических реализаций), но некоторые из них являются «практическими» (т.е. их можно реализовать, например, ИС на основе LSH из подразд. 3.2).

**1.8. Структура обзора.** Востребованной в применениях (и быстрой) разновидностью поиска по $\mathrm{dist}_{\mathrm{edit}}$ является выполнение точного запроса rNN с радиусом $r=1$ (разд. 2). ИС с теоретическими гарантиями сублинейного поиска для данных худшего случая (подразд. 2.1) используют идеи, аналогичные теоретическим ИС для поиска по $\mathrm{dist}_{\mathrm{Ham}}$ с $r=1$ [4]. Практические ИС (подразд. 2.2) также основаны на этих идеях, но дополнительно применяют эвристики, что приводит к потере теоретических гарантий.

ИС для выполнения запроса rNN с $r>1$ рассмотрены в разд. 3 и 4. ИС для точного поиска с $r>1$ и с теоретическими гарантиями для данных худшего случая, основанные на k-errata tree (подразд. 3.1), имеют экспоненциальные от $r$ за-

траты памяти и/или время выполнения запроса. ИС на основе локально-чувствительного хэширования с гарантией сублинейного от $N$ времени (подразд. 3.2) используют вложения строк в векторы с различными расстояниями и применяют ИС LSH для быстрого поиска векторов.

Практические ИС для выполнения точного запроса rNN (разд. 4) включают эвристики и не обеспечивают гарантий сублинейного времени поиска. Они применяют подходы для точного поиска по мерам сходства множеств. Элементами множества (признаками, извлеченными из строки) часто являются $n$-граммы.

В разд. 5 рассмотрены ИС различных типов для выполнения запросов ближайшего соседа по $\mathrm{dist}_{\mathrm{edit}}$: запроса $(c, \delta)$-NN с теоретическими гарантиями, но без практической реализации, на основе вложения строк в векторы с различными расстояниями и практические ИС для выполнения запросов kNN, но без теоретических гарантий.

## 2. ТОЧНЫЙ ПОИСК С ЕДИНИЧНЫМ РАДИУСОМ ЗАПРОСА

Поиск по $\mathrm{dist}_{\mathrm{edit}}$ строк с единичным радиусом запроса применяется [15, 16] для проверки и коррекции правописания (например, поисковых запросов), проверки паролей на надежность и как этап решения задачи поиска с бóльшим радиусом (разд. 3, 4) и др.

**2.1. Поиск с теоретическими гарантиями.** Некоторые алгоритмы для точного поиска бинарных векторов по $\mathrm{dist}_{\mathrm{Ham}}$ из [4, подразд. 2.1.1] можно использовать для более общей задачи поиска строк с небинарными символами алфавита по $\mathrm{dist}_{\mathrm{edit}}$. Алгоритмы с теоретическими гарантиями, предложенные в [17–19], разработаны непосредственно для $\mathrm{dist}_{\mathrm{edit}}$, но могут решать задачу поиска и для бинарных векторов с $\mathrm{dist}_{\mathrm{Ham}}$.

ИС [17] на основе сочетания префиксных деревьев (подразд. 4.5), минимальных идеальных хэш-функций (т.е. без коллизий [20]) и сигнатур Карпа–Рабина [21] выполняет запрос rNN с $r=1$ за время $O(D+\mathrm{occ})$ в худшем случае, где $D$ — число символов в строке. Затраты памяти составляют $O(ND \log |\Sigma|)$ битов, где $|\Sigma|$ — размер алфавита символов.

В [18] предложены варианты ИС (включая рандомизированные) с уменьшенными за счет сжатия затратами памяти. Для строк из конечного алфавита постоянного размера получено [19] оптимальное время запроса $O(D/B+\mathrm{occ})$ при небольших дополнительных затратах памяти ($B$ — размер машинного слова в битах).

**2.2. Поиск без теоретических гарантий.** Практические решения для символьных строк и $\mathrm{dist}_{\mathrm{edit}}$ разработаны в [15, 22, 23, 16] (см. также ссылки к ним).

ИС [15] основана на фильтре Блума [24]. Для поиска слов (строк) с $r=1$ по $\mathrm{dist}_{\mathrm{edit}}$ при конструировании ИС каждое слово базы преобразуют в $2D+1$ строку, где $D+1$ строка содержит вставки спецсимвола в позициях от 0 до $D$, а $D$ строк — замену на него буквы в позиции от 0 до $D$. Из $N$ слов базы конструируют $N(2D+1)$ слов, хэшируют $k$ хэш-функциями и помечают в хэш-таблице соответствующие ячейки. Модифицированные строки запоминать не требуется. Чтобы по слову-запросу определить, имеется ли в базе слово, отличающееся от него одной буквой (или совпадающее), конструируют $2D+1$ модификацию слова-запроса и выполняют $2D+1$ запрос на совпадение. Для ускорения этой ИС генерируют хэш-значения так, чтобы они оказались на одной странице компьютерной памяти. Для уменьшения количества модификаций [22, 23] в ИС запоминают исходные строки базы, а также строки, полученные удалением каждого их символа (для каждой такой модификации запоминают указатель на исходную строку). Запрос выполняется поиском по совпадению для всех $D$ модификаций строки-запроса, полученных удалением одного символа. Обобщение на бóльшие радиусы запроса рассмотрено в подразд. 4.4.

В ИС [16] предложены изменения теоретической ИС [17], предназначенные для улучшения практических характеристик. Эвристики при этом не использовались, поскольку они могут значительно замедлить ИС для данных худшего случая. Вместо идеального хэширования применяют более эффективное линейное зондирование (при коллизии хэш-значений ищется ближайшая следующая свободная ячейка и хэш помещается в нее, при поиске ищется совпадение хэшей в последовательности ячеек или пустая ячейка, если объекта в таблице нет). Затраты памяти на хэш-таблицы минимизируют за счет исключения пустых ячеек. Префиксных деревьев и сигнатур Карпа–Рабина не используют, вместо этого выполняется непосредственное сравнение строк. Затраты памяти составляют $O(ND \log |\Sigma|)$ битов, время выполнения запроса $O(D\lceil(D \log |\Sigma|)/B\rceil)$ (в модели RAM с длиной слова $B = \Omega(\log(DN))$, т.е. со всеми операциями, выполняемыми за постоянное время). Эти характеристики получены для величины их математического ожидания по реализациям случайных чисел, используемых в алгоритме. Реализован также алгоритм для $r = 2$ с суперлинейными затратами памяти, который на практике в десятки раз более медленный, чем ИС для $r = 1$. Для этих значений $r$ ИС из [16] быстрее, чем ИС из [25] и FastSS (подразд. 4.4).

### 3. ПОИСК ДЛЯ НЕЕДИНИЧНОГО РАДИУСА ЗАПРОСА С ТЕОРЕТИЧЕСКИМИ ГАРАНТИЯМИ

Рассмотрим ИС с теоретическими гарантиями для выполнения точных диапазонных запросов (подразд. 3.1) и приближенных диапазонных запросов на основе LSH (подразд. 3.2). ИС для выполнения запросов поиска приближенного ближайшего соседа с теоретическими гарантиями приведена в подразд. 5.1.

**3.1. Точный поиск с теоретическими гарантиями.** Для выполнения запроса rNN с фиксированным радиусом запроса $r > 1$ по $\mathrm{dist}_{\mathrm{edit}}$ с гарантиями худшего случая используется ИС k-errata tree [26], предложенная для поиска по $\mathrm{dist}_{\mathrm{Ham}}$. При конструировании ИС применяется сжатое префиксное дерево (compressed prefix tree или trie) для строк базы и рекурсивно создаются поддеревья замены, вставки и удаления. При выполнении запроса исследуются все пути, пока не получают $r$ ошибок относительно строки-запроса (на каждом пути рассматриваются различные возможные комбинации операций редактирования). По сравнению с аналогичной ИС для $\mathrm{dist}_{\mathrm{Ham}}$ [26] в случае $\mathrm{dist}_{\mathrm{edit}}$ время выполнения запроса включает вместо occ слагаемое $\mathrm{occ} \times 3^r$ и увеличиваются соответствующие константы, что дает время $O(D + ((c \log N)^r / r) \log\log(DN) + \mathrm{occ} \times 3^r)$. Экспоненциальные от $r$ затраты памяти не позволяют применять эту ИС на практике для $r > 1$ [7]. ИС этого типа с линейными затратами памяти [27] также не используется на практике вследствие экспоненциальных зависимостей от $r$ времени выполнения запроса.

**3.2. Приближенный поиск на основе локально-чувствительного хэширования.** В локально-чувствительном хэшировании (LSH), в отличие от обычного, LSH-функции генерируют хэш-значения так, что вероятность их совпадения у сходных объектов выше, чем у несходных [12, 2, 5]. Объекты с одинаковым хэшем образуют корзину. В ИС на основе LSH при поступлении объекта-запроса генерируют его LSH-хэш, извлекают из соответствующей корзины объекты базы и используют их в качестве кандидатов для уточнения по исходному расстоянию. Хэш-значения сходных объектов, сгенерированные одной LSH-функцией, могут оказаться различными. Поэтому для повышения вероятности нахождения близких к запросу объектов-кандидатов объединяют списки кандидатов из корзин нескольких независимых LSH-функций. ИС на основе LSH — одни из немногих практически реализованных ИС для приближенного вероятностного поиска по сходству с гарантиями сублинейного времени в худшем случае.

ИС на основе LSH для приближенного поиска по $\mathrm{dist}_{\mathrm{edit}}$ впервые реализована в [28, 29]. Строки вкладывались в пространство $L_1$ (т.е. в векторы с $\mathrm{dist}_{\mathrm{Man}}$) извлечением $n$-грамм различной длины и формированием векторов частоты их встречаемости в строке (компонент вектора соответствует $n$-грамме). Затем применялось LSH-хэширование на основе случайного проецирования с элементами случайных матриц из устойчивого распределения Коши (см. [30, 31] и обзоры [2, 5]). Для выполнения запроса приближенного ближайшего соседа использовалась ИС LSH-forest [32, 33].

Для приближенного решения задачи APSS в EmbedJoin [34] применяют приближенное вложение CGK [35] строк с $\mathrm{dist}_{\mathrm{edit}}$ в пространство бинарных векторов с $\mathrm{dist}_{\mathrm{Ham}}$. На практике размерность вложения выбирают как утроенную среднюю длину строки в базе (более длинные строки укорачивают). Затем по сэмплированным бинарным векторам (SamplingLSH [2, 5]) строят ИС LSH из $L$ таблиц для поиска по $\mathrm{dist}_{\mathrm{Ham}}$. Применение нескольких ИС с различными реализациями вложения CGK улучшает результаты. Для строк-кандидатов выполняют фильтрацию по длине (подразд. 4.1), а для оставшихся кандидатов точно вычисляют $\mathrm{dist}_{\mathrm{edit}}$.

Биологическим базам генома и белков свойственно наличие сходных строк с длинным префиксом вставок в одной из них. При больших значениях порога $r$ такая структура баз приводит к тому, что бóльшая часть величины расстояния редактирования может быть обусловлена префиксами вставок. Это увеличивает количество false negatives в EmbedJoin. Для баз такого типа используют суффиксы исходных строк, начинающиеся с позиций, отстоящих от начала с некоторым шагом. Кроме того, чтобы уменьшить количество кандидатов, в их множество включают только строки базы с неоднократной коллизией LSH-хэшей с LSH-хэшами строки-запроса. Метод хорошо масштабируется с длиной строки и величиной порога на расстояния и по скорости значительно превосходит PassJoin [36], EDJoin [37], AdaptJoin [38], QChunk [39] (разд. 4), немного уступая им в точности. Для строк малой длины алгоритм не дает преимуществ.

## 4. ПРАКТИЧЕСКИЙ ТОЧНЫЙ ПОИСК С НЕЕДИНИЧНЫМ РАДИУСОМ ЗАПРОСА

Рассмотрим ИС для выполнения точного диапазонного запроса rNN с $r>1$. Эти ИС реализованы на практике, но используют эвристики и поэтому не гарантируют сублинейного времени выполнения запроса для данных худшего случая. Часто ИС конструируют для выполнения запроса с фиксированным значением $r>1$. Для поиска применяют стратегию F&R с комбинацией различных фильтров, аналогичных применяемым для поиска множеств или бинарных векторов [13, 4], но специализированных для строк.

В целях быстрой фильтрации из строк обычно извлекают признаки, такие как непересекающиеся или пересекающиеся $n$-граммы без учета или с учетом их положения в строке. Используют также другие признаки, которые можно охарактеризовать как глобальные, например длина строки. Для строк, близких по $\mathrm{dist}_{\mathrm{edit}}$, необходимо совпадение определенного количества признаков либо должны оказаться похожими глобальные признаки. Чтобы быстро найти такие строки-кандидаты (и исключить остальные), применяют обратное индексирование. Как и для бинарных векторов [4], для длинных строк используют разбиение на непересекающиеся части и строят ИС для частей; кандидатами являются строки, извлеченные из всех ИС для частей.

Для строк используют различные типы специализированной фильтрации: по количеству совпавших признаков, длине строк и положению признаков в строке (подразд. 4.1). Иногда удобно представлять признаки в виде векторов и применять фильтрацию на основе расстояний между векторами (подразд. 4.2). Затраты памяти на ИС уменьшают отбором подмножества извлеченных признаков (подразд. 4.3). Как и для $\mathrm{dist}_{\mathrm{Ham}}$, используют модификацию строк и поиск по совпа-

дению (подразд. 4.4). В целях экономии вычислений для строк с общими префиксами применяют префиксные деревья (подразд. 4.5). Сокращения количества признаков для фильтрации добиваются разбиением строк на непересекающиеся части (подразд. 4.6).

**4.1. Фильтрация строк по совпадению и положению *n*-грамм.** Для строк $a$, $b$, преобразованных в $n$-граммы, если $\mathrm{dist}_{\mathrm{edit}}(a, b) \le r$, то количество совпавших $n$-грамм больше или равно $T = \max(|a|, |b|) - n + 1 - rn$ [40]. Положение $n$-грамм не учитывается. Иногда строки наращивают конкатенацией $n-1$ «префиксного» и $n-1$ «суффиксного» спецсимвола (не из алфавита), тогда выражение для $T$ изменяется [41].

Таким образом, для строки-запроса $x$ условиям запроса rNN по $\mathrm{dist}_{\mathrm{edit}}$ удовлетворяют строки базы $y$, количество совпавших $n$-грамм в которых не меньше $|x| - n + 1 - rn$ (если не учитывается длина строк $y$). Здесь $|x| - n + 1$ — количество $n$-грамм в строке, $rn$ — количество $n$-грамм, измененных $r$ операциями редактирования. Такие строки становятся кандидатами, для которых проверяется точное выполнение условия $\mathrm{dist}_{\mathrm{edit}} \le r$. Для быстрого нахождения строк-кандидатов, имеющих превышающее порог количество $n$-грамм, общих со строкой-запросом, применяют обратное индексирование (см. подразд. 1.7).

Чтобы избежать работы как с очень часто, так и с очень редко встречающимися $n$-граммами, используют их различную длину [42, 43]. Такие $n$-граммы позволяют повысить эффективность исключения кандидатов. Отметим, что применение $n$-грамм различной длины для уменьшения искажения при вложении $\mathrm{dist}_{\mathrm{edit}}$ в $\mathrm{dist}_{\mathrm{Man}}$ рассмотрено в [28].

Фильтрация по совпадению $n$-грамм использует ответ на вопрос: «какое максимальное количество $n$-грамм портят $r$ операций редактирования?». Фильтрация за счет учета несовпадающих $n$-грамм в EdJoin [37] отвечает на вопрос: «какое минимальное количество операций редактирования может привести к наблюдаемым несовпадающим $n$-граммам?». На основе анализа положения несовпадающих $n$-грамм (являются ли они непересекающимися или частично пересекающимися) предложен жадный алгоритм, который дает нижнюю границу $\mathrm{dist}_{\mathrm{edit}}$ (как для всего множества несовпадающих $n$-грамм, так и для любого его подмножества). Эта граница может использоваться и на этапе фильтрации, и на этапе уточнения. Фильтр по содержанию несоответствующих $n$-грамм в EdJoin работает с расстояниями векторов $n$-грамм (подразд. 4.2) и применяется на этапе уточнения.

Фильтр по длине строки [41] учитывает, что при $\mathrm{dist}_{\mathrm{edit}} \le r$ разность длин строк не может превышать $r$. Позиционный фильтр [41] (фильтр по положению $n$-грамм) основан на том, что при $\mathrm{dist}_{\mathrm{edit}} \le r$ $n$-грамма в одной строке не может соответствовать $n$-грамме в другой строке, которая отстоит более чем на $r$ позиций. Здесь для строки генерируются и запоминаются $n$-граммы с их положением. Отметим, что применение различных фильтров до слияния обратных списков объектов из обратного индекса не всегда ускоряет получение кандидатов, так как может создавать большое количество коротких списков, работа с которыми неэффективна. Недостатком $n$-граммной фильтрации является необходимость выполнения условия $|a| \ge (r+1)n$, что не позволяет работать с короткими строками и большими радиусами запроса $r$, а также применять для сокращения множества объектов-кандидатов длинные, более селективные $n$-граммы.

**4.2. Преобразование строк в векторы и фильтрация по содержанию.** Пусть для каждой строки построен вектор частоты встречаемости символов (размерность векторов равна размеру алфавита символов). Нижней границей $\mathrm{dist}_{\mathrm{edit}}$ является частотное расстояние $\mathrm{dist}_{\mathrm{freq}}$ между векторами $\mathbf{a}$, $\mathbf{b}$ частоты встречаемости символов в строках [44, 7]: $\mathrm{dist}_{\mathrm{freq}}(\mathbf{a}, \mathbf{b}) = 1/2\,(|\,\mathrm{dist}_{\mathrm{Man}}(\mathbf{a}, \mathbf{0}) - \mathrm{dist}_{\mathrm{Man}}(\mathbf{b}, \mathbf{0})\,| + \mathrm{dist}_{\mathrm{Man}}(\mathbf{a}, \mathbf{b}))$.

Строку включают в список кандидатов, если $\mathrm{dist}_{\mathrm{freq}} \le r$, так как $\mathrm{dist}_{\mathrm{freq}} \le$ $\le \mathrm{dist}_{\mathrm{edit}}$. Строку исключают при $\mathrm{dist}_{\mathrm{freq}} > r$. Это выполняется, если, например, для любого символа разность частот превышает $r$. Другим условием исключения является $\mathrm{dist}_{\mathrm{Man}}(\mathbf{a}, \mathbf{b}) > 2r - ||a| - |b||$. Действительно, из $1/2\,(|\mathrm{dist}_{\mathrm{Man}}(\mathbf{a}, \mathbf{0}) - \mathrm{dist}_{\mathrm{Man}}(\mathbf{b}, \mathbf{0})| + \mathrm{dist}_{\mathrm{Man}}(\mathbf{a}, \mathbf{b})) > r$ следует $\mathrm{dist}_{\mathrm{Man}}(\mathbf{a}, \mathbf{b}) > 2r - |\mathrm{dist}_{\mathrm{Man}}(\mathbf{a}, \mathbf{0}) - \mathrm{dist}_{\mathrm{Man}}(\mathbf{b}, \mathbf{0})| = 2r - ||a| - |b||$. Очевидно $2r - ||a| - |b|| \le 2r$, т.е. можно использовать значение $2r$ для произвольных строк. Поэтому вместо запроса по $\mathrm{dist}_{\mathrm{edit}}$ можно выполнять запрос по $\mathrm{dist}_{\mathrm{freq}}$ с тем же $r$, или по $\mathrm{dist}_{\mathrm{Man}}$ с модифицированным порогом, в том числе с использованием ИС.

В фильтре по содержанию EdJoin [37] рассматривают окно на строках (в реализации применяют специальную схему дополнения строк символами). Для любого окна выполняется $\mathrm{dist}_{\mathrm{Man}} \le 2\,\mathrm{dist}_{\mathrm{edit}}$, где $\mathrm{dist}_{\mathrm{Man}}$ вычислено по векторам частот символов в окне, $\mathrm{dist}_{\mathrm{edit}}$ определяется между подстроками в окне и не превышает $\mathrm{dist}_{\mathrm{edit}}$ между полными строками. Это следует из того, что любая операция редактирования увеличивает величину $\mathrm{dist}_{\mathrm{Man}}$ не более чем на 2.

Так как для векторов, полученных бинаризацией исходных векторов частот, $\mathrm{dist}_{\mathrm{Ham}} \le \mathrm{dist}_{\mathrm{Man}}$, то для них $\mathrm{dist}_{\mathrm{Ham}} \le 2\mathrm{dist}_{\mathrm{edit}}$. Например, фильтр по содержанию EdJoin [37] формирует бинарный вектор (32 бита) $\mathbf{h}(a)$ строки $a$ так, что имеющемуся символу соответствует единичное значение. Условие $\mathrm{dist}_{\mathrm{edit}}(a, b) < r$ может выполняться только в случае, если $\mathrm{dist}_{\mathrm{Ham}}(\mathbf{h}(a), \mathbf{h}(b)) \le 2r$ [40, 7].

Рассмотрим векторы частот не отдельных символов, а $n$-грамм. Для строк, элементами которых являются $n$-граммы, выполняется $\mathrm{dist}_{\mathrm{freq}} \le \mathrm{dist}_{\mathrm{edit}}$. Согласно [40] количество $n$-грамм, которое модифицирует одна операция редактирования, не превышает $n$. Поэтому $\mathrm{dist}_{\mathrm{freq}} / n \le \mathrm{dist}_{\mathrm{edit}}$, где $\mathrm{dist}_{\mathrm{freq}}$ — частотное расстояние между векторами частот $n$-грамм строк, а $\mathrm{dist}_{\mathrm{edit}}$ — между строками символов. Соответственно для бинарных векторов $n$-грамм строк-кандидатов выполняется $\mathrm{dist}_{\mathrm{Ham}} \le 2rn$.

Для уменьшения размерности векторов применяют хэширование. Если уменьшить размер алфавита хэшированием символов, то $\mathrm{dist}_{\mathrm{edit}}$ между получившимися строками является нижней границей исходного $\mathrm{dist}_{\mathrm{edit}}$ [7]. Поэтому $\mathrm{dist}_{\mathrm{freq}}$ между векторами частот символов уменьшенной (за счет хэширования) размерности также является нижней границей исходного $\mathrm{dist}_{\mathrm{edit}}$. В [45] хэширование интерпретируется как разбиение символов алфавита на подмножества.

Применяют также хэширование $n$-грамм [46]. Перед извлечением $n$-грамм строки дополняют в начале и в конце $n-1$ спецсимволом. Хэш-функция преобразует $n$-грамму в целое число, которое является номером компонента бинарного вектора. При хэшировании также учитывается частота встречаемости $n$-граммы (добавлением числа в конец $n$-граммы как дополнительного символа). В случае коллизии значение бинарного компонента вектора модифицируют операцией OR или XOR. Бинарные векторы сравнивают по $\mathrm{dist}_{\mathrm{Ham}}$. Если $\mathrm{dist}_{\mathrm{edit}}(x, y) \le r$, то $\mathrm{dist}_{\mathrm{Ham}}(\mathbf{h}(x), \mathbf{h}(y)) \le 2nr - ||x| - |y||$.

В [46] для полученных бинарных векторов выполнялся линейный поиск по $\mathrm{dist}_{\mathrm{Ham}}$ с последующим уточнением по $\mathrm{dist}_{\mathrm{edit}}$ строк. Эффективность исключения по $\mathrm{dist}_{\mathrm{Ham}}$ растет с размерностью бинарного вектора (использовались размерности до 2560). Во многих случаях достаточно размерности 640. Показано ускорение выполнения запросов по сравнению с Flamingo [47], AdaptSearch [38], Pivotal [48] и другими алгоритмами при на порядок меньших затратах памяти.

Со сформированными векторами могут также применяться соответствующие ИС поиска по сходству (древовидные и другие [5, 6]).

**4.3. Префиксная фильтрация для строк.** Префиксная фильтрация [8] позволяет снизить затраты памяти на ИС и время фильтрации за счет рассмотрения не всех, а только части $n$-грамм строк. Для этого в All-Pairs-Ed [49] $n$-граммы упорядочивают согласно некоторому глобальному порядку, например по увели-

чению частоты встречаемости. Если общее количество совпадающих $n$-грамм двух строк не меньше порогового $T$, то в первых $|a_{n\text{-gramm}}|-T+1$ $n$-граммах (т.е. в префиксах) двух строк должна оказаться одинаковая $n$-грамма (здесь $|a_{n\text{-gramm}}|$ — количество $n$-грамм в строке $a$). Учитывая, что количество $n$-грамм в строке равно $|a|-n+1$ и выражение для $T=|a|-n+1-rn$ (см. подразд. 4.1), получаем длину префикса $nr+1$.

Применяя фильтр по положению, при префиксной фильтрации сравнивают $n$-граммы, положения которых отличаются не более чем на $r$ позиций (см. подразд. 4.1). Фильтр несовпадений из EdJoin (см. подразд. 4.2) позволяет уменьшить длину префикса с $nr+1$ до некоторой величины в диапазоне $[r+1, nr+1]$.

Для префиксной фильтрации используется малое количество $n$-грамм. Поэтому, когда строки длинные, эффективность исключения для малого количества $n$-грамм уменьшается. Кроме того, длина префикса зависит от $r$. Если конструировать ИС по длинным префиксам, рассчитанным на большие $r$, размер ИС увеличится.

**4.4. Поиск на основе модификации строк.** Одним из недостатков $n$-граммного подхода (см. подразд. 4.1, 4.3) является невозможность работы с короткими строками. Подход к поиску сходных строк посредством всех возможных модификаций строк и последующего поиска по совпадению (аналогично поиску бинарных векторов [4]) осложняется необходимостью большого количества различных модификаций вследствие как применения $\text{dist}_{\text{edit}}$, так и перебора большого количества сочетаний символов из алфавита (оценка $O(D^r|\Sigma|^r)$ дана в [50]).

Частичное решение этой проблемы состоит в использовании вместо всех возможных модификаций только модификаций удалением ([22, 23, 7] и см. подразд. 2.2). Пусть $\text{dist}_{\text{edit}} \le r$ и удаляется от 0 до $r$ символов двух строк. Показано, что хотя бы одна из получившихся пар строк совпадет [51, 25]. Таким образом, количество модификаций одной строки уменьшается до $\sum_{i\le r} C_D^i$, т.е. до количества модификаций запроса для бинарных векторов [4].

При конструировании ИС FastSS (http://fastss.csg.uzh.ch/ [51]) генерирует все возможные модификации удалением до $r$ символов всех строк базы и формирует обратный индекс по этим модификациям и соответствующим им исходным строкам. При выполнении запроса модифицируют удалениями строку-запрос и для каждой модификации извлекают кандидатов, с которыми вычисляют $\text{dist}_{\text{edit}}$. Отметим, что если для каждой модификации сохранять положения удаленных символов, то можно определить $\text{dist}_{\text{edit}}$ между объектом-запросом и объектом-кандидатом за время $O(r)$. Расширение на различные размеры строк и радиусы запроса разбиением строк на части предложено в [52, 25] и превосходит FastSS по скорости.

Метод модификации строк посредством удаления их символов в отличие от подхода $n$-грамм (см. подразд. 4.1, 4.3) позволяет работать с короткими строками. Модифицированные строки длиннее $n$-грамм, что позволяет проводить более эффективную фильтрацию кандидатов для коротких строк и малых значений $r$. Так, для баз с очень короткими строками FastSS показал лучшее быстродействие относительно всех сравниваемых алгоритмов [8] (подразд. 4.6).

Однако подход модификации удалением требует значительных затрат памяти на хранение всех модифицированных строк и большого количества модификаций и запросов по ним при поиске кандидатов и поэтому на практике работает только для коротких строк.

**4.5. Поиск строк на основе префиксных деревьев.** Вследствие перекрытия $n$-грамм их обратное индексирование требует объема памяти в несколько раз бóльшего, чем объем базы, и плохо работает с короткими строками (см. подразд. 4.1). ИС на основе модификации удалением хорошо работает только с короткими строками, так как требует еще бóльших затрат памяти и значительного количества запросов на совпадение (см. подразд. 4.4). Кроме того, при работе

с каждым объектом-кандидатом выполняются операции случайного ввода-вывода, так как кандидаты расположены в случайном месте памяти, поэтому для хранения базы нужна быстрая память.

Для преодоления некоторых из этих недостатков применяют древовидные ИС (см. обзор [6]). Рассмотрим ИС на основе префиксных деревьев (ИС на основе B+tree описаны в разд. 5, об использовании других древовидных ИС для поиска строк см. подразд. 4.2).

Подход к созданию префиксных деревьев [53], включающий TrieTraverse [54] и TriePathStack [55], основан на идее, что сходные строки имеют сходные префиксы. И действительно, если для префиксов двух строк уже выполняется $\mathrm{dist}_{\mathrm{edit}} > r$, эти строки можно исключить, так как последующие символы могут только увеличить $\mathrm{dist}_{\mathrm{edit}}$.

Из базы строк формируют префиксное дерево. Его корень соответствует спецсимволу «пустая строка». Так как многие строки имеют одинаковые префиксы, затраты памяти можно оптимизировать, несмотря на расходы памяти на конструирование дерева, особенно для сжатых вариантов префиксного дерева. Эффективность при выполнении запроса достигается за счет инкрементного вычисления $\mathrm{dist}_{\mathrm{edit}}$ при обходе дерева, повторного использования результатов вычисления $\mathrm{dist}_{\mathrm{edit}}$ для строк с общим префиксом, рассмотрения только тех модификаций строки-запроса, которые соответствуют строкам базы (аналогично ИС для $\mathrm{dist}_{\mathrm{Ham}}$ [4]).

Строки с одинаковыми префиксами имеют одинаковые узлы-предки. Это можно использовать для исключения не одной, а групп строк. Активные узлы дерева — это узлы, отличающиеся на $\mathrm{dist}_{\mathrm{edit}} \le r$ в префиксах строк, соответствующих узлам. Если узел не является активным для всех префиксов исходной строки, для его потомков $\mathrm{dist}_{\mathrm{edit}} > r$. Поэтому простой алгоритм поиска заключается в вычислении по строке-запросу множества активных узлов.

При выполнении запроса используется инкрементная процедура TrieTraverse. Дерево обходят, посимвольно приращивая строку-запрос. По строке, соответствующей пути в префиксном дереве, и по строке-запросу строится матрица динамического программирования для вычисления $\mathrm{dist}_{\mathrm{edit}}$ и активации узлов. По префиксам (если для них $\mathrm{dist}_{\mathrm{edit}} > r$) исключают множество поддеревьев, корневые узлы которых не являются активными. Если активному узлу соответствует строка базы, для нее $\mathrm{dist}_{\mathrm{edit}} \le r$ и, следовательно, она принадлежит подмножеству строк базы, которые являются ответом на запрос. Таким образом, не требуется уточнения прямым вычислением расстояния $\mathrm{dist}_{\mathrm{edit}}$ от строки-запроса до этих строк (необходимые $\mathrm{dist}_{\mathrm{edit}}$ были уже вычислены инкрементно при обходе дерева и активации узлов).

Результаты выполнения запроса APSS (поиск всех строк базы с $\mathrm{dist}_{\mathrm{edit}} \le r$) можно получить, используя в качестве строк-запросов все строки базы. Однако при этом не учитывается тот факт, что в базе строк-запросов многие строки также могут иметь общие префиксы. Более эффективный алгоритм TriePathStack [55] строит общее префиксное дерево для строк обеих баз и определяет для каждого узла одной базы активные узлы из другой базы.

Недостатки TriePathStack проявляются при работе с длинными несхожими строками. Количество активных узлов становится большим. Например, если $\mathrm{dist}_{\mathrm{edit}} = 3$, то все узлы префиксного дерева на первых четырех уровнях активны. Ситуация усугубляется для строк с большими алфавитами символов. Различные методы исключения [55] применяются после стадии генерации активных узлов. Отсюда большие вычислительные затраты и затраты памяти, что делает эти подходы неэффективными для больших баз, длинных строк и больших порогов на $\mathrm{dist}_{\mathrm{edit}}$.

В PreJoin [56] активные узлы исключают не после их генерации, а используют правила исключения в процессе генерации. Для этого генерируют множество активных узлов не только узлов-сыновей, но одновременно и узлов-внуков. По-

рядок обхода нефиксированный, в первую очередь выявляют и посещают «важные» поддеревья. Новый метод генерации активных узлов предназначен для уменьшения их количества. При этом минимизируются затраты памяти и вычислительные затраты на исключение уже активных узлов.

PreJoinPlus [56] отличается применением идеи [49] разбиения строк на две части. По крайней мере для одной из двух частей должно выполняться $\mathrm{dist}_{\mathrm{edit}} \le r/2$. В отличие от обычного префиксного дерева множество кандидатов, полученных для каждой из двух частей, необходимо уточнить. Сравнение с деревьями TrieTraverse, TriePathStack и $n$-граммным подходом EdJoin (см. подразд. 4.1) показало преимущество PreJoinPlus.

**4.6. Признаки и фильтрация на основе разбиения строк без пересечения.** Как отмечалось в подразд. 4.5, обратный индекс для перекрывающихся $n$-грамм требует в несколько раз больше памяти, чем хранение базы. Ряд алгоритмов фильтрации используют разбиение представления объекта на непересекающиеся части (см., например, [4, 13, 15]). Если $\mathrm{dist}_{\mathrm{edit}} \le r$ и одна строка разбита на непересекающиеся части (сегменты), количество которых $r+1$, то по крайней мере одна часть совпадает с некоторой подстрокой другой строки [57, 7]. В отличие от частей подстроки пересекаются. При этом положение начальных символов частей двух строк отличается не более чем на $r$ позиций (фильтрация по положению). Оптимальное разбиение может определяться с учетом статистики встречаемости частей строк.

Поиск в строках базы подстроки, совпадающей с частью строки-запроса, можно выполнить с помощью обратного индекса позиционных $n$-грамм [7]. Каждую строку базы дополняют $n-1$ спецсимволом (это обеспечивает наличие $n$-грамм, начинающихся со всех символов строки) и строят обратный индекс всех $n$-грамм, включающий информацию об их положении в строках. Для всех строк, имеющих определенную длину, создают отдельный обратный индекс.

При выполнении запроса строку-запрос разбивают на $r+1$ часть без пересечения. Для каждой части находят строки, содержащие эту часть, причем в положении, отличающемся не более чем на $r$. При длине части $n$ поиск тривиален (длина части равна длине $n$-граммы и надо выполнить поиск по совпадению).

При длине части, бóльшей $n$, рассматривают все входящие в часть $n$-граммы. По каждой из них извлекают обратные списки (идентификаторы строк базы) с учетом ограничения на их положение. Строки, которые содержатся во всех извлеченных обратных списках, являются кандидатами на то, что их подстроки совпадут с частью строки-запроса. Для выявления реального совпадения выполняют посимвольное сравнение.

При длине части, меньшей $n$, извлекают строки базы, $n$-граммы которых имеют префикс, совпадающий с частью (с учетом ограничения на их положение в строках). Таким образом находят строки базы, совпадающие со всеми частями строки-запроса. Эти строки являются кандидатами на уточнение прямым вычислением $\mathrm{dist}_{\mathrm{edit}}$ или точной проверкой условия $\mathrm{dist}_{\mathrm{edit}} \le r$.

Очевидно, алгоритм не применим при длине строки-запроса, меньшей $r+1$. Кроме того, для строк длиной, меньшей $2r+1$, при $r$ операциях удаления строки-запросы будут иметь количество символов, меньшее $r+1$. Поэтому требуется отдельная ИС для строк длиной, меньшей $2r+1$, и выполнения запросов с длиной строки-запроса, меньшей $r+1$. Так как строки короткие, в этом случае можно применять ИС с модификацией запроса (см. подразд. 4.4).

Аналогично [4, подразд. 2.2.1] и в соответствии с принципом «pigeonhole» [58] существуют варианты ИС с нахождением не точно совпадающих частей. Так, при разбиении строки-запроса на $m \ge 1$ непересекающихся частей по крайней мере одна часть будет иметь $\mathrm{dist}_{\mathrm{edit}} \le \lfloor r/m \rfloor$ до некоторой подстроки второй

строки, причем положения части и подстроки отличаются не более чем на $r$ [59]. Для такого поиска можно использовать например суффиксное дерево [7], модификацию строк с последующим поиском по совпадению (см. подразд. 4.4), поиск по непересекающимся $n$-граммам строки-запроса [60].

QChunk (IndexGram и IndexChunk) [39] извлекают различные признаки из строк базы и строк запроса — пересекающиеся и непересекающиеся $n$-граммы. Строки дополняют спецсимволами ($n-1$ спецсимвол для строк, из которых извлекаются пересекающиеся $n$-граммы, а при извлечении непересекающихся $n$-грамм спецсимволы обеспечивают длину $n$ последней из них). Идентичные признаки в различных положениях рассматриваются как разные. IndexGram использует $n$-граммы для строк базы и непересекающиеся $n$-граммы для строки-запроса, а IndexChunk наоборот. Для IndexGram минимальное количество совпадающих признаков (нижняя граница) равно $\lceil |x|/n \rceil - r$, а для IndexChunk оно равно $\lceil |y|/n \rceil - r$, где $\lceil |x|/n \rceil$ — количество непересекающихся $n$-грамм в строке $x$. Аналогичные условия существуют для признаков с учетом положения, отличающегося не более чем на $r$.

IndexGram и IndexChunk можно применять и к префиксам. При этом длина префикса для непересекающихся $n$-грамм равна $r+1$, а для пересекающихся $n$-грамм выражение для длины префикса более сложное. Вначале выполняется извлечение строк базы, содержащих признак из префикса, и их фильтрация по длине и по положению, затем — дополнительная фильтрация с учетом описанных ранее нижних границ с дальнейшими оптимизациями.

PassJoin [36] разбивают строку на $r+1$ часть без пересечения (сегмент). Если какая-то строка имеет с этой строкой $\mathrm{dist}_{\mathrm{edit}} \le r$, то у нее должна быть подстрока, совпадающая с сегментом первой строки. Используют разбиение строк базы на почти равные сегменты, отличающиеся по длине не более чем на 1. По ним строят обратный индекс, в котором обратные списки формируют для строк одинаковой длины. Предложен также алгоритм разбиения строки запроса на подстроки, минимизирующий количество подстрок, по которым выполняется запрос, без потери кандидатов (т.е. без false negatives).

На этапе уточнения списка кандидатов строки разбивают на части (совпадающую, левую, правую) и используют пороги для исключения строк. Так, для соответствия запросу rNN должно выполняться $\mathrm{dist}_{\mathrm{edit}} \le i-1$ для левой части и $\mathrm{dist}_{\mathrm{edit}} \le r+1-i$ для правой части, где $i$ — номер сегмента строки, по которому она извлечена из обратного списка. Отметим, что величины порогов для исключения частей меньше величин порогов для исключения целых строк, что потенциально позволяет получить более эффективное исключение. В экспериментах для длинных строк [8] PassJoin оказался быстрее ИС Bed-tree, ListMerger, PartEnum, AllPair, PPJoin, EDJoin, QChunk, VChunk, AdaptJoin, PassJoin, TrieJoin, FastSS, M-Tree. Для коротких строк конкурентоспособен также FastSS (см. подразд. 4.4), но при больших затратах памяти.

ИС Pivotal [48] также использует количество непересекающихся $n$-грамм, равное $r+1$, но из префикса длиной $nr+1$ (см. подразд. 4.3). Как обычно, $n$-граммы префикса отсортированы в глобальном порядке (в Pivotal — по частоте встречаемости). Если последняя $n$-грамма префикса строки $b$ предшествует последней $n$-грамме префикса строки $a$, то среди непересекающихся $n$-грамм префикса $b$ (при любом разбиении на $r+1$ часть) должна найтись по крайней мере одна, совпадающая с $n$-граммой префикса $a$.

По базе, отсортированной по длине строк, строят два обратных индекса: для префиксных позиционных $n$-грамм с пересечением и без него. Для работы с переменным порогом $r$ обратные индексы реализованы как инкрементные: каждый последующий обратный индекс соответствует большему значению $r$ и включает

объекты, не вошедшие в предшествующие обратные индексы. При выполнении запроса используют префиксные $n$-граммы строки-запроса с пересечением и без него, выполняя поиск в соответствующих обратных индексах. С каждой $n$-граммой в ИС ассоциированы длины строк, которым она принадлежит, что позволяет выполнять фильтрацию по длине (см. подразд. 4.1).

Непересекающиеся $n$-граммы можно выбирать в префиксе различными способами. Для выбора высококачественных непересекающихся $n$-грамм префикса (с короткими обратными списками и соответственно с большей эффективностью исключения) используют динамическое программирование. Это позволяет эффективно исключать строки, в которых символы, отличающиеся от строки-запроса, расположены непоследовательно.

Чтобы исключить большое количество несходных строк, в которых символы, отличающиеся от строки-запроса, расположены последовательно, используют фильтр постановки в соответствие (alignment filter), основанный на следующем утверждении. Найдем сумму минимальных расстояний редактирования между каждой из множества непересекающихся $n$-грамм (их $r+1$) префикса одной строки (с начальным положением $\mathrm{pos}_i$) и любой подстрокой второй строки. Если для двух строк $\mathrm{dist}_{\mathrm{edit}} \le r$, то эта сумма не превышает $r$. При этом позиционный фильтр (см. подразд. 4.1) позволяет ограничить вторую строку символами, расположенными между $\mathrm{pos}_i - r$ и $\mathrm{pos}_i + n - 1 + r$. Так как сложность вычисления $\mathrm{dist}_{\mathrm{edit}}$ велика, в alignment filter используют нижнюю границу на основе $\mathrm{dist}_{\mathrm{Ham}}$ между бинарными векторами символов (см. подразд. 4.2).

Комбинация с pigeonring [58] ускоряет Pivotal до 3.1 и 3.9 раз. Возможно, это самая быстрая ИС для поиска строк в настоящее время.

## 5. ПОИСК БЛИЖАЙШИХ СОСЕДЕЙ

При работе со строками выполнение запроса kNN обычно называют top-k similarity search. Рассмотрим ИС для выполнения запросов поиска приближенного ближайшего соседа с теоретическими гарантиями (подразд. 5.1) и практические ИС для точного поиска, но без теоретических гарантий сублинейного времени (подразд. 5.2).

**5.1. Приближенный поиск с теоретическими гарантиями.** Для выполнения запроса $(c, \delta)$-NN по $\mathrm{dist}_{\mathrm{edit}}$ с существенно сублинейным временем используют вложения строк, а также ИС для полученных векторов. Отметим, что ИС, описанные в этом разделе, не реализованы на практике.

Так, в [61] используют вложение строк в векторное пространство с $\mathrm{dist}_{\mathrm{Man}}$ на основе разбиения исходных строк на непересекающиеся подстроки и их сдвигов, для этого вложения с высокой вероятностью выполняется ограничение на искажение расстояний. Затем применяют ИС для выполнения вероятностных приближенных запросов по $\mathrm{dist}_{\mathrm{Man}}$ (например, [62] или ИС LSH для $\mathrm{dist}_{\mathrm{Man}}$, см. ссылки в [11, 12] и в подразд. 3.2). В результате получают итоговую ИС для поиска строк по $\mathrm{dist}_{\mathrm{edit}}$ с затратами памяти $(DN)^{O(1)}$ и временем выполнения запроса $(D \log N)^{O(1)}$ при множителе аппроксимации $2^{\sqrt{O(\log D \log\log D)}}$, где $D$ — длина строки.

ИС для $\mathrm{dist}_{\mathrm{edit}}$ [63] основана на ИС для приближенного поиска по метрике sum-product [11], в которой метриками-компонентами являются $\mathrm{dist}_{\mathrm{edit}}$ коротких частей строк. ИС для поиска ближайшего соседа можно построить для коротких строк запоминанием ближайшего соседа для всех возможных запросов [4]. Это обеспечивает время выполнения запроса $O(D)$ с экспоненциальными затратами памяти $(DN|\Sigma|)^{O(D^{\alpha})}$, $\alpha > 0$, и постоянным множителем аппроксимации. При множителе аппроксимации $D^{\alpha}$ можно получить полиномиальные затраты памяти.

Используются два преобразования: разбиение длинной строки на короткие, позволяющее вычислить исходное $\mathrm{dist}_{\mathrm{edit}}$, а также вложение sum-product в max-product, которое выполняется с искажением и вероятностью, достаточными для поиска ближайшего соседа по sum-product через ИС для поиска по max-product [64].

Для приближенного поиска по метрике Улама ($\mathrm{dist}_{\mathrm{edit}}$ на строках без повторяющихся символов) применяют [65] вложение строк с постоянным искажением в векторы из iterated product space $\oplus_{(L_2)^2}^{D} \oplus_{L_\infty}^{O(\log D)} L_1^{2D}$ с расстоянием между векторами $\mathrm{dist}_{\mathrm{NEG},\infty,1}(\mathbf{a}, \mathbf{b})$. Это забывчивое вложение отображает строки в тщательно подобранный набор характеристических векторов их подстрок (т.е. частей строк со смежными символами). Использование ИС (которую можно считать обобщением ИС LSH для $\mathrm{dist}_{\mathrm{Man}}$), разработанной для приближенного поиска по метрике iterated product space (и вероятностного приближенного вложения), позволило получить лучший известный алгоритм поиска по метрике Улама с аппроксимацией $O(\log\log N)$ и с возможностью улучшения до $O(\log\log D)$ [65]. Конкретнее, эта теоретическая ИС обеспечивает время выполнения запроса $D^{O(1)} N^{\varepsilon}$ и затраты памяти $(DN)^{O(1)}$ при величине множителя аппроксимации $c = O(\varepsilon^{-3} \log\log N)$ для любого $\varepsilon > 0$. Отметим, что вложение метрики Улама с постоянным искажением в более простые пространства, например в $L_1$, невозможно [65].

ИС для выполнения приближенных запросов kNN по $\mathrm{dist}_{\mathrm{edit}}$ на основе вложения в векторное пространство с $\mathrm{dist}_{\mathrm{Man}}$ [28, 29] и LSH-forest упоминалась в подразд. 3.2.

**5.2. Практический точный поиск без теоретических гарантий.** Даже если ИС позволяет выполнять запросы rNN с переменным $r$, реализация запросов kNN последовательностью запросов rNN с различными $r$ (см. подразд. 1.2) очень трудоемка. Поэтому разрабатываются другие решения для запросов kNN, обычно без теоретических гарантий.

В ИС AQ [66] предложен один из первых алгоритмов для выполнения точного запроса kNN по $\mathrm{dist}_{\mathrm{edit}}$. Фильтрация выполняется по пороговому значению $T$ количества совпадающих $n$-грамм и по отличию длины строк (см. подразд. 4.1), которые изменяют в процессе выполнения запроса.

Строки разной длины запоминают в различных обратных индексах. При выполнении запроса устанавливают $\mathrm{length}_{\mathrm{diff}} = 0$, $T = 1$. Извлекают строки $y$, для которых $||y| - |x|| = \mathrm{length}_{\mathrm{diff}}$. Для них с использованием порога $T$ находят строки-кандидаты, ранжируют их по $\mathrm{dist}_{\mathrm{edit}}$ и получают $k$ текущих ближайших соседей. Если расстояние от объекта-запроса до $k$-го текущего ближайшего соседа не превышает $\mathrm{length}_{\mathrm{diff}} + 1$, ближайшие $k$ строк являются ответом на запрос. В противном случае вычисляют значение порога $T$, соответствующее текущему радиусу запроса, увеличивают значение $\mathrm{length}_{\mathrm{diff}}$ на 1 и повторяют процедуру. Кроме того, в $n$-граммах изменяют значение $n$ (вычисляют из формулы для текущего значения $T$). При этом $n$ увеличивается, что уменьшает размеры обратных списков и общее время выполнения запроса. Недостатком ИС AQ является большие затраты памяти на обратные индексы для $n$-грамм различной длины.

ИС Bed-tree [67] построена на основе B+tree. Это требует упорядочения строк на основе глобального порядка. Например, такой порядок задается сортировкой строк по длине, а затем упорядочением в лексикографическом порядке. При этом полезная информация содержится только в префиксах строк. Больше информации обеспечивает порядок $n$-грамм. Хэш-функция отображает каждую $n$-грамму в хэш-значение (возможны коллизии). Затем полученный хэш-вектор строки отображают в значение Z-порядка [68]. Для упорядочения $n$-грамм в дополнение к хэш-значению используется информация об их положении (см. подразд. 4.1).

Применение таких упорядочений позволяет вычислить нижнюю границу $\mathrm{dist}_{\mathrm{edit}}$ между строкой-запросом и строками базы в интервале строк, заданном конкретным порядком, и выполнять запросы поиска по сходству с помощью B+tree. Так, при выполнении запроса rNN алгоритм рекурсивно посещает узлы B+tree с нижней границей $\mathrm{dist}_{\mathrm{edit}}$ не более $r$ и использует строки из листовых узлов в качестве кандидатов. Отметим, что переход может происходить сразу в несколько узлов. Запрос kNN выполняется посредством модификации порога в величину расстояния до текущего $k$-го ближайшего соседа.

К недостаткам Bed-tree относится малая эффективность поиска для коротких строк базы (поиск вырождается во множество линейных поисков на частях B+tree). Кроме того, каждое изменение порога при запросе kNN требует вычисления большого количества расстояний $\mathrm{dist}_{\mathrm{edit}}$ до строк-кандидатов.

ИС PBI на основе B+tree [69] является аналогом метрической ИС iDistance [70], но для строк с $\mathrm{dist}_{\mathrm{edit}}$. Предложены эвристики выбора опорных объектов-строк и присвоения их областям других строк базы таким образом, чтобы минимизировать количество кандидатов. Хотя PBI превосходит ИС Flamingo [47] и Bed-tree в большинстве экспериментов, для очень длинных строк все эти алгоритмы работают плохо.

В ИС Range [71] повышают эффективность алгоритма динамического программирования для вычисления $\mathrm{dist}_{\mathrm{edit}}$ за счет вычисления не всех элементов матрицы, используемой в динамическом программировании. Дублирования вычислений избегают группированием элементов, соответствующих общим подстрокам.

В ходе обработки запроса быстро находят строки с малым $\mathrm{dist}_{\mathrm{edit}}$ до строки-запроса и используют эти расстояния в качестве границ. Чтобы не выполнять множество запросов rNN с различными пороговыми значениями $r$, применяют префиксное дерево (см. подразд. 4.5). Находят префиксы, совпадающие со строкой-запросом или отличающиеся на некоторое значение $\mathrm{dist}_{\mathrm{edit}}$ (активные узлы, см. поздразд. 4.5). Затем из узлов, соответствующих этим префиксам, продолжают обход дерева (с тем же или увеличивающимся значением $\mathrm{dist}_{\mathrm{edit}}$), пока не посетят $k$ листьев. Строки в них являются ближайшими соседями (т.е. не требуется их уточнения вычислением до них расстояний $\mathrm{dist}_{\mathrm{edit}}$). В экспериментах показано преимущество над Flamingo, AQ, Bed-tree.

Range обеспечивает эффективное исключение для коротких строк. Для длинных строк количество общих префиксов уменьшается и это снижает эффективность процедуры исключения. Кроме того, ИС на основе префиксных деревьев сложны в конструировании и предназначены только для работы с быстрой памятью.

ИС AppGram [72] предлагает стратегию фильтрации на основе количества $n$-грамм не с полным, а с приближенным совпадением. Это позволяет использовать более длинные $n$-граммы и уменьшить число false positives. Применяется двухуровневый обратный индекс. Верхний уровень работает с $n$-граммами, индексирующими строки, нижний — с частями $n$-грамм, индексирующими $n$-граммы. При поступлении запроса по его $n$-граммам нижний обратный индекс находит сходные $n$-граммы с $\mathrm{dist}_{\mathrm{edit}} \le t$ между $n$-граммами на основе подсчета совпавших частей $n$-грамм. Затем эти $n$-граммы используют для получения списков строк-кандидатов на верхнем уровне при возрастающих значениях $t$. К спискам строк-кандидатов применяют фильтры. Если строки $a$, $b$ имеют $\mathrm{dist}_{\mathrm{edit}} < r$, то у них должно быть не менее $\max(|a|,|b|)-n+1-\max(1, n-2t)-(n-t)(r-1)$ таких $n$-грамм, расстояние между которыми $\mathrm{dist}_{\mathrm{edit}} \le t$. Более плотную нижнюю границу $\mathrm{dist}_{\mathrm{edit}}$ дает расстояние на основе оптимального отображения между мультимножествами $n$-грамм [72]. Текущее значение $r$ равно $\mathrm{dist}_{\mathrm{edit}}$ от объекта-запроса до $k$-го текущего ближайшего соседа.

Затраты памяти и время конструирования AppGram больше, чем у Bed-tree, но меньше, чем у Flamingo. Время выполнения запроса при $k \ge 2$ меньше, чем

у Flamingo, Bed-tree, Range, а при $k=1$ Flamingo и Range быстрее. Время уточнения кандидатов доминирует во времени выполнения запроса.

ИС HS-tree [73] выполняет запросы rNN и kNN. Строки группируют по длине. Для каждой группы строк конструируют полное бинарное дерево. В каждом узле дерева (под)строку разбивают на две непересекающиеся части (равной или отличающейся на единицу длины), префикс и суффикс. Разбиение проводят рекурсивно до достижения одного символа в сегменте. В каждом узле хранят обратный индекс его сегментов (со строками базы, которым принадлежат сегменты).

При выполнении запроса rNN фильтру по длине удовлетворяют индексные структуры HS-tree для строк длиной от $|x|-r$ до $|x|+r$. На $i$-м уровне (как обычно, дерево «растет» вниз, т.е. расположенный ниже уровень имеет бóльшее значение $i$) строки разбиты на $2^i$ сегментов. Для $2^i \geq r+1$ строка базы не может являться ответом на запрос, если строка-запрос не имеет общего сегмента со строкой базы (исключение по принципу pigeonhole [58]). Более того, в строке-запросе должно быть не менее $2^i - r$ сегментов строки базы, чтобы последняя стала объектом-кандидатом. Так как с увеличением номера уровня уменьшается размер сегмента и эффективность исключения, используют уровень $\lceil \log(r+1) \rceil$. Для дополнительной фильтрации применяют разницу в положении сегментов и длинах строк (см. подразд. 4.1).

Дополнительно количество кандидатов уменьшают устранением «конфликтов», которые заключаются в том, что сегменты строки-запроса должны быть непересекающимися. Для этого используют динамическое программирование [36]. При уточнении кандидатов вместо вычисления значения $\mathrm{dist}_{\mathrm{edit}}$ применяют эффективный алгоритм сравнения с $r$ на основе индивидуальных пороговых значений для сегментов.

При выполнении запроса kNN для каждого уровня проверяют условие $2^{i-1} \geq \mathrm{dist}^*_{\mathrm{edit}} + 1$, где $\mathrm{dist}^*_{\mathrm{edit}}$ — расстояние до текущего $k$-го ближайшего соседа. Если условие выполняется, поиск прекращают, при этом найденные текущие ближайшие соседи являются истинными. В противном случае находят строки базы, для которых количество совпавших подстрок больше или равно $2^i - \mathrm{dist}^*_{\mathrm{edit}}$. Строки группируют по значениям этого количества и посещают такие группы в порядке его уменьшения. Для каждой строки вычисляют истинное $\mathrm{dist}_{\mathrm{edit}}$ и модифицируют приоритетную очередь текущих ближайших соседей и $\mathrm{dist}^*_{\mathrm{edit}}$, если встречаются строки с $\mathrm{dist}_{\mathrm{edit}} < \mathrm{dist}^*_{\mathrm{edit}}$.

Кроме того, используют фильтрацию строк с последовательными ошибками, которые описанная фильтрация не исключает. Вместо вычисления $\mathrm{dist}_{\mathrm{edit}}$ для строк, прошедших через фильтрацию на уровне $i$, переходят на уровень $i+1$ и оценивают на нем более плотную нижнюю границу $\mathrm{dist}_{\mathrm{edit}}$. Если количество совпавших подстрок меньше $2^{i+1} - \mathrm{dist}^*_{\mathrm{edit}}$, их исключают. В противном случае проверяют уровень $i+2$. Если строка не исключена на уровне листьев дерева, то до нее вычисляют $\mathrm{dist}_{\mathrm{edit}}$ (с такими же оптимизациями, как для запроса rNN).

В экспериментальном исследовании вариантов ИС для быстрой памяти при выполнении запроса rNN ИС HS-tree оказалась быстрее лучших алгоритмов AdaptSearch [38], QChunk ([39], см. подразд. 4.6), PassJoin ([36], см. подразд. 4.6), а при запросе kNN — быстрее Range и AQ. В экспериментах с вариантами ИС для медленной (внешней) памяти при выполнении запроса rNN HS-tree оказалось быстрее Bed-tree и PassJoin, а при запросе kNN быстрее Bed-tree и AppGram [72].

Преимуществом HS-tree является возможность использования внешней памяти, а недостатком — неэффективность исключения для больших значений $r$.

## ЗАКЛЮЧЕНИЕ

Индексные структуры, предложенные для поиска символьных строк базы, сходных со строкой-запросом по расстоянию редактирования строк $\mathrm{dist}_{\mathrm{edit}}$, работают на тех же принципах, что и ИС для поиска по сходству векторных данных [4–6], множеств [4] и данных, для которых известны только величины расстояний/сходств между объектами [3]. В них широко применяется стратегия фильтрации и уточнения, при помощи которой на первом этапе быстро отбирают строки-кандидаты базы, которые могут являться ответом на запрос, а затем окончательный ответ определяют вычислением их $\mathrm{dist}_{\mathrm{edit}}$ до строки-запроса.

Некоторые ИС непосредственно работают с исходными строками.

Так, ИС на основе расстояний [3] пригодны для поиска по сходству любых объектов, в том числе и строк. Обычно при конструировании таких ИС используется иерархическое разбиение объектов базы на (под)множества, а при выполнении запроса принимается решение, какие из подмножеств можно исключить, а какие продолжать обрабатывать. Многие из ИС этого класса используют метод ветвей и границ (branch and bound, B&B). В данном обзоре применение таких ИС для поиска строк не рассматривалось, так как ввиду отсутствия специализации под строки этот подход уступает по скорости другим [7].

В ИС на основе модификации строк применяют некоторые модификации строки-запроса и ищут строки базы, совпадающие с полученными модификациями. Для уменьшения количества модификаций используют также запоминание модификаций объектов базы. Этот подход эффективен только для коротких строк и малых радиусов запроса.

Специализированные для строк префиксные деревья фактически используют только те модификации строки-запроса, которые содержатся в базе. Кроме того, зачастую не требуется выполнять этап уточнения, так как кандидаты являются ответом на запрос. Однако эти ИС требуют быстрой памяти и плохо работают с базами коротких строк, которые содержат почти все возможные строки заданной длины, так как это не уменьшает количества модификаций. Для больших баз с длинными несходными строками и большими порогами на $\mathrm{dist}_{\mathrm{edit}}$ мала эффективность исключения строк, которые не являются ответом на запрос.

Другой класс ИС работает с признаками, которые извлекают из строк, например с $n$-граммами (различной длины, без учета или с учетом их положения, с перекрытием или без него и др.). Анализ количества совпадающих или сходных $n$-грамм строки-запроса и строк базы, эффективно выполняющийся с помощью обратного индекса, позволяет отобрать строки-кандидаты для уточнения их соответствия исходному запросу вычислением точной величины или границы на $\mathrm{dist}_{\mathrm{edit}}$ строк. Выбор оптимального размера признака для наивысшей производительности ИС является непростой задачей [8]. Учет только части $n$-грамм (префиксная фильтрация) позволяет повысить скорость отбора строк-кандидатов. Разбиение строк на непересекающиеся части уменьшает количество выделяемых признаков.

Кроме того, расстояния между векторами частот встречаемости (или индикаторами наличия) признаков строк ограничивают $\mathrm{dist}_{\mathrm{edit}}$ исходных строк. Это позволяет применять ИС, разработанные для быстрого поиска по сходству векторов [4–6], для быстрого поиска строк-кандидатов по векторам их признаков. Отметим возможность повышения эффективности поиска векторов за счет снижения их размерности, например случайным проецированием [1, 2, 74, 75] — рандомизированным алгоритмом, который применяется и в других задачах [76–78].

ИС для выполнения запросов точного поиска с теоретическими гарантиями времени поиска для худшего случая, сублинейного от количества строк в базе, имеют существенный недостаток — экспоненциальную зависимость времени запроса и/или затрат памяти от радиуса запроса. Это аналог «проклятия размерности» для векторных пространств [11, 4, 5], а также для данных с «внутренней»

размерностью [3]. Экспоненциальная зависимость не позволяет реализовать и использовать такие ИС на практике, кроме как для очень малых радиусов. Многие классы практически реализуемых ИС, не имеющие гарантий худшего случая, дают выигрыш над линейным поиском только для малых радиусов. Так, в [7] реализованы ИС с радиусом запросов до трех и отмечается, что ускорение линейного поиска достигает четырех порядков, но быстро нивелируется с увеличением радиуса. Однако полезные практические применения существуют и для единичного радиуса запроса.

ИС для приближенного поиска строк с существенно сублинейным (полилогарифмическим от $N$) временем запроса и полиномиальными затратами памяти (см. подразд. 5.1) используют вложения в векторные пространства с разными типами расстояний и известным искажением вложения. Для поиска по полученным представлениям применяют ИС с гарантиями сублинейного времени. Эти ИС не реализованы на практике. ИС для приближенного поиска на основе LSH, которые также применяют вложения строк в векторы, имеют практические реализации, и для них также можно ожидать теоретических гарантий сублинейного (хотя и не логарифмического) времени выполнения запросов [79]. Однако в [80] предложена условная нижняя граница времени запроса приближенного ближайшего соседа для любых алгоритмов, включая рандомизированные, которая следует из условия выполнения SETH. Эта граница требует почти линейного времени запроса при полиномиальном времени предобработки. А именно, если выполняется SETH, то для любых констант $\beta, \alpha > 0$ существует константа $\varepsilon = \varepsilon(\beta, \alpha)$ такая, что никакой алгоритм не может предобработать множество из $N$ векторов за время $O(N^{\alpha})$ и затем ответить на запрос $(1+\varepsilon)$-NN за время $O(N^{1-\beta})$.

Нерешенной проблемой является обучение другим мерам сходства, отличным от $\mathrm{dist}_{\mathrm{edit}}$, на основе обучающих выборок, в том числе сформированных суждениями специалистов о сходстве строк [9].

СПИСОК ЛИТЕРАТУРЫ


1. Rachkovskij D.A. Real-valued vectors for fast distance and similarity estimation. *Cybernetics and Systems Analysis*. 2016. Vol. 52, N 6. P. 967–988.
2. Rachkovskij D.A. Binary vectors for fast distance and similarity estimation. *Cybernetics and Systems Analysis.* 2017. Vol. 53, N 1. P. 138–156.
3. Rachkovskij D.A. Distance-based index structures for fast similarity search. *Cybernetics and Systems Analysis*. 2017. Vol. 53, N 4. P. 636–658.
4. Rachkovskij D.A. Index structures for fast similarity search for binary vectors. *Cybernetics and Systems Analysis.* 2017. Vol. 53, N 5. P. 799–820.
5. Rachkovskij D.A. Index structures for fast similarity search for real-valued vectors. I. *Cybernetics and Systems Analysis.* 2018. Vol. 54, N 1. P. 152–164.
6. Rachkovskij D.A. Index structures for fast similarity search for real-valued vectors. II. *Cybernetics and Systems Analysis.* 2018. Vol. 54, N 2. P. 320–335.
7. Boytsov L. Indexing methods for approximate dictionary searching: Comparative analysis. *J. Exp. Algorithmics.* 2011. Vol. 16. P. 1.1:1–1.1:91.
8. Jiang Y., Li G., Feng J., Li W. String similarity joins: An experimental evaluation. *Proc. VLDB Endowment.* 2014. Vol. 7, N 8. P. 625–636.
9. Yu M., Li G., Deng D., Feng J. String similarity search and join: a survey. *Frontiers of Computer Science.* 2016. Vol. 10, N 3. P. 399–417.
10. Backurs A., Indyk P. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). Proc. STOC'15. 2015. P. 51–58.
11. Andoni A., Indyk P. Nearest neighbors in high-dimensional spaces. In: *Handbook of Discrete and Computational Geometry*. 3rd edition. Chap. 43. Boca Raton, USA: CRC Press, 2017. P. 1133–1153.

12. Andoni A., Indyk P. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*. 2008. Vol. 51, N 1. P. 117–122.
13. Mann W., Augsten N., Bouros P. An empirical evaluation of set similarity join techniques. *Proc. VLDB Endow*. 2016. Vol. 9, N 9. P. 636–647.
14. Jia L., Zhang L.,Yu G., You J., Ding J., Li M. A survey on set similarity search and join. *International Journal of Performability Engineering*. 2018. Vol. 14, N 2. P. 245–258.
15. Manber U., Wu S. An algorithm for approximate membership checking with application to password security. *Inf. Process. Lett.* 1994. Vol. 50, N 4. P. 191–197.
16. Chegrane I., Belazzougui D. Simple, compact and robust approximate string dictionary. J. *Discrete Algorithms.* 2014. Vol. 28. P. 49–60.
17. Belazzougui D. Faster and space-optimal edit distance "1" dictionary. *Proc. CPM'09*. 2009. P. 154–167.
18. Belazzougui D., Venturini R. Compressed string dictionary search with edit distance one. *Algorithmica*. 2016. Vol. 74, N 3. P. 1099–1122.
19. Chan T., Lewenstein M. Fast string dictionary lookup with one error. *Proc. CPM'15*. 2015. P. 114–123.
20. Fredman M.L., Komlos J., Szemeredi E. Storing a sparse table with O(1) worst case access time. *Journal of the ACM*. 1984. Vol. 31, N 3. P. 538–544.
21. Karp R.M., Rabin M.O. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*. 1987. Vol. 31, N 2. P. 249–260.
22. Mor M., Fraenkel A.S. A Hash code method for detecting and correcting spelling errors. *Communications of the ACM*. 1982.Vol. 25, N 12. P. 935–938.
23. Muth R., Manber U. Approximate multiple string search. *Proc. CPM'96.* 1996. P. 75–86.
24. Broder A., Mitzenmacher M. Network applications of bloom filters: A survey. *Internet Mathematics.* 2004. Vol. 1, N 4. P. 485–509.
25. Karch D., Luxen D., Sanders P. Improved fast similarity search in dictionaries. *Proc. SPIRE'10*. 2010. P. 173–178.
26. Cole R., Gottlieb L.-A., Lewenstein M. Dictionary matching and indexing with errors and don't cares. *Proc. STOC'04.* 2004. P. 91–100.
27. Chan H., Lam T.W., Sung W., Tam S., Wong S. Compressed indexes for approximate string matching. *Algorithmica*. 2010. Vol. 58, N 2. P. 263–281.
28. Sokolov A.M. Vector representations for efficient comparison and search for similar strings. *Cybernetics and System Analysis. 2007*. Vol. 43, N 4. P. 484–498.
29. Sokolov A.M. Investigation of accelerated search for close text sequences with the help of vector representations. *Cybernetics and Systems Analysis*. 2008. Vol. 44, N 4. P. 493–506.
30. Datar M., Immorlica N., Indyk P., Mirrokni V.S. Locality-sensitive hashing scheme based on p-stable distributions. *Proc. SCG'04*. 2004. P. 253–262.
31. Andoni A., Datar M., Immorlica N., Indyk P., Mirrokni V. Locality-Sensitive Hashing using stable distributions. In: *Nearest-Neighbor Methods in Learning and Vision: Theory and Practice*. Shakhnarovich G., Darrell T., Indyk P. (Eds.). Cambridge, MA: MIT Press, 2006. P. 61–72.
32. Bawa M., Condie T., Ganesan P. Lsh forest: self-tuning indexes for similarity search. *Proc. WWW'05.* 2005. P. 651–660.
33. Andoni A., Razenshteyn I., Shekel Nosatzki N. Lsh forest: Practical algorithms made theoretical. *Proc. SODA'17*. 2017. P. 67–78.
34. Zhang H., Zhang Q. EmbedJoin: efficient edit similarity joins via embeddings. *Proc. KDD'17*. 2017. P. 585–594.
35. Chakraborty D., Goldenberg E., Koucky M. Streaming algorithms for embedding and computing edit distance in the low distance regime. *Proc. STOC'16*. 2016. P. 712–725.
36. Li G., Deng D., Wang J., Feng J. Pass-join: A partition-based method for similarity joins. P*roc. VLDB Endowment.* 2011. Vol. 5, N 3. P. 253–264.
37. Xiao C., Wang W., Lin X. Ed-Join: an efficient algorithm for similarity joins with edit distance constraints. *Proc. VLDB Endowment.* 2008. Vol 1, N 1. P. 933–944.
38. Wang J., Li G., Feng J. Can we beat the prefix filtering?: An adaptive framework for similarity join and search. *Proc. SIGMOD'12*. 2012. P. 85–96.
39. Qin J., Wang W., Lu Y., Xiao C., Lin X. Efficient exact edit similarity query processing with the asymmetric signature scheme. *Proc. SIGMOD'11.* 2011. P. 1033–1044.

40. Jokinen P., Ukkonen E. Two algorithms for approximate string matching in static texts. *Proc. MFCS'91*. 1991. P. 240–248.
41. Gravano L., Ipeirotis P.G., Jagadish H.V., Koudas N., Muthukrishnan S., Srivastava D. Approximate string joins in a database (almost) for free. *Proc. VLDB'01*. 2001. P. 491–500.
42. Li C., Wang B., Yang X.. VGRAM: Improving performance of approximate queries on string collections using variable-length grams. *Proc. VLDB'07*. 2007. P. 303–314.
43. Yang X., Wang B., Li C. Cost-based variablelength-gram selection for string collections to support approximate queries efficiently. *Proc. SIGMOD'08*. 2008. P. 353–364.
44. Kahveci T., Singh A. An efficient index structure for string databases. *Proc. VLDB'01*. 2001. P. 351–360.
45. Jiang Y., Deng D., Wang J., Li G., Feng J. Efficient parallel partition-based algorithms for similarity search and join with edit distance constraints. *Proc. EDBT'13*. 2013. P. 341–348.
46. Wei H., Yu J.X., Lu C. String similarity search: a hash-based approach. *IEEE Transactions on Knowledge and Data Engineering*. 2018. Vol. 30, N 1. P. 170–184.
47. Vernicaand R., Li C. Efficient top-k algorithms for fuzzy search in string collections. *Proc. KEYS'09*. 2009. P. 9–14.
48. Deng D., Li G., Feng J. A pivotal prefix based filtering algorithm for string similarity search. *Proc. SIGMOD'14*. 2014. P. 673–684.
49. Chaudhuri S., Ganti V., Kaushik R. A primitive operator for similarity joins in data cleaning. *Proc. ICDE'06*. 2006. P. 5–16.
50. Ukkonen E. Approximate string-matching over suffix trees. In: *Combinatorial Pattern Matching. CPM 1993*. *Lecture Notes in Computer Science*. Apostolico A., Crochemore M., Galil Z., Manber U. (Eds.). 1993. Vol 684. P. 228–242.
51. Bocek T., Hunt E., Hausheer D., Stiller B. Fast similarity search in peer-to-peer networks. *Proc. NOMS'08*. 2008. P. 240–247.
52. Wang W., Xiao C., Lin X., Zhang C. Efficient approximate entity extraction with edit distance constraints. *Proc. SIGMOD'09*. 2009. P. 759–770.
53. Chaudhuri S., Kaushik R. Extending autocompletion to tolerate errors. *Proc. SIGMOD'09*. 2009. P. 707–718.
54. Li G., Ji S., Li C., Feng J. Efficient fuzzy full-text type-ahead search. *The VLDB Journal*. 2011. Vol. 20, N 4. P. 617–640.
55. Feng J., Wang J., Li G. Trie-Join: a trie-based method for efficient string similarity joins. *The VLDB Journal*. 2012. Vol. 21, N. 4. P. 437–461.
56. Gouda K., Rashad M. Efficient string edit similarity join algorithm. *Computing and Informatics*. 2017. Vol. 36. P. 683–704.
57. Wu S., Manber U. Fast text searching allowing errors. *Communications of the ACM*. 1992. Vol. 35, N 10. P. 83–91.
58. Qin J., Xiao C. Pigeonring: A principle for faster thresholded similarity search. *Proc. VLDB Endow*. 2018. Vol. 12, N 1. P. 28–42.
59. Baeza-Yates R., Navarro G. Faster approximate string matching. *Algorithmica*. 1999. Vol. 23, N 2. P. 127–158.
60. Navarro G., Sutinen E., Tarhio J. Indexing text with approximate q-grams. *Journal of Discrete Algorithms*. 2005. Vol. 3, N 2–4. P. 157–175.
61. Ostrovsky R., Rabani Y. Low distortion embedding for edit distance. *Journal of the ACM*. 2007. Vol. 54, N 5. P. 23–36.
62. Kushilevitz E., Ostrovsky R., Rabani Y. Efficient search for approximate nearest neighbor in high dimensional spaces. *SIAM Journal on Computin*g. 2000. Vol. 30, N 2. P. 457–474.
63. Indyk P. Approximate nearest neighbor under edit distance via product metrics. *Proc. SODA'04*. 2004. P. 646–650.
64. Indyk P. Approximate nearest neighbor algorithms for frechet metric via product metrics. *Proc. SoCG'02*. 2002. P. 102–106.
65. Andoni A., Indyk P., Krauthgamer R. Overcoming the L1 non-embeddability barrier: Algorithms for product metrics. *Proc. SODA'09*. 2009. P. 865–874.
66. Yang Z., Yu J., Kitsuregawa M. Fast algorithms for top-k approximate string matching. *Proc. AAAI'10*. 2010. P. 1467–1473.
67. Zhang Z., Hadjieleftheriou M., Ooi B.C., Srivastava D. Bed-tree: An all-purpose index structure for string similarity search based on edit distance. *Proc. SIGMOD'10*. 2010. P. 915–926.

68. Morton G.M. A computer oriented geodetic data base; and a new technique in file sequencing. Technical Report. 1966. Ottawa, Canada: IBM Ltd. 20 p.
69. Lu W., Du X., Hadjieleftheriou M., Ooi B.C. Efficiently supporting edit distance based string similarity search using B+-trees. *IEEE Transactions on Knowledge and Data Engineering*. 2014. Vol. 26, N 12. P. 2983–2996.
70. Jagadish H.V., Ooi B.C., Tan K.-L., Yu C., Zhang R. iDistance: An adaptive b+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst*. 2005. Vol. 30, N 2. P. 364–397.
71. Deng D., Li G., Feng J., Li W.-S. Top-k string similarity search with edit-distance constraints. *Proc. ICDE'13.* 2013. P. 925–936.
72. Wang X., Ding X., Tung A.K.H., Zhang Z. Efficient and effective kNN sequence search with approximate n-grams. *Proc. VLDB Endowment.* 2013. Vol. 7, N 1. P. 1–12.
73. Yu M., Wang J., Li G., Zhang Y., Deng D., Feng J. A unified framework for string similarity search with edit-distance constraint. *The VLDB Journal*. 2017. Vol. 26. P. 249–274.
74. Rachkovskij D.A. Formation of similarity-reflecting binary vectors with random binary projections. *Cybernetics and Systems Analysis.* 2012. Vol. 51, N 2. P. 313–323.
75. Рачковський Д.А., Гриценко В.І. Розподілене подання векторних даних на основі випадкових проекцій. Київ: Інтерсервіс, 2018. 216 с.
76. Rachkovskij D.A., Revunova E.G. A randomized method for solving discrete ill-posed problems. *Cybernetics and Systems Analysis.* 2012. Vol. 48, N 4. P. 621–635.
77. Revunova E.G. Model selection criteria for a linear model to solve discrete ill-posed problems on the basis of singular decomposition and random projection. *Cybernetics and Systems Analysis*. 2016. Vol. 52, N 4. P. 647–664.
78. Revunova E.G. Averaging over matrices in solving discrete ill-posed problems on the basis of random projection. *Proc. CSIT'17*. 2017. P. 473–478.
79. McCauley S. Approximate similarity search under edit distance using locality-sensitive hashing. arXiv:1907.01600. 2019.
80. Rubinstein A. Hardness of approximate nearest neighbor search. *Proc. STOC'18.* 2018. P. 1260–1268.

*Надійшла до редакції 22.02.2019*

## Д.А. Рачковський
### ІНДЕКСНІ СТРУКТУРИ ДЛЯ ШВИДКОГО ПОШУКУ СХОЖИХ СИМВОЛЬНИХ РЯДКІВ

**Анотація.** Наведено огляд індексних структур для швидкого пошуку за схожістю об'єктів, що представлені бінарними символьными рядками. Розглянуто індексні структури як для точного, так і для наближеного пошуку за відстанню редагування. Описано індексні структури на основі оберненого індексування, гешування, що зберігає схожість, деревовидних структур. Викладено ідеї алгоритмів, відомих та нещодавно запропонованих.

**Ключові слова:** пошук за схожістю, відстань редагування, найближчий сусід, ближній сусід, індексні структури, обернене індексування, *n*-грами, локально-чутливе гешування, деревовидні структури.

## D.A. Rachkovskij
### INDEX STRUCTURES FOR FAST SIMILARITY SEARCH FOR SYMBOLIC STRINGS

**Abstract.** We survey index structures for fast similarity search of objects represented by symbolic strings. Index structures for both exact and approximate search by the edit distance are considered. Mainly, we present index structures based on inverted indexing, similarity-preserving hashing, tree structures. The ideas of specific algorithms, including the recently proposed ones, are outlined.

**Keywords:** similarity search, edit distance, nearest neighbor, near neighbor, index structures, inverted indexing, *n*-grams, locality-sensitive hashing, treelike structures.

**Рачковский Дмитрий Андреевич,**
доктор техн. наук, ведущий научный сотрудник Международного научно-учебного центра информационных технологий и систем НАН Украины и МОН Украины, Киев, e-mail: dar@infrm.kiev.ua.