электронное научно-техническое издание
НАУКА и ОБРАЗОВАНИЕ
Эл № ФС 77 - 30569. Государственная регистрация №0421100025. ISSN 1994-0408

Применение обобщенного дерева поиска для нечеткого поиска строки

 # 03, март 2011

авторы: Коротков А. Е., Панферов В. В.

Национальный исследовательский ядерный университет «МИФИ»

aekorotkov@gmail.com

Введение

Традиционные запросы к базам данных используют ограниченный набор классов предикатов для поиска. С одной стороны, это органичение вызвано отсутствием реализации необходимых типов данных и поисковых предикатов в системах управления базами данных (СУБД). С другой стороны, это ограничение вызвано ограничениями структур данных, на которых основаны поисковые индексы. Для того чтобы база данных была масштабируемой с точки зрения количества данных, часто требуется индексная структура, поддерживающая поисковый предикат. Традиционно, индексы баз данных поддерживают предикаты равенства и принадлежности линейному диапазону [17]. Помимо этого, планировщик базы данных может раскрывать некоторые более сложные предикаты поисковых запросов в виде комбинации таких предикатов.

Современные приложения баз данных постоянно расширяют свою функциональность. Некоторая часть этой функциональности может быть реализована без расширения функциональности СУБД, но другая часть – нет. Совеременным приложениям баз данных часто требуется поддержка нестандартных типов данных и нестандартных пердикатов поисковых запросов со стороны СУБД. Примером их применения могут служить геринформационные системы (ГИС). В ГИС используются геометрические типы данных, предикаты пересечения и включения и т.д. Для оптимизации поиска в ГИС используются пространственные индексы, такие как R-дерево [18].

Востребованными бывают не только предикаты поисковых запросов на нестандартных типах данных, но также и нестандартные предикаты на страндартных типах данных. Предикаты частичного совпадения текста могут служить примером этому.

Данная работа рассматривает реализацию индекса базы данных для нечеткого поиска строки. Поисковый предикат основан на расстоянии Левештейна между строками. Индекс базы данных был реализован как расширение *GiST* – универсального фреймворка для разработки поисковых индексов.

Нечеткий поиск строки

Под нечетким поиском строки подразумевается такой поиск строки, когда поисковый шаблон или массив данных может подвергаться определенным искажениям. Примером применения нечеткого поиска строки может служить поиск подпоследовательностей ДНК после возможных мутаций [5, 6, 7] или поиск текста, подверженного ошибкам набора и правописания [8, 9, 10].

В данной работе в качестве массива данных рассматривается набор S строк s_i; $S = \{s_1, s_2, \dots, s_n\}$. Поисковым предикатом является утверждение о том, что расстояние между элементом массива данных s_i и искомой строкой p не превышает заданного числа d, т.е. $ed(s_i, p) \leq d$. Расстоянием Левенштейна [13] между строками s_1 и s_2 является минимальное число элементарных операций редактирования, необходимых для того, чтобы преобразовать строку s_1 в строку s_2. Такой поисковый предикат может быть, к примеру, применен для поиска в словаре слова, которое содержит ошибки набора.

Уже существуют различные реализации индексов баз данных для нечеткого поиска текста [4]. В данной работе рассматривается резализация такого индекса на основе обобщенного дерева поиска. Для нечеткого поиска строки уже существует модуль *pg_trgm*, являющийся расширением *GiST* [14]. Однако в *pg_trgm* в качестве меры расстояния между строками используется доля совпадающих триграмм, в то время как в данной работе используется расстояние Левенштейна.

Связанные работы

Есть 4 структуры данных, традиционно используемые для нечеткого поиска строк: дерево суффиксов, массив суффиксов, индексы *k*-грам и *k*-образцов [4]. Отметим, что эти структуры данных используются для решения задачи поиска по тексту, которая является более широкой, чем рассматриваемая в данной работе.

Деревья суффиксов [21] – это широко применяемая в обработке текстов структура данных. Для любой позиции i в тексте T можно определить суффикс T_i. Дерево суффиксов строится по всему множеству суффиксов текста. Каждый лист дерева указывает на суффикс текста, при этом каждый путь от корня к другой вершине определяет подстроку текста, а все суффиксы внутри поддерева начинаются с одной и той же подстроки. Массив суффиксов [22, 23] – это структура данных, которая перечисляет все суффиксы текста в лексографическом порядке.

В индексе *k*-грам длины подстрок ограничены числом *k*. Индекс *k*-грам содержит набор позиций текста для каждой *k*-грамы. Индекс *k*-образцов – это менее требовательная к памяти альтернатива индексу *k*-грам.

Существуют различные подходы к использованию этих структур данных для нечеткого поиска строк: генерация соседей, фрагментирование к точному поиску и промежуточное фрагментирование [4].

Поисковый предикат задается поисковой строкой *p* и максимальным расстоянием *d*. Если *d* – мало, то можно генериовать все строки, расстояние от которых до *p* не превосходит *d*, и использовать точный поиск с этими строками. Этот подход называется генерацией соседей [24, 25].

Искомая строка может быть разбита на фрагменты, один или более из которых должны совпадать точно. Можно и использовать точный поиск для этих фрагментов, и последовательное сравнение после этого. Этот подход называется фрагментированием к точному поиску [3].

Также существует промежуточный подход. Поисковая строка может быть разбита на фрагменты, некоторые из которых должны совпадать с небольшим числом ошибок. Преимущество по сравнению с фрагментарованием к точному поиску состоит в том, что эти фрагменты имеют большую длину. К полученным фрагментам применяется генерация соседей. Этот подход называется промежуточным фрагментированием [27, 20].

Данная работа сфокусирована на разработке индекса, который разделяет строки на кластеры, заданные регулярными выражениями из определенного класса. Этот подход к созданию подобного индекса может дать значимые результаты, как в направлении нечеткого поиска текста, так и в направлении обобщенного дерева поиска.

Обобщенное дерево поиска

Расширения СУБД с точки зрения методов доступа к данным, являются едва ли не самыми сложными для реализации. Проблема состоит не в отсутствии у СУБД необходимых интерфейсов для реализации новых методов доступа, а в том, что эти интерфейсы, как правило, не изолируют саму логику новых методов доступа от операций, необходимых для поддержания используемых структур данных. Для реализации нового метода доступа, в большинстве случаев, нужно иметь дело с упаковкой записей в страницы, поддержанием связей между страницами, чтением страниц в память и так далее. Таким образом, реализация новых методов доступа требует значительных трудозатрат и глубоких знаний в области внутреннего устройства СУБД [2].

Вместо того чтобы создавать новые структуры для доступа к данным, можно расширять существующие структуры данных. Например, *B*+-дерево может быть легко расширено для применения к любым типам данных, на которых можно задать линейных порядок. При этом полученное поисковое дерево будет поддерживать поиск по предикатам равенства и принадлежности линейному диапазону. Однако, таким образом достигается расширяемость с точки зрения индексируемых типов данных, но не с точки зрения поисковых запросов. Аналогично с R-деревом: вне зависимости от индексируемого типа данных оно может поддерживать поиск только по предикатам равенства, пересечения и включения. [1]

Обобщенное дерево поиска представляет более общее решение данной проблемы. *GiST* – это структура данных, расширяемая как с т.з. индексируемых типов данных, так и с т.з. ускоряемых поисковых запросов. *GiST* определяет набор из интерфейсных функций, реализации которых достаточно для создания поискового индекса. При этом эта реализация зависит от индексируемого типа данных, но абстрагирована от страниц данных, записей, конкурентного доступа, восстановления после сбоев и так далее. Таким образом, для реализации поискового индекса с помощью *GiST* не требуется писать код, отвечающий за поддержание структуры данных[15]. Поимо этого, *GiST* обобщает большинство существующих поисковых деревьев. К примеру, *B*+-дерево и *R*-дерево могут быть реализованы как расширения *GiST* [2].

На текущий момент *GiST* полностью реализован в открытой постреляционной *СУБД PostgreSQL*. Хотя результаты исследований о *GiST* используются в большинстве коммерческих СУБД, таких как *Oracle* и *DB2*. В данной работе было выбрано *GiST* по следующим причинам:

- Предоставление открытого и лицензионно-свободного решения для нечеткого поиска текста
- Результаты данной работы могут быть ценными для развития *GiST*, т.к. в ней реализовано новое применение *GiST*
- Простота реализации расширения *GiST*

Применение *GiST* для нечеткого поиска строки

В качестве меры рассмотряния между стоками используется расстояние Левенштейна. Расстояние Левенштейна между двумя строками – это минимальное число элементарных операций, необходимых для приведения одной строки к другой. Элементарными считаются следующие операции:

- Вставка произвольного символа в произвольную позицию строки
- Замена произвольного символа строки на любой другой символ
- Удаление произвольного символа строки

Для того чтобы рассчитать расстояние между строками ***a*** и ***b***, может быть использован алгоритм выравнивания двух последовательностей [11, 19]. В данной работе рассматривается две модификации этого алгоритма, поэтому его следует рассмотреть подробно.

Путь строки $\boldsymbol{a} = a_1a_2\ldots a_n$ и $\boldsymbol{b} = b_1b_2\ldots b_m$ имеют длины n и m. Выравнивание происходит, когда в строки вставляются пустые символы «–» таким образом, что новые строки имеют одинаковую длину L. После вставки символов «–» $\boldsymbol{a} = a_1a_2\ldots a_n$ становится $\boldsymbol{a}^* = a_1^*a_2^*\ldots a_n^*$ и $\boldsymbol{b} = b_1b_2\ldots b_m$ становится $\boldsymbol{b}^* = b_1^*b_2^*\ldots b_m^*$. Выравнивание задаётся двумя такими последовательностями, написанными одна под другой:

$$\begin{matrix} a_1^* & a_2^* & \ldots & a_L^* \\ b_1^* & b_2^* & \ldots & b_L^* \end{matrix}$$

Расстояние между **a** и **b** вводится как:

$$D(a,b) = \min \sum_{i=1}^{L} d(a_i^*, b_i^*)$$

Где d(a,b) представляет собой расстояние между символами a и b. В случае расстояния Левештейна d(a,b) определяется следующим образом:

$$d(a,-) = d(-,a) = 1$$
$$d(a,b) = \begin{cases} 0, a = b \\ 1, a \neq b \end{cases}$$

Матрица D вводится как расстояние между префиксами строк **a** и **b**.

$D_{i,j} = D(a_1a_2\ldots a_i, b_1b_2\ldots b_j)$

Таблица 1. Матрица выравнивания

	–	b_1	b_2	…	b_m
–	$D_{0,0}$	$D_{0,1}$	$D_{0,2}$	…	$D_{0,m}$
a_1	$D_{1,0}$	$D_{1,1}$	$D_{1,2}$	…	$D_{1,m}$
a_2	$D_{2,0}$	$D_{2,1}$	$D_{2,2}$	…	$D_{2,m}$
…	…	…	…	…	…
a_n	$D_{n,0}$	$D_{n,1}$	$D_{n,2}$	…	$D_{n,m}$

Алгоритм двумерного выравнивания сводится к заполнению матрицы D по следующим правилам:

$$D_{0,0} = 0$$

$$D_{0,j} = \sum_{k=1}^{j} d(-, b_k) \quad D_{i,0} = \sum_{k=1}^{i} d(a_k, -)$$

$$D_{i,j} = min\{D_{i-1,j} + d(a_i, -), D_{i-1,j-1} + d(a_i, b_j), D_{i,j-1} + d(-, b_j)\}$$

Правый нижний элемент матрицы содержит расстояние между строками.

$D_{n,m} = D(a_1 a_2 \ldots a_n, b_1 b_2 \ldots b_m) = D(\boldsymbol{a}, \boldsymbol{b})$

Поисковый предикат

Был выбран поисковый предикат *П(x) = (levenshtein(s,x) ≤ d)*, где *levenshtein* – расстояния Левештейна, ***s*** – заданная строка, ***d*** – заданное неотрицательное число. Этот предикат истинен для тех и только тех строк, для которых расстояние Левенштейна до заданной строки не превышает заданного числа. Если рассматривать расстояние по Левенштейну как метрику (а это можно сделать, т.к. для него выполняется неравенство треугольника), то множество строк, удовлетворяющих этому предикату, будет представлять собой шар с центром в строке ***s*** и радиусом ***d***.

Для реализации данного предиката в СУБД *PostgreSQL* был реализован тип *word_query* и оператор @@ между типом *text* (встроенный текстовый тип) и типом *word_que*ry. Тип *word_query* представляет собой пару *(**s**, **d**)*. В текстовом представлении word_query **s** и **d** разделены знаком «;». То есть пара («собака», 2) будет выглядеть как «собака;2». Таким образом, запрос по поиску в столбце *word* таблицы *dictionary* слов, расстояние от которых до слова «собака» не превышает 2, будет выглядеть следующим образом:

*SELECT * FROM dictionary WHERE word @@ "собака;2"::word_query;*

Предикат узла дерева

Выбор предиката узла дерева очень важен при реализации расширения GiST. Все характеристики получившегося дерева ключевым образом зависят от выбранного предиката. В данной работе был выбран предикат соответствия определенному подклассу регулярных выражений. Выбранный подкласс регулярных выражений можно описать следующим образом. Каждое

выражение представляет собой конкатенацию n (n – целое неотрицательное число) подвыражений. Каждое подвыражение может быть задано одним из следующих образов:

1) Один из m символов (выражение вида «$[a_1a_2...a_m]$»)

2) Один из m символов или пустая строка (выражение вида «$[a_1a_2...a_m]$?»)

3) Произвольный символ или пустая строка (выражение вида «.?»)

БНФ для рассматриваемого класса регулярных выражений выглядит следующим образом:

<выражение> ::= <выражение><подвыражение>|<пусто>

<подвыражение> ::= .?|[<набор_символов>]|[<набор_символов>]?

<набор_символов> ::= <набор_символов><символ>|<символ>

<символ> ::= *a1*|*a2*|...|*ak*

Везде далее в данной работе под термином «регулярное выражение» подразумевается регулярное выражение из данного класса, если явно не упомянуто другое.

Реализация интерфейсных методов *GiST*

Для создания расширения *GiST* необходимо определить 7 интерфейсных методов:

1) *compress* и *decompress* – отвечают за компрессию и декомпрессию ключей (в оперативной памяти ключи должны храниться в виде, удобном для работы с ними, однако для хранения их на диске часто бывает целесообразно сжать их)

2) *consistent* – вычисляет совместимость ключа узла дерева с поисковым запросом (за счет этого метода и производится поисковая оптимизация, если предикат узла дерева не совместим с поисковым предикатом, то всё поддерево можно не проверять)

3) *union* – возвращает объединение двух ключей (все значения удовлетворявшие предикатам исходных ключей должны удовлетворять и предикату результирующего ключа)

4) *penalty* – возвращает меру увеличения исходного ключа при добавлении к нему нового (это значение должно отражать меру расширения множества значений, удовлетворяющих предикату ключа)

5) *picksplit* – разделяет массив ключей на два, при этом желательно, чтобы объединенные ключи двух частей массива имели минимальный размер (под размером ключа здесь понимается мощность множества значений, удовлетворяющих предикату ключа)

6) *same* – проверяет, совпадают ли два ключа

Компрессия ключей перед записью на диск не осуществлялась, поэтому реализация методов *compress* и *decompress* была тривиальной. Реализация *same* также была тривиальной,

поскольку регулярные выражения хранятся однозначным образом. Методы *penalty* и *picksplit* были определены на основе операций объединения ключей и оценки размера ключей. Метод penalty вычисляет разность между размером объединения и исходным размером ключа. Метод *picksplit* реализован на основе квадратичного алгоритма Гутмана для разделения узлов R-дерева. Для методов union и consistent использовались разновидности алгоритма двумерного выравнивания, рассмотренные далее.

Метод *Consistent*

Для реализации метода consistent используется разновидность двумерного выравнивания[11], которая позволяет найти минимальное расстояние по Левенштейну между строкой, удовлетворяющей регулярному выражению, и строкой поискового запроса. Находимое минимальное расстояние можно записать в виде выражения:

$\boldsymbol{d} = min\{levenstein(\boldsymbol{s},\boldsymbol{x})|\boldsymbol{x}\sim\boldsymbol{r}\}$

где ***s*** – строка поискового запроса, ***r*** – регулярное выражение, «~» - оператор соответствия регулярному выражению.

Решение о совместимости между запросом и регулярным выражением принимается на основе сравнения результирующего значения и максимального расстояния в поисковом запросе.

Для вычисления минимального расстояния используется модификация алгоритма выравнивания двух строк. В этой модификации строится выравнивание $\boldsymbol{s} = s_1 s_2 \ldots s_n$ и $\boldsymbol{r} = r_1 r_2 \ldots r_m$. Минимальное рассотояние между строкой, которая соответствует **r** и **s** вычисляется выражением:

$$D(s,r) = min \sum_{i=1}^{L} d(s_i^*, r_i^*)$$

Где $d(\boldsymbol{s},\boldsymbol{r})$ определяется следующим образом:

$$d(s,r) = \begin{cases} 0, \text{если } r \text{ допускает } s \\ 1, \text{если } r \text{ не допускает } s \end{cases}$$

$$d(s,-) = 1$$

$$\mathrm{d}(-,r) = \begin{cases} 0, \text{если } r \text{ допускает пустую строку} \\ 1, \text{если } r \text{ не допускает пустую строку} \end{cases}$$

В остальном алгоритм совпадает с исходным. Докажем, что результат выполнения алгоритма действительно является минимальным расстоянием между строкой, которая удовлетворяет ***r***, и ***s***.

Доказательство. Пусть строка ***x*** длины k – это строка, удовлетворяющая ***r***, и имеющая при этом минимальное расстояние до ***s***. Если ***x*** удовлетворяет ***r***, то существует выравнивание между ***x*** и ***r***. В этом выравнивании некоторые из r_i могут быть пропущены, если r_i допускает пустую строку.

$$\begin{array}{cccc} r_1^* & r_2^* & \dots & r_k^* \\ x_1 & x_2 & \dots & x_k \end{array}$$

Также существует оптимальное выравнивание между ***x*** и ***s***.

$$\begin{array}{cccc} s_1^* & s_2^* & \dots & s_L^* \\ x_1^* & x_2^* & \dots & x_L^* \end{array}$$

Составим объединенное выравнивание, добавив в ***r**** пустые элементы в тех местах, где их содержит ***x****.

$$\begin{array}{cccc} s_1^* & s_2^* & \dots & s_L^* \\ x_1^* & x_2^* & \dots & x_L^* \\ r_1^{**} & r_2^{**} & \dots & r_L^{**} \end{array}$$

Из определения d очевидно, что $d(s_i^*, r_i^{**}) \le d(s_i^*, x_i^*)$

$$v = \sum_{i=1}^{L} d(s_i^*, r_i^{**}) \le \sum_{i=1}^{L} d(s_i^*, x_i^*)$$

Поскольку в r^* пропущены только подвыражения, допускающие пустую строку, v не может быть меньше, чем расстояние между ***s*** и ***r***, вычисленное рассмотренным выравниванием. Таким образом, минимальное расстояние между строкой, удовлетворяющей ***r***, и ***s*** не может быть меньше, чем расстояние, вычисленное рассмотренным выравниванием. Также очевидно, что оно не может быть больше, поскольку легко можно построить пример строки, удовлетворящей ***r***, расстояние от которой до ***s*** равно расстоянию, вычисленному по алгоритму.

Рассмотрим пример. Найдём минимальное расстояние между строкой “дом” и выражением “«[дк][узо]?[оч][мх]»”.

Таблица 2. Пример матрицы выравнивания для нахождения расстояния между строкой и регулярным выражением

		[дк]	[узо]?	[оч]	[мх]
	0	1	1	2	3
д	1	0	0	1	2
о	2	1	0	0	1
м	3	2	1	1	0

Метод *Union*

В методе union используется другая модификация алгоритма выравнивания двух строк. Используется следующая мера расстояния между подвыражениями:

$$d(r_1,r_2) = \frac{u_1}{c+u_2} + \frac{u_2}{c+u_1},$$

Где u_1 – число уникальных символов в первом подвыражении (число символов, которые допустимы первым подвыражением и не допустимы вторым подвыражением) , u_2 - число уникальных символов во втором подвыражении, а c – число общих символов в подвыражениях. При этом пустая строка рассматривается как отдельный символ.

Случай, когда одно из подвыражений равно “.?”, следует рассмотреть отдельно (когда оба подвыражения равны “.?”, очевидно, что расстояние следует принять равным нулю). В этом случае используется следующая мера расстояния:

$$d(".?",".?") = 0$$

$$d(".?",r_2) = \frac{n-c_2}{n}$$

Где c_2 – число символов во втором подвыжении n – общее число символов в алфавите.

В случае пропуска одного из подвыражений используется следующая мера расстояния:

$$d(r,-) = d(-,r) = 1 + \frac{u}{u+c},$$

Где $u = 0$, если пустая строка допускается подвыражением, $u = 1$, иначе; c – число символов в подвыражении.

В данной модификации выравнивания нужно не только вычислить расстояние, но и построить объединение выражений. Рассмотрим выравнивание выражений $\boldsymbol{a} = a_1 a_2 \ldots a_n$ и $\boldsymbol{b} = b_1 b_2 \ldots b_m$.

$$\begin{array}{cccc} a_1^* & a_2^* & \ldots & a_L^* \\ b_1^* & b_2^* & \ldots & b_L^* \end{array}$$

Результирующее выражение $\boldsymbol{c} = c_1 c_2 \ldots c_m$ может быть вычислено как $c_i = u(a_i, b_i)$, где u – это функция вычисления подвыражений.

$$u(".?", a) = u(a, ".?") = ".?"$$

$$u("[a_1 a_2 \ldots a_n]","[b_1 b_2 \ldots b_m]") = "[a_1 a_2 \ldots a_n b_1 b_2 \ldots b_m]"$$

$$u("[a_1 a_2 \ldots a_n]","[b_1 b_2 \ldots b_m]?") = u("[a_1 a_2 \ldots a_n]?","[b_1 b_2 \ldots b_m]")$$

$$= u("[a_1 a_2 \ldots a_n]?","[b_1 b_2 \ldots b_m]?") = "[a_1 a_2 \ldots a_n b_1 b_2 \ldots b_m]?"$$

В операции объединения подвыражений, если число символов в результирующем подвыражении превысит пороговое значение k, то это подвыражение заменяется на “.?”. Эта замена производится для того, чтобы уменьшить длину подвыражения и улучшить производительность.

Рассмотрим в качетсве пример процесс объединения “[абв][где][жз]?.?” И “.?[аг][бде]?з?з?”. Результирующая матрица представлена ниже.

Таблица 3. Пример матрицы выравнивания объединения двух подвыражений

	-	.?	[аг]	[бде]?	з?	з?
-	0,00	1,00	2,33	3,33	4,33	5,33
[абв]	1,25	0,88	1,88	2,88	3,88	4,88
[где]	2,50	2,13	1,75	2,75	3,75	4,75
[жз]?	3,50	3,13	2,75	2,63	3,63	4,63
.?	4,50	3,50	3,75	3,59	3,53	4,53

Результирующее выравнивание выглядит следующим образом:

выражение1	[абв]	[где]	[жз]	.?	–
выражение2	.?	[аг]	[бде]?	з?	з?
объединение	.?	[агде]	[бдежз]	.?	з?

Объединенное выражение: “.?[агде][бдежз]?.?з?”.

Тестирование производительности

Для того, чтобы провести синтетическое тестирование индекса базы данных, нужно решить две задачи: получить набор тестовых данных и набор тестовых запросов. В качестве тестовых данных в данной работе использовался словарь английского языка, включающий в себя 61505 слов.

Далее генерировались тесты. Генерируемые тесты делятся на две категории: тесты со случайно сгенерированными словами и тесты с внесением случайных искажений в существующие слова.

В тестах со случайно сгенерированными словами генерировалась последовательность случайных букв латинского алфавита длиной *n* от 3 до 18 символов. Далее генерировалось случайное число от 1 до $\lceil n/5 \rceil$, которое использовалось в качестве радиуса поискового запроса. Соотношение $\lceil n/5 \rceil$ использовалось в качестве верхней границы для того, чтобы радиус поискового запроса не был слишком большим по сравнению с длиной слова.

В тестах с внесением случайных искажений в существующие слова выбиралось случайное исходное слово из словаря длиной *n*. Далее в это слово вносилось от 1 до $\lceil n/5 \rceil$ случайных элементарных изменений (вставка, замена или удаление символа). А после этого случайным образом выбирался радиус поискового запроса длиной от 1 до $\lceil n/5 \rceil$.

Результаты проведенных тестов представлены в таблицах.

В таблице 4 представлены результаты запросов на основе внесения случайных изменений в существующие слова. В этой таблице отображена зависимость среднего ускорения (У), среднего времени поиска без использования индекса (БИ) и среднего времени поиска с использованием индекса (И) от радиуса поискового запроса и числа изменений, внесенных в исходное слово. При этом ускорение (У) вводилось как $У = T_{би} / T_{и}$, где $T_{би}$ и $T_{и}$ – это время поиска без использования индекса и с использованием индекса соответствен. Как видно из таблицы коэффициент ускорения увеличивался с увеличением числа вносимых изменений в исходное слово и уменьшался с увеличением радиуса поискового запроса.

Таблица 4. Результаты тестирования индекса с внесением случайных искажений в существующие слова

	Радиус поискового запроса									Среднее		
	1			2			3					
Искажение	У	БИ	И	У	БИ	И	У	БИ	И	У	БИ	И
0	2,29	124	66	1,36	141	115	1,529	186	132	1,72	150	104
1	2,91	124	58	1,54	143	108	1,751	180	126	2,07	149	98
2	3,52	142	63	1,62	142	102	1,851	184	121	2,38	156	95
3	10,7	180	36	6,66	187	063	2,549	182	116	6,64	183	72
Среднее	4,85	142	56	2,80	153	097	1,920	183	124	3,19	160	92

В таблице 5 представлены результаты тестирования поисковых запросов со случайно сгенерированными словами. В этой таблице представлены те же данные, что и в таблице выше, но они зависят от длины генерируемых слов и радиуса поискового запроса. Из таблицы видно, что коэффициент ускорения возрастает с увеличением длины генерируемого слова и уменьшается с увеличением радиуса поиска.

Таблица 5. Результаты тестирования индекса со случайными словами

	Радиус поискового запроса												Среднее		
	1			2			3			4					
Длина	У	БИ	И	У	БИ	И	У	БИ	И	У	БИ	И	У	БИ	И
3	5,3	80	16										5,3	80	16
4	4,5	89	24										4,5	89	24
5	4,6	99	29										4,5	99	29
6	4,6	109	33	1,8	109	68							3,2	109	51
7	5,7	119	30	2,2	119	71							4,0	119	50
8	6,6	128	27	2,7	128	63							4,7	128	45
9	8,4	139	22	3,2	138	58							5,8	138	40
10	11	148	16	6,1	148	35							8,5	148	26
11	12	157	14	7,3	156	28	3,4	157	58				7,7	157	33
12	16	166	11	9,8	167	20	5,4	167	47				10,5	166	26
13	21	175	9,4	13,2	174	14	8,5	175	25				14,1	175	16
14	28	183	7,5	15,4	184	13	9,3	184	23				17,6	184	14
15	49	193	4,9	20,3	192	10	12,7	194	17				27,4	193	11
16	88	201	2,8	32,3	202	7,4	16,1	200	13	10,58	202	20	36,7	201	11
17	201	211	1,6	63,6	210	4,4	25,5	209	10	13,74	211	16	76,1	210	7,9
18	353	220	0,8	116	218	2,2	43,6	218	5,8	17,90	220	13	132	219	5,4
Среднее	51,3	151	16	22,6	165	30	15,6	188	25	14,07	211	16	25,9	179	22

Результаты тестирования показали, что разработанный индекс работоспособен и способен дать существенный прирост производительности. При этот индекс показывает наилучшую про-изводительность, когда обрабатываются запросы, которые запрашивают области далекие от реально хранящихся данных. Также производительность индекса лучше, когда запрашиваются более узкие радиусы поисковых запросов. Обе эти тенденции можно объяснить долей узлов дерева, которую можно отсечь при поиске.

Заключение

Данная работа повящена разработке поискового индекса на основе *GiST* для нечеткого поиска текста. Был разработан поисковый индекс, позволяющий искать в массиве строк $S = (s_1, s_2, \ldots s_n)$ такие s_i, что $levenshtein(s_i, p) \leq d$. Было проведено тестирование индекса, в котором в качестве массива данных выступал словарь английского языка объемом 61505 слов. Тестирование показало ускорение поиска в 3,2 раза для тестов на основе изменения существующих слов и в 26 раз для тестов на основе случайно сгенерированных слов.

Возможны следующие направления для дальнейших исследований:

1) Исследовать разработанный индекс на различных массивах данных. Выяснить, на каких массивах данных данный индекс может давать значимый прирост производительности поиска, а на каких – нет, и почему.

2) Улучшить результаты индекса. Для улучшения результатов, можно попробовать изменить реализацию некоторых интерфейсных методов *GiST*, в частности, метода *PickSplit*. Также, возможно, следует изменить класс регулярных выражений, используемый в качестве предиката узла дерева.

3) Применить разработанный индекс для поиска с другими поисковыми предикатами. С расстоянием редактирования, отличным от расстояния по Левенштейну. Например, можно использовать расстояние редактирования, которое рассматривает замену соседних букв как одну ошибку. Такой поисковый предикат можно применять для поиска слов с ошибками набора, так как замена соседних букв является распространенной опечаткой. Также можно рассматривать оптимизацию запросов, в которых регулярное выражение выступает в качестве поискового предиката.

Ссылки

1. J.M. Hellerstein, J.F. Naughton и A. Pfeffer Generalized Search Trees for Database Systems // Proc. Int'l Conf. on Very Large Data Bases – 1995. – №21. – С. 562-573.
2. Paul M. Aoki, Generalizing “search” in generalized search trees // Proc. Int'l Conf. on Data Engineering – 1998. – №14.
3. G. Navarro A guided tour to approximate string matching // ACM Computing Surveys – 2001. – Т.33, №1. – С.31 – 88.
4. G. Navarro, R. Baeza-Yates, E. Sutinen, J. Tarhio Indexing methods for approximate string matching // IEEE Data Engineering Bulletin – 2001. – №24. – С. 19-27.

5. SF Altschul, TL Madden, AA Schaffer и др. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs // Nucleic Acids Res. – 1997. – T.25, №17. – C. 3389-3402.
6. Zemin Ning, Anthony J. Cox, и James C. Mullikin METHODS: SSAHA: A Fast Search Method for Large DNA Databases // Genome Res. – 2001. – №11. – C. 1725-1729.
7. Maria B. Chaley, Eugene V. Korotkov, and Konstantin G. Skryabin Method Revealing Latent Periodicity of the Nucleotide Sequences Modified for a Case of Small Samples // DNA Res. – 1999. – №6. – C.153-163.
8. Sun Wu, Udi Manber Fast text searching: allowing errors // Communications of the ACM – 1992. – T.35, №10. – C.83-91.
9. James L. Peterson Computer programs for detecting and correcting spelling errors // Communications of the ACM archive // 1980. – T.23,№12. – C.676-687.
10. Fred J. Damerau A technique for computer detection and correction of spelling errors // 1964. – T.7, №3. – C.171-176.
11. Robert A. Wagner, Michael J. Fischer The String-to-String Correction Problem // Journal of the ACM – 1974. – T.21, №1. – C.168-173.
12. R. Mc. Naugnton, H. Yamada Regular Expressions and State Graphs for Automata // IRE Trans. Electronic Computers – 1960. – C.39-47.
13. В. И. Левенштейн Двоичные коды с исправлением выпадений, вставок и замещений символов // Советская физика – 1966. – C.707-710.
14. R. C. Angell, G. E. Freund, P. Willett Automatic spelling correction using a trigram similarity // Information Processing and Management – 1983. – T.19,№4 – C.255-262.
15. M. Kornacker, C. Mohan, J.M. Hellerstein Concurrency and recovery in generalized search trees // In Proceedings of the ACM-SIGMOD Conference – 1997. – T.26,№2 – C.62-72.
16. Gonzalo Navarro, Ricardo Baeza-Yates A New Indexing Method for Approximate String Matching // Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching – 1999 – №1645 – C.163-185.
17. Douglas Comer The Ubiquitous B-Tree // Computing Surveys – 1979. – T.11,№2 – C.121-137.
18. Antonin Guttman, R-Trees: A Dynamic Index Structure For Spatial Searching // In Proc. ACM SIGMOD International Conference on Management of Data – 1984. – C.47-57.
19. Michael S. Waterman Introduction to computational biology: maps, sequences and genomes, – M.: CRC Press, 1995. – 432 с.

20. Navarro, R. Baeza-Yates A practical q-gram index for text retrieval allowing errors // CLEI Electronic Journal – 1998.
21. Apostolico, Z. Galil Combinatorial Algorithms on Words // Springer-Verlag – 1985.
22. T. Takaoka Approximate pattern matching with samples // Proc. 5th Int'l. Symp. on algorithms and Computation – 1994. – №834. – C.234-242.
23. G. Gonnet, R. Baeza-Yates и T. Snider Information Retrieval: Data Structures and Algorithms // Prentice-Hall – 1992. – C.66-82.
24. T. Takaoka Approximate pattern matching with samples // In Proc. 5th Int'l. Symp. on Algorithms and Computation – 1994. – №834 – C.234-242.
25. E. Ukkonen Finding approximate patterns in strings // Journal of Algorithms – 1985. – №6. – C.132-137.
26. R. Baeza-Yates and G. Gonnet A fast algorithm on average for all-against-all sequence matching // In Proc. 6th Symp. on String Processing and Information Retrieval – 1990.
27. S. Muthukrishnan and C. Sahinalp Approximate nearest neighbors and sequence comparisons with block operations // In Proc.of ACM Symposium on Theory of Computing – 2000. – C.416-422