Бойцов Л.М.

ИСПОЛЬЗОВАНИЕ ХЕШИРОВАНИЯ ПО СИГНАТУРЕ ДЛЯ ПОИСКА ПО СХОДСТВУ

Введение

За последние 30-40 лет количество информации, представленной в электронном виде, существенно возросло и продолжает возрастать экспоненциально. В связи с этим особую значимость получила задача полнотекстового поиска в этом постоянно увеличивающемся массиве данных. Первые информационно-поисковые системы (ИПС) появились более тридцати лет назад, и с тех произошли существенные изменения, как в поисковых алгоритмах, так и в техническом оснащении. Современные поисковые системы (ПС) «научились» автоматически собирать информацию в интернете, учитывать морфологические особенности и производить оценку значимости (релевантности или соответствия документа запросу) найденных документов. Тем не менее, такая проблема, как поиск по сходству, до сих пор остается по ряду причин недостаточно изученной.

Так, например, поиск по сходству занимает больше времени чем поиск на точное равенство. Кроме того, с одной стороны использование поиска по сходству увеличивает полноту выборки, то есть количество действительно релевантных документов, а с другой стороны увеличивает объем «шума». Однако в некоторых случаях он просто необходим, например, если пользователь не знает точного написания редкого слова или термина.

Поэтому целью данной работы является рассмотрение и сравнение методов поиска на неточное равенство, в том числе, методов поиска термина (ключевого слова) в словаре. В первых двух частях статьи содержится обзор основных поисковых методов, используемых в настоящее время, а также обзор методов поиска в словаре по сходству. В третьей и четвертой частях я описываю разработанный мною метод поиска в словаре: хеширование по сигнатуре, и привожу оценки эффективности данного алгоритма. В третьей части приводится также сравнительная характеристика алгоритмов индексирования словаря. В последнем разделе можно прочесть о результатах сравнения производительности созданного мною макета поисковой системы, основанной на сжатых инвертированные файлах и хешировании словаря по сигнатуре, с программой glimpse.

Нужно отметить, что проблема поиска на неточное равенство очень многообразна: это может быть поиск в массиве текстовой информации [9], в базе данных ДНК, на множестве графов или изображений. Сразу хочу отметить, что в данной работе рассматриваются только текстовые

поисковые системы. Текстовым документом я буду называть список слов в некотором алфавите (это могут быть слова естественного языка или языка программирования), полагая при этом, что слова имеют относительно небольшую длину: 1-20 символов.

1. Обзор поисковых методов

Наиболее распространенные в настоящее время методы поиска, используемые информационно-поисковыми системами (ИПС), можно разделить на две группы:

- поиск по ключевым словам (терминам)
- кластерные методы

В первую группу попадает, по-видимому, абсолютное большинство современных ИПС, в том числе и ПС, которые обрабатывают запросы на естественных языках (Яндекс, Yahoo, и.т.д.). В процессе поиска такие системы, как правило, производят выборку всех документов, содержащих хотя бы одно ключевое слово (грамматическую форму данного слова или синонима), а затем ранжируют эти документы по убыванию степени соответствия (релевантности) поисковому запросу.

Кластерные методы основаны на следующем наблюдении: близкие в смысле значения функции соответствия документы обычно релевантны одним и тем же поисковым запросам. Таким образом, разбив все документы на группы или кластеры и выбрав или построив автоматически в каждой группе характерного представителя: центроид кластера, можно сравнивать запрос не с каждым документом по отдельности, а сначала только с центроидами. Если центроид релевантен запросу, то нужно продолжать поиск внутри кластера, нет - перейти к рассмотрению другого множества документов.

Поиск по ключевым словам (терминам). Обратимся к более подробному описанию поиска по ключевым словам. Ввиду эффективности и простоты реализации данный метод получил наибольшее распространение. Основными методами организации индекса для поиска по ключевым словам являются:

- инвертированные файлы (ИФ) [2,5,6,9]
- сигнатурные файлы (СФ) [5,6,9].

ИФ являются аналогом предметного указателя в конце книги. ИФ - это множество пар: ключевое слово, адрес вхождения ключевого слова в документ. Детализация адреса термина может быть различной: вхождение слова может быть задано на уровне документа, абзаца, или даже на

уровне точного местоположения в тексте. Основным недостатком ИФ является его большой объем. Если не применять специальные методы кодирования списков вхождений, то его размер может превышать размер информационного массива документов в 2-3 раза.

Алгоритмы СФ разрабатывались как альтернатива ИФ с более компактным индексом. В простейшем варианте СФ выбирается хэш-функция $f(x)$, отображающая множество ключевых слов в отрезок целых чисел от 1 до n, и каждому документу ставится в соответствие битовый вектор (или по другому сигнатура документа, компоненты которого могут принимать значения только 0 и 1), у которого i-ый компонент равен 1, если в документе есть ключевое слово w такое, что $f(w)=i$, и нулю в противном случае. В процессе поиска по ключевому слову w последовательно перебираются все сигнатуры и находятся те, у которых компонент с номером $f(w)$ равен 1. Очевидно, что только такие документы могут содержать ключевое слово w, поэтому они последовательно сканируются на предмет проверки соответствия запросу.

Размер СФ составляет около 50% от размера исходных данных. В статье [6] приводятся результаты, которые подтверждаются также моими собственными экспериментами, что сжатый ИФ занимает меньше места, чем СФ, а поиск по нему происходит в несколько раз быстрее. Так, например, для статической коллекции размером 200М индекс занимает 7% от исходного объема, а время поиска на точное равенство в зависимости от объема выборки колеблется в пределах от 0.001 до 0.1 секунды. В данной статье я собираюсь продемонстрировать, что ИФ также очень эффективен и при поиске на неточное соответствие.

Рассмотрим типы запросов в ИПС. Типы поисковых выражений можно классифицировать по нескольким параметрам. Прежде всего, их можно разделить на контекстно-зависимые и контекстно-независимые. В случае контекстно-независимых запросов предполагается, что документ состоит из некоторых неделимых компонентов: в большинстве случаев, слов. Контекстно-зависимые запросы могут быть очень разнообразны, начиная с поиска подстроки в документе, и заканчивая условием на наличие определенных ключевых слов в одном предложении, абзаце, и.т.д.

Язык задания поискового запроса может быть, как формальным (булевым), так и естественным. По этому признаку типы запросов можно разделить на **булевы** или **логические** и так называемые **ранжированные**.

В случае булева поиска пользователь задает условие вхождения ключевых слов в документ в виде булевской функции. При поиске по ранжированным запросам текст запроса обычно записывается на естественном языке. ИПС разбирает запрос, выделяя в нем слова, или устойчи-

вые выражения, расширяет запрос синонимами терминов поискового запроса и производит поиск всех документов, содержащих хотя бы один термин. Найденные документы ранжируются по степени релевантности запросу.

Возможны различные варианты сопоставления ключевых слов запроса и документа:

- точное соответствие.
- с учетом изменяемости слова.
- по сходству.

Возвращаясь еще раз к тому, что в данной статье понимается под поиском на неточное соответствие, или поиском по сходству, я хочу уточнить, что под сходством я понимаю сходство терминов запроса и слов в текстовых документах некоторого информационного массива.

Название первого пункта говорит само за себя: при поиске на точное равенство ищутся только те документы, в которые ключевое слово входит точно в той форме, которую указал в запросе пользователь.

К сожалению, поиск на точное соответствие не позволяет найти слово, если в документе оно встречается в другом падеже, спряжении, и.т.д. Не удивительно, что большинство ИПС осуществляет поиск с учетом изменяемости слова. Современные системы также отождествляют такие слова как **делать** и **дело,** осуществляя поиск по словоформе. Поиск по словоформе и поиск с учетом изменяемости слова, собственно говоря, уже являются одним из вариантов поиска по сходству, учитывающего только определенный тип ошибок.

Пользователь не всегда набирает термины в запросе правильно. Иногда, чтобы найти нужный документ, ИПС должна идентифицировать термины словаря «похожие» на термин запроса. Как я уже отмечал, поиск с учетом изменяемости слов уже является поиском по сходству. Однако наибольшие трудности возникают в случае поиска, как с учетом изменяемости слова, так и с учетом возможных ошибок в поисковом образце. Ключевым моментом поиска по сходству является выбор меры степени «похожести». Я буду использовать метрику Левенштайна, которую также называют расстоянием редактирования.

Определение 1. *Будем называть строкой или словом последовательность символов некоторого алфавита A.*

Определение 2. *Расстоянием редактирования Левенштайна* $L(u,v)$ *между строками u и v будем называть минимальное количество вставок, замен и удалений символов, необходимых для преобразования u в v.*

Выбор в качестве функции расстояния метрики Левенштайна обусловлен двумя причинами. Во-первых, расстояние Левенштайна формализует интуитивное понятие об «ошибке», а во-вторых, существует мно-

жество алгоритмов эффективного его вычисления [4,8,10,15,18]. Фактически, при поиске требуется не столько значение расстояния между строками u и v, сколько знание превышает ли $L(u,v)$ некоторое наперед заданное пороговое значение. А для проверки условия $L(u,v)<k$ (см. [15]) для небольших строк (не более 32-64 символов) требуется $O((k+1)\times(|u|+|v|))$ операций, где $|x|$ - длина строки x. Это означает, что проверка условия слово u «похоже» на слово v может происходить очень эффективно, если оба слова загружены в оперативную память компьютера. Опыт использования программы agrep [15] показывает, что время, затрачиваемое на сравнение слов, на порядок меньше времени считывания с диска, в том числе, при поиске по сходству. Таким образом, оптимизация поиска по сходству почти напрямую зависит от сокращения объема дисковых операций.

Как я уже говорил, для поиска на неточное равенство также можно использовать ИФ. Для этого в словаре нужно обязательно хранить **все** термины, содержащиеся в массиве документов. Если поиск происходит с учетом изменяемости слова по падежам, или спряжениям, то индексирующий алгоритм должен, где это возможно, вместе с термином w, найденном в некотором документе, заносить в словарь все формы этого слова. Например, если программа находит в документе слово **кот** и обнаруживает, что в словаре оно не содержится, то она должна занести в словарь все формы этого слова в единственном и множественном числе: **кот, кота, коту,** и.т.д. Поскольку в поисковом запросе слово **кот** может быть указано с ошибкой, например: **котова**, программа не сможет корректно определить, как поисковый термин должен изменяться по падежам. В данном случае, если в словаре будет только форма именительного падежа, то при поиске на одну ошибку слово не будет найдено.

Одним из ключевых моментов поиска по терминам является методика организации словаря. Особенно это актуально в случае поиска на неточное равенство, который, как правило, требует больших временных затрат.

2. Поиск в словаре по сходству

В настоящее время для поиска термина в словаре применяют, в основном, следующие методы:

- полный перебор всех терминов словаря, или последовательный поиск.
- метод расширения выборки, или метод спел-чекера.
- метод n-грамм (триад).

Полный перебор. Данный метод осуществляет последовательное считывание данных с диска и сравнение с термином запроса. Основным достоинством этого метода является простота реализации и, как это ни странно, довольно высокая скорость работы. Дело в том, что при последовательном считывании больших файлов (при условии невысокой фрагментации на диске) достигается пиковая скорость чтения. Кроме того, этот метод позволяет реализовать многофункциональный поиск, например, по подстроке или регулярным выражениям, без каких-либо существенных ограничений. Как это ни печально, скорость произвольного доступа на современных дисках может быть почти на два порядка меньше! Однако у этого метода есть один очень существенный недостаток: далеко не всегда скорость обработки данных намного больше времени дисковых операций. Например, в случае архивации словаря или учета изменяемости слов, время поиска может существенно увеличиться. Другим существенным недостатком является низкая скорость поиска на точное равенство.

Метод расширения выборки. Суть данного метода заключается в следующем: строится множество всевозможных «ошибочных» слов, например множество всех слов, получающихся из термина запроса в результате одной операции редактирования, и каждое из этих слов ищется в словаре. В случае, когда словарь может быть загружен в память, этот метод для алфавита размером около 30 символов и поиске, допускающем одну-две ошибки в терминах запроса, работает несколько быстрее, чем другие методы, однако у него есть свои очень серьезные недостатки. Во-первых, он существенно зависит от размера и состава алфавита. Кроме того, если поиск по одной ошибке требует расширения выборки примерно пятьюстами ключевыми словами, то для поиска по двум ошибкам размер такого расширения будет, как минимум, в сто раз больше. На больших объемах данных и поиске на одну ошибку метод не слишком эффективен. В моем распоряжении, к сожалению, пока нет простой реализации этого метода, поэтому я оценивал скорость его работы косвенно: по времени считывания 500 случайно выбранных блоков данных размером в 1 килобайт. Индекс, основанный на хешировании по сигнатуре, позволяет найти все термины, отличающиеся на одну операцию редактирования, за 1.5-2.5 сек, а время считывания случайно выбранных страниц составляет 4-5сек.

По-моему мнению, время считывания 500 случайно выбранных блоков позволяет довольно точно оценить время поиска на неточное равенство спел-чекером, основанном на хешировании. В данном случае хэш-функция случайно «разбрасывает» различные термины по спискам хэш-таблицы. Для большого объема данных, а я использовал словарь в 14 млн. строк, вероятность того, что два разных термина из 500 возможных

окажутся в одном списке, довольна мала. Заметим, что для прочтения списка требуется считывание (если чтение происходят напрямую с диска, а не из кэша), как минимум, одного блока размером 1К. А оценить вероятность того, что все ячейки разместятся менее чем в 400 списках при условии, что в хэш-таблице всего 10 тыс. списков, можно следующим образом: списки хэш-таблицы будем рассматривать как ячейки, способные вместить произвольное количество шаров, а термины как неразличимые шары. Будем предполагать, что хэш-функция случайным образом распределяет шары по ячейкам. Если есть N ячеек и m шаров, то возможно C^m_{N+m-1} (см.[3] Гл. 2 § 5) размещений этих шаров по N ячейкам, и C^{m-1}_{N-1} при условии, что ни одна из N ячеек не останется незаполненной. Таким образом, количество размещений шаров, при котором ровно l ячеек из N занято равно $C^l_N C^{l-1}_{m-1}$ ($C^m_{N+m-1} = \sum_{l=1}^{m} C^l_N C^{l-1}_{m-1}$, l из N заполненных ячеек можно выбрать C^l_N способами), а вероятность события, что все шары разместятся не более чем в n ячейках, равна:

$$p_n = \frac{1}{C^m_{N+m-1}} \sum_{l=1}^{n} C^l_N C^{l-1}_{m-1} \qquad (1)$$

В сумме (1) отношение двух слагаемых с индексами l и $l-1$ равно:

$$\frac{C^l_N C^{l-1}_{m-1}}{C^{l-1}_N C^{l-2}_{m-1}} = \frac{(N-l+1)}{l-1}\frac{(m-l+1)}{l} = \frac{(N-(l-1))}{l-1}\frac{(m+1-l)}{l} = \left(\frac{N}{l-1}-1\right)\left(\frac{m+1}{l}-1\right)$$

В нашем случае N=10000, m=500, N=20m. Таким образом, при $l-1<l<\frac{9}{10}m<\frac{9}{10}(m+1)$ отношение двух соседних слагаемых больше следующей величины:

$$\left(\frac{10N}{9m}-1\right)\frac{1}{10} > \frac{N}{10m}$$

то есть слагаемые убывают быстрее геометрической прогрессии со знаменателем 2 при $l<m-50$. Отсюда сразу получаем оценку для p_l, где l<m-50. Очевидно, что $p_l \le p_{50} \times 2^{-l+50}\left(1+\frac{1}{2}+\frac{1}{4}+\ldots\right) \le 2^{-l+49}$. Получается, что если хэш-функция случайно распределяет слова по ячейкам, то вероятность того, что 500 слов-терминов займут меньше 400 ячеек из 10000 возможных, пренебрежимо мала, а программа, читающая 500 случайно выбранных блоков с диска, для своего выполнения потребует в среднем примерно столько же времени, сколько и программа 500 раз считываю-

щая термины из хэш-индекса.

Хочу отметить, что оценить производительность спел-чекера, основанного на хешировании без списков переполнений сложнее хотя бы потому, что существует очень много различных способов разрешения коллизий.

В случае хеширования без списков переполнений элементы хранятся в одном массиве, а хэш-функция отображает элементы в ячейки данного массива. При отображении в одну ячейку, то есть при так называемой коллизии, для элемента находится некоторая свободная ячейка, например ближайшая слева или справа. Хотя практически очевидно, что и в данном случае для прочтения, скажем, пятисот слов, требуется в среднем также прочесть около 400-500 дисковых страниц.

Метод n-грамм (метод триад). Идея этого метода заключается в следующем: если строка v «похожа» на строку u, то у них должны быть какие-либо общие подстроки. Эта идея подкрепляется следующим простым фактом: пусть расстояние Левенштайна между u и v не превышает k $\left(L(u,v) \le k\right)$, тогда если представить строку u в виде последовательности из k+1-ой строки: $u = u_1 u_2 \ldots u_{k+1}$, то хотя бы одна из u_i является подстрокой v. Если, например, для всевозможных строк длины 3, которые будем называть три-граммами или триадами, хранить на диске список ключевых слов, в которых есть данная подстрока и ссылка на исходное ключевое слово, то можно осуществлять поиск следующим образом: для ключевого слова запроса строить множество всех три-грамм t_i этого слова и для каждой t_i последовательно считывать все ключевые слова, содержащие данную триаду, возвращая пользователю только те термины, которые «похожи» на термины запроса.

У этого метода есть свои минусы: для каждого указателя в списке три-граммы t_i нужно считать ключевое слово из словаря. Поскольку таких ключевых слов относительно немного: их доля составляет порядка одной пятисотой от общего количества терминов, а разбросаны эти термины по словарю случайно, то, проведя несложные выкладки (см. описание метода спел-чекера), получим, что для большинства терминов, нужно считывать целый блок размером 1-4К! Таким образом, чтение только одного списка приводит к считыванию данных объемом порядка одной десятой от исходного объема. Поскольку считывание осуществляется с большим количеством позиционирований, скорость считывания в несколько раз меньше, чем скорость последовательного доступа. То есть выигрыш от сокращения количества дисковых операций практически полностью «съедается» потерей в скорости считывания и эксперимент подтверждает это.

Эту проблему, насколько мне известно, можно решить двумя способами:
хранить вместо указателя ключевое слово целиком, либо хранить в индексе некоторые статистические данные, позволяющие эмпирическим путем отсечь «бесперспективные» термины без обращения к словарю. Первый метод, видимо, является самым быстрым из существующих на сегодня. Данные читаются практически без позиционирований и их объем составляет несколько процентов от исходного. Возможно, что он же и самый избыточный по объему индекса. Без архивации размер индекса, как минимум, в пять-шесть раз превышает размер словаря. Если сами ключевые слова еще и могут быть сжаты в два-три раза, то для указателей на списки вхождений ключевых слов в документы степень сжатия будет явно меньше. Еще одна проблема связана с поиском по коротким терминам. Пусть пользователь ввел поисковый термин: **пагом**, ошибочно написав в слове **«паром»** г вместо р. Нетрудно видеть, что у слов **паром** и **пагом** нет ни одной общей подстроки длины три. Получается, что нужно еще хранить и двухбуквенные начала слов. Однако если пользователь переставит буквы **а** и **р**, то слово все равно не будет найдено. Кроме того, процент вхождения подстроки из двух букв в слова намного выше, а это снижает скорость поиска.

Второй способ был придуман Джоном Вилбуром совместно с Олегом Ховайко (исходники и краткое описание алгоритма были любезно предоставлены Олегом Ховайко http://olegh.spedia.net). В этом алгоритме для каждой триады хранится ее общий вес *Gw* в словаре равный логарифму отношения числу триад в словаре к числу вхождений данной триады в словарь, а также локальный вес этой триады в каждом ключевом слове *Lw* равный логарифму числа вхождения триады в ключевое слово плюс единица:

$$Gw(t) = \log\left(\frac{N}{N_t}\right), \qquad Lw(t) = \log(n+1)$$

Процесс поиска полностью аналогичен описанному варианту с тем отличием, что для каждого термина u, содержащего хотя бы одну триаду t_j, вычисляется функция соответствия u термину запроса v:

$$M(u,v) = \sum_{t_j} Lw_u(t_j) \times Lw_v(t_j) \times Gw(t_j),$$

где $Lw_s(t)$ - локальный вес триады t в ключевом слове s равный логарифму отношений числа вхождений t в слово к общему числу триад в слов, а t_j -общая триада слов u и v. Если значение $M(u,v)$ превышает некоторое пороговое значение k, то термин добавляется в список найденных, либо производится дополнительная проверка (например, что

$L(u,v) \le 2$). Несложно видеть, что в данном методе обращения к словарю чрезвычайно редки.

С моей точки зрения это очень неплохой метод, который выдает пользователю найденные ключевые слова, отсортированные по степени соответствия. Кроме того, небольшая модификация этого алгоритма позволяет существенно ускорить поиск по часто используемым терминам. Однако тестирование этого метода показало, что он работает со скоростью близкой к скорости метода хеширования по сигнатуре, в некоторых случаях даже несколько медленнее, при том, что объем индекса почти в четыре раза больше.

3. Хеширование по сигнатуре

Рассмотрев наиболее распространенные методы поиска, допускающие ошибки в терминах запроса, перейдем к описанию нового метода, который я буду называть хешированием по сигнатуре. Я не нашел работ, описывающих предлагаемый метод, хотя идея, лежащая в его основе, проста и вполне могла быть уже использована ранее.

Пусть задан непустой алфавит A и известны вероятности появления различных символов алфавита. Пусть также на множестве символов A задана функция $f(\alpha)$, отображающая буквы в числа от 1 до m $(|A| > m > 0)$.

Определение 3. Сигнатурой $sign(w)$ слова w будем называть вектор размерности m, k-ый элемент которого равняется единице, если в слове w есть символ α такой, что $f(\alpha) = k$, и нулю в противном случае. Номером сигнатуры слова будем называть число $H(w) = \sum_{i=0}^{m-1} 2^i sign(w)_{i+1}$.

$H(w)$ является хэш-функцией, отображающей множество слов в отрезок целых чисел от 0 до $2^m - 1$. С ее помощью можно проиндексировать множество терминов словаря, объединяя термины с одинаковыми значениями $H(w)$ в списки. Указатели на начала списков необходимо хранить в отдельном массиве размерности 2^m. Как известно, хэш-функция обеспечивает наиболее эффективный поиск на точное равенство при условии, что вероятности попадания слов в различные списки примерно одинаковы. К сожалению, функция $H(w)$ не обладает этим свойством из-за чего она несколько проигрывает в скорости поиска на точное равенство, но в отличие от других хэш-функций она ускоряет поиск по сходству в пределах одной-двух ошибок в ключевых словах запроса.

Пусть слово $\tilde{u}$ получено из u в результате одной операции редак-

тирования: замены, добавления (удаления) буквы или перестановки символов. В силу определения сигнатуры, у векторов $sign(\tilde{u})$ и $sign(u)$ отличаются не более одного компонента в случае добавления (удаления), и не более двух в случае замены. При этом если при замене символов поменяются два элемента сигнатуры, то общее количество единиц (модуль сигнатуры) останется неизменным. Перестановки же, как несложно видеть, вообще не влияют на сигнатуру.

Таким образом, в процессе поиска на одну ошибку нужно просмотреть только те списки хэш-таблицы, у которых расстояние Хэмминга между сигнатурой списка и сигнатурой термина меньше либо равно двум, при этом если оно равно двум, то модули сигнатуры совпадают. Аналогичные соображения можно использовать при поиске по двум и более ошибкам. Кроме того, вероятность события, что $k > 1$ операций редактирования - замены, невелика (при $k = 2$ не более 0.1), да и сам поиск более чем по двум ошибкам для коротких слов (а длинные реже встречаются) малоинформативен, потому что возвращает слишком большую выборку. Поэтому можно считать, что максимально возможное расстояние Хэмминга между сигнатурой ключевого слова запроса и сигнатурой списка хэш-таблицы равно $2k-1$, где k - максимально допустимое расстояние Левенштайна между найденным словом и термином запроса. При поиске по двум-трем ошибкам это несколько увеличивает скорость обработки без особой потери качества. Кроме поиска на соответствие целой строке хеширование по сигнатуре также несколько ускоряет поиск по подстроке, или, другими словами, поиск терминов, содержащих данную подстроку.

Хеширование по сигнатуре - это расширение выборки наоборот. Алгоритм расширения выборки сначала строит возможные ключевые слова, отличающиеся на одну операцию редактирования, а потом проверяет есть ли полученная строка в словаре. Метод хеширования по сигнатуре использует обратный алгоритм: сначала находит «потенциальные» списки словаря, а затем соответствующие запросу термины. За счет того, что значительное количество «похожих» слов (с одинаковыми сигнатурами) располагаются физически (и логически) в смежных страницах памяти, удается избежать многократных и неэффективных чтений с большим количеством позиционирований.

Метод хеширования по сигнатуре обладает следующими достоинствами:

1. Позволяет осуществлять с высокой скоростью поиск на точное равенство и поиск, допускающий одну-две ошибки в поисковом запросе.

2. Работает быстро, как в случае «прямых» чтений с диска, так и из кэша. Метод работает с приемлемой скоростью даже при очень больших размерах словаря. Так на машине Pentium II 400 с 128M памяти (скорость

работы диска 3М/С), поиск в пределах одной ошибки занимает 3-5сек для словаря объемом 30млн. записей, и около 6-10 сек при поиске по двум ошибкам. При тех же условиях программа agrep осуществляет поиск за две минуты, а реализация алгоритма Вилбура-Ховайко за 5-10сек.

3. Характеризуется достаточно компактным индексом. Объем индекса обычно не более чем на 10-20% превышает размер файла, содержащего список терминов словаря.

4. Обладает скоростью поиска, которая не уменьшается при увеличении размера алфавита.

5. Отличается простотой реализации.

6. Следует отметить, что даже при типичных запросах выбирается не более 10% страниц словаря, поэтому сжатие словаря не замедляет процесс поиска так сильно, как, например, при последовательном сканировании. Дело в том, что при считывании сжатых данных процесс распаковки занимает почти столько же времени, а нередко даже больше, чем само считывание с диска, поэтому десятикратное уменьшение объема считываемых данных может скомпенсировать потерю в скорости.

Тем не менее, этому методу присущ и один довольно существенный недостаток: он медленно работает, если словарь дефрагментирован: то есть списки слов с одинаковыми сигнатурами разбросаны по несмежным секторам на диске.

К сожалению, нет никаких системно-независимых стандартизированных средств физической дефрагментации. Единственное, что можно сделать: перестроить индекс таким образом, чтобы термины записывались на диск в порядке обхода списка. Сначала записать на диск все слова w с $h(w)$ равной 1, затем 2, и.т.д. После этого можно использовать утилиты дефрагментации диска, которые существуют для большинства операционных систем. Например в операционной системе Линукс существует автоматическая поддержка файловой системы в дефрагментированном виде. Поскольку структура индекса после перестройки такова, что слова с одинаковыми сигнатурами занимают логически смежные адреса, то после дефрагментации диска ОС разместит их в смежных секторах.

4. Оценки эффективности хеширования по сигнатуре

Функция $f(\alpha)$ (определенная выше на стр. 8) разбивает все множество символов алфавита на непересекающиеся группы: все элементы группы отображаются в одно число. Если вероятности попадания символов в различные группы «близки», то можно для оценки эффективности хеширования по сигнатуре использовать модель алфавита с m символами, вероятности появления которых равны (факторизовать алфавит по

отношению эквивалентности: $\alpha \sim \beta \Leftrightarrow f(\alpha) = f(\beta)$). В этом случае для вероятности $q_n(k)$ события, заключающегося в том, что у слова w длиной n символов будет некоторая заранее выбранная сигнатура $sign(w)$, модуль которой (количество единичных элементов) равен k, верна следующая формула:

$$q_n(k) = \left\{ {n \atop k} \right\} \frac{k!}{m^n} \quad (2)$$

где $\left\{ {n \atop k} \right\}$ - число Стирлинга второго рода (см. [1] глава 6), которое задается следующими рекуррентными соотношениями:

$$\left\{ {0 \atop 0} \right\} = 1, \left\{ {k \atop 0} \right\} = 0, \; k > 0$$

$$\left\{ {n \atop k} \right\} = k \left\{ {n-1 \atop k} \right\} + \left\{ {n-1 \atop k-1} \right\}$$

Пусть p_l - вероятность появления слова длины l, n - максимальная длина слова, L - количество терминов в хэш-таблице, тогда математическое ожидание количества слов, располагающихся в списке сигнатуры модуля k равно:

$$L \sum_{l_2=0}^{n} p_{l_2} q_{l_2}(k)$$

а математическое ожидание количества слов, располагающихся в списке сигнатуры, случайно выбранного слова равно:

$$L \sum_{k=0}^{m} \sum_{l_1=0}^{n} C_m^k p_{l_1} q_{l_1}(k) \sum_{l_2=0}^{n} p_{l_2} q_{l_2}(k) = \sum_{k,l_1,l_2} C_m^k p_{l_1} p_{l_2} q_{l_1}(k) q_{l_2}(k) <$$

$$< L \max^2 p_l \sum_{k,l_1,l_2} C_m^k q_{l_1}(k) q_{l_2}(k) \quad (3)$$

Как несложно видеть, формула (3) дает оценку для математического ожидания количества сканируемых слов при поиске на точное равенство. Воспользовавшись рекуррентной формулой для $q_n(k)$:

$$q_n(k) = k/m\left(q_{n-1}(k) + q_{n-1}(k-1)\right), \; q_0(0) - 1, \; q_k(0) = 0, \; k > 0 \quad (4)$$

можно численно оценить величину суммы в формуле (3) и доказать, что относительная погрешность такого вычисления будет очень мала.

В компьютерной арифметике число $\tilde{x}$ представляется в двоичной системе в нормализованной форме в виде произведения мантиссы: $1.\tilde{x}_1\tilde{x}_2 \ldots \tilde{x}_n$, содержащей n-знаков, и числа 2 в степени $\deg(x)$:

$$1.\tilde{x}_1\tilde{x}_2\ldots\tilde{x}_n \times 2^{\deg(\tilde{x})}$$

При перемножении двух чисел $\tilde{x}$ и $\tilde{y}$ вычисляется точное произведение мантиссы с так называемой двойной точностью, степени чисел складываются, а затем у мантиссы отсекаются n наименее значимых разрядов. Таким образом, если не происходит переполнения в степени, а в нашем случае все числа небольшие и переполнения быть не может, то функцию «приближенного» умножения $Mult(x,y)$ можно описать с помощью функции $Map(\cdot)$, отображающей вещественные числа во множество представимых компьютером чисел:

$$Mult(\tilde{x},\tilde{y}) = Map(\tilde{x}\times\tilde{y})$$

В силу того, что старший разряд мантиссы всегда равен 1, то относительная погрешность функции Map, а также функции «приближенного» умножения, точно представимых чисел будет не больше $\Delta = 2^{-n}$.

Пусть теперь известно, что величины a и b положительны, а также положительны их приближенные значения $\tilde{a}$ и $\tilde{b}$, полученные в результате каких-либо вычислений. Пусть также относительная погрешность вычисления составляет $\theta_1 < 1$ и $\theta_2 < 1$, соответственно:

$$\left|\frac{a-\tilde{a}}{a}\right| < \theta_1 < 1, \left|\frac{b-\tilde{b}}{b}\right| < \theta_2 < 1$$

Посчитаем теперь погрешность вычисления произведения a на b с использованием уже «неточных» значений сомножителей:

$$Mult(\tilde{a},\tilde{b}) = (1+\theta_3)\tilde{a}\tilde{b}$$

где $\theta_3 < \Delta$, а $\Delta = 2^{-n}$, где n - число двоичных знаков мантиссы. Отсюда получаем оценку относительной погрешности:

$$\left|\frac{Mult(\tilde{a},\tilde{b})-ab}{ab}\right| < (\theta_1+\theta_2+\theta_2\theta_1)+(1+\theta_1+\theta_2+\theta_2\theta_1)\theta_3 \qquad (5)$$

Для формулы (4) одно из значений θ_1, θ_2 также не превышает Δ, поэтому (5) можно упростить:

$$\left|\frac{Mult(\tilde{a},\tilde{b})-ab}{ab}\right| < (\theta+\Delta+\theta\Delta)+\Delta(1+\theta+\Delta+\Delta\theta) < \theta+6\Delta \qquad (6)$$

При сложении чисел одного знака, во-первых, происходит расширение мантисс n двоичными разрядами, а затем их выравнивание. Пусть $\tilde{x} > \tilde{y} > 0$. Тогда при выравнивании у меньшего по модулю числа $\tilde{y}$ степень увеличивается на $\deg(\tilde{x}) - \deg(\tilde{y})$, а мантисса сдвигается вправо на ту же величину: $\deg(\tilde{x}) - \deg(\tilde{y})$. При этом может происходить потеря значимых разрядов. После этого происходит само сложение и результат при-

водится к нормализованному виду: отсекаются n младших разрядов. Поскольку, как при выравнивании, так и при усечении младших разрядов, мы делаем ошибку в разрядах, следующих за n-ым, то относительная погрешность вычисления также не превышает величину Δ. Опять-таки, как и в случае умножения, операция «приближенного» сложения чисел $Add(x,y)$ представима в виде:

$$Add(\tilde{x},\tilde{y}) = Map(\tilde{x}+\tilde{y})$$

а относительная погрешность, как я уже упомянул, имеет порядок $\Delta = 2^{-n}$.

Вновь вернемся к случаю, когда числа $\tilde{a}$ и $\tilde{b}$ являются положительными приближениями положительных чисел a, b и получены в результате некоторых вычислений. Оценим возможную погрешность вычисления суммы a и b с использованием полученных приближений:

$$\left|\frac{Map(\tilde{a}+\tilde{b})-a-b}{a+b}\right| = \left|\frac{(\tilde{a}+\tilde{b})(1+\theta_3)-a-b}{a+b}\right| <$$

$$\left|\frac{\theta_1 a+\theta_2 b+\theta_3(a(1+\theta_1)+b(1+\theta_2))}{a+b}\right| < \max(\theta_1,\theta_2)+2\Delta \qquad (7)$$

при условии, что все погрешности не превышают единицу. Применив формулы (6), (7) к рекуррентности (4), получим, что при вычислении величин $q_n(k)$ через $q_{n-1}(k)$, погрешность возрастает линейно по n.

Условие положительности величин является необходимым. Рассмотрим пример чисел: $x = 1.000\ldots0\times2^0$, $y = 1.111\ldots101\times2^{-1}$. Двоичная запись в числах x и y содержит $n+2$ знака после запятой, где n - число знаков в мантиссе. Разность их равна $2^{-n-1}+2^{-n-2}$. При представлении чисел x и y в нормализованном виде два последних знака отбрасываются, и, в результате, разность этих чисел, вычисленная на компьютере, будет равна 2^{-n}, а относительная погрешность вычисления разности $4/3$.

Для n=18 и m=16 значение суммы в формуле (3) равно приблизительно 0.09. Несколько сложнее с величинами p_l. Рассмотрев список, содержащий около 80000 тысяч различных английских слов, я выяснил, что количество слов, приходящихся на определенную длину, не превосходит 10-15%, поэтому для p_l можно выбрать оценку 0.15. Таким образом, количество сканируемых слов, согласно оценке не превысит и полпроцента от общего количества терминов словаря. Более того, можно показать, что если слов достаточно много (несколько сот тысяч), то эта ве-

личина (поделенная на общее количество слов L) является также оценкой доли страниц индекса, считываемых с диска, при поиске на точное равенство. Аналогичным образом можно получить оценку для поиска в пределах одной ошибки:

$$L\max^2 p_l \sum_{k,l_1,l_2} C_m^k q_{l_1}(k)\Big((1+m(m-k))q_{l_2}(k)+k(k-1)q_{l_2}+(m-k)(k+1)q_{l_2}(k+1)\Big) \tag{8}$$

В предположении, что $p_l < 0.15$, m=16 для поиска на точное равенство получаем оценку $0.002L$, а для поиска, допускающего одну ошибку в термине запроса, $0.05L$.

5. Практические результаты

Для демонстрации целесообразности использования ИФ и хеширования по сигнатуре с точки зрения эффективности и компактности индекса для поиска на неточное равенство, я реализовал прототип поисковой системы. Словарь в программе - хэш-таблица по сигнатурам слов, а списки вхождений хранятся в виде ИФ сжатого методом Голомба или дельты [7] в зависимости от длины списка. Сравнение производилось с программой glimpse [17]. glimpse не хранит список вхождения ключевого слова целиком, а только список вхождений в одну из 256 групп, поэтому при поиске в индексе сначала определяется список файлов, содержащих ключевые слова, соответствующие запросу, а затем осуществляется сканирование файлов, входящих в группы. Основными аргументами авторов в пользу применения такого метода являются:

- Компактность индекса
- Высокая скорость индексирования
- Возможность поиска на неточное равенство

Благодаря исследованиям Х. Вильямса, Д. Зоббеля, А. Мофата и других ученых, созданы алгоритмы кодирования ИФ, позволяющие значительно сжать индекс. Фактически, сжатый ИФ имеет примерно тот же размер, что и индекс glimpse'a. Индекс этой программы, согласно утверждениям авторов, в зависимости от выбора пользователя может занимать 3-5%, 8-10% или 10-20%. Для информационного массива в 100 тыс. документов объемом в 230М индекс на базе ИФ имеет размер порядка 7%. Так что это преимущество алгоритма Glimpse не слишком велико. В коммерческих ИПС индекс ИФ действительно может занимать больше места, но это связано с дополнительными издержками на хранение статистики вхождения терминов в документы и резервированием места в индексе для ускорения процесса его обновления.

Что касается скорости индексирования, то разница также не очень

велика, например, на массиве в 10 тыс. документов моя программа осуществляет индексацию за 1 мин 44 сек без удаления стоп-слов и за 1 мин 10 сек, если стоп-слова не заносятся в индекс и словарь, в то время как Glimpse решает эту задачу за 1 мин 5 сек. Размер индекса (минимально возможный) составляет 1.5М, а у ИФ - 3М. При этом скорость поиска может отличаться на порядок. Если же Glimpse создает индекс среднего размера, то его размер соответствует размеру индекса ИФ.

В случае больших словарей хеширование по сигнатуре позволяет осуществлять поиск существенно быстрее метода полного сканирования: на персональной ЭВМ поиск в словаре, содержащем пять миллионов записей, в большинстве случаев занимает не более двух секунд (например, поиск по подстроке), и менее одной секунды при поиске в пределах одной ошибки.

Поскольку в словаре ИФ, как правило, хранятся словоформы, то есть слова без окончаний и суффиксов, то поиск на неточное равенство в таком «каноническом» словаре невозможен. Однако никто не мешает хранить в словаре полные формы ключевых слов. Если же необходимо осуществлять поиск по словоформам, то нужно на этапе индексирования объединять списки вхождений таких слов в один. В случае не очень флективных языков таких, как английский, это не вызовет большого увеличения размера словаря, основанного на хешировании по сигнатуре, и не приведет к потери эффективности.

Для русского языка, например, это большая проблема. Однако ее можно решить, хотя и с некоторыми потерями в эффективности, особенно при поиске по подстроке. В большинстве языков, если не во всех, окончание - только небольшая часть слова. В русском языке, например, не бывает окончаний больше трех символов (если не считать частицу «ся»). Поэтому сигнатуру можно строить по слову без последних трех букв. Опять-таки, при поиске нужно отсекать последние три буквы и строить сигнатуру по оставшейся части слова. При этом расстояние Левенштайна между укороченными терминами, при условии, что «отрезается» всегда одно и то же количество символов, не больше, чем между исходными.

Выгода от такого подхода заключается в несколько более компактном способе хранения слов: для терминов в разных падежах или спряжениях общее начало хранится в одном списке хэш-таблице и его можно не повторять для некоторых грамматических форм. Например, рассмотрим слово **машина**, которое в различных падежах имеет вид: **машина, машины, машине, машину, машиной, машине.**

Для него, в соответствием с правилом отсечения трех последних букв, будет две разных общих части **маши** и **маш**. Их можно хранить

следующим образом:

машин | а | ы | е | у,

машин | ой.

При этом нужно хранить 21 символ (и два указателя на списки вхождений), вместо пяти указателей и 36 символов. Резюмируя вышесказанное: хранение всех грамматических форм слов в словаре, позволяет осуществлять поиск на неточное равенство при относительно небольших издержках на хранение грамматических форм.

По сравнению с glimpse поиск в ИФ происходит намного быстрее. По предварительным результатам (при условии, что данные читаются с диска а не из кэша) glimpse выполняет запрос на порядок медленнее. ИФ остается на сегодняшний день не только одним из самым наиболее популярных методов индексирования, но и также одним из самых эффективных и экономных в смысле размера индекса. ИФ является гибкой структурой, которую можно использовать для поиска на точное соответствие и поиска с учетом морфологической и синонимической информации. ИФ можно также эффективно использовать и для поиска на неточное равенство. ИФ во многих случаях превосходит другие методы в смысле скорости индексирования, поиска, компактности индекса и функциональности.

Хеширование по сигнатуре, в свою очередь, позволяет осуществлять эффективный поиск по сходству даже в большом словаре. При этом данный метод, с моей точки зрения, позволяет достичь оптимального сочетания функциональности, скорости поиска, компактности индекса и простоты реализации. Как показал опыт, его использование вместе с методом инвертирования документов позволяет создавать поисковые системы, эффективно осуществляющие поиск на неточное равенство в больших информационных массивах.

Литература

1. Р. Грэхем, Д. Кнут, О. Паташник, Конкретная математика, М. Мир 1998.
2. Э. Озкарахан, Машины баз данных и управление базами данных, М. Мир 1989.
3. В. Феллер, Введение в теорию вероятностей и ее приложения. Т. 1., М. Мир 1967.
4. A. Ehrenfeucht, D. Haussler. A New Distance Metric on Strings Computable in Linear Time. Discrete Applied Mathematics, 20:191-203, 1988.
5. Christos Faloutsos and Douglas Oard, Survery of Information Retrieval and Filtering Methods, University of Maryland.

6. J. Zobel and A. Moffat and K. Ramamohanarao, Inverted files versus signature files for text indexing, Collaborative Information Technology Research Institute, Departments of Computer Science, RMIT and The University of Melbourne, Australia, feb 1995, Technical report No TR-95-5
7. Hugh E. Williams and Justin Zobel, Compressing Integers for Fast File Access, Computer Journal, Volume 42, Issue 03, pp. 193-201.
8. U. Masek, M. S. Peterson. A faster algorithm for computing string-edit distances. Journal of Computer and System Sciences, 20(1), 785-807,1980.
9. C.J. van Rijsbergen, Information Retrieval, London: Butterworths, 1979 (http://www.dcs.glasgow.ac.uk/Keith/Preface.html)
10. P.H. Sellers, The Theory of Computation of Evolutionary Distances: Pattern recognition. Journal of Algorithms, 1:359-373, 1980.
11. E. Ukkonen, Algorithms for approximate string matching, 1985, Information and Control, 64, 100-118.
12. E. Ukkonen, Finding approximate patterns in strings, $O(k \times n)$ time. Journal of Algorithms 1985, 6 , 132-137.
13. E. Ukkonen, Approximate String Matching with q-Grams and maximal matches. Theoretical Computer Science, 92(1), 191-211,1992.
14. E. Ukkonen, Approximate String Matching over Suffix-Trees. In Proceedings of the Fourth Annual Symposium on Combinatorial Pattern Matching, Padova, Italy, June, pp. 229-242, 1993.
15. Wu S. and U. Manber, Agrep - A Fast Approximate Pattern-Matching Tool, Usenix Winter 1992. Technical Conference, San Francisco (January 1992), pp.153-162. (ftp://ftp.cs.arizona.edu/)
16. U. Manber, "A text compression scheme that allows fast searching directly in the compressed file," in Combinatorial Pattern Matching. 5th Annual Symposium, CPM 94. Proceedings, p. 113-24. Asilomar, CA, USA, 5-8 June 1994.
17. Wu S., U. Manber, GLIMPSE: A Tool to Search Through Entire File Systems, Winter USENIX Technical Conference, 1994 (ftp://ftp.cs.arizona.edu/)
18. R.A. Wagner and M.J. Fisher, The String to String Correction Problem. Journal of the ACM, 21(1):168-173, 1974.