Базы данных
методы доступа
хеширование

В.Н.Лукин

22.11.2020

Метод хеширования (1)

Другое название – метод перемешанных таблиц.

Представляет собой расширение метода прямого доступа, если нет однозначной зависимости между адресом записи и ключом.

Адресная функция (хеш-функция), которая по ключу формирует адрес, может один и тот же адрес выделить разным ключам.

Эта ситуация называется *коллизией*, а соответствующие ключи — *синонимами*. Алгоритм хеширования включает в себя механизм разрешения коллизий.

Метод хеширования (2)

Эффективность метода во многом зависит от эффективности разрешения коллизий. Кроме того, существенно влияет распределение ключей и размер таблицы: в больших таблицах (по отношению к информационным строкам) вероятность коллизий меньше.

Если исходить из предположения о равномерном распределении значения ключей, реализация метода заключается в отображении их равномерно на множество допустимых адресов.

Если есть априорные сведения о распределении ключей, можно построить хеш-функцию, отображающую их равномерно, что заметно повысит эффективность даже при не очень эффективном алгоритме разрешения коллизий.

Метод хеширования (3)

Эффективность доступа при методе хеширования зависит от распределения ключей, от равномерности распределения адресов хеш-функцией, что влечет уменьшение числа коллизий, и от алгоритма разрешения коллизий.

Эффективность хранения зависит от соотношения между возможным количеством ключей и реальным размером таблицы. Она хуже при слабо заполненных таблицах, обеспечивающих высокую эффективность доступа.

Метод хеширования

пример

Рассмотрим простейший способ разрешения коллизий.

Если при попытке размещения по указанному адресу выясняется, что там уже что-то лежит, циклически ищется первое свободное место.

Если свободный адрес найден, данные помещаются в эту строку, в противном случае мест нет.

При выборке проверяется наличие данных по указанному адресу. Если их нет — поиск неудачен, в противном случае проверяется совпадение ключей.

При несовпадении нужный ключ ищется циклически. Если встретился свободный адрес или вернулись к исходному адресу, поиск неудачен, в противном случае при совпадении ключей успешен.

Алгоритмы хеширования (1)

Пусть

- M — число записей в таблице,
- $k \in K$ — ключ из множества допустимых значений ключа K,
- a — адрес в таблице,
- $h(k)$ — хеш-функция, которая по значению ключа формирует адрес: $a = h(k)$, то есть отображает множество ключей K на интервал адресов $[0, M-1]$.

Будем считать, что M больше количества возможных записей. Так как взаимно однозначного соответствия ключей и адресов нет, допускается существование коллизии: таких k_1 и k_2, что при $k_1 \neq k_2$ выполняется $h(k_1) = h(k_2)$.

Ключи k_1 и k_2 называются синонимами.

Алгоритмы хеширования (2)

Будем считать, что ключ k представлен целым неотрицательным числом.

Если ключ символьный, его несложно преобразовать в числовой. Достаточно лишь разбить его на отрезки и циклически сложить их двоичные представления. Недостаток этого метода — он нечувствителен к порядку следования символов. Чтобы избавиться от него, достаточно двоичное представление каждого следующего отрезка сдвинуть циклически на m разрядов влево (умножить на 2^m).

Метод деления

Хеш-функция вычисляется как остаток от деления значения ключа на *M: h*(*k*)=*k mod M.*

Большое значение для эффективности распределения ключей имеет выбор числа *M*. Например, если оно чётное, то адрес будет чётным при чётном ключе и наоборот. Ещё хуже, если оно является степенью системы счисления, тогда значением функции будут младшие разряды ключа.

Лучший вариант для уменьшения группировки ключей — выбор в качестве *M* простого числа.

Метод деления

пример (1)

На вход хеш-функции поступает последовательность слов

$$S = s_1, \ldots, s_6.$$

Приведём слова к числовому виду преобразованием k = length(s).

Пусть $M = 7$, $h(k)=k \bmod M$, *S = это*, *я*, *знаю*, *и*, *помню*, *прекрасно*.

Тогда соответствующая последовательность значений k: 3, 1, 4, 1, 5, 9.

Порядок размещения ключей представлен в таблице. В строке 1, на которой возникла коллизия, для наглядности размещены оба синонима. Реально там может быть единственный ключ, второй должен будет помещён в другое место функцией разрешения коллизий.

Метод деления

пример (2)

№	*k*	*h*(*k*)	*Порядок записи*	*Количество коллизий*
0				
1	я, и	1	2, 4	1
2	прекрасно	2	6	
3	это	3	1	
4	знаю	4	3	
5	помню	5	5	
6				

Метод умножения (1)

Метод умножения — это второй из двух наиболее популярных методов.

Он обладает неплохим свойством рассеивания, а соответствующая хеш-функция работает быстрее, чем предыдущая.

Функция определяется следующим образом:

$$h(k) = \lfloor M \times \{k\alpha\} \rfloor,$$

где $M = 2^m$, α — произвольное дробное число, $\{k\alpha\}$ — дробная часть произведения, $\lfloor x \rfloor$ — ближайшее целое, не превосходящее x.

Метод умножения (2)

В качестве α лучше брать достаточно длинный отрезок иррационального числа.

Неплохие результаты получаются, если использовать «золотое сечение»: $\alpha = (\sqrt{5} - 1) / 2$. В этом случае хорошо рассеиваются арифметические прогрессии ключей.

К данному методу близок метод середины квадратов: в качестве значения функции берутся средние двоичные разряды числа k^2. Однако он хуже метода умножения.

Метод умножения
пример (1)

На вход хеш-функции поступает то же, что и в предыдущем примере, приведение к числовому виду такое же.

Пусть $M = 8$, $h(k) = \lfloor M \times \{k\alpha\} \rfloor$, $\alpha = (\sqrt{5} - 1) / 2 \approx 0{,}618$. Порядок размещения ключей представлен в таблице. При значении ключа *я*, *и*, *прекрасно* (числовые значения 1, 1, 9) в строке 4 образовались коллизии. В данном случае результат хуже, чем в методе деления.

Метод умножения

пример (2)

№	*k*	*h*(*k*)	*Порядок записи*	*Количество коллизий*
0	помню	0	5	
1				
2				
3	знаю	3	3	
4	я, и, прекрасно	4	2, 4, 6	2
5				
6	это	6	1	
7				

Метод деления многочленов (1)

Запишем ключ в двоичной форме:

$$k = 2^n b_n + 2^{n-1} b_{n-1} + \ldots + 2b_1 + b_0,$$

где b_i принимают значения 0 или 1.

Его можно представить многочленом вида

$$k_n(t) = b_n t^n + b_{n-1} t^{n-1} + \ldots + b_1 t + b_0.$$

Выберем простой неприводимый многочлен со случайными двоичными коэффициентами:

$$c_{n-m}(t) = t^{n-m} + c_{m-1} t^{n-m-1} + \ldots + c_1 t + c_0.$$

Остаток от деления многочлена $k_n(t)$ на $c_{n-m}(t)$ для $t = 2$ будет значением функции $h(k)$. При условии близких, но неравных ключей $k_1 \neq k_2$ для этого метода выполняется $h(k_1) \neq h(k_2)$, то есть метод характеризуется хорошим рассеиванием ключей.

Алгоритмы разрешения коллизий

Методы разрешения коллизий делятся на два класса: метод цепочек и метод открытой адресации.

Метод цепочек, в свою очередь, делится на методы внутренних и внешних цепочек. Результат первого исследования в этой области был опубликован в работе А. Думи в 1956 г.

Из методов открытой адресации будут рассмотрены линейное и квадратичное опробование и повторное хеширование. Метод открытой линейной адресации впервые предложил А. П. Ершов в 1957 г.

Метод внешних цепочек (1)

В методе внешних цепочек каждая позиция таблицы служит для записи не ключа, а начальной ссылки на линейный однонаправленный список, в который помещается ключ со связанными с ним данными и все его синонимы.

Список располагается в памяти, внешней по отношению к таблице.

При записи адресная функция указывает на строку таблицы. Если она пуста, в нее записывается адрес начала списка, в первое его звено помещается ключ и данные. Если строка занята, регистрируется коллизия, синоним записывается в очередное звено списка.

При поиске ключа проверяется соответствующая строка таблицы. Если она пуста, поиск неудачен, в противном случае ключ ищется в списке. Если его нет и в списке, поиск неудачен.

Метод внешних цепочек (2)

Рассмотрим вопрос эффективности поиска при данном методе.

Понятно, что значительная часть времени уходит на просмотр списков, если они длинные. Но они обычно невелики: средняя их длина — N / M.

Если размер таблицы небольшой, списки будут расти. То же самое произойдёт при неудачной хеш-функции, которая порождает много синонимов.

Производительность работы со списками можно увеличить обычным образом, например, поддерживать упорядоченность списка по ключам, это сократит время поиска вдвое.

Недостаток метода — использование дополнительной памяти. Сохранение информации во внешней памяти и её обратная загрузка в оперативную может вызвать некоторые трудности.

Метод внешних цепочек

пример (хеширование методом умножения)

№	Length(*k*)	*h*(*k*)	*Порядок записи*	*Количество коллизий*
0	5	0	5	
1				
2				
3	4	3	3	
4	1, 1, 9	4	2, 4, 6	2
5				
6	3	6	1	
7				

→ помню

→ знаю

→ я → и → прекрасно

→ это

Метод внутренних цепочек (1)

Если записи невелики по размеру, можно хранить их в таблице.

Первичное размещение производится по адресу, указанному хеш-функцией, а в случае коллизии запись размещается в свободное место таблицы.

Для списка строка таблицы дополняется полем для ссылки на элемент этой же таблицы.

При записи обращаются к строке, указанной хеш-функцией. Если она свободна, данные в неё записываются, а в поле ссылки записывается признак конца списка. Если строка занята (коллизия), проверяется строка, на которую указывает поле ссылки. Процесс продолжается, пока поле ссылки не пусто.

Далее в таблице ищется свободное место, в него размещаются данные, а ссылка на него записывается в бывшую последнюю строку, которая стала предпоследней.

Метод внутренних цепочек (2)

При поиске ключ ищется по всей цепочке строк, начиная с той, на которую указала хеш-функция. Если начальная строка свободна или был достигнут конец цепочки, поиск неудачен.

Метод внутренних цепочек имеет особенность: списки могут срастаться, когда для ключа без синонимов хеш-функция указывает на занятую строку (при разрешении коллизии она была свободной). Тогда ключ и данные записывают в конец списка чужих синонимов. Так два списка срослись, время поиска увеличилось по обеим группам синонимов.

Если поисков больше, чем вставок, при коллизии выгоднее переместить чужой ключ на свободное место и с текущей строки образовать новый список для размещаемого ключа.

Метод внешних цепочек называется ещё *методом срастающихся цепочек*.

Метод внутренних цепочек

пример (хеширование методом умножения)

№	*k*	*h*(*k*)	*Порядок записи*	*Ссылка*
0	и	4	4	1
1	помню	0	5	2
2	прекрасно	4	6	-1
3	знаю	3	3	-1
4	я	4	2	0
5				
6	это	6	1	-1
7				

Метод внутренних цепочек
пример (порядок заполнения таблицы)

1. Ключ *это* записывается в строку 6.
2. Ключ *я* записывается в строку 4.
3. Ключ *знаю* записывается в строку 3.
4. Ключ *и* синоним ключа *я*. Находим пустую строку с начала таблицы, это строка 0. В строку 4 записываем ссылку 0, в строку 0 помещаем ключ *и*.
5. Для ключа *помню* значение хеш-функции 0, синонимов у него нет, но место в таблице занято. Приходится находить свободное место, это строка 1. Списки срослись.
6. Ключ *прекрасно* — синоним ключей *я* и *и*. Ищем свободное место, проходим цепочку 0, 1 и находим его в строке 2.

Метод линейного опробования (1)

Этот простейший метод открытой адресации уже приводился в качестве примера.

Если при выполнении записи возникла коллизия, начиная с текущего места циклически (с переходом через границу таблицы) с постоянным шагом ищется свободная строка.

Если она найдена, на это место записываются данные. Если процесс привёл к исходной строке, а свободное место не обнаружилось, таблица переполнена.

При выборке из таблицы таким же способом ищется уже не свободное место, а ключ. Если процесс привёл к свободной или к исходной строке, поиск неудачен.

Метод линейного опробования (2)

Величина шага и размер таблицы должны быть взаимно просты, иначе часть таблицы использоваться не будет. Так, при шаге 2, чётном M и чётном $h(k)$ поиск будет проводиться только в чётных строках, а при нечётном значении $h(k)$ — в нечётных.

Шаг должен быть достаточно велик: если хеширование плохо рассеивает ключи, арифметические прогрессии будут приводить к сгущениям.

Пусть c — шаг, $h_0(k) = h(k)$ — начальное значение номера строки, $h_i(k)$ — номер строки на i-м шаге. Тогда $h_i(k) = (h_0(k) + ci) \bmod M = (h_{i-1}(k) + c) \bmod M$.

Метод линейного опробования (3)

Метод линейного опробования (линейной адресации) приводит к локальным сгущениям ключей, особенно в случае срастания двух или более групп, относящихся к различным синонимам.

Понятно, что вероятность сгущения существенно снижается, если таблица заполнена слабо. Д. Кнут считает, что при $N / M < 0{,}75$ метод ещё работает неплохо, а при большем заполнении «работает медленно, но верно».

Наш опыт работы с большими хеш-таблицами во внешней памяти в говорит о том, что уже при $N / M = 0{,}5$ практически наступает коллапс.

Метод линейного опробования

пример

При размещении ключей методом умножения применялось разрешение коллизий методом линейного опробования.

Очередной номер строки при коллизии

$$h_i(k) = (h_{i-1}(k) + c) \bmod M,\ c = 3.$$

Результат размещения приведён на следующем слайде.

Последний столбец в таблице — номер итерации при разрешении коллизий. В данном примере вторичного сгущения, то есть объединения групп различных синонимов, не произошло.

Метод линейного опробования

пример (хеширование методом умножения)

№	*k*	*h*(*k*)	*Порядок записи*	*i*
0	помню	0	5	0
1				
2	прекрасно	2=(7+3) mod 8	6	2
3	знаю	3	3	
4	я	4	2	0
5				
6	это	6	1	0
7	и	7=(4+3) mod 8	4	1

Метод квадратичного опробования

Метод квадратичного опробования представляет собой модификацию предыдущего, когда в формулу добавляется нелинейный член:

$$h_i(k) = (h_0(k) + ci + di^2) \bmod M =$$

$$= (h_{i-1}(k) + c + d(2i - 1)) \bmod M.$$

Здесь d — дополнительная константа. Нелинейность позволяет существенно уменьшить число проб при поиске, но для плотно заполненной таблицы избежать срастания групп различных синонимов не удаётся.

Метод кваратичного опробования

пример

При размещении ключей методом умножения применялось разрешение коллизий методом квадратичного опробования.

Очередной номер строки при коллизии

$$h_i(k) = (h_{i-1}(k) + c + d(2i - 1)) \bmod M$$

$c = 3$, $d = 2$.

Результат размещения приведён на следующем слайде. В данном примере вторичного сгущения нет.

Метод квадратичного опробования

пример (хеширование методом умножения)

№	*k*	*h*(*k*)	*Порядок записи*	*i*
0	помню	0	5	0
1	и	1=(4+3+2) mod 8	4	1
2	прекрасно	2=(1+3+6) mod 8	6	2
3	знаю	3	3	
4	я	4	2	0
5				
6	это	6	1	0
7				

Метод двойного хеширования (1)

Идея метода в том, чтобы отказаться от постоянной или линейной величины шага в пользу случайной, зависящей от ключа.

В этом случае необходима ещё одна хеш-функция $g(k)$, возможно, похожая на $h_0(k)$, но не тождественная ей.

Формула для вычисления следующего адреса:

$$h_i(k) = (h_0(k) + ig(k)) \bmod M = (h_{i-1}(k) + g(k)) \bmod M.$$

В качестве примера $g(k)$ рассмотрим следующие функции (M — простое число):

$$h_0(k) = k \bmod M,\ g(k) = 1 + k \bmod (M–2),$$

$$h_0(k) \text{ — любая},\ g(k)=M–h_0(k).$$

Метод двойного хеширования (2)

Во втором примере $h_0(k)$ и $g(k)$ зависимы.

Это может ускорить вычисления, но если $h_0(k)$ приводила к сгущениям, то и $h_i(k)$ будет обладать тем же свойством.

Для уменьшения подобных неприятностей вводится некоторая независимая величина r, которая управляет последовательностью проб.

Тогда после $h_i(k)$ будет следовать не $h_{i+1}(k)$, а $h_{i+r}(k)$.

Если h_0 и g — независимые функции, вероятность коллизии будет $1/M^2$

Алгоритмы удаления (1)

Методы хеширования не влияют на стиль удаления данных из хеш-таблиц. Если бы не было коллизий, удаление ничем не отличалось бы от такового для метода прямого доступа.

При коллизии требуется найти удаляемый ключ в группе синонимов, удалить его и позаботиться о сохранении корректного состояния этой группы.

Рассмотрим способы удаления данных для различных методов разрешения коллизий.

Алгоритмы удаления (2)

Метод *внешних цепочек* не представляет особых трудностей для удаления: производятся обычные действия по удалению звена из линейного однонаправленного списка. При удалении последнего элемента начальная ссылка в таблице обнуляется.

Удаление при методе *внутренних* цепочек сложнее. Если в списке нет синонимов элемента, он удаляется.

Удаление элемента из списка могло быть похожим на удаление из внешнего, но мешают сросшиеся цепочки. Из-за того, что хеш-функция может указывать на строку, где должен лежать «чужой» элемент, данные в ней не могут быть удалены.

Алгоритмы удаления (3)

Для двух сросшихся списков нетрудно найти эффективное решение, но если их больше, алгоритм становится слишком громоздким. Поэтому применяют алгоритм удаления сдвигом хвоста списка на один элемент, начиная с удаляемого, с проверкой на сращивание списков и пропуска «чужих» элементов без сдвига.

Если удаления производятся достаточно часто, а вероятность срастания цепочек велика, каждый список синонимов лучше начинать с той строки, на которую указывает хеш-функция. Тогда при записи первого ключа нужно освободить эту строку, если она была занята чужим синонимом.

Алгоритмы удаления (4)

Удалить данные при использовании методов открытой адресации непосредственно из таблицы невозможно: ключ может входить в цепочку синонимов, и его удаление приведёт к разрушению этой цепочки.

В этом случае вместо удаления ключа его заменяют специальным значением. При поиске это место пропускается, а при вставке оно считается пустым.

Итак, строка хеш-таблицы будет в одном из состояний: свободная, занятая, удаленная. Однажды занятая позиция не может стать свободной, поэтому с исчерпанием свободных мест время поиска будет заметно расти, особенно при неудачном поиске.

Переразмещение (рехеширование)

Сложность метода хеширования — выбор размера таблицы. При большой таблице неэффективно используется память, при маленькой заметно растёт время доступа.

Наблюдения говорят о том, что при некотором уровне заполнения работа с таблицей становится невозможной. Увеличить размер таблицы нельзя: функции хеширования от него зависят.

Тогда создают таблицу большего размера и переносят в нее содержимое старой. Эта процедура требует времени, её невозможно выполнить в оперативном режиме. Это *переразмещение*, или *рехеширование*.

Анализ метода (1)

Метод хеширования всегда предполагает коллизии.

Разрешение коллизий требует больше времени, чем вычисление адреса.

Доказано, что при хеш-функции, равномерно распределяющей ключи, среднее число проб для вставки ключа минимально. Поэтому эффективность во многом зависит от качества рассеивания ключей.

Оно, в свою очередь, зависит от характера данных, один и тот же метод может показать хорошие результаты в одном случае, но быть неприемлемым в другом. Анализ алгоритмов хеширования довольно сложен. Приведём только некоторые характеристики.

Анализ метода (2)

Обозначим $\alpha = N / M$ — коэффициент заполненности таблицы, C — среднее число проб для поиска существующего ключа, C' — среднее число проб для вставки ключа при больших N и M.

Тогда для случая отсутствия коллизий

$C' = 1 / (1 - \alpha) + O(1 / M)$

$C = (1 / \alpha) \ln(1 / (1 - \alpha)) + O(1 / M)$

При наличии коллизий эти величины зависят от метода их разрешения, кроме метод двойного хеширования, характеристики которого практически совпадают с приведенными.

.

Анализ метода (3)

В следующих оценках подразумевается наличие слагаемого $O(1/M)$.

Метод	***C'***	***C***
Внешние цепочки	$\alpha + e^{-\alpha}$	$1 + \alpha / 2$
Внутренние цепочки	$1 + 1/4(e^{2\alpha} - 1 - 2\alpha)$	$1 + \alpha / 4 + 1 / (8\alpha) (e^{2\alpha} - 1 - 2\alpha)$
Линейное опробование	$\frac{1}{2}(1 + 1 / (1 - \alpha))$	$\frac{1}{2}(1 + 1 / (1 - \alpha)^2)$
Двойное хеширование	$1 / (1 - \alpha)$	$(1 / \alpha) \ln(1 / (1 - \alpha))$

Литература

1. *Когаловский М. Р.* Энциклопедия технологий баз данных.
2. *Кнут Д.* Искусство программирования для ЭВМ.
3. *Лукин В.Н.* Введение в проектирование баз данных.
4. *Сибуя М., Ямамото Т.* Алгоритмы обработки данных
5. *Тиори Д., Фрай Дж.* Проектирование структур баз данных.