Санкт-Петербургский государственный университет
Математико-механический факультет

Кафедра системного программирования

Структура хранения индексов закэшированных страйпов и ссылок на них

Курсовая работа студента 345 группы
Дудина Виктора Дмитриевича

Научный руководитель Короткевич А. И., RaidiX,
руководитель научного направления

Санкт-Петербург
2012

Оглавление

Введение

В основе большинства современных систем хранения данных лежат RAID-массивы на базе жёстких дисков, которые не могут обеспечить достаточной производительности без применения кэширования. В связи с этим, производительность механизма кэширования является важным аспектом в современных СХД. Улучшение любой из частей механизма кэширования может привести к существенному росту производительности всей СХД.

В данной работе был изучен механизм кэширования, использующийся в СХД Avrora от компании AvroRAID. Было замечено, что одна из частей этого механизма использует, возможно, не самую оптимальную структуру данных для выполнения своих целей. А именно, сомнению подверглась структура хранения индексов страйпов, находящихся в кэше.

Цель данной работы – изучить существующие структуры хранения индексов и выбрать из них самую оптимальную в рамках ограничений, накладываемых СХД.

Постановка задачи

В рамках данной курсовой работы ставились следующие задачи:

- Исследовать существующие структуры хранения индексов и закрепленных за ними данных. Фактически, необходимо было изучить структуры-«Словари», хранящие пару «Ключ, Значение»;
- Провести сравнительное тестирование существующих структур и выявить из них наиболее производительные;
- Сравнить производительность существующих структур и структуры, реализованной в СХД Avrora.

Какие задачи ставились перед изучаемыми структурами? Иными словами, какие целевые функции нас интересовали? Можно выделить три главных функции:

- Поиск элемента по заданному ключу
- Добавление нового элемента
- Удаление элемента по заданному ключу

Здесь элементом называется пара «Ключ, Значение», о которой говорилось ранее.
Стоит добавить, что функции удаления и добавления всегда выполняются в связке, то есть за удалением элемента всегда будет следовать добавление нового элемента. Таким образом, мы можем объединить удаление и добавление в одну функцию. Назовем ее «Замещение». Итак, остается 2 целевые функции: поиск и замещение. Быстродействие структур при выполнении этих функций мы и будем сравнивать.

Какие ограничения накладывает СХД Avrora?

- Количество элементов, хранящихся в структуре, фиксировано. Так как после первичного заполнения структуры выполняются только поиск и замещение, количество элементов в структуре остается неизменным.
- Количество элементов, хранящихся в структуре, не очень велико и обычно составляет всего несколько тысяч. Это связано с ограниченностью размера кэша.
- Искомая структура имеет фиксированный размер. Это означает, что в ходе добавления и удаления элементов мы не сможем выделять для структуры дополнительную память. Память выделяется один раз - в момент инициализации - и не перераспределяется со временем.

Обзор существующих структур

В ходе данной исследовательской работы было найдено много различных структур, которые выполняют требуемую функциональность. Эти структуры можно поделить на 2 крупных класса: *деревья* и *хеш-таблицы*. В каждом из классов присутствует большое количество структур, каждая со своими особенностями, асимптотиками, тонкостями реализации. Сразу же возник справедливый вопрос: все ли эти структуры стоит рассматривать? После некоторых рассуждений было решено исследовать только хеш-таблицы, а деревья не трогать совсем.

Почему именно так? Структура всякого дерева подразумевает наличие ссылок между вершинами. Использовать ссылочную структуру – невыгодно, т.к. переходя по ссылкам, мы часто будем получать промахи процессорного кэша, что в столь нагруженной структуре крайне плохо отражается на производительности всей системы. Можно было бы решить эту проблему, упаковав дерево в массив. Но проходить по такой структуре и изменять ее – значит бегать по всем кускам массива (соседние вершины дерева могут оказаться не рядом в массиве), что тоже приведет к частым промахам процессорного кэша. В то же время, структура большинства хеш-таблиц подразумевает последовательное расположение в памяти элементов, находящихся в одной «корзине». Такая организация «корзин» приводит к минимизации промахов процессорного кэша при выполнении целевых функций.

Кроме того, при некоторых допущениях можно сказать, что в хеш-таблицах целевые функции выполняются за время $O(1)$. В деревьях выполнение данных функций занимает в среднем $O(logN)$ времени, где N – количество элементов в структуре.

Исходя из этих соображений, вся дальнейшая исследовательская работа была связана только с хеш-таблицами.

Хеш-таблицы

Хеш-таблицы делятся на 2 вида (по способу разрешения коллизий):

- Метод цепочек (separate chaining)
- Открытая адресация (open addressing)

Из хеш-таблиц, реализующих метод цепочек, можно выделить:

- *Chained hashing* – элементы внутри корзины организованы в связанный список, т.е. каждый элемент хранит ссылку на следующий элемент;
- *Clustered chained hashing* – аналогичен *chained hashing,* но элементы группируются в блоки, и каждый блок хранит ссылку на следующий блок.

Из хеш-таблиц с открытой адресацией можно выделить:

- *Array hashing* – корзина является массивом, который расширяется при добавлении элемента и сужается при удалении;
- *Linear probing, Quadratic probing, Double hashing* – вся хеш-таблица является одним большим массивом. Номер корзины задает место в этом массиве, начиная с которого элемент может присутствовать в структуре. Алгоритм вставки элемента проверяет ячейки массива в некотором порядке до тех пор, пока не будет найдена первая свободная ячейка, в которую и будет записан новый элемент. Данные методы задают различный порядок обхода ячеек массива.
- *Cuckoo hashing* – главной особенностью этой хеш-таблицы является то, что для любого элемента существует ровно 2 возможных ячейки во всей структуре. При поиске нам достаточно проверить только эти 2 ячейки, чтобы узнать, находится ли элемент в структуре. Таким образом, максимальное время поиска фиксировано.
- *Bucketized cuckoo hashing* – аналогичен *cuckoo hashing*, но ячейки структуры – это корзины небольшого размера. Любой элемент может храниться в одной из двух фиксированных корзин.

Опираясь на статьи [1] и [2], сразу же были отброшены варианты с *Bucketized cuckoo hashing* и *Chained hashing*. Так же был отброшен *Array hashing*, поскольку в рамках данной СХД наша структура не допускает перераспределения памяти в ходе работы. По этой же причине был отброшен *Cuckoo hashing*, поскольку возможны случаи, когда хеш-таблицу необходимо перестраивать с изменением размера, а это недопустимо.

Clustered chained hashing. Несмотря на то, что в статье [1] этот метод показал не очень высокую производительность, было решено опробовать его, внеся некоторые изменения в реализацию.

Linear probing, Quadratic probing, Double hashing. Данная хеш-таблица имеет один существенный минус: элементы из одной корзины могут «наползти» на область, предназначенную для другой корзины. В связи с этим, удаление элементов из структуры становится практически невозможным, поскольку при последующем поиске элемента, не присутствующего в структуре, мы не будем знать, где остановиться. Данный метод было решено опробовать, но с внесением изменений, решающих описанную проблему.

Описание исследованных структур

Перейдем к более подробному описанию исследованных структур.

Clustered chained hashing

В основе структуры лежит «Блок», хранящий не более чем *K* элементов структуры, а также ссылку на другой «Блок». За каждой корзиной закреплен определенный «Блок». «Блоки» хранятся в пуле «Блоков», память под который выделяется во время инициализации структуры. Как выполняются целевые функции:

- *Поиск элемента по ключу:* сначала вычисляем корзину, соответствующую данному ключу. Затем делаем поиск в «Блоке», закрепленном за данной корзиной. Если элемент присутствует в «Блоке», то ОК, поиск завершен. Если элемента в «Блоке» нет, и при этом ссылка на следующий «Блок» пуста, значит искомого элемента в структуре нет. В противном случае, переходим по ссылке на следующий «Блок» и продолжаем поиск в нем.
- *Добавление элемента:* вычислив корзину, добавляем элемент в соответствующий «Блок». Если «Блок» заполнен, то переходим по ссылке на следующий «Блок» и добавляем элемент в него. Если ссылка на следующий «Блок» была пуста, то мы выделяем из пула «Блоков» очередной «Блок» и запоминаем ссылку на него, делая его следующим «Блоком».
- *Удаление элемента по ключу:* пытаемся удалить элемент из соответствующего «Блока». Если элемента в этом «Блоке» нет, то переходим по ссылке на следующий «Блок» и выполняем удаление в нем. В случае, если мы удалили элемент из «Блока», не являющегося последним в цепи, мы перебрасываем самый последний элемент в цепи на место удаленного элемента. Если при этом последний «Блок» опустел, то мы возвращаем его в пул «Блоков» и удаляем ссылку на него как на следующий «Блок».

Modified linear probing

В данном методе хеш-таблица – это один большой массив, в котором могут храниться элементы. За каждой корзиной закреплен определенный кусок этого массива размера *K*. Размер куска меньше, чем количество элементов, которые теоретически могут попасть в одну корзину. В случае, когда в одну корзину попадает больше *K* элементов, первые *K* элементов хранятся в основном массиве, а остальные хранятся в дополнительном массиве. Этот дополнительный массив – один для всей хеш-таблицы. Поэтому в него могут попадать элементы с любой корзины. В дополнительном массиве элементы хранятся

последовательно, один за другим, независимо от корзин, к которым они относятся. Как выполняются целевые функции:

- *Поиск элемента по ключу:* сначала вычисляем корзину, соответствующую данному ключу. Далее последовательно проверяем элементы, лежащие в корзине. Если элемент не найден, и при этом корзина заполнена не полностью, значит элемента в структуре нет, поиск завершен. Если же корзина заполнена полностью, то поиск продолжается по дополнительному массиву. Массив обходится последовательно либо до нахождения искомого элемента, либо до последнего элемента в массиве.
- *Добавление элемента:* вычислив корзину, добавляем в нее элемент. Если корзина уже заполнена, то добавляем элемент в конец дополнительного массива.
- *Удаление элемента по ключу:* пытаемся удалить элемент из соответствующей корзины. Если элемента в самой корзине нет, то удаляем этот элемент из дополнительного массива, перемещая последний элемент в доп.массиве на место удаленного элемента. Если элемент есть в самой корзине, то удаляем его из корзины. Если, при этом, перед удалением корзина была заполнена, то нам необходимо проверить доп.массив на наличие элементов, принадлежащих данной корзине. Если такие элементы есть, то мы перемещаем их из доп.массива в саму корзину.

Original structure

Так будем называть структуру, реализованную в СХД Avrora. Не вдаваясь в детали реализации, можно сказать, что эта структура основана на наборе закольцованных массивов, каждый из которых хранит несколько элементов. В отличие от хеш-таблиц, эта структура занимает ровно столько места, сколько элементов хранится в структуре.

Реализация этой структуры была взята из работающей СХД Avrora, то есть это ровно та реализация, которая используется в текущих СХД.

Сравнение производительности будем проводить между этими тремя структурами.

Тестирование

Условия тестирования

Чтобы понять, насколько производительны будут предложенные структуры в рамках СХД, тестирование проводилось на той операционной системе, которую использует сама СХД. Конкретнее, тестирование производилось на 64-битной версии Scientific Linux 6.

Другой не менее важный вопрос: на каких данных проводить тесты? Понятно, что для максимально достоверных показателей производительности тестирование следует проводить именно на тех данных, с которыми наша структура будет работать внутри СХД. Но для этого каждую из структур пришлось бы встраивать в существующую систему кэширования вместо уже использующейся. Поскольку это было невозможно, было решено проводить тестирование на автоматически сгенерированных запросах.

Тестирование проводилось для разного количества элементов в структуре. Конкретнее, тестирование проводилось в условиях, когда в структуре находилось 500, 2 000, 5 000, 15 000 и 30 000 элементов. Откуда именно такие цифры? Для данных алгоритмов не имеет значения, сколько дисков используется в СХД, важно лишь количество страйпов. Количество страйпов рассчитывалось как 4Тб / 128Кб = 2^24, где 4Тб – максимальный размер жесткого диска на сегодняшний день, 128Кб – размер одного участка страйпа (стрипа), используемый в СХД Avrora. Во всех тестах предполагалось, что максимальное кол-во страйпов 2^24. Про размер кэша. Исходя из физических характеристик самой крупной СХД Avrora (32 диска по 4Тб, 128Гб оперативной памяти), можно вычислить, что максимальное количество страйпов в кэше составляет около 30 000. В свою очередь, минимальное кол-во страйпов - 500. Поэтому было решено тестировать структуры для 500, 2 000, 5 000, 15 000 и 30 000 страйпов.

Время измерялось в тактах процессора.

Запросы на поиск генерировались так: 75% запросов искали элемент, присутствовавший в структуре, остальные 25% искали отсутствующий элемент.

Запросы на замещение генерировались так: каждый запрос удалял элемент, который точно был в структуре, и добавлял элемент, которого в структуре точно не было.

Вариация параметров

Обе тестируемые структуры – и *clustered chained hashing (CCH),* и *modified linear probing (MLP)* – имеют настраиваемые параметры. Для *CCH* это количество корзин и размер «Блока», для *MLP* это количество корзин и размер всей структуры (этот размер больше, чем кол-во элементов, хранящихся в структуре). Так как было не очевидно, при каких параметрах структура будет выдавать наилучшую производительность, пришлось проводить тестирование для различных комбинаций этих параметров. По этой причине, параметры варьировались:

- для *CCH*:
    - количество корзин: 2^9, 2^11, 2^13
    - размер «Блока»: 8, 12, 16
- для *MLP*:
    - количество корзин: 2^9, 2^11, 2^13
    - размер структуры: 2*N, 4*N, 8*N (N – кол-во элементов в структуре)

Результаты тестирования

Для начала приведем результаты тестирования структур *clustered chained hashing* и *modified linear probing* с разными комбинациями параметров.

Количество корзин	Размер «Блока»	Время работы при количестве элементов:				
		500	2 000	5 000	15 000	30 000
2^9	8	62 234 087	85 326 360	119 178 420	260 401 017	473 869 887
2^9	12	62 351 770	85 680 133	115 863 987	260 141 898	461 167 286
2^9	16	61 502 296	85 481 627	109 965 030	238 518 630	446 353 381
2^11	8	41 769 633	68 633 645	90 580 917	125 552 144	177 801 226
2^11	12	43 023 853	68 040 792	91 963 672	121 686 771	187 913 814
2^11	16	43 547 828	70 343 346	93 131 968	124 082 184	171 794 538
2^13	8	40 864 051	46 912 111	62 423 365	92 424 138	110 067 105
2^13	12	39 731 215	48 249 405	64 768 833	95 393 429	111 609 432
2^13	16	42 328 759	50 854 503	68 492 416	100 440 205	116 831 386

Выполнение 1 000 000 запросов на **поиск** в *clustered chained hashing*

Количество корзин	Размер «Блока»	Время работы при количестве элементов:				
		500	2 000	5 000	15 000	30 000
2^9	8	127 925 317	169 891 115	257 244 329	503 870 948	829 213 941
2^9	12	128 661 124	163 159 114	203 902 478	402 639 761	652 461 593
2^9	16	129 295 175	161 255 910	191 308 327	356 313 390	588 343 893
2^11	8	123 203 951	177 280 528	211 365 417	292 942 881	409 862 897
2^11	12	128 165 817	183 022 037	218 001 309	249 813 326	350 013 154
2^11	16	129 801 692	186 381 657	222 593 681	246 369 892	306 146 402
2^13	8	140 346 354	169 321 417	210 776 495	261 078 124	280 702 422
2^13	12	144 172 387	165 607 438	220 145 871	271 469 854	276 613 614
2^13	16	140 299 127	171 079 172	225 303 968	276 611 061	292 330 471

Выполнение 1 000 000 запросов на **замещение** в *clustered chained hashing*

Количество корзин	Размер структуры	Время работы при количестве элементов:				
		500	2 000	5 000	15 000	30 000
2^9	2*N	106 420 984	121 347 267	150 391 628	234 063 444	356 711 533
2^9	4*N	91 479 526	123 698 094	152 816 064	235 336 822	357 357 032
2^9	8*N	93 927 811	126 826 661	156 696 248	238 919 064	354 247 135
2^11	2*N	80 569 516	130 302 670	126 694 340	156 497 152	193 156 424
2^11	4*N	50 735 454	77 525 152	125 266 446	161 389 704	200 128 504
2^11	8*N	64 952 583	97 222 515	129 253 527	162 301 614	195 101 295
2^13	2*N	59 982 215	106 995 556	102 136 726	152 300 710	141 818 833
2^13	4*N	60 304 245	108 039 177	88 118 426	126 572 303	142 779 362
2^13	8*N	60 019 831	67 883 962	86 837 618	121 933 155	159 090 060

Выполнение 1 000 000 запросов на **поиск** в *modified linear probing*

Количество корзин	Размер структуры	Время работы при количестве элементов:				
		500	2 000	5 000	15 000	30 000
2^9	2*N	331 661 993	297 711 468	320 348 082	416 570 015	556 861 517
2^9	4*N	249 235 288	293 947 981	330 454 979	418 719 120	561 739 227
2^9	8*N	254 952 076	306 601 116	333 077 055	424 532 220	576 241 965
2^11	2*N	273 877 780	561 404 850	375 791 231	354 198 811	402 221 415
2^11	4*N	273 813 487	262 698 402	308 391 844	363 796 329	407 564 853
2^11	8*N	195 255 043	272 261 941	320 211 405	368 913 035	436 210 705
2^13	2*N	180 480 177	488 500 538	450 197 219	656 986 911	401 092 424
2^13	4*N	180 893 502	487 138 233	266 361 060	335 910 153	366 835 315
2^13	8*N	179 853 532	207 577 444	266 950 536	345 199 932	394 290 053

Выполнение 1 000 000 запросов на **замещение** в *modified linear probing*

Чтобы дать справедливую оценку, какая из протестированных комбинаций лучше, необходимо дополнительно учесть количество памяти, которое использует структура. Приведем информацию о расходуемой памяти.

Количество корзин	Размер «Блока»	Расход памяти (Кб) при количестве элементов:				
		500	2 000	5 000	15 000	30 000
2^9	8	815	815	815	815	815
2^9	12	832	832	832	832	832
2^9	16	856	856	856	856	856
2^11	8	518	518	518	701	1 067
2^11	12	643	643	643	822	1 180
2^11	16	769	769	769	946	1 300
2^13	8	1 389	1 389	1 480	1 709	2 028
2^13	12	1 900	1 900	1 989	2 213	2 525
2^13	16	2 412	2 412	2 500	2 721	3 030

Расход памяти *clustered chained hashing* при различных комбинациях параметров

Количество корзин	Размер структуры	Расход памяти (Кб) при количестве элементов:				
		500	2 000	5 000	15 000	30 000
2^9	2*N	146	194	290	602	1 074
2^9	4*N	162	258	450	1 074	2 010
2^9	8*N	194	386	762	2 010	3 882
2^11	2*N	168	200	296	616	1 096
2^11	4*N	168	264	456	1 096	2 024
2^11	8*N	200	392	776	2 024	3 912
2^13	2*N	288	288	416	672	1 184
2^13	4*N	288	288	544	1 184	2 080
2^13	8*N	288	416	800	2 080	4 000

Расход памяти *modified linear probing* при различных комбинациях параметров

Стоит отметить, что при любой комбинации параметров, исследуемые структуры занимают не более 4 Мб памяти, что в системах с несколькими Гб памяти является очень маленькой величиной.

Как видно из этих таблиц, лучшая производительность практически во всех тестах у *clustered chained hashing* достигается при кол-ве корзин 2^13 и размере «Блока» 8, а у *modified linear probing* при кол-ве корзин 2^13 и размере структуры 8*N.

Теперь оценим время работы *original structure* и сравним производительности *original structure* и исследованных структур.

Original Structure	Количество элементов в структуре:				
	500	2 000	5 000	15 000	30 000
Время выполнения 1 000 000 запросов на **поиск** (такт)	417613955	533483900	627867031	781457812	885180407
Время выполнения 1 000 000 запросов на **замещение** (такт)	2616327325	6252848926	13130085601	35055250986	46959606857
Расход памяти (Кб)	143	190	284	596	1193

Результаты тестирования структуры, реализованной в СХД Avrora

Теперь посмотрим, насколько производительность исследованных структур лучше, чем производительность *original structure*, реализованной в СХД Avrora.

Название структуры	Во сколько раз быстрее, чем *original structure*, при количестве элементов:				
	500	2 000	5 000	15 000	30 000
Clustered chained hashing	10.22	11.37	10.06	8.46	8.04
Modified linear probing	6.96	7.86	7.23	6.41	5.56

Сравнение производительности исследованных структур со структурой, реализованной в СХД Avrora, при выполнении 1 000 000 запросов на **поиск**

Название структуры	Во сколько раз быстрее, чем *original structure*, при количестве элементов:				
	500	2 000	5 000	15 000	30 000
Clustered chained hashing	18.64	36.93	62.29	134.27	167.29
Modified linear probing	14.55	30.12	49.19	101.55	119.10

Сравнение производительности исследованных структур со структурой, реализованной в СХД Avrora, при выполнении 1 000 000 запросов на **замещение**

Из проведенных тестов можно сделать вывод, что *Clustered chained hashing* – отличная замена для *Original structure*. Комбинация с кол-вом корзин 2^13 и размером «Блока» 8 занимает не более 2 Мб памяти, но дает существенный прирост производительности.

Заключение

В ходе выполнения данной курсовой работы были исследованы различные варианты хеш-таблиц и проведено измерение быстродействия некоторых из этих вариантов. Из результатов тестирования видно, что хеш-таблицы более производительны, чем структура, реализованная в СХД Avrora. Хеш-таблицы быстрее исходной структуры в несколько раз, а это означает, что мы добились того, к чему стремились.

Стоит помнить, что мы сравнивали скорость работы структур, которые занимаются хранением индексов закэшированных страйпов. Несмотря на то, что их производительность отличается в несколько раз, мы не можем сказать, какой прирост производительности получит вся система кэширования при замене исходной структуры на хеш-таблицу. Это станет известно лишь после внедрения хеш-таблицы в систему кэширования и проведения тестов производительности на реальных данных.

Список используемой литературы

[1] Nikolas Askitis. Fast and compact hash tables for integer keys (2009).

[2] Rasmus Pagh, Flemming Friche Rodler. Cuckoo hashing (2003).

[3] http://en.wikipedia.org/wiki/Hash_table