Yang Du

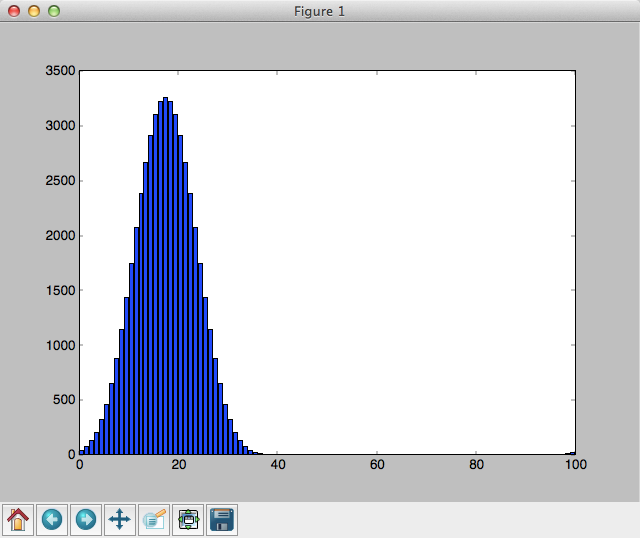
CS145

Problem Set 3

1.

1. Tables. Since T1 and T2 both work on A=6. But T2 does an insert so that just put a lock on the tuple can not guarantee because it is an insert and actually change the table, it not an update where A can be fixed. It is not necessary to put the lock on the whole database because we just update one table and it doesn’t show that there is any trigger to update another table. Thus, put a lock on the table is fine for this problem.
2. Pages. Since there is an index on A, and we only worry about the situation when A = 6, put a lock on the pages containing A=6 is fine. Because we do not need to worry about the update or insert of A=100 since it is in a different page. An index on A makes A to be sorted. Thus, we can use a finer granularity when A has an index.
3. When T1 select the count, it might get n. And at the time T1 do some work, T2 insert (6, 20). Then T1 select all the tuples when A=6. Thus, T1 will get n+1 tuples because T2 just inserted the one tuple into table R.
4. Phantom Deadlock is a deadlock, which is detected due to network latency or some other delay, but actually it is not a deadlock. For example, on one machine, R depends on P. On another machine, Q depends on R and S depends on Q. So globally the coordinator will have S depends on Q, Q depends R, R depends on P. When Q release S and P requests s, message may arrive in wrong order or delay, the coordinator may detect that there is a cycle on S depends on Q, Q depends R, R depends on P, and P depends on S, and believe deadlock and kill P or Q, but actually it is not necessary.

2. （a)



(b) The buckets I got is:

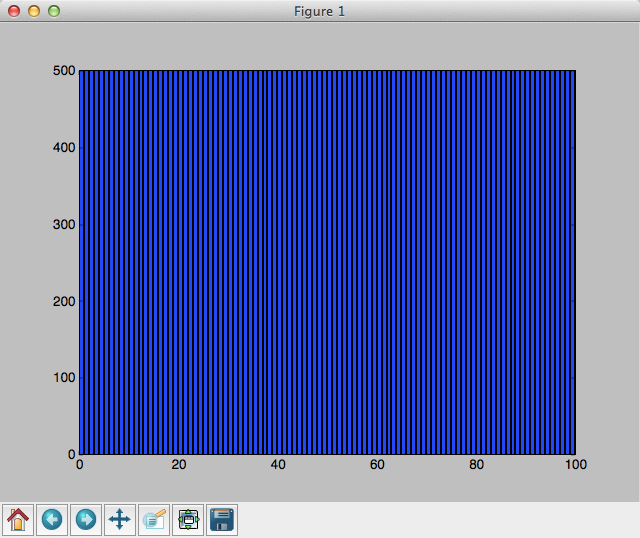
[35, 70, 125, 205, 315, 460, 645, 871, 1135, 1430, 1745, 2065, 2375, 2660, 2905, 3095, 3215, 3256, 3215, 3095, 2905, 2660, 2375, 2065, 1745, 1430, 1135, 871, 645, 460, 315, 205, 125, 70, 35, 15, 5, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 5, 15]

And I use my program to calculate the cost: 35^2 + 70^2+… and the result is 117126320.

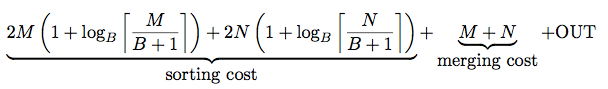
117126320/(100 \* 500 ^2) = 4.685.

Thus, the actual running time is 4.685 times slower than what he expected

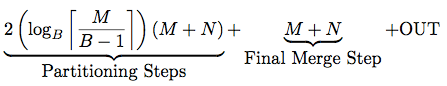
(c) The new hash function I design is to first take out of the TI since all the string has the same TI. And then convert the rest of the string to int, and divided by 500.

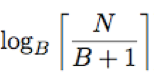
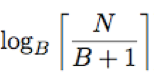
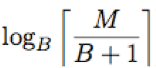


The bucket I got is [500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500], and it is perfectly uniformly distributed. The cost I got is 25000000, which is equal to the expected time.

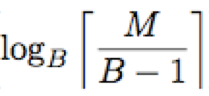
3. 1 (a) The cost for the sort-merge join is 

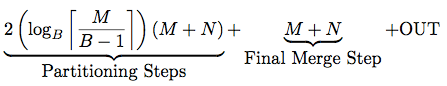
and the cost for the hash join is



So we can see the only differences are B+1 versus B-1, and the term , where in sort-merge it is N, and in hash is M, and sort-merge has an extra 2(M+N). Thus, even if we make much smaller than , it still can not catch the difference of 2M + 2N. Thus, sort-merge can not beat hash by simply change the input size and memory pages.

There are two situations that sort-merge can beat hash join. One is if the data is sorted, then sort-merge only need to use M+N+Out to output the result, where hash join still need to do many times of partition.

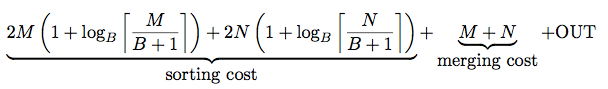
The other situation is if the data is skewed, then the hash join has to run many more time to partition the data because of the skewed partition is too large, and it has to fit in memory, which increase , but sort-merge will still have the same cost so that sort-merge join can beat hash join.

(b) The cost for BNLJ is , and the cost for hash join is still . The M + Out is the same so we do not worry at the comparison.

Thus, if we make M = N = 1024 ^ 3, and B = 2.

Hash join need 2\*30\*(M + N) + N = 60M + 61N = 121\*1024^3, but BNLJ need MN = 1024 ^ 6. It is obvious that 121\*1024^3 < 1024 ^ 6

So in my scenario R and S has size 1024^3 with 2 RAM pages.

(c) We know that sort-merge join has the cost , and BNLJ has the cost . Ignore M+Out since they are the same.

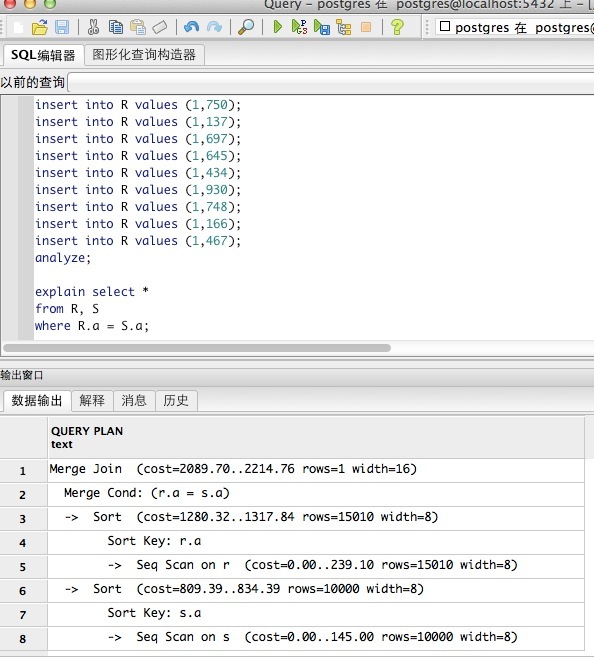
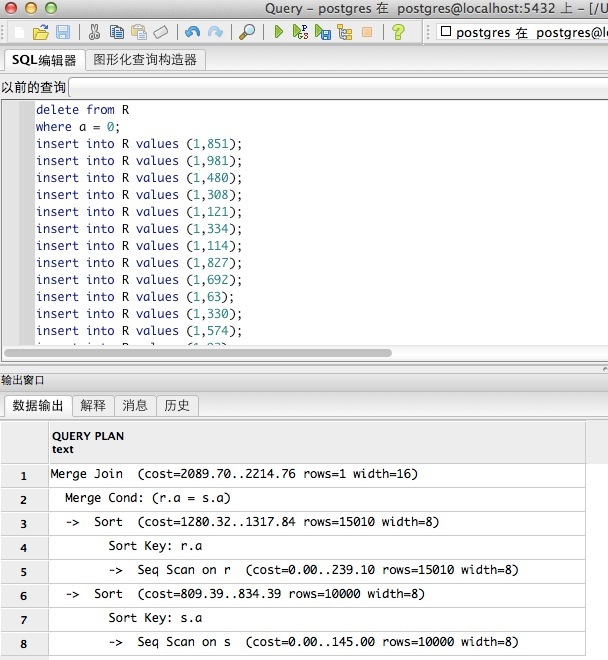
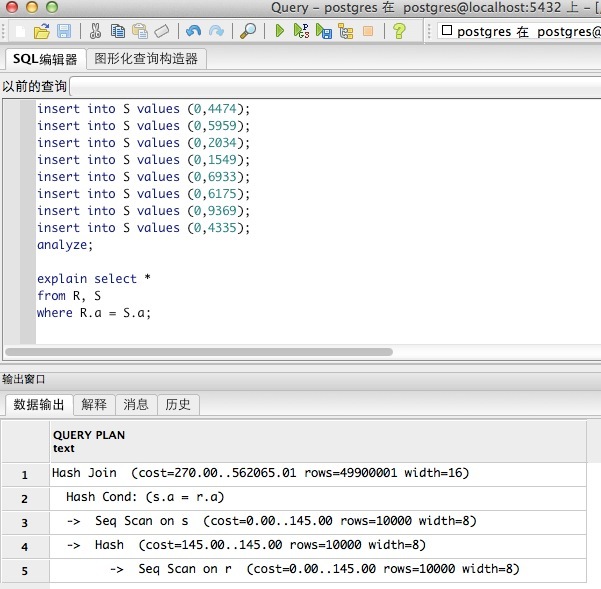
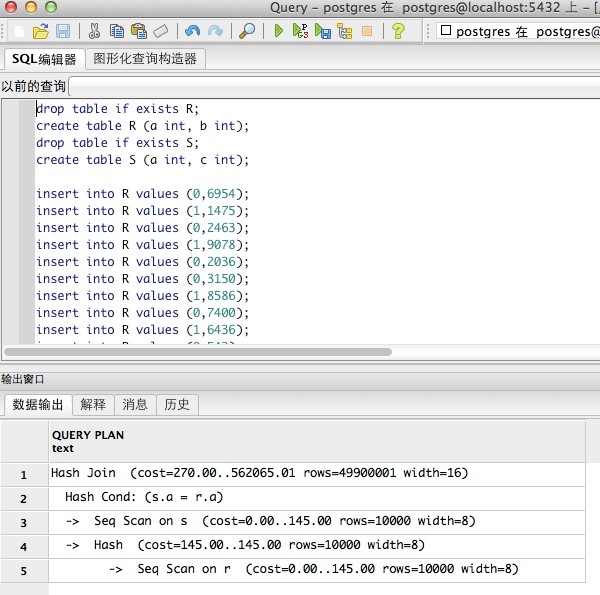
If we make the size small, M=N=4, and B=3, sort-merge join cost 2\*4\*log31+2\*4\*log31+2\*4+2\*4+4 = 16\* log31 + 20, but BNLJ only cost 4/2\*4+4 = 12.

Thus, when R and S has size 4, and 3 RAM pages are available, BNLJ is cheaper than sort-merge join.

3.2

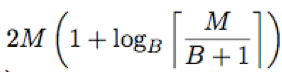
The following screenshots shows what I did for this problem. The first two is the one before updating. The later two is the one after updating. In my algorithms, I am trying to create skewed distribution. And I discovered that 10000 1s and 10000 0s will generate merge join, and it is so easy to generate hash join. So I generate 10000 number which will either be 0s or 1s for R, and 10000 0s for S. Then for step 2, I remove all the 0s from R, and insert additional 10000 1s to R. And it uses merge join instead.

My algorithm is based on my understanding that if a distribution is too skews, the system will use hash instead of merge because hash has a higher cost because it needs to do partition for the skewed partition. I mean partition all the data because one partition has too many data in it.

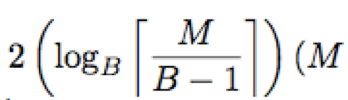


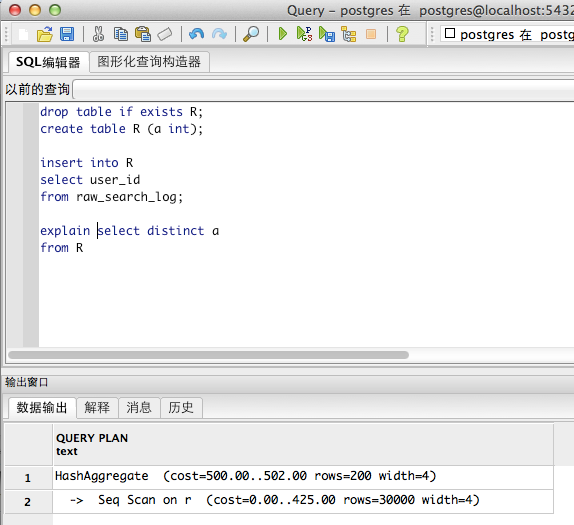
4.

(a) First sort the N integer using external merge algorithms. Where you sort each page first, and then take two pages and merge, and the two of the sorted to merge until all the N integer is sorted. Then, iterate through the N sorted integer, store the current integer as a, and go to see the next integer, if it is equal to a, then ignore it, if it is not equal to a, then output a. Repeat this step until the N integers is finished.

Cost: +M+OUT

(b) First, partition the data using the hash algorithms, which is load 1 page of the data in memory, and partition into B-1 partition. If each partition can fit in B-1 pages in memory, then stops, otherwise repeat to partition until each partition can fit in B-1 pages. Second, for each partition, remove the repeated number either by sort the numbers with the page using quicksort, or simply for each integer, search through to see if there is another number with the same value, this process will cost M + x pages where x is when we partition, some whole pages might be broke up.

Cost: )+M+x

(c) 

The data uses Hash Aggregate, which is very similar to my hashing-based algorithm.

drop table if exists R;

create table R (a int);

insert into R

select user\_id

from raw\_search\_log;

explain select distinct a

from R

This the code I wrote to run the explain. I basic import the data from the OLAP activity, and create a table R using the user\_id.

5.

Relation: R(A,B)

Description of Data: A is the key, to be simple, consider A to be an integer, B can be anything. The data should not be too skewed. Hopefully it will not be A = 1 for all the data so that there will not be any difference. And the data should be large enough so that it will use multiple pages to make a difference.

1.

Query:

select \*

from R

Description:

In the above query, the whole table will be returned so that cluster the data will not make any difference because no matter the data is clustered or not, there is no search need to do, the whole table will be returned so that the database will just return what it has not worrying about search.

Cost:

If the data uses M pages, both index will have M I/Os.

2.

Query:

select \*

from R

where A>6

Description:

In this query, the clustered index will first find A = 6, and then since the data is clustered, it will just read all the data blocks after A=6. However, in the unclustered index, even the index finds A = 6 and then move to right, it need to read random blocks since the data are saved randomly.

Cost:

Both clustered and unclustered takes about same time to find A = 6. Then the clustered data only need to read the adjacent blocks. If A in table R is 1,2,3,4,5,6,7,8,9,10,11, so that 6 is the mean. Clustered index only need to touch about half of the data block if the data distributed evenly, but Unclustered may need to touch all the data. In addition clustered index only need to read each blocks once, but unclustered may need to read a block multiple times if the memory is limited and if data block one contains A = 6 and A = 11 so that the data block might be swapped out of the memory when computes A = 7, and read it back when A = 11.

3.

Query:

Insert into R values (6, 8)

Description:

Both index will use the same time to find 6 in the index. However, for the unclustered index, it will just put this tuple at the end of the data block and create a pointer to point to the data. However, the clustered index will try to insert this tuple in the right place, so it might need to shift the entire data block if the data block is very full. It might cost O(n) to do that will unclustered only cost O(1).