## Chapter 4.5 Two - Pass Algorithm based on Hash

***Basic Thought:***

* If the data is too much to be put into main memory block, then using an available ***hash - key*** to hash all tuples of one or more Operation Objects. For all normal operations, there exists one hash - key method, it enables us to assign all tuples into one same bucket when under consideration.
* Operate one bucket each time. *(Under the situation of Binary Operation, deal with a pair bucket with the same hash value.)* Actually, the size of Operation Object has been decreased, the ratio of decreasing equals to the number of bucket. Its size is M.

### Chapter 4.5.1 Divide Relation Through Hash

***Assumption:***

* There exist Relation R, and M main memory blocks.
* h is hash function, hash function takes the whole Relation as the parameter.

***Algorithm:***

*Using M - 1 buffer blocks to initialize M - 1 buckets;*

*For each block b in Relation R DO BEGIN*

*Read block b into Mth buffer block;*

*For each tuple t in block b DO BEGIN*

*If the h(t) buffer block has no more space for the current tuple t THEN*

*BEGIN*

*Copy the h(t) buffer block to disk;*

*Using a new empty buffer block to initialize this buffer block;*

*END;*

*Copy tuple t into the h(t) buffer block;*

*END;*

*END;*

*END;*

*For each bucket DO*

*If the current bucket has tuples THEN*

*Write the current bucket back to disk;*

### Chapter 4.5.2 Remove Duplication Algorithm based on Hash

***Principle:***

* Divide Relation R into M - 1 buckets, two copies of the same tuples t will be put into the same bucket.
* Check one bucket for one time, and execute Remove Duplication Algorithm, and set the result as *Ri*.
* Merge all *Ri* collections and write the final collection back to disk.

***Cost:***

* B(R) <= M \* M.
* The total disk I/O is 3 \* B(R).

### Chapter 4.5.3 Grouping and Aggregation Algorithm based on Hash

***Procedure:***

1. Divide Relation R into M - 1 buckets, and the chosen hash function is decided by Grouping Key.
2. Then use *One - Pass Grouping Algorithm* to deal with each bucket.
3. In the second pass, only need one tuple in each group. So, even the size of bucket is larger than M, as long as the number of tuple in all groups is smaller than M, then we can use one pass to deal with the bucket.

***Cost:***

1. As long as *B(R) <= M \* M*, then we can deal with each bucket in the main memory.
2. The total disk I/O is *3 \* B(R)*.

### Chapter 4.5.4 Union, Intersection and Difference Algorithm based on Hash

***Procedure:***

1. Divide Relation R and Relation S into 2 \* (M - 1) buckets. Make sure to use the same hash function to hash tuples of R and S.

*(For example, if calculate R Union S, then hash Relation R and Relation S into M - 1 buckets, R1, R2... R(M - 1) and S1, S2... S(M - 1). For each pair Ri and Si, get Union Collection and output result.)*

*Relation R:*

|  |  |  |  |
| --- | --- | --- | --- |
| Block ONE | Block TWO | ...... | Block M - 1 |
| R1 | R2 |  | R(M - 1) |

*Relation S:*

|  |  |  |  |
| --- | --- | --- | --- |
| Block ONE | Block TWO | ...... | Block M - 1 |
| S1 | S2 |  | S(M - 1) |

*(Block M is used to calculate the Union, Intersection and Difference Collection.)*

***Attention:***

* If tuple t exists in the Relation R and Relation S, then for i, we can find tuple t in the Relation Ri and Si. Output tuple t once and neglect the other copy of tuple t.
* In order to calculate Union, Intersection and Difference Collection of Relation R and Relation S, then calling the corresponding One - Pass Algorithm. *(One - Pass Algorithms only need B(S) + B(R) disk I/O.)*
* Hash Relation R and Relation S to the buckets, it needs *(B(R) + B(S))* disk I/O.
* Store Relation R and Relation S into the disk, then it needs *(B(R) + B(S))* disk I/O.

***Cost:***

1. As long as to get the Union, Intersection and Difference collection, then just make sure the size of Ri and Si as B(R) / M - 1 and B(S) / M - 1.
2. The total disk I/O equals to 3 \* (B(R) + B(S)).
3. min(B(R), B(S)) <= M \* M.

### Chapter 4.5.5 Hash Join Algorithm

***Instruction:***

1. Join Relation R and Relation S, we need to use join *Property Y* as the hash key. Therefore, make sure that when join tuple in Relation R and S, the tuple must appears in Bucket Ri and Si.
2. Finally, *One - Pass Join Algorithm* can be used on pair buckets. *Hash - Join* finish.

***Example:***

Relation R and Relation S, their size are 1000 blocks and 500 blocks, and there has 101 buffer memory blocks are available.

Divide Relation R into 100 buckets, then each bucket will be 10 blocks; while Relation S into 100 buckets, then each bucket has 5 blocks. Because 5 blocks is far less than 101 blocks. Therefore *One - Pass Join Algorithm* works.

*Cost:*

* When hash Relation R and Relation S into buckets, it needs 1500 times disk I/O.
* When write all buckets of Relation R and Relation S back to disk, it needs 1500 times disk I/O.
* Execute *One - Pass Join Algorithm* on Relation R and Relation S, it needs 1500 times disk I/O.
* In all, it needs 4500 times disk I/O.

***Cost:***

* Hash - Join needs 3 \* (B(R) + B(S)) disk I/O.
* min(B(R), B(S)) <= M \* M, then Two - Pass Hash Join works.

Chapter 4.5.6 Save some Disk I/O

(Need Second Time Reading, Not easy to understand.)

***Principle:***

When the number of available memory is much more than one block of each bucket, then we can save disk I/O.

***Method:***

1. One choice is to use several blocks for each buckets, and write them back to the continuous blocks on the disk. *(This method can not save disk I/O, but it can make the disk I/O faster, because it decreases path searching time and rotational delay time.)*
2. The most efficient method is *Mixed - Hash - Join*. Normally, assume that join Relation R and Relation S. Relation S is the smaller relation, then build k buckets. *(k buckets is much smaller than the available M memory.)* When hash Relation S, then choose m buckets to store into the main memory but for each one in k - m buckets, just save one block of these buckets. *(As long as m \* B(S) / k + k - m <= M, then it works.)*

*(The expected size of bucket is B(S) / k, and there have m buckets in the main memory.)*

***Algorithm Improvement:***

*Prerequisite:*

* Read m buckets which have never been visited of Relation S into main memory.
* For each block in k - m buckets of Relation R, they need to be written into main memory.
* The k - m buckets of Relation S need to be written into disk.

*Steps:*

1. Read each tuple of Relation R, If the tuple t hashed to one of the m buckets of Relation S, then join the tuple t with the bucket of Relation S, just as the one - pass hash join. *(In order to make the join process quicker, then organize each bucket with some efficient search query.)*
2. If tuple t hashed to the bucket of Relation S in the disk, then just as two - pass hash join, tuple t will be sent to the corresponding block in the main memory, of course tuple t will be moved onto the disk.
3. In the second round, just connect the corresponding buckets of Relation R and Relation S. Of course, no need to join the buckets of Relation S which have been left in the main memory, since they have already been joined and output.

***Cost:***

For each bucket in Relation S and each block in Relation R, the saved time of disk I/O equals to 2. Since the bucket ratio in the main memory equals to *m/k*, so the saved disk I/O equals to *2 \* ( m/k ) \* ( B(R) + B(S) )*.

*( In order to make m/k as large as it can, then make m equal to 1 and make k as small as it can. )*

Assume that k equals to *B(S) / M*, m = 1. The largest saved disk I/O equals to:

*2 \* M / B(S) \* ( B(R) + B(S) )*

The total cost equals to:

*3 \* ( B(R) + B(S) ) - 2 \* M / B(S) \* ( B(R) + B(S) )*

***Example:***

Relation R has the size of 1000 blocks and Relation S has the size of 500 blocks, and M = 101.

1. Assume that k equals to 500 / 101 = 5, then each bucket contains 100 blocks. If we tried to put one of these buckets and one block of other four buckets, then all these blocks will be up to 104. Then it may has the main memory overflow issue.
2. Assume that k equals to 6, then each bucket contains 500/6 = 83 blocks. In First Pass, the number of disk I/O to read Relation S equals to 500, and the left times of disk I/O equals to 500 - 83 = 417 times. We also read all five buckets into disk which cost 833 times disk I/O.
3. In Second Pass, we read all buckets from disk, or do 417 + 833 = 1250 times disk I/O. Then the total disk I/O equals to 1500 + 1250 + 1250 = 4000 times disk I/O. *(This number can be used to directly compare with the 4500 times disk I/O of Sort Join or Hash Join.)*

Chapter 4.5.7 Conclusion on Algorithm based on Hash

|  |  |  |
| --- | --- | --- |
| Operator | M | Disk I/O |
| Grouping | B^1/2 | 3 \* B |
| Remove Duplication | B^1/2 | 3 \* B |
| Union | B(S)^1/2 | 3 \* (B(R) + B(S)) |
| Intersection | B(S)^1/2 | 3 \* (B(R) + B(S)) |
| Difference | B(S)^1/2 | 3 \* (B(R) + B(S)) |
| Join | B(S)^1/2 | 3 \* (B(R) + B(S)) |
| Join | B(S)^1/2 | (3 - 2 \* M / B(S)) \* (B(R) + B(S)) |

***Attention:***

1. Algorithm based on Binary Operation has a requirement about Relation Size, it only depends on the smaller object, but not like the Algorithm based on Sort depends on the sum size of two Relation.
2. Algorithm based on Sort Operation admits us to generate one result with Sequenced list, and after that this sorted list will be used. *(Maybe used in another Sort Algorithm, it may also be used as the result of Sequenced Query.)*
3. Algorithm based on Hash depends on the size equal buckets.
4. In Algorithm based on Sort, if we organize the disk, then the sorted table will be wrote on the continuous disk blocks. So If three times disk I/O of each block needs shorter rotation latent time or search time, then the algorithm may be quicker than Algorithm based on Hash.
5. If M is much larger than the amount of sorted sub - table, then we can read some continuous blocks in order to save some time latent and query time.
6. In Algorithm based on Hash, we can choose that the number of buckets is much smaller than M, then we can write several blocks of one bucket.