## Chapter 4.2 One Pass Algorithm

***Question:***

How could we execute each single operation of Logical Query Plan? (i.e. Join or Selection) Selecting an operator algorithm is a necessary part to convert Logical Query Plan to Physical Query Plan.

***Classification based on Algorithm:***

1. *Method based on Sort.*
2. *Method based on Hash.*
3. *Method based on Index.*

***Classification based on Difficulty and Cost:***

1. *One – Pass Algorithm:*

Reading data from disk for one time. This method requires that at least one Operation Object of the Operation can be stored into the main memory. (Especially for Projection and Selection)

1. *Two – Pass Algorithm:*

Data Size is too big to be put into the available main memory, but it is not the biggest data set. The feature of two – pass algorithm is to read data from disk and deal with them in one method and re-write them back to the disk, then in next pass, read the data again.

1. *Three – Pass (Multi – Pass) Algorithm:*

The Three – Pass (Multi - Pass) Algorithm has no limitation for the data size. They can finish work in three or multi - pass algorithm. They are also extension of Recursive Algorithm.

***Classification for Operator:***

1. *One Single Tuple, Unitary Operation:*

The Operation *(Selection and Projection Operator)* does not need to store the whole Relation or even the most part of the Relation into the main memory. The Operation reads one Block for one time, and generate the output by using the main memory.

1. *Whole Relation, Unitary Operation:*

The Operation *(Grouping and Deduplication Operator)* does need the whole or main part of tuples, then One – Pass Algorithm is limited to M size *(The buffer number that can be used in main memory.)* or smaller Relation.

1. *Whole Relation, Dual Operation:*

*(Union[并集], Intersection[交集], Difference[差集], Joint[连接], Product[积集] and Packet[包集])* If we want to use One – Pass Algorithm, then the size of Operator is limited to M size.

### Chapter 4.2.1 One Pass Algorithm for Single Tuple Operation

***Principle:***

No matter whether Relation R can be stored into main memory, Operations *(Selection and Projection)* have their obvious algorithms.

***Process:***

* Read one block of Relation R into the input buffer, operates on each tuple, and move the selected/projected tuple into the output buffer.

*(The output buffer maybe another operation input buffer, so the output buffer doesn’t count to the whole buffer.)*

* No matter how big the block B is, it requires that it must satisfies M >= 1.



***Cost:***

* The cost of operations *(Selection and Projection)* is decided by how Operation Objects in Relation R are provided. If Relation R is on the disk, then the cost is once execution of Table – Scan or Index – Scan.
* Normally, if Relation R is cluster, then the cost is *B(R)*; Otherwise, if Relation R is non – cluster, then cost is *T(R)*.

### Chapter 4.2.2 One Pass Algorithm for Single Tuple Operation in the Whole Relation

Let’s consider One – Pass Algorithm for Single Tuple in whole Relation: *Deduplication and Grouping*.

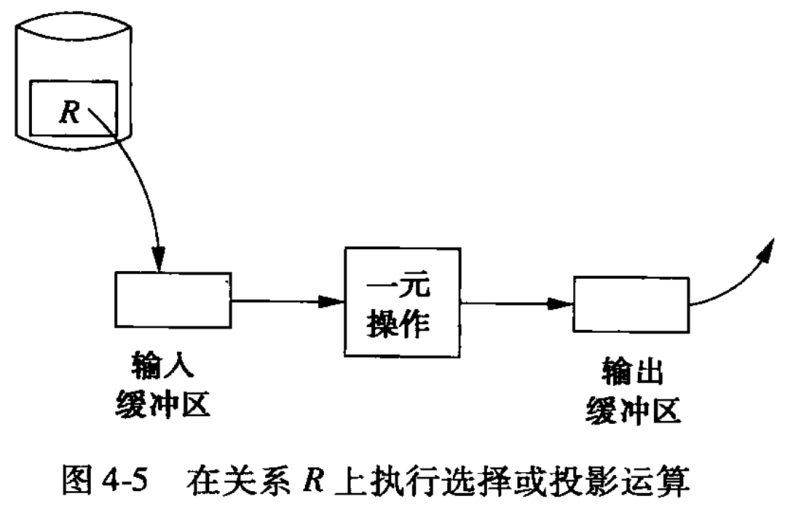
***Deduplication:***

1. ***Prerequisite:***
2. If the first time visiting the tuple, just duplicate it to the output buffer and output it.
3. If we had visited the tuple before, then no need to output it.

*(In order to support two prerequisites, then keep one copy for each tuple in main memory. Using one memory buffer to keep one block of one tuple of Relation while other M - 1 buffer area can be used to save every tuple that we saw before so far.)*

1. ***Process:***

* Take the data structure into consideration when we store the visited tuple. We compare a new tuple in Relation R with tuples that have been visited before.
* If the current tuple doesn’t equal to any in the tuple collection, then we need to copy it into output buffer and add it to the visited tuple collection.



1. ***Cost:***

Store the same number tuples into M - 1 buffer memory as M - 1 blocks of Relation R. We hope that every different tuple of Relation R can be stored into the main memory, then *B(Deduplication(R)) <= M - 1*. In the same time, *M > 1*, so *B(Deduplication(R)) <= M.*

***Attention:***

*Deduplication(R)* can not be evaluated in the Relation R. If we underrate it’s value, while *B(Deduplication(R))* is much bigger than M, then we need pay large cost for it, since the tuples in Relation R need to go back and forth into main memory.

***Grouping:***

1. ***Definition:***
2. Grouping Operation gives us zero or multiple grouping attributes or possibly one or multiple cluster attributes.
3. Create an attribute for one group in main memory, then scan tuples in Relation R one block for one time.
4. The attribute of each group includes group value and one or multiple aggregated property value.
5. ***Classification:***
6. *MIN(a)* or *MAX(a)* aggregation, *MIN(a)* is used to record the minimum value in property a of all tuples so far. While *MAX(a)* is used to record the maximum value in property a of all tuples so far. When visits one tuple in the group, if possible, then change the minimum or maximum value.
7. Any *COUNT* aggregation, if one tuple exists, then add 1 to COUNT value.
8. *SUM(a)* aggregation, if a does not equal to NULL, then add the value of property a to *SUM(a)* when scan the group so far.
9. *AVG(a)*. Two accumulated values are needed to keep. One is the number of tuples among the group and *SUM(a)* on the property a. After scanning, calculate *AVG(a)* equals to *SUM(a)* divided by *COUNT*.

*(When all tuples of Relation R have been read into input buffer memory, and have been used to calculate for each Group Aggregation, then generates one tuple output for each group.)*

***Attention:***

Not until we finish scan all tuples, then we create output. So this algorithm doesn’t fit for the iterator structure. Before using GetNext can get the next tuple, method Open must be used to do the grouping.

***Supplement:***

One main memory data structure is needed to help us find each Group attribute of known property value. Hash data structure and balanced binary tree can be effective.

*(The Key of Query can only be Grouping Attribute.)*

1. ***Cost:***
2. One Pass algorithm requires B(R) disk I/O, which is same as the One Pass of any Unitary Operation. Although M is less than B.
3. The attribute of Group is not longer than the tuple of Group, and the number of Group is less than the number of tuples.

### Chapter 4.2.3 One Pass Algorithm for Binary Operation

***Principle:***

1. The Binary Operation includes *Union*, *Intersection*, *Difference*, *Product*, and *Join*. In order to make difference between *Collection* and *Package*, we use *B* and *S* to represent *Package* and *Collection*, for example, *UB* is to get the Union Collection of Package while *-S* is to get Different Set from Collection.
2. To simplify the discussion, take *Natural Join* into consideration.
3. After rename the property, realize *Equal Join* based on the same method.
4. *Theta - Join* can be considered as adding other conditions that can not be represent in *Equal Join* after *Product Operation* or *Equal Join Operation*.

***Package Union:***

1. Process - Copy each tuple in Relation R to output, and in the same way copy each tuple in S.
2. Cost - *B(R) + B(S)*, also *M = 1*.

***Attention:***

Other Binary Operations need to read Relation R and S with smaller dataset into main memory and construct an appropriate data structure which can make tuples insert into main memory and read quickly. Same as the operation before, hash or balanced binary tree can satisfy this condition.

***Cost:***

Binary Operation on Relation R and S requires *min(B(R), B(S)) <= M*.

*(One buffer is used to read the block of the bigger Relation while M buffers are used to contain the whole smaller Relation and its data structure in the main memory.)*

***Assumption:***

R is the Relation with bigger dataset while Relation S is with the relative smaller dataset. Put Relation S into the main memory.

***Set Union:***

1. Read Relation S into M - 1 buffers into the main memory and construct one search structure, and key is the whole tuple.
2. Copy all tuples of Relation S to output.
3. Read each block of Relation R into Mth buffer and make sure whether for each tuple t in Relation R belongs to S. If t doesn’t belong to Relation S, then copy it to output; Otherwise, skip tuple t.

***Set Intersection:***

1. Read Relation S into M - 1 buffers into the main memory and construct one search structure, and key is the whole tuple.
2. Read each block of Relation R and make sure whether for each tuple t in Relation R belongs to S. If t belongs to Relation S, then copy tuple t to output; Otherwise, skip tuple t.

***Set Difference:***

Since difference can not be changeable, so we must differentiate R - S and S - R and make assumption that Relation R is with the bigger dataset. Under two situations, read Relation S into M - 1 buffers into the main memory and construct one search structure, and key is the whole tuple.

*R - S* - Read each block of Relation R and check each tuple t in this block. If tuple t is in Relation S, then skip tuple t; Otherwise, copy tuple t to output.

*S - R* - Read each block of Relation R and check each tuple t in this block. If the tuple t is in Relation S, then delete tuple t of Relation S from mian memory; Otherwise, do nothing. After consider each tuple of Relation R, then copy the remaining tuples of Relation S to output.

***Package Intersection:***

1. Read Relation S into M - 1 buffers, but for each tuple, we give it a flag ‘count’ which equals to the times that the tuple appears in Relation S. Multi - Copy tuple t doesn’t store for multi - times. Reversely, just store one copy of tuple t and connect the flag ‘count’ with tuple t, the value of *count(t)* is the times that the tuple t exists in the Relation S. *(Assume that B(S) <= M - 1.)*
2. Read each block in Relation R, for each tuple t in Relation R, check whether tuple t exists in Relation S. If not, then skip it; Otherwise, if it exists in Relation S, then check the value of count(t); If count(t) >= 1, then output t and minus 1 from count(t). Otherwise, if count(t) = 0, then do not output t.
3. Finally, we generate the copy tuple t that with the same times as in Relation S.

*Buffer Main Memory:*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Relation | S | | | | |
| Tuple | TupleS(1) | TupleS(2) | TupleS(3) | ...... | TupleS(M-1) |
| Count | T(1) | T(2) | T(3) | ...... | T(M-1) |

*Seconday Main Memory:(As Output)*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Relation | R | | | | |
| Tuple | TupleR(1) | TupleR(2) | TupleR(3) | ...... | TupleRM |

***Package Difference:***

*S - R* - Read tuples of Relation S into main memory, and for each tuple t, we give it a flag ‘count’ which equals to the times that the tuple t exists in Relation S. For each tuple t in Relation R, check whether the tuple t appears in Relation S. If it does appear, then minus one to count(t). At last, copy each tuple t from main memory to output and the time equals to *count(t)*.

*R - S* - Read tuples of Relation S into main memory, and for each tuple t, we give it a flag ‘count’ which equals to the times that the tuple t exists in Relation S. For each tuple t in Relation R, check whether the tuple t appears in Relation S. If it doesn’t appears, then we copy the tuple t to output. Otherwise, if the count(t) equals to 0, then output tuple t; Or if the count(t) > 0, then do not output tuple t but just minus 1 from count(t).

*Buffer Main Memory: (As Output)*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Relation | S | | | | |
| Tuple | TupleS(1) | TupleS(2) | TupleS(3) | ...... | TupleS(M-1) |
| Count | T(1) | T(2) | T(3) | ...... | T(M-1) |

*Seconday Main Memory:*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Relation | R | | | | |
| Tuple | TupleR(1) | TupleR(2) | TupleR(3) | ...... | TupleRM |

***Package Product:***

1. Read Relation S into M - 1 buffers into the main memory, do not need special data structure.
2. Read each block of Relation R and for each tuple t in Relation R, join tuple t with the each tuple in Relation S. After the new connected tuple is formed, then output tuple t.
3. For each tuple in Relation R, the cost of this algorithm is too much, because each tuple must match with the tuples filled in M - 1 blocks. The output space is big but the time to output each tuple is small.

***Natural Join:***

***Prerequisite:***

R(X, Y) connects with S(Y, Z), Y represents all public properties of Relation R and S, X is the property that belongs to Relation R but not Relation S, but Z is the property that belongs to Relation S but not Relation R. Assume that Relation S is with the smaller dataset.

***Process:***

1. Read all tuples in Relation S, and construct a main memory query data structure with key property Y with M - 1 buffer in the main memory.
2. Read each block of Relation R into Mth buffer. For each tuple t in Relation R, find the tuple t that matches with the tuple t on Property Y in Relation S. For each matched tuple in Relation S, form the final tuple and output it.

*(Same with all other Binary Operation algorithms, it needs B(R) + B(S) disk I/O.)*