Query Execution: Part 3

One-Pass Algorithms

I. One-pass algorithms.

I.1. Tuple-at-a-time, unary operations.

- Selection
- Projection

I.2. Full-relation, unary operations.

- Duplicate Elimination
- Grouping

I.3. Full-relation, binary operations.

- Union (set)
- Intersection (set, bag)
- Difference (set, bag)
- Product
- Join

Tuple-at-at-time Algorithms

Operations: Selection, Projection.

Projection operation requires each tuple in a relation to be accessed and manipulated.

Selection operation may require access to each tuple.

The algroithms are as follows:

```
Algorithm OnePassSelection(R, Condition)

 Open(R);          // start the iterator

                   //main loop
 do
    t := GetNext(R);  //  get next tuple
    if (CheckCondition(Condition, t) == true) output t; //check condition
 while t is NOT EMPTY;

 Close(R);

end Algorithm


Algorithm OnePassProjection(R, AttList)

 Open(R);          // start the iterator

                   //main loop
 do
    t := GetNext(R);       //  get next tuple
    if (t is NOT EMPTY)    //  if next tuple exists
      { t' = Project(AttList, t);   // perform projection
        output t';
      }
 while t is NOT EMPTY;

end Algorithm
```

The algorithms utilize the *iterators* discussed earlier for performing a full *table-scan* of the relation and providing tuple-by-tuple access to the tuples.

Evaluation

Constraints	None
I/O Cost	$B(R)$
Memory Footprint	$O(1)$

Notes: These are universal algorithms - they can can be used with relations of any size. The required memory is just 1 disk block, as the iterators/table-scan process can load data one block at a time.

Algorithms for Unary Full-Relation Operations

Duplicate Elimination

If relation R can (almost) fit in main memory we can use the following one pass algorithm to compute $\delta(R)$.

```
Algorithm OnePassDuplicateElimination(R)

Open(R);  // start the iterator

Initialize Hash Table HashTable in M-1 blocks of the buffer;

do
   t:= GetNext(R);
                // if t is in the hash table, skip it
  // otherwise, insert it
   if (seek(t,HashTable) == false) insert(t, HashTable);

while t is NOT EMPTY;
output HashTable;      // The hash table contains the restult

Close(R);

end Algorithm
```

We use $M-1$ blocks of the buffer for a data structure (hash table in the example above, also can be a binary search tree or another data structure) which stores all unique tuples found in R. When a tuple is read from R we test if it has been observed, but searching for it in the data structure. If it is there, we skip it, otherwise we insert it.

Evaluation

Constraints	$B(\delta(R)) \leq M)$
I/O Cost	$B(R)$
Memory Footprint	M

Notes: The algorithm can be used if all unique tuples in R, i.e., the result of $\delta(R)$ fits in $M-1$ pages. We simplfy this condition to $B(\delta(R)) \leq M$ for large enough M. The algorithm uses the full buffer - one page for the iterator, and $M-1$ pages to store the result.

Grouping and Aggregation

The idea behind the grouping operation γ_L is simple. The result of this operation is a collection of tuples, one tuple per group. We scan the relation R, and each time a new group needs to be created we put a container tuple for it in the buffer. For each tuple we identify which group the tuple belongs to, and modify the container according to the aggregation operations needed to perform.

The overall algorithm is as follows.

```
Algorithm OnePassGrouping(R, AttList)

Open(R);  // start the iterator
```

```
          // the hash table will contain tuple containers for groups
Initialize Hash Table HashTable in M-1 blocks of the buffer;

do
   t:= GetNext(R);
                // check to see if a new group tuple needs to
  // be added
   group := FindGroup(t,AttList,HashTable);
   if (group == -1)                          // if no group is found
      { group := insertNewGroup(t,AttList,HashTable);} // create a new group

   aggregate(group, t, AttList);// modify the contents of group's tuple according
                               // the desired aggregate operations

while t is NOT EMPTY;

output HashTable;      // The hash table contains the restult

Close(R);

end Algorithm
```

For each aggregate that needs to be computed `aggregate()` function douce the following:

`MIN`, `MAX`	compare current value in the group tuple with the value in the tuple `t`, record the smaller (larger) of the two in `group`
`COUNT`	add one to the counter
`SUM`	add the value from `t` to the value in `group`
`AVG`	maintain both the sum and the count values, change as above

Evaluation

Constraints	$B(\gamma_L(R)) \leq M)$
I/O Cost	$B(R)$
Memory Footprint	M

Full-relation, binary operations

The overall idea behind all these algorithms is the same. Almost all one-pass algorithms for binary operations work as follows:

1. Load the smaller of the two relations into main memory (typically store it in a data structure that allows for easy seek() operation).
2. Scan the larger relation, one block at a time (using iterators for tuple-by-tuple access), check the relationship between current tuple and the tuples from the smaller relation, form output.

The scan requires just 1 block, so, the remaining buffer space, $M - 1$ blocks, can be allocated to storage of the smaller relation.

Union

Bag Union operation is very simple. Because no duplicate elimination is needed, it can be achieved, by simply scanning and outputing the first relation, followed by scanning and outputing the second one.

```
Algorithm BagUnion(R, S)

Open(R);                  // first, scan R
while not EOF(R)
   { t := GetNext(R);
     output t;
    }
Close(R);

Open(S);                  // next, scan S
while not EOF(S)
    { t:= GetNext(S);
      output t;
    }
Close(S);

end Algorithm
```

Set Union is somewhat more complex, as it requires duplicate elimination "on the go". The algorithm, thus is similar to the one-pass algorithm for $\gamma(R)$, except, the starting point is, the smaller of the two relations is loaded in the buffer.

```
Algorithm SetUnion(R,S)

// we assume that B(R) > B(S)

 Initialize HashTable;
 Load S into HashTable; // relation S is hashed in main memory;

  Open(R);              // start the iterator on R
  do
    t = GetNext(R);
    if (seek(t, HashTable) == false) output t; // if t is not in S output it
  while t is NOT EMPTY;

  Close(R);

  output HashTable;  // output S
  end Algorithm
```

Notes: Only one disk block is needed for the iterator on R, therefore, S can be hashed in all remaining blocks. The iterator is used to compute $R - S$, while the final `output HashTable;` operation, adds S: $(R - S) \cup S = R \cup S$.

Evaluation

Algorithm	`BagUnion`	`SetUnion`
Constraints	none	$B(S) \leq M)$
I/O Cost	$B(R) + B(S)$	
Memory Footprint	O(1)	M

Intersection

Set Intersection operation can be implemented using a modification of the `SetUnion()` algorithm.

```
Algorithm SetIntersection(R,S)

// we assume that B(R) > B(S)

 Initialize HashTable;
 Load S into HashTable; // relation S is hashed in main memory;

  Open(R);              // start the iterator on R
  do
    t = GetNext(R);
    if (seek(t, HashTable) == true) output t; // if t is in S output it
  while t is NOT EMPTY;

  Close(R);

 end Algorithm
```

Notes: There are two differences between SetUnion and SetIntersection. First, the hash table is not returned. Second, `t` is output when it is found in the hash table.

Bag Intersection algorithm is more tricky. Here, we need to keep track of the count for each tuple. The algorithm works in two steps. First, for the smaller relation, S, we perform an analog of the one-pass duplicate elimination algorithm, except, we store count of number of occurrences for each tuple. On the second step, we scan R, and any time we detect a tuple in the hash table, we output it, and decrease its counter.

```
Algorithm SetIntersection(R,S)
// we assume that B(R) < B(S)

// Step 1: initialize the hash table for S

 Open(S);  // start the iterator
```

```
Initialize Hash Table HashTable in M-1 blocks of the buffer;

do
   t:= GetNext(S);
                  // if t is in the hash table, skip it
                  // otherwise, insert it
   if (seek(t,HashTable) == false)
               {insert(t, HashTable)
counter[t] := 1;}
   else  // tuple t is already in the hash table
        counter[t] := counter[t] + 1;  // increment the counter
while t is NOT EMPTY;
Close(S);

// Step 2: scan R, compute intersection

 Open(R);
  do
    t := GetNext(R);
    if (seek(t, HashTable) == true)  // t is found in HashTable
       {
         output t;
 counter[t] := counter[t] - 1;   // decrement the counter
                                              // if the counter drops to 0
   if (counter[t] == 0) delete(t, HashTable);  //remove the tuple
        }
   while t is NOT EMPTY;
   Close(R);

 end Algorithm
```

Evaluation

Algorithm	`BagIntersection`	`SetIntersection`
Constraints	$B(S) \leq M$	
I/O Cost	$B(R) + B(S)$	
Memory Footprint	M	

Notes: For `SetIntersection` algorithm the constraint is somewhat different. On one hand, the true condition is $\gamma(B(S)) \leq M - 1$. On the other hand, there is an overhead incurred by the algorithm, because it has to store counters. But $B(S) \leq M$ is a reasonable approximation.

Difference

Difference is not commutative. So, we need two one-pass algorithms: one for $R - S$, when $B(R) > B(S)$ and one for $R - S$ when $B(R) < B(S)$, for both bag and set differences.

Set Difference algorithms. For the case $B(R) > B(S)$ a modification of

SetUnion will work. As we saw above, this algorithm computes $R - S$ within the main iterator loop. For the case of $B(R) < B(S)$, we need to store R in main memory, and change the processing slightly.

```
Algorithm SetDifferenceS(R,S)

 // we assume that B(R) > B(S)

  Initialize HashTable;
  Load S into HashTable; // relation S is hashed in main memory;

   Open(R);              // start the iterator on R
   do
     t = GetNext(R);
     if (seek(t, HashTable) == false) output t; // if t is not in S output it
   while t is NOT EMPTY;

   Close(R);
   end Algorithm


Algorithm SetDifferenceR(R,S)

 // we assume that B(R) < B(S)

  Initialize HashTable;
  Load R into HashTable; // relation R is hashed in main memory;

   Open(S);              // start the iterator on S
   do
     t = GetNext(S);
     if (seek(t, HashTable) == true) delete(t, HashTable);
                           // if t is in R, delete it from the hash table
   while t is NOT EMPTY;

   Close(S);
   output HashTable;

   end Algorithm
```

For Bag Difference we must use the same trick as for *Bag Intersection* — store the counters for each tuple.

```
Algorithm SetDifferenceS(R,S)
// we assume that B(R) < B(S)

// Step 1: initialize the hash table for S

 Open(S);  // start the iterator
```

```
 Initialize Hash Table HashTable in M-1 blocks of the buffer;

 do
    t:= GetNext(S);
                  // if t is in the hash table, skip it
                  // otherwise, insert it
    if (seek(t,HashTable) == false)
                {insert(t, HashTable)
 counter[t] := 1;}
    else  // tuple t is already in the hash table
         counter[t] := counter[t] + 1;  // increment the counter
 while t is NOT EMPTY;
 Close(S);

 // Step 2: scan R, compute difference

  Open(R);
   do
     t := GetNext(R);
     if (seek(t, HashTable) == true)  // t is found in HashTable
        {
  counter[t] := counter[t] - 1;   // decrement the counter
                                          // if the counter drops to 0
    if (counter[t] == 0) delete(t, HashTable);  //remove the tuple
         }
      else // t is NOT / NO LONGER in hash table
          {output t;}

    while t is NOT EMPTY;
    Close(R);

  end Algorithm


Algorithm SetDifferenceR(R,S)
// we assume that B(R) > B(S)

// Step 1: initialize the hash table for R

 Open(R);  // start the iterator

 Initialize Hash Table HashTable in M-1 blocks of the buffer;

 do
    t:= GetNext(R);
                  // if t is in the hash table, skip it
                  // otherwise, insert it
    if (seek(t,HashTable) == false)
                {insert(t, HashTable)
 counter[t] := 1;}
```

```
   else  // tuple t is already in the hash table
        counter[t] := counter[t] + 1;  // increment the counter
while t is NOT EMPTY;
Close(R);

// Step 2: scan S, compute difference

 Open(S);
  do
    t := GetNext(S);
    if (seek(t, HashTable) == true)  // t is found in HashTable
       {
 counter[t] := counter[t] - 1;   // decrement the counter
                                          // if the counter drops to 0
    if (counter[t] == 0) delete(t, HashTable);  //remove the tuple
         }

    while t is NOT EMPTY;
    Close(R);

    output HashTable;

  end Algorithm
```

Evaluation

Algorithm	`BagDifferenceS`	`BagDifferenceR`	`SetDifferenceS`	`SetDifferenceR`
Constraints	$B(S) \leq M$	$B(R) \leq M$	$B(S) \leq M$	$B(R) \leq M$
I/O Cost	$B(R) + B(S)$			
Memory Footprint	M			

Product

Product is a simple but time-consuming operation. A one-pass algorithm for product is striaghtforward. One relation is read into the buffer, the second relation is scanned and for each tuple from the first relation and the current tuple from the second, their product is formed and output.

```
Algorithm OnePassProduct(R,S)

// assume B(R) > B(S)

Read S into main memory;

Open(R);
do
  t := GetNext(R);
  if (t is NOT EMPTY)
      for each t' in S  output (t,t');
```

```
while t is NOT EMPTY;
Close(R);

end Algorithm
```

Evaluation

Constraints	$B(S) \leq M$
I/O Cost	$B(R) + B(S)$
Memory Footprint	M

Join

The algorithms below work for natural join and equijoins. They can be extended easily to arbitrary join conditions.

One-pass algorithm for join can be build out of one-pass algorithm for product. Here, instead of outputing every pair of tuples, we will first test, if the join condition is satisfied.

```
Algorithm NaiveOnePassJoin(R,S, JoinCondition)

// assume B(R) > B(S)

Read S into main memory;

Open(R);
do
  t := GetNext(R);
  if (t is NOT EMPTY)
      for each t' in S
         if SatisfyJoinCondition(t,t', JoinCondition) output Join(t,t');

while t is NOT EMPTY;
Close(R);

end Algorithm
```

This algorithm is naive, as it still requires checking every pair of tuples. By arranging S in main memory indexed by the values from the `JoinCondition`, we can speed up memory computations.

```
Algorithm OnePassJoin(R,S, JoinCondition)

// assume B(R) > B(S)
Initialize Index;
Read S into Index, index on attributes from JoinCondition;
```

```
Open(R);
do
  t := GetNext(R);
  if (t is NOT EMPTY)
     { T = seek(Index, t); // find in Index, tuples with
                           // values of JoinCondition equal
   // to those in tuple t, return result
   // as set T
for each t' IN T  output Join(t,t',JoinCondition);
      }

while t is NOT EMPTY;
Close(R);

end Algorithm
```

Evaluation

Algorithm	`NaiveOnePassJoin`	`OnePassJoin`
Constraints	$B(S) \leq M$	
I/O Cost	$B(R) + B(S)$	
Memory Footprint	M	