Solutions Manual

Chapter 15

Section 15.2

Exercise 15.2.1

(a) The pseudocode for the iterator for projection $\pi_{A_1,A_2,..,A_n}$(R) is as follows:

```
Open ()
{
  R.Open();
}

GetNext ()
{
  xTuple ← R.GetNext();
  if (xTuple ≠ NotFound)
  {
    xTuple ← Construct the tuple consisting of the
             attributes $A_1, A_2, ..., A_n$;
  }
  return xTuple;     // Note that xTuple could be
                     // either NotFound or some tuple
}

Close ()
{
  R.Close();
}
```

(b) The pseudocode for the iterator for distinct δ(R) is as follows:

```
Open ( )
{
   R. Open ( ) ;
   Allocate Search Structure H for tuples of R and
   initialize it to empty;
}

GetNext ( )
{
   repeat forever
   {
      xTuple ← R. GetNext ( ) ;
      if (xTuple = NotFound) return NotFound ;
      / *
        * At this point we got the tuple. Search for it using H.
        * If found, ignore the tuple and get next one. Otherwise,
        * insert the tuple into H and return it.
        */
      if (H. Find ( xTuple ) = false )
      {
        H. Insert ( xTuple ) ;
        return xTuple ;
      }
   }  // of repeat forever
}

Close ( )
{
   R. Close ( ) ;
   Deallocate Search Structure H;
}
```

(c) The pseudocode for the iterator for grouping γ_L(R) is as follows:

```
Open ( )
{
   R. Open ( ) ;
   Allocate Search Structure H for grouping attributes of R and
   initialize it to empty;
   repeat  forever
   {
      xTuple ← R. GetNext ( ) ;
      if  ( xTuple  =  NotFound )
      {
         / *
           * We are done retrieving tuples. At this point H contains all the
           * groups. We position to the first group in H and return.
           * Note that there may not be any groups (i.e. R is empty), in
           * which case the position will be set to NotFound.
           */
         nextGroup ← H. GetFirst ( ) ;        // nextTuple can be NotFound
         R. Close ( ) ;
         return ;
      }
      / *
        * At this point we have a new tuple, so search for the group in H
        * that corresponds to the grouping attributes of the tuple.
        * If group is not found, insert new group into H and perform the
        * needed accumulation(s) for each aggregation operator in L.
        * If group is found, perform the needed accumulation(s) for each
        * aggregation operator in L.
        */
      xGroup ← H. Find (grouping attributes of  xTuple ) ;
      if  ( xGroup  =  NotFound )
      {
         xGroup ← create new group ;
         H. Insert (grouping attributes of  xTuple ,  xGroup ) ;
      }
      Perform accumulation(s) for each aggregation
      operator in L for  xGroup ;
   }  // of repeat forever
}
```

```
GetNext ( )
{
  / *
   * All groups were created during Open. Note that we did not finalize
   * groups during Open since it is not known when a certain group ends
   * until R is read completely. And it would be wasteful to loop over
   * all groups at the end of Open. Open will position nextGroup to the
   * first group (if any). Here, we take nextGroup and finalize it. Then
   * return the corresponding tuple, and set nextGroup to the next group
   * in H (if any). If no more groups, we return NotFound.
   */
  if (nextGroup ≠ NotFound)
  {
    currGroup ← nextGroup;
    Finalize currGroup;
    nextGroup ← H.GetNext;                  // position to next group
    return tuple associated with the currGroup;
  }
  return NotFound;
}

Close ( )
{
  // note that R was closed already (at completion of Open)
  Deallocate Search Structure H;
}
```

(d) The pseudocode for the iterator for set union R∪S is as follows:

```
Open ( )
{
   S . Open ( ) ;
   Allocate Search Structure H for tuples of S and
   initialize it to empty;
   workingWithS ← true ;              // start processing with S
}

GetNext ( )
{
   / *
    * We first process all tuples in S, then all tuples in R. For each
    * tuple, if it is from S insert it into H and return it. If it is
    * from R, search H. If found, ignore the tuple, otherwise return it.
    */
   if ( workingWithS = true )         // if working with S
   {
      xTuple ← S . GetNext ( ) ;
      if ( xTuple ≠ NotFound )        // got tuple from S
      {
         H. Insert ( xTuple ) ;
         return xTuple ;
      }
      // we get here if there are no more tuples in S
      S . Close ( ) ;
      R. Open ( ) ;
      workingWithS ← false ;          // working with R now
   }
   // we get here if we are working with R
   repeat
   {
      xTuple ← R. GetNext ( ) ;
   }
   until ( xTuple = NotFound ) or
         (H. Find ( xTuple ) = true ) ;

   return xTuple ;                    // return matched tuple or NotFound
}
```

```
Close ( )
{
    R.Close ( );
    Deallocate Search Structure H;
}
```

(e) The pseudocode for the iterator for set intersection R∩S is as follows:

```
Open ( )
{
   S . Open ( ) ;
   Allocate Search Structure H for tuples of S and
   initialize it to empty;
   // Insert all tuples of S into H. To be searched later.
   repeat forever
   {
      xTuple ← S . GetNext ( ) ;
      if ( xTuple = NotFound )          // no more tuples in R
      {
         S . Close ( ) ;
         R . Open ( ) ;
         return ;
      }
      H. Insert ( xTuple ) ;
   }  // of repeat forever
}

GetNext ( )
{
   // For each tuple in R, search H. If found, return it.
   repeat
   {
      xTuple ← R . GetNext ( ) ;
   }
   until ( xTuple = NotFound ) or
         (H. Find ( xTuple ) = true ) ;
   return xTuple ;                  // return matched tuple or NotFound
}

Close ( )
{
   R . Close ( ) ;
   Deallocate Search Structure H;
}
```

(f) The pseudocode for the iterator for set difference R–S is as follows:

```
Open ( )
{
   S . Open ( ) ;
   Allocate Search Structure H for tuples of S and
   initialize it to empty;
   // Insert all tuples of S into H. To be searched later.
   repeat forever
   {
      xTuple ← S . GetNext ( ) ;
      if ( xTuple = NotFound )          // no more tuples in S
      {
         S . Close ( ) ;
         R . Open ( ) ;
         return ;
      }
      H . Insert ( xTuple ) ;
   }  // of repeat forever
}

GetNext ( )
{
   // For each tuple in R, search H. If not found, return it.
   repeat
   {
      xTuple ← R . GetNext ( ) ;
   }
   until ( xTuple = NotFound ) or
         (H . Find ( xTuple ) = false ) ;
   return xTuple ;            // return tuple or NotFound
}

Close ( )
{
   R . Close ( ) ;
   Deallocate Search Structure H;
}
```

The pseudocode for the iterator for set difference S–R is as follows:

```
Open ( )
{
   S . Open ( ) ;
   Allocate Search Structure H for tuples of S and
   initialize it to empty;
   / *
    * Insert all tuples of S into H. Then read R, and
    * for each tuple in R, search H. If found, delete
    * the H entry for this tuple.
    */
   xTuple ← S . GetNext ( ) ;
   if ( xTuple = NotFound ) return ;     // S is empty, H will be empty
   repeat
   {
      H. Insert ( xTuple ) ;
      xTuple ← S . GetNext ( ) ;
   }
   until ( xTuple = NotFound ) ;
   S . Close ( ) ;
   R. Open ( ) ;
   repeat forever
   {
      xTuple ← R. GetNext ( ) ;
      if ( xTuple = NotFound )          // no more tuples in R
      {
         Prepare to iterate through entries of H ;
         R. Close ( ) ;
         return ;
      }
      if (H. Find ( xTuple ) = true )
      {
         H. Delete ( xTuple ) ;
      }
   }  // of repeat forever
}
```

```
GetNext ( )
{
   / *
     * At this point the only entries in H are the ones we need
     * to return, so return them one at the time. That is, return
     * tuple associated with the next H entry or NotFound if we
     * are done iterating through H.
     */
   return  H. GetNext ( ) ;
}

Close ( )
{
   Deallocate Search Structure H;
}
```

(g) The pseudocode for the iterator for bag intersection R∩S is as follows:

```
Open ( )
{
   S . Open ( ) ;
   Allocate Search Structure H for tuples of S and
   initialize it to empty; In addition to storing tuples,
   we keep a counter for each entry in H, which represents
   the number of times a given tuple appears in S.
   / *
    * We first insert all tuples of S into H, maintaining the
    * counter accordingly (i.e. if tuple is in H already,
    * increment the counter. Otherwise, insert tuple into H
    * initializing counter to 1).
    */
   xTuple ← S . GetNext ( ) ;
   if  ( xTuple  =  NotFound )
   {
      currTuple ← NotFound ;     // early out
      S . Close ( ) ;
      return ;
   }
   repeat
   {
      H. Insert ( xTuple ) ;       // will either insert new entry with
                                   // counter = 1, or will increment
                                   // the counter for existing entry
      xTuple ← S . GetNext ( ) ;
   }
   until  ( xTuple  =  NotFound )
   S . Close ( ) ;
   R. Open ( ) ;
   currTuple ← R. GetNext ( ) ;
}
```

```
GetNext ( )
{
   / *
    * At this point, we have Search Structure H that has tuples
    * along with the counter. For each tuple in R, search H.
    * If found, and the corresponding counter is not zero,
    * decrement the counter and return the tuple.
    */
   if ( currTuple = NotFound )
   {
      return NotFound ;
   }
   repeat
   {
      Search H for currTuple entry with non-zero counter ;
      if ( entry found )
      {
         Decrement the associated counter ;
         return currTuple ;
      }
      currTuple ← R. GetNext ( ) ;
   }
   until ( currTuple = NotFound ) ;
   return NotFound ;
}

Close ( )
{
   R. Close ( ) ;
   Deallocate Search Structure H ;
}
```

(h) The pseudocode for the iterator for bag difference R–S is as follows:

```
Open ()
{
    R.Open ();
    currTuple ← R.GetNext ();
    if (currTuple = NotFound)  // R is empty
    {
        return;                // early out
    }
    S.Open ();
    Allocate Search Structure H for tuples of S and
    initialize it to empty; In addition to storing tuples,
    we keep a counter that represents the number of times
    a given tuple occurs in S.
    /*
     * We first insert all tuples of S into H, maintaining the
     * counter accordingly (i.e. if tuple is in H already,
     * increment counter. Otherwise, insert tuple into H
     * initializing counter to 1).
     */
    xTuple ← S.GetNext ();
    repeat while(xTuple ≠ NotFound);
    {
        H.Insert(xTuple);      // will either insert new entry with
                               // counter = 1, or will increment
                               // the counter for existing entry
        xTuple ← S.GetNext ();
    }
    S.Close ();
    return;
}
```

```
GetNext ( )
{
   / *
     * For each tuple in R, search H. If not found, return it.
     * If found, check the counter. If counter is zero, return
     * the tuple. Otherwise, decrement the counter.
     */
   if (currTuple = NotFound)
   {
      return NotFound;
   }
   repeat
   {
      Search H for currTuple entry with non-zero counter;
      if (entry not found)
      {
         return currTuple;
      }
      // we come here if entry is found
      Decrement the associated counter;
      currTuple ← R.GetNext();
   }
   until (currTuple = NotFound);
   return NotFound;
}

Close ( )
{
   R.Close();
   Deallocate Search Structure H;
}
```

The pseudocode for the iterator for bag difference S–R is as follows:

```
Open ( )
{
    S . Open ( ) ;
    Allocate Search Structure H for tuples of S and
    initialize it to empty; In addition to storing tuples,
    we keep a counter that represents the number of times
    a given tuple occurs in S.
    / *
      * We first insert all tuples of S into H, maintaining the
      * counter accordingly (i.e. if tuple is in H already,
      * increment counter. Otherwise, insert tuple into H
      * initializing counter to 1).
      */
    xTuple ← S . GetNext ( ) ;
    if  ( xTuple  =  NotFound )          // S is empty
    {
        currTuple ← NotFound ;           // early out
        S . Close ( ) ;
        return ;
    }
    repeat
    {
       H. Insert ( xTuple ) ;        // will either insert new entry with
                                     // counter = 1, or will increment
                                     // the counter for existing entry
        xTuple ← S . GetNext ( ) ;
    }
    until  ( xTuple  =  NotFound ) ;
    S . Close ( ) ;
    / *
      * We now read R, and for each R tuple we search H.
      * If found, and the associated counter is not zero,
      * decrement the counter.
      */
    R . Open ( ) ;
    xTuple ← R . GetNext ( ) ;
    if  ( currTuple  =  NotFound )   // R is empty
    {
        return ;                         // early out
    }
    repeat
    {
        Search H for xTuple entry with non-zero counter;
        if  (entry found)
        {
            Decrement the associated counter;
        }
        xTuple ← R.GetNext();
    }
    until (xTuple = NotFound);
    Prepare to iterate through entries of H;
    R.Close();
    return;
}

GetNext()
{
    /*
     * At this point the only entries in H we need to return
     * are the ones with the positive counter. The counter also
     * designates the number of times the associated tuple needs
     * to be returned, so we reuse it and decrement it for each
     * returned tuple.
     */
    Get next H entry that has counter > 0;
    if  (no more such entries)
    {
        return  NotFound;
    }
    Decrement the counter for this entry;
    return  tuple associated with this entry;
}

Close()
{
  Deallocate Search Structure H
}
```

(i) The pseudocode for the iterator for product R×S is as follows:

```
Open()
{
   S.Open();
   // Read all tuples of S and save them into memory
   xTuple ← S.GetNext();
   if (xTuple = NotFound)      // S is empty
   {
      currTupleR ← NotFound;    // early out
      S.Close();
      return;
   }
   repeat
   {
      Save xTuple into dedicated memory location;
      xTuple ← S.GetNext();
   }
   until (xTuple ← NotFound);
   S.Close();
   R.Open();
   currTupleR ← R.GetNext();
   currTupleS ← get first S tuple from memory;
   return;
}

GetNext()
  {
   /*
    * At this point we have S in memory, so for each tuple read from R
    * we concatenate that tuple with all of the tuples of S. Note that
    * at the entry to GetNext we are already positioned to the R and S
    * tuples that need to be concatenated and returned
    */
   if (currTupleR = NotFound)
   {
      return NotFound;
   }
   xTuple ← concatenation of currTupleR and currTupleS;
   // setup for the next invocation
   currTupleS ← get next S tuple from memory;
```

```
    if (currTupleS = NotFound)              // S is exhausted
    {
        currTupleR ← R.GetNext();
        currTupleS ← get first S tuple from memory;
    }
    return xTuple;
}

Close()
{
    R.Close();
}
```

(j) The pseudocode for the iterator for natural join R(X,Y)⋈S(Y,Z) is as follows:

```
Open()
{
  S.Open();
  Allocate Search Structure H for attributes of Y and
  initialize it to empty;
  // Insert all tuples of S into H. To be searched later.
  repeat forever
  {
    xTuple ← S.GetNext();
    if (xTuple = NotFound)          // no more tuples in S
    {
      S.Close();
      R.Open();
      currTuple ← R.GetNext();
      return;
    }
    H.Insert(xTuple);
  }  // of repeat forever
}

GetNext()
{
  /*
   * For each tuple in R, search H based on attributes of Y.
   * Return any matching tuples (joined together). Note that
   * there could be multiple tuples that match current R tuple.
   * We assume that H.FindNext will return all such tuples
   * one at the time, and that H.Reset will re-initialize H
   * so that H.FindNext starts a new search.
   */
  if (currTuple = NotFound)
  {
    return NotFound;
  }
  repeat
  {
    if (H.FindNext(currTuple) = true)
    {
            return joined currTuple and matched tuple from H;
        }
        // We come here if did not find a match; read next tuple in R
        currTuple ← R.GetNext();
        // Re-init H to ensure duplicate tuples can be searched again
        H.Reset();
    }
    until (currTuple = NotFound);
    return NotFound;
}

Close()
{
    R.Close();
    Deallocate Search Structure H;
}
```

Exercise 15.2.2

(a) Projection is not a blocking operator.

(b) Distinct is not a blocking operator.

(c) Grouping is a blocking operator.

(d) Set union is not a blocking operator.

(e) Set intersection is not a blocking operator.

(f) Set difference can be blocking or not depending on which relation is read first. For instance, when computing R–S, if S is read into M-1 blocks and then each block of R is read, the operation is not blocking. However, if R is read into M-1 blocks and then each S block is read, the operation is blocking.

(g) Bag intersection is not a blocking operator.

(h) Bag difference can be blocking for the same reason as in (f)

(i) Product is not a blocking operator.

(j) Natural join is not a blocking operator.

Exercise 15.2.3

If one or both arguments were not clustered, the entries for the Approximate M required would not change, since the sizes of the relations when copied into designated memory buffers is not affected by clustering. What would change is the Disk I/O cost. The cost would increase based on the cluster ratio of the relations. The cost could go up as high as T (i.e. in all listed formulas B would be replaced by T).

Exercise 15.2.4

(a)
```
Read R into memory.
For each tuple x in S:
   if x joins with any tuples T of R that are in memory,
   then output all those tuples T and then delete them
        from the memory.
```

```
(b) Read S into memory.
    For each tuple x in R:
       if x joins with any tuples of S that are in memory,
       then output x.

(c) Read R into memory.
    For each tuple x in S:
       if x joins with any tuples T of R that are in memory,
       then delete those tuples T from the memory.
    Output all tuples left in memory (if any).

(d) Read S into memory.
    For each tuple x in R:
       if x joins with any tuples of S that are in memory,
       then output x.

(e) Read R into memory.
    For each tuple x in S:
       if x joins with any tuples T of R that are in memory,
       then output all those joined tuples and then mark
            the tuples T as "joined".
    Output any tuples of R that are not marked "joined",
    padding them with nulls.

(f) Read S into memory.
    For each tuple x in R:
       if x joins with any tuples T of S that are in memory,
       then output all those joined tuples.
       Otherwise (x does not join with any tuples of S),
            pad x with nulls and output it.

(g) Notice that R $\overset{\circ}{\bowtie}_R$ S is equivalent to S $\overset{\circ}{\bowtie}_L$ R,
    thus the algorithm is identical to the one in (e) with
    R and S exchanged.

(h) Notice that R $\overset{\circ}{\bowtie}_R$ S is equivalent to S $\overset{\circ}{\bowtie}_L$ R,
    thus the algorithm is identical to the one in (f) with
    R and S exchanged.
```

```
(i) Read R into memory.
    For each tuple x in S:
       if x joins with any tuples T of R that are in memory,
       then output all those joined tuples and then mark
            the tuples T as "joined".
       Otherwise (x does not join with any tuples of R),
            pad x with nulls and output it.
    Output any tuples of R that are not marked "joined",
    padding them with nulls.
```

Section 15.3

Exercise 15.3.1

```
Open()
{
  m ← NotFound;
  R.Open();
  r ← R.GetNext();
  if (r ≠ NotFound)
  {
    S.Open();
    Create relation M in memory (size M-1 blocks)
    // insert tuples from S into M
    repeat
    {
      xTuple ← S.GetNext();
      if (xTuple = NotFound)
      {
        S.Close();
        leave loop
      }
      insert xTuple into M
    } until M-1 blocks are filled in
    M.Open();
    m ← M.GetNext();
  }
}
```

```
GetNext()
{
  if (m = NotFound)            // handle the cae when one
  {                            // of the relations is empty
    return NotFound;
  }

  repeat while(r and m do not join)
  {
    if (r ≠ NotFound)          // R was not exhausted
    {
      r ← R.GetNext();
    }
    if (r = NotFound)          // R exhausted, reset R
    {                          // for next tuple in M
      R.Close();
      m ← M.GetNext();
      if (m = NotFound)        // M exhausted, need to
      {                        // read more of S
        if (S is closed)       // already done with S
        {
          return NotFound;
        }
        // insert more tuples from S into M
        Re-init M to empty;
        repeat
        {
          xTuple ← S.GetNext();
          if (xTuple = NotFound)
          {
            S.Close();
            leave loop
          }
          insert xTuple into M
        } until M-1 blocks are filled in
        m ← M.GetNext();
        if (m = NotFound)
        {
          return NotFound;
```

```
                }
            }
            R.Open();
            r ← R.GetNext();
        }
    }
    xTuple ← join of r and m;
    r ← R.GetNext();
    return xTuple;
}

Close()
{
    R.Close();
    M.Close();    // this will free storage
}
```

Exercise 15.3.2

Using the formula: Disk I/O cost = $B(S) + \dfrac{B(R)B(S)}{M-1}$

we get: Disk I/O cost = $10000 + \dfrac{10000 \times 10000}{999} \approx 110101$.

Note, however, that the formula in the textbook is an approximation in the sense that it assumes that M-1 divides B(S). If a more precise formula is used:

Disk I/O cost = $B(S) + B(R)\left\lceil \dfrac{B(S)}{M-1} \right\rceil$

the result for this exercise would be: $10000 + 10000\left\lceil \dfrac{10000}{999} \right\rceil = 120000$.

Exercise 15.3.3

Using the formula: Disk I/O cost = $B(S) + \dfrac{B(R)B(S)}{M-1}$

We need to find out the value of M, such that the number of disk I/O's is no more than some value x. That is, $B(S) + \dfrac{B(R)B(S)}{M-1} \leqslant x$,or $M \geqslant 1 + \dfrac{B(R)B(S)}{x - B(S)}$.

Substituting for known values $B(R) = B(S) = 10000$, we get $M \geqslant 1 + \dfrac{100000000}{x - 10000}$.

Now we have a generic formula which, given some upper bound for the disk I/O x, will give us the value of M that would be needed.

(a) For $x = 100000$, M would need to be at least 1113.

(b) For $x = 25000$, M would need to be at least 6668.

(c) For $x = 15000$, M would need to be at least 20001.

Exercise 15.3.4

(a) One way to do better than $\frac{T(R)T(S)}{M}$ would be to use only one memory block to read the smallest relation, say S, one block at the time. For each block read, move only the tuples of S to fill the remaining M-1 blocks of memory. This way we essentially reduce the problem to the case where one of the relations (S) is not clustered any more and we can apply the block-based Nested-loop join algorithm discussed in Section 15.3.3. The cost formula, therefore, would be: Disk I/O cost = $T(S) + \frac{T(R)B(S)}{M-1}$, or in the simplified version: $\frac{T(R)B(S)}{M}$

We can do even better than that if we are allowed to write R to disk. That is, we first preprocess by reading blocks of R into memory, constructing non-clustered blocks of R and writing them to disk. After that, we would have a non-clustered copy of R on disk which we could then use in conjunction with the algorithm in (a). This leads to the following formula:
Disk I/O cost = $T(R) + B(R) + T(S) + \frac{B(S)B(R)}{M-1}$.

(b) If larger relation R is unclustered, we could minimize the number of disk I/O's for large relations by first using one memory block to read R and fill in the remaining M-1 blocks of memory with R tuples. Then read the all tuples of S into the memory one block at the time and join to the M-1 blocks of R. In other words, R would be an outer table and read only once, and S would be an inner table read multiple times. The cost of this algorithm would be $T(R) + \frac{B(R)B(S)}{M-1}$ which is much better than the cost of making S an outer relation $B(S) + \frac{B(S)T(R)}{M-1}$.

(c) Similarly to (a), when both relations are large, it is more beneficial to pick the unclustered relation as an outer relation, to minimize the number of disk I/O's. When the smaller relation S is unclustered, and is used as an inner relation of the join, we would incur an approximate cost of $T(S)\frac{B(R)}{M}$, and if S is used as an outer relation of the join, the cost would roughly be $B(R)\frac{B(S)}{M}$. Thus, it is best to pick S as an outer relation of the join.

Exercise 15.3.5

```
Open()
{
  R.Open();
  r ← R.GetNext();
  if (r = NotFound)
  {
    s ← NotFound;
  }
  else
  {
    S.Open();
    s ← S.GetNext();
  }
}

GetNext()
{
  if (s = NotFound)          // handle the cae when one
  {                          // of the relations is empty
    return NotFound;
  }

  repeat while(r and s do not join)
  {
    if (r ≠ NotFound)        // R was not exhausted
    {
      r ← R.GetNext();
    }
    if (r = NotFound)        // R exhausted, reset R
    {                               // for next tuple in M
      R.Close();
      s ← S.GetNext();
      if (s = NotFound)
      {
        S.Close();
        return NotFound;
      }
      R.Open();
      r ← R.GetNext();
    }
  }
  xTuple ← join of r and s;
  r ← R.GetNext();
  return xTuple;
}

Close()
{
  R.Close();
}
```

Section 15.4

Exercise 15.4.1

(a)
```
Open()
{
  R.Open();
  repeat forever
  {
    if (there is memory left)
    {
      xTuple ← R.GetNext();
      if (xTuple = NotFound)
      {
        R.Close();
        save current sorted sublist $R_i$ to disk;
        do (For all $n$ created sorted sublists)
        {
            $R_i$.Open();
            $r_i \leftarrow R_i$.GetNext()
            Maintain searched structure for $r_i$
            (based on the grouping attributes of L);
          }
          **return**;
        }
        insert xTuple into searched structure
        (based on the grouping attributes of L);
      }
      **else**
      {
        save current sorted sublist $R_i$ to disk;
        start new sublist $R_{i+1}$;
      }
    }
  }
```

```
  **GetNext**()
  {
    xTuple ← get smallest tuple $r_i$ from the searched structure;
    Remove tuples from sorted sublists that have the same sort key
    as xTuple;
    **return** xTuple;
  }

  **Close**()
  {
    Close all sorted sublists.
  }
```

(b)
```
  **Open**()
  {
    R.Open();
    **repeat forever**
    {
      **if** (there is memory left)
      {
        xTuple ← R.GetNext();
        **if** (xTuple = NotFound)
```

```
          {
            R.Close();
            save current sorted sublist $R_i$ to disk
            do (For all $n$ created sorted sublists)
            {
              $R_i$.Open();
              $r_i \leftarrow R_i$.GetNext()
              Maintain searched structure for $r_i$
            }
            return;
          }
          insert xTuple into searched structure
        }
        else
        {
          save current sorted sublist $R_i$ to disk
          start new sublist $R_{i+1}$
        }
      }
    }

GetNext()
{
  xTuple ← get smallest tuple $r_i$ from the searched structure;
  if (xTuple = NotFound)
  {
    return NotFound;
  }
  // we have a new group
  Initialize for all aggregates of L;
  repeat
  {
    accumulate for aggregates of L;
  } until (there are no more tuples with same sort keys as xTuple)
  Finalize for all aggregates of L;
  return tuple consisting of grouping attributes of L and
         the associated values of the aggregations;
}

Close()
```

```
{
  Close all sorted sublists.
}
```

(c)
```
Open()
{
  R.Open();
  Create the sorted sublists for R (logic similar to (a));
  R.Close();
  S.Open();
  Create the sorted sublists for S (logic similar to (a));
  S.Close();
  Initialize search structure to hold first tuples from the
  sorted sublists;
  return;
}

GetNext()
{
  xTuple ← get the smallest tuple from the search structure;
  repeat while(xTuple does not appear in both R and S)
  {
    Remove xTuple from the sublist (use GetNext() for that list);
    xTuple ← get the smallest tuple from the search structure;
    if (xTuple = NotFound)
    {
      return NotFound;
    }
  }
  Remove xTuple from the sublists (use GetNext() for those lists);
  return xTuple;
}

Close()
{
  Close all sorted sublists.
}
```

(d)
```
Open()
{
    R.Open();
    Create the sorted sublists for R (logic similar to (a));
    R.Close();
    S.Open();
    Create the sorted sublists for S (logic similar to (a));
    S.Close();
    Initialize search structure to hold first tuples from the
    sorted sublists;
    return;
}

GetNext()
{
    xTuple ← get the smallest tuple from the search structure;
    Count the number of times x the xTuple appears in R
    (using GetNext(), which will also replenish the memory list);
    Count the number of times y the xTuple appears in S
    (using GetNext(), which will also replenish the memory list);
    xNum ← x - y;
    if (xNum > 0 )
    {
        output xTuple xNum times one at the time;
    }
    else
    {
        repeat from the start (get next smallest tuple);
    }
}

Close()
{
    Close all sorted sublists.
}
```

(e)
```
Open()
{
    // note, that a join key is used as the sort key
    // for the sorting/comparing operations below
    R.Open();
    Create the sorted sublists for R (logic similar to (a));
```

```
  R.Close();
  S.Open();
  Create the sorted sublists for S (logic similar to (a));
  S.Close();
  Initialize search structure to hold first tuples from the
  sorted sublists;
  return;
}

GetNext()
{
  xTuple ← get the smallest tuple from the search structure;
  repeat while(xTuple does not appear in both R and S)
  {
    Remove xTuple from the sublist (use GetNext() for that list);
    xTuple ← get the smallest tuple from the search structure;
    if (xTuple = NotFound)
    {
      return NotFound;
    }
  }
  output all the tuples formed by joining tuples from R and S
  having the same join key as xTuple;
}

Close()
{
  Close all sorted sublists.
}
```

Exercise 15.4.2

(a) 3*(10000+10000) = 60000

(b) 5*(10000+10000) = 100000

(c) 3*(10000+10000) = 60000

Exercise 15.4.3

We could use the extra buffers to read more than one tuple from a given sublist. Depending on the operation we may want to either evenly split the extra buffers among the sublists or use all of the extra buffers for one (or several, if can fit more than one sublist) sublist.

Exercise 15.4.4

(a) Nested-loop join needs to be used to join for each of the two Y-values. For each such join, 500 blocks of R and 250 blocks of S need to be joined. For each join, we read S into 100 blocks and then use the remaining block to read R. The number of I/O's is thus, 250 + 3*500 = 1750. The total I/O cost is then, 6000 (for sorting) + 2*1750 = 9500.

(b) Nested-loop join needs to be used to join for each of the five Y-values. For each such join, 200 blocks of R and 100 blocks of S need to be joined, so we can read S into 100 blocks and then use one block to read R. The total I/O cost is then, 6000 (for sorting) + 5*(100+200) = 7500.

(c) Nested-loop join needs to be used to join for each of the ten Y-values. For each such join, 100 blocks of R and 50 blocks of S need to be joined. Therefore, we can load S into 50 blocks and use remaining blocks to read R. The total I/O cost is then, 6000 (for sorting) + 10*(50+100) = 7500.

Exercise 15.4.5

As in 15.4.4, we need to perform nested-loop joins, however only 86 buffers are available to store the tuples.

(a) Nested-loop join needs to be used to join for each of the two Y-values. For each such join, 500 blocks of R and 250 blocks of S need to be joined. We read S into 85 blocks and use the remaining block to read R. The total I/O cost is then, 3000 (for sorting) + 2*(250+3*500) = 6500.

(b) Nested-loop join needs to be used to join for each of the five Y-values. For each such join, 200 blocks of R and 100 blocks of S need to be joined, so we can read S into 85 blocks and then use one block to read R. The total I/O cost is then, 3000 (for sorting) + 5*(100+2*200) = 5500.

(c) Nested-loop join needs to be used to join for each of the ten Y-values. For each such join, 100 blocks of R and 50 blocks of S need to be joined. Therefore, we can load S into 50 blocks and use remaining blocks to read R. The total I/O cost is then, 3000 (for sorting) + 10*(50+100) = 4500.

Exercise 15.4.6

(a) $\sqrt{10000} = 100$

(b) $\sqrt{10000} = 100$

(c) $\sqrt{10000 + 10000} = 142$

Exercise 15.4.7

(a) Algorithm for the semijoin $R(X, Y) \ltimes S(Y, Z)$

```
Create sorted sublists of size M for R and S,
using Y as the sort key.
Bring the first block of each sublist into a buffer
(assuming there are no more than M sublists).
Repeat until all tuples of S have been examined
  Find the next tuple with the least Y-value $y$ among
  the tuples of the sublists for S.

  For each sublist of R
    Proceed through the R tuples stopping when Y-value of
    the R tuple it at least $y$ (loading new blocks from
    the corresponding sublist if necessary).
    If the Y-value of the R tuple is equal to $y$,
      output the R tuple
```

(b) Algorithm for the antisemijoin $R(X, Y) \overline{\ltimes} S(Y, Z)$

Create sorted sublists of size M for R and S, using Y as the sort key. Bring the first block of each sublist into a buffer (assuming there are no more than

M sublists). Repeat until all tuples of R have been examined Find the next tuple with the least Y-value y among the tuples of the sublists for R.

For each sublist of S Proceed through the S tuples stopping when Y-value of the S tuple it at least y (loading new blocks from the corresponding sublist if necessary). If the Y-value of the S tuple is equal to y, no need to look at other sublists of S, iterate main loop to go get next tuple of R.

//note, we come here if we iterated over all sublists of S //and did not find a match Output the R tuple

(c) Algorithm for the left outer join R $\overset{\circ}{\bowtie}_L S$ is similar to the algorithm of section 15.4.8, except in the case when no tuples have been identified that have the same Y-value y, and the y was for the tuple in R, we need to output the tuple of R joined with the null-padded side of S.

(d) Algorithm for the full outer join R $\overset{\circ}{\bowtie} S$ is similar to the algorithm of section 15.4.8, except in the case when no tuples have been identified that have the same Y-value y, we need to output the tuple anyway, null-padding the other side. In other words, if the tuple was from R, we null-pad the S side, and if the tuple was form S, we null-pad the R side.

Exercise 15.4.8

In the worst case, we could have each tuple being so big that it does not fit into M blocks (or that no two tuples fit into M blocks) and so the memory requirement is that tuple length (or the relevant set of the attributes) be lass than M. Assuming that it is, the next requirement would be based on the fact that we can only have up to M-1 sublists and so we must have $T(R) \leqslant M - 1$.

Exercise 15.4.9

For operations involving one relation (e.g. grouping) we can avoid writing and reading the last sublist. This means that the total number of saved I/O's would be 2B(L) where B(L) is the number of blocks per sublist.

Section 15.5

Exercise 15.5.1

(a) Choose the number of buckets k such that the entire bucket plus one block of each $k - 1$ buckets will fit into M. Then we keep all the distinct tuples of the first bucket in main memory and avoid having to store and read blocks for one bucket out of k. Since the bucket size would be approximately M, we can save about 2M disk I/O's.

(b) Similarly to (a), we keep one entire bucket in main memory.

(c) Similar to the hybrid-hash-join, we choose the smallest relation S and pick number of buckets k such that the entire bucket plus one block of each $k - 1$ buckets will fit into M. If a tuple of R hashes into this in-memory bucket, we perform the union operation right away. If a tuple of R does not hash into the in-memory bucket, we send it to the main memory block associated with that bucket.

(d) Similarly to (c), we keep one entire bucket of S in main memory.

Exercise 15.5.2

We pick k=11 so that the expected size of the in-memory bucket is 909 blocks, which hopefully leaves enough memory for the 10 blocks for the remaining buckets. The number of disk I/O's we use for S for the first pass is 10000 to read all of S, and 10000-909 = 9091 to write ten buckets to disk. When we process R on the first pass, we need to read 10000 blocks and write ten buckets (9091 I/O's) to disk.
On the second pass, we read all buckets written to disk, or 9091+9091=18182 I/O's. The total number of I/O's would be 10000+9091+10000+9091+18182 = 56364 (this is close to 56000, which we can get if approximate formula from 15.5.6 is used).

Exercise 15.5.3

(a) **Open**()
Perform R.Open(). Read R using R.GetNext() and hash its tuples into the

buckets. Then, we can treat each bucket $R_1, ..., R_n$ as its own relation. Perform R_1.Open(), and set $r = R_1$. Perform R.Close();

GetNext()

Preform duplicate elimination on current bucket r as if it were its own relation. We can use the iterators developed for one-pass algorithm for this (e.g. 15.2.1 (b)). Once we processed all tuples from the current bucket R_i, close the bucket and switch to the next bucket R_{i+1}. If no more buckets, return NotFound.

Close()

Deallocate search structure if needed.

(b) **Open**()

Perform R.Open(). Read R using R.GetNext() and hash its tuples into the buckets. Then, we can treat each bucket $R_1, ..., R_n$ as its own relation. Perform R_1.Open(), and set $r = R_1$. Perform R.Close();

GetNext()

Preform one-pass grouping algorithm on current bucket r as if it were its own relation. We can use the iterators developed in 15.2.1 (c) for this purpose. Once we processed all tuples from the current bucket R_i, close the bucket and switch to the next bucket R_{i+1}. If no more buckets, return NotFound.

Close()

Deallocate search structure if needed.

(c) **Open**()

Perform R.Open(). Read R using R.GetNext() and hash its tuples into the buckets. Then, we can treat each bucket $R_1, ..., R_n$ as its own relation. Perform S.Open(). Read S using S.GetNext() and hash its tuples into the buckets. Then, we can treat each bucket $S_1, ..., S_n$ as its own relation. Perform S_1.Open(), and set $s = S_1$. Perform R.Close();

GetNext()

Preform one-pass intersection algorithm on current buckets r and s as if they were relations R and S. We can use the iterators developed in 15.2.1 (g) for this purpose. Once we processed all tuples from the current buckets R_i and S_i, close the buckets and switch to the next buckets R_{i+1} and S_{i+1}. If no more buckets, return NotFound.

Close()

nothing to do

(d) Similar to (c)

(e) Similar to (c)

Exercise 15.5.4

No modifications are needed, since each group needs only one tuple in memory.

Exercise 15.5.5

(a) Assuming tuples distribute evenly among buckets, we can pick as little as 5 buckets. This way, S will contain 100 blocks per bucket, and R will contain 200 blocks per bucket. We would need to read S with random I/O initially (since we do not know how the tuples are distributed on disk initially) and hash the tuples into the five buckets. Once 100 blocks of memory is taken (assuming there will be 20 blocks per bucket), we write all 100 blocks contiguously to disk. We do the same with R. Then we perform a one-pass join of all pairs of corresponding buckets, by reading corresponding S bucket completely into the memory (into 100 blocks) and using the remaining block to read R. Since, on average 20 blocks for each bucket will be in consecutive storage, we will save a random I/O on every 20 blocks.
The total I/O time would be, 100.5*500 (to read S) + 100.5*1000 (to read R) + (100+100/2)*5 (to write S) + (100+100/2)*10 (to write R) + 5*(100+20/2) (reads per S bucket) * 5 (buckets) + 50*(100+20/2) (reads per R bucket) * 5 (buckets) = 183250 ms.

(b) The difference with (a) is that we now have six buckets, and we only have one block for each of the five buckets that will be stored to disk. So the total I/O time would be: 100.5*500 (to read S) + 100.5*1000 (to read R) + 100.5*417 (to write S) + 100.5*833 (to write R) + 100.5*417 (reads S) + 100.5*833 (read R) = 402000 ms.
We can improve on this if we store each bucket in consecutive storage (assuming this is possible to control), then we will save on reads for the second pass. The total time would be: 100.5*500 (to read S) + 100.5*1000 (to read R) + 100.5*417 (to write S) + 100.5*833 (to write R) + (100 + 83/2) (reads per S bucket) * 5 (buckets) + (100 + 166/2) (reads per R bucket) * 5 (buckets) = 277998 ms.

(c) If we store the sorted sublists in consecutive storage, we can save I/O's by reading several blocks of each sublist at the time. Since there would be 5 sublists for S and 10 for R, for the merge phase we could read about 20 blocks for each S sublist and about 10 blocks for each R sublist. The I/O time for the sort phase then would be 100.5*500 (to read S) + 100.5*1000 (to read R) + (100 + 100/2)*5 (to write S) + (100+100/2)*10 (to write R) + (100+20/2)*25 (to read S for merge) + (100+10/2)*100 (to read R for merge) + (100 + 100/2)*5 (to write final S sublist) + (100 + 100/2)*10 (to write final R sublist) = 168500 ms.
For the join phase, we will need to read R and S once, however we may not be able to take advantage of the fact that both sublists are in consecutive storage on disk. This is because we usually would only use one block for each relation (reserving the memory for those cases where multiple tuples join). Therefore, in the worst case we will alternate the reads between R and S (i.e. all reads are random) and in the best extreme case we will read all of the tuples of one relation and then all of the tuples of another. Here, we assume the worst case and so the I/O time would be 100.5*500 + 100.5*1000 = 150750 ms.
The total time is then 168500 + 150750 = 319250 ms.

Section 15.6

Exercise 15.6.1

(a) Read and output all tuples of R. Then for each tuple *t* in S, use index for R.*a* to retrieve all tuples of R that match on *t.a*. If *t* is not among those tuples, output *t*. This method is efficient when S is small and R is large.

(b) Read S and for each tuple *t* of S, use index for R.*a* to retrieve all tuples of R that match *t.a*. If *t* is among those tuples, output *t*. This method is efficient when S is small and R is large.

(c) We could use index for R.*a* to process R in batches - each batch for a given index key. If there is only one tuple for a given index key, output that tuple. Otherwise, perform duplicate elimination (via hash or sort, etc.) for the set of tuples. This method is efficient when index on R.*a* is unique or when V(R.*a*) is small.

Exercise 15.6.2

(a) $\left\lceil \frac{10000}{k} \right\rceil$

(b) $\left\lceil \frac{500000}{k} \right\rceil$

(c) 10000

Exercise 15.6.3

Assuming $\frac{k}{10}$ keys will be in the range:

(a) $\frac{10000}{k} * \frac{k}{10} = 1000$

(b) $\frac{500000}{k} * \frac{k}{10} = 50000$

(c) 10000

Exercise 15.6.4

(a) If we use index, we need about $\frac{500000}{k}$ I/O's and if we do not use index we need 10000. Thus, for $k > 50$ we would want to use the index.

(b) If we use index, we need 50000 I/O's and if we do not use index we need 10000. Thus, we would not want to use the index.

Exercise 15.6.5

Index join:
Since the T(R) is very small we can ignore the cost of accessing it and so the dominant cost is from accessing S. The cost of accessing S is either $T(R)(max(1, \frac{B(S)}{V(S,Y)}))$ (if index is clustering) or $T(R)\frac{T(S)}{V(S,Y)}$ (if index is non-clustering). Note that since T(R) is very small, most of S will never have to be accessed.
Sort-based join:

The cost would be $3B(R) + 3B(S)$. Note that even though T(R) is very small and so 3B(R) is small as well, we still need to examine all of S.
Therefore, the cost for index-based join is much better when S is much larger than R, and V(S,Y) is large.

Exercise 15.6.6

The method outlined in the example, performs a simple sort-join with just the sorting phase skipped. Thus, it uses one block to read S and one block to read R, and it will work as long as the number of tuples of R and S that have the same Y-value fit into M blocks.

Section 15.7

Exercise 15.7.1

(a) One of the relations must fit in memory, so $min(B(R), B(S)) \geqslant \frac{M}{2}$.

(b) In the worst case we only have $\frac{M}{2}$ blocks and so the requirement is $min(B(R), B(S)) \leqslant \frac{M^2}{4}$.

(c) In the worst case we only have $\frac{M}{2}$ blocks and so the requirement is $max(B(R), B(S)) \leqslant \frac{M^2}{4}$.

Exercise 15.7.2

For both (a) and (b), the disk I/O will not improve (unless of course the number of buffers is enough to fit all the blocks of R). The reason is that we are reading R sequentially from first tuple to last every time, and so by the time we need to read first tuple again, its buffer will not be in the buffer pool.

Exercise 15.7.3

Using the k blocks for buffering R, the I/O cost would be:

1: $B(S) + B(R) + \left(\frac{B(S)}{M-1} - 1\right)(B(R) - k)$.

Using the k blocks for buffering S, the I/O cost would be:

2: $B(S) + \frac{B(S)}{M-1+k}B(R)$.

We can omit B(S) term appearing in both equations. Then, simplifying (1), we get: $B(S)\frac{B(R)-k}{M-1} + k$.

Simplifying (2), we get: $\frac{B(S)B(R)}{M-1+k}$.

Dividing both sides by positive $B(S)$, we get:

$\frac{B(R)-k}{M-1} + \frac{k}{B(S)}$ vs. $\frac{B(R)}{M-1+k}$.

Further manipulations yield:

$B(R) + 1 + \frac{(M-1+k)(M-1)}{B(S)}$ vs. $M + k$.

Which is essentially $B(R)$ vs. $M + k$, and we conclude that (1) is greater than (2), and that using the extra k buffers for S yields less I/O's.

Section 15.8

Exercise 15.8.1

(a) R is partitioned into 100 groups $R_{a1}, R_{a2}, ..., R_{a100}$, since each group has 200 blocks and still does not fit into M, each group is partitioned into 100 groups again $R_{b1}, R_{b2}, ..., R_{b100}$, 2 blocks each. Now each R_{bi} group is read into memory, sorted and stored back, then all 100 groups are merged, producing the sorted sublists for group R_{ai}. Next each group R_{ai} is merged producing the final sublist for R.
Similarly, S is partitioned into 100 groups, 500 blocks each, then each group partitioned into 100 groups, 50 blocks each. Next, the 50-block groups are read, sorted and saved. Then they are merged as well as their resulting sublist are merged producing the final sublist for S.
R and S are then joined using their sublists.

(b) R and S are hashed into 100 buckets each. Assuming the tuples are evenly distributed, there would be about 200 blocks in each R bucket and 500 blocks in each S bucket. Thus, each R bucket would be hashed into 100 buckets again along with hashing the corresponding S bucket. Next the corresponding second level buckets would be joined producing the output.

Exercise 15.8.1

(a) Cannot be used due to the storage requirement ($\frac{mB(S)}{k} + k - m \leqslant M$).

(b) Yes, we can store the sorted sublists in consecutive storage on disk, saving on the seek time.

(c) Yes, similarly to (b), we can store the buckets in consecutive storage on disk, saving on the seek time.