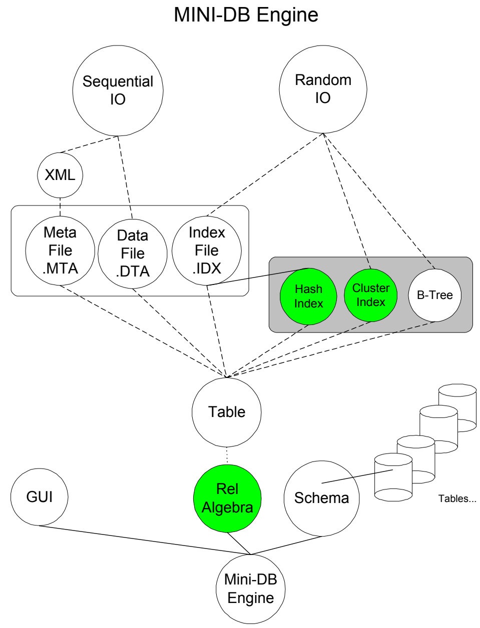
Assignment-4	MINI DB-ENGINE (Phase III) Building the Relational Algebra Class (Continued)	

Complete the development of the MINI_Relational_Algebra class.



Step 1: (Review, Reflect, Brainstorm and Reorganize)

The outcome of this process should be a list of recommendations which will guide you to redesign and re-code some aspects of assignment 3. Specifically, you may identify a better way of organizing your classes, such that they are more reusable, flexible, etc. Also, this process will provide you with the opportunity to clean up and document your code.

Step 2: (Completion of the Relational Algebra Operators)

Extend your MINI_Relational_Algebra class to include the following operators.

(join, union, intersection, difference)

Your final implementation should include the following methods:

```
Class Mini_Rel_Algebra {
       bool create(relation_name);
       bool insert(relation_name, attribute_1, value_1, ....attribute_n, value_n);
       bool delete(relation_name, attribute_name, attribute_value);
       bool modify(relation_name, attribute_name, attribute_value);

       result_rel select(relation_name, attribute_name, condition, attribute_value);
       result_rel project(relation_name, attribute_list);
       result_rel cartesian_product(relation_1, relation_2);

       result_rel join(relation_1, relation_2,
                       condition_list);             // condition in the form
                                                    // "attrib_name^condition^attrib_value~" or
                                                    // "attrib_name^condition^attrib_name~"

       result_rel union(relation_1, relation_2);       // make sure the two relations are union
                                                    compatible
       result_rel intersect(relation_1, relation_2);  // make sure the two relations are union
                                                    compatible
       result_rel difference(relation_1, relation_2); // make sure the two relations are union
                                                    compatible
}
```

Functional Specification:

- *result_rel join(relation_1, Relation_2, condition_list);*

  Join the two relations based on the join condition(s) provided in "condition_list". The condition_list may appear as:

  "attrib_name^condition^attrib_value~" or
  "attrib_name^condition^attrib_name~"

  Once the condition list is parsed into the corresponding attribute_name, condition, and attribute_value. The Join module should call the cartesian_product() followed by a call to select() operation.

  For now the **condition** is only restricted to equality.

- *result_rel   union(relation_1, Relation_2);*

  First make sure the two relations are union compatible by checking the meta-data for relation_1 and relation_2. Once union compatibility is established, create a temporary relation for maintaining the union. Note that based on our design of a table, each new relation must have a key attribute. Therefore, we must add a new attribute which will serve as the primary key for the result relation. The new attribute can be called RRN (Relative Record Number) and will be a unique number for each record in the union.   (essentially an autonumber field)

  Don't forget to eliminate duplicate records.

- *result_rel   intersect(relation_1, Relation_2);*

  Similar issues discussed in the implementation of union() must be considered.

- *result_rel   difference(relation_1, Relation_2);*

  Similar issues discussed in the implementation of union() must be considered.

Step 3: Extending your Index File Class (Hashing Algorithm)

Overload one or more methods in your index-file class to allow hashing of keys.

Current index addresses are calculated by multiplying the key with the index record size:

    Address = key * Index_Record_Size

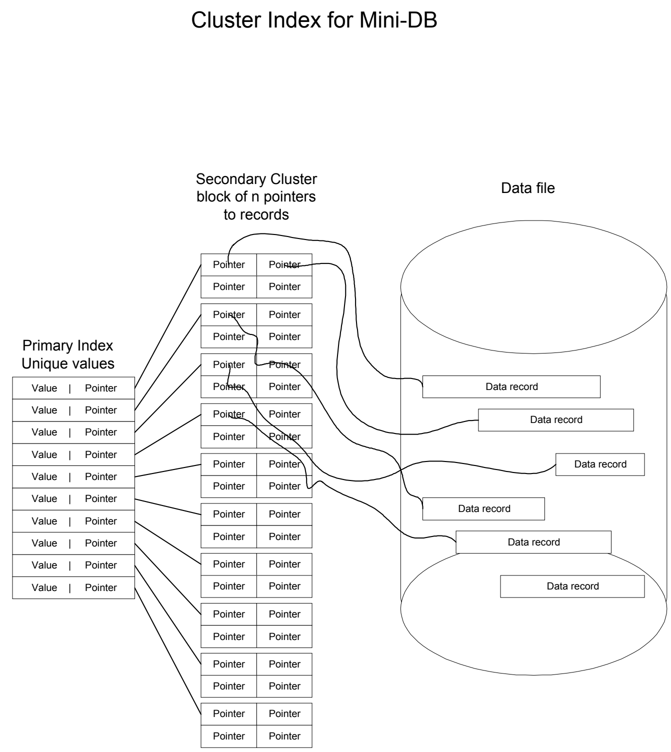
Overload this method so that the client can use a hash algorithm instead:

    Address = hash(key) * Index_Record_Size   // the key may be numeric or string

Of course your new hashing algorithm should also handle collisions.   (Will be discussed in class)

This new index structure and its access algorithm should still have a O(1) performance, but provide much better and more efficient space utilization for the index file. In addition, the new algorithm will accommodate non-numeric key values.

Extra Credit: Extending your Index File Class (Cluster Index) (20 points)



Extend your Index File Class or simply create a new class to allow cluster indexes.

In order to provide a cluster index on a non-key attribute, you need to have a primary and a secondary index structure:

1) create a primary index on all the **unique values** in that attribute. This structure will be very similar to your current .idx file. With the primary difference being that it's pointers point to another index structure, and not to the records in the data file. (Note that the uniqueness property specified above will make this index similar to an index on a primary key!)

2) create a secondary index file (cluster of pointers) which will point to records in the data file. For the purpose of our assignment, each cluster should accommodate a block of up to 4 pointers. (This can of course be easily expanded to 8, 16, 32, etc.) In addition, you may designate the last pointer as chaining pointer. Which means that, if we have a block with 4 pointers, 3 of them refer to data records, and the 4^{th} will refer to a new cluster block.

What to hand in:

- **Cover page** with proper title, your name, course # and name, assignment #, date, etc....
- **Source code** (documented)