BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-90-M8

Collections, Tuples and Iterators in Object-Oriented Database Systems

by
Shin Y. Lee

Collections ,Tuples and Iterators in Object-Oriented Database systems

Shin Y. Lee
Computer Science
Brown University

May 7, 1990

Submitted in partial fulfillment of the requirements
for the Degree of Master of Science in
the Department of Computer Science at Brown University

Stanley B. Zdonik

Professor Stanley B. Zdonik
Adivisor

Collections ,Tuples and Iterators in Object-Oriented Database systems

Shin Y. Lee
Computer Science
Brown University

May 11, 1990

Abstract

This paper describes the behavioral specification and implementation of **Collections, Tuples** and **Iterators** in the **Encore** object-oriented Database system. The concept of Collection as a set of instances provides the basis on which reference of objects in persistent storage may be performed. Type Collection supports operations for *Encore* queries and maintain Btree as an index. We created a Btree as an abstract type in *Encore.* A query languge retrieves and manipulate objects in collections. Since the properties of existing types may not reflect all relationships in a database, some query operations create new relationships. The **Tuple** type is also created to store new relations produced by query results. We defined *Collection* and *Tuple* types as parameterized types. As in most programming languages, **Iteration** methods are useful to allow users to iterate over an arbitrary collection of objects. We defined an iteration type, Iterator as an abstract type, to be applied to any collections.

1. Introduction

An Object-Oriented Database system should support some concepts of Object-oriented languages and provide persistent storage for the objects and schema. Without the notion of the Collection, or a set of objects, the value-based reference of objects in persistent storage might be difficult. As an abstract set type in an Object-oriented database, the Collection type can also be used as the value-class of properties of user-defined abstract types, providing multi-valued properties.

The **Encore** is an Object-oriented database system which is being developed in Computer Science Department at Brown University. *Encore* supports a type system to model objects with a set of properties and operations, and also provides the notion of Collections to store all objects of type. Collection defines query operation of Encore and also provide key-based indexes, which are used for efficient access to objects. Collection provides operations for creation and deletion of indexes, and retrieval of instance objects using indexes. A Key index are implemented as a B-tree, which is created as an abstract type in ENCORE, including insert, delete, search operations. Basically the B-tree type and its operations are designed using the top-down B-tree Algorithm of [Sedgewick]. From the users' point of view, B-tree is a part of the Collection Type and is only accessible through the operations of Collection. From the system's view point, B-tree is an independent type having a structure of B-tree composed of nodes, and operations applicable to B-tree structure. The nodes of the B-tree are designed as instances of type Segment.

In addition to Collection Type, we have another parameterized type, $Tuple[< A_1, T_1 >, \ldots, < A_n, T_n >]$. Tuple Type is created as a reflection of the structure of tuples in relational database. In relational database system a query can make a new relationship as a result by producing a new relational table. Similarly in Object-oriented database system, a query may introduce new relationship, which is not already defined in database. In such a case, **Tuple** type acts as a template for storing new relationships. When a query creates a new relationship, a tuple type is automatically generated from the type **Tuple**.

In addition to value-based retrieval of objects, searching through all objects in a collection in order is useful. An obvious use, for example, is when users want to iterate over objects performing some action on each object. The **Iterator** Type was designed as an abstract type to be applicable to any type of object. An iterator returns objects in some order, as defined by the collection. For example if objects are sorted depending on property value, iterator would iterate over objects in that sorted order.

2. Data Model

Collections, tuples and iterators are supported by, and also a part of the ENCORE Object-Oriented Data model. *Encore* is a typed system which supports object identity, type inheritance and abstract data types encapsulating properties and operations. An object has an interface and an implementation. The interface is modeled or prototyped by the type which has Property definitions and Operation definitions. In its implementation, every object has values of properties which represent the abstract state of the object, while objects might accomplish some actions using operations.

Unlike the key valued identification in the relational data model, Object identifiers, or unique identifiers(UID), serve for unique identification of objects. The Type system also defines some equality to be used by the query model. Objects are *identical* if they have same identifiers(identical to itself), and objects are *i- equal* if they have same values to some depth. The Unique identifier is similar to pointers of programming languages and key values in the relational data model.

The **inheritance** between types allows reuse of operations and structure of properties. The relationship through Inheritance leads to hierarchical relationships of objects by subtyping. The subtype P of type T inherits all the properties and operations of type T and has them as part of P itself. In order to allow substitutability of instances of sybtype in the *Encore* Type model, subtyping should not restrict supertype behavior, but be an extensions of its behavior. For example an instance of type T should be allowed to be substituted in place where the instances of its supertype S is expected.

In addition to the abstract type system, Encore includes parameterized types, namely **Collection and Tuple** types. A Collection type is parameterized by an abstract type T, called member type. Collections are considered to include the objects of a type, keeping itself homogeneous. A Collection type is automatically created when a type is defined; the type has property *instances* which is a collection of the corresponding collection type. The collection all instances of a type, which is the value of *instances* property of the type, has all objects created by the type.

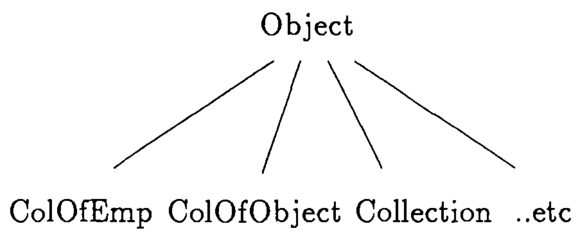
The tuple type is created to store new relations produced by query results. The parameterized type Tuple[<A1,T1>,...,<An,Tn>], is parameterized by Ai's, names of attributes, and Ti's, types of attributes. As for Collections, a template type *Tuple* is defined as prototype of other tuple types. When a tuple type is created, it becomes a direct subtype of *Object* and gets copies of the properties and operations of the type *Tuple* .

The Iterator type is implemented with a pointer, which points to the current object, and exploring operations. In this report, only one iteration type is introduced; it is designed to iterate over *Collection* types. An extension would be to create subtype iterators which will be able to iterate over other aggregate types, such as B-tree, Bytes, etc.

3. Collection

In this section, we shall discuss in more detail the Collection Type in ENCORE. Collections are set objects which collect together instances of a particular object type. Basically collections are a generalization of the set type of programming languages and the relational tables of the relational Database system. Collections support set operations of Programming languages and operations for Object-oriented queries. Particularly, Collection provides an interface for key indexes which can serve for efficient query processing.

Collection Types are generated automatically from Type Collection and are represented as Collection[T], or ColOfT, where T is some type. Collection[T] is an abstract parameterized type which is parameterized by the type of elements. When an abstract type(e.g. Person, Emp etc) is created by users, the corresponding collection type(e.g. ColOfPerson, ColOF Emp, etc) is created and it becomes a subtype of type *Object* Although a collection is considered to be homogeneous, the system defined collection,ColOfObject is to serve as a heterogeneous collection of all objects in a Database. The hierarchical structure of Collection types are shown below:



3.1. Collection Operations for queries

Collection maintains operations for *Encore* queries, which are performed over a collection. The Collections are considered to be homogeneous such that all elements of a collection have the same type. We define algebraic query operations in two categories in [G. Shaw 1989]:

1) Operations that retrieve data such as Select, Image, Project, Ojoin,Union, Intersection and Difference.

2) Operations that support data retrieval through manipulation of result structure and object identity : Flatten, Next, UnNest, DupEliminate, Coalesce.

$Select(s;p) = \{s | (s \text{ in } S) \wedge p(s)\}$
$\text{Image}(S,f{:}T) = \{f(s) \mid s \text{ in } S\}$
$\text{Project}(S, < (A_1, f_1), \ldots, (A_n, F_n) >) =$
$\quad \{< A_1 : f_1(s), \ldots, A_n : f_n(s) > \mid s \text{ in } S\}$
$\text{Ojoin}(S,R,A_1,A_2,p) = \text{Relational\_Join}(S',R',p')$
$\text{Flatten}(S) = \{r | \exists t \text{ t in } S \wedge r \text{ in } t\}$

Figure 1)

So far the operations of category **1)** and Flatten of **2)** were implemented in collection type as methods. Since we do not have any optimization schemes yet developed, each of the operations are developed in a quite straightforward manner.

The functional representation of each query operation is given in Figure 1). Their definitions and semantics are explained in [G. Shaw 1989] in more detail. The Select operation creates a collection of objects, which is a subset of the target collection, satisfying a selection predicate. Predicates are processed by preprocessor that generates an operation for each predicate[W. Wong 1989]. The select operation applies these generated operations to each object in the target collection and collects objects that satisfy the predicate operation into a collection. In all other query operations, Predicates are processed in the same way as in selection. For example, when a Selection query :

Select(Student, $s s@name == " Shin Lee")

is accepted by the preprocessor, following procedure is created as an operation of *Student*:

```
ENObject
Student_PredOp(c,args)
Object c;
Object args;
{
if(c.name == "Shin Lee")
then
        return TRUE;
else
         return FALSE
}
```

The Preprocessor passes the new operation to the *Select* operation of Collection. The Select operation works as follows:

```
Select(ColOfPerson, Student_PredOp, args)
{
for(every object of ColOf Person)
     {
     execute Student_PredOp
     if(Student_PredOp returns True)
        then the object is put into result collection
      }
}
```

The Project and Ojoin operations can create new relationship not already defined in the object types. These operations produce tuple types to store new relationships. Each of the functions of Project are converted to a method of a type by the query preprocessor, as of Select operation. The Project operation of collection applies the methods of predicates to every object of the collection and creates one tuple for each of the objects. The Ojoin operation is an explicit join applied to two objects from two collections in the database. As in selection the predicate is converted to a method which is executed for every possible pair of objects from two collections. Ojoin makes a tuple for every pair satisfying the predicate.

Union and Intersection are common set operations of Programming languages. Union/intersection operation plainly merges/intersects two collections creating a new collection. The memberType of two collections have to have a common anscestor.

3.2. Index System

Collection supports an operation *FIND* which retrieves objects from a collection depending on values of properties. The Find operation takes a pair of property name and property value, and uses an index of the property to find objects. If a proper index does not exist, it will go through the collection sequentially. When the query optimization method is developed, the *Find* operation will be used properly for Select, Project and Ojoin query operations; actually the *Find* operation is the interface between Collections and Indexes.

Our index system is an implementation of a top-down B-tree scheme [R. Sedgewick, 1988]. The advantage of this method is that it provides log(N) search time, N is linearly dependent on the size of a Btree in terms of number of elements. As is well-known to us, this scheme prevents bad worst case performance by balancing the levels of trees. This method also has an advantage over hash methods by permitting the collection listed or searched in sorted order. In Encore, a Btree stores identifiers(UIDs) of member objects, not physical pointers of objects or objects itself. Because this Btree does not belong to the file system, it is not appropriate to have physical pointers to objects; thus the Btree is not dependent on the physical changes of objects. Since Btree only has identifiers, there are some disadvantages in that it will need time to not only traverse the Btree but also to search the object itself using the identifier. We will propose some extensions to traditional Btree to help correct this problem.

The index system is composed hierarchically of three types, List, Btree and Segment as follows:

$$\begin{tikzcd}
List \arrow[d, dash] \\
(subtype) \arrow[d, dash] \\
B-Tree \arrow[d, dash] \\
(property\_of) \arrow[d, dash] \\
A\_tree\_structure\_of\_Segments
\end{tikzcd}$$

Currently the List type is an abstract type and behaves as a supertype of B-tree, supporting operations independent from the structure of the Btree. The List type should also be able to behave as a supertype for any other possible indexing scheme which might be developed as subtypes of List.

The Segment type was created for the nodes of the Btree, which has a structure of a tree of nodes. Each Segment corresponds to one node and has an array of records, which are pairs of key values and UID's of objects, in sorted order by key values. Since the purpose of Btree is to reduce disk access time during the search and manipulation of objects, the size of the *Segment* Objects are the same as of the page size of physical disk system. In this sense the *Segment* type could be said to be physical storage dependent. The Segment type adopts binary search method for the search inside the segment(node). B-tree type stores B-tree structure of Segments as a property called *members*, in this structure Segments are related by Links from parent Segments to children Segments. The structure of a typical B-tree object is shown below:

$$\begin{tikzcd}
\text{B-Tree} \arrow[d, no head] & \text{Segment} \arrow[d, no head] \\
\text{property of} \arrow[d, no head] & \text{property of} \arrow[d, no head] \\
\text{a tree structure of Segment} & \text{an array of records}
\end{tikzcd}$$

B-tree type has only the root of the tree structure, the other Segments are accessible through the parent-child Links between segments; these pointers are actually UID's of segments.

Search: To look for a object, the Btree appliess its operation to find the segment which this object might belongs to. Then the Btree sends a lookup-request message to the segment. This segment searches itself sequentially and returns the identifier for the object or the link to the next appropriate node(Segment). In addition to the Search operation, Segment also supports iterating operations to allow users to iterate from the first to last.

In Btree we search for a record following links from the root down to the terminal nodes. The Btree accepts return values from each segments and keep searching until a segment returns objects or it reaches a terminal segment.

Insert: To insert a new object, the insertion method starts to search for the proper node to insert into. We are using top-down insertion algorithm of B-tree. In the bottom-up Btree, if during the insertion the node is found to be full, the nodes will be split from the leaf up to parents nodes. Thus, in the worst case the split may occur all the way up to the root node. In top-down Btree the method splits nodes(Segments), during the search period when it sees a full Segment. This prevents traversing the segments twice, one from root down to terminal for search and the other from root back up to root for split. The split operation is also implemented as a method of the Btree.

Delete: To delete a record, the deletion method begins with search to find the segment S containing this record. After the deletion if the segment S becomes less than half full this method will search a segment T, which resides immediately to the left or right to the current segment and has the same parent. If the neighbor segment is exactly half full, two segments are put into a segment. If the neighbor segment is more than half full, we distribute the records of S and T evenly as possible keeping the sorted order. Then we modify the parent record of S and T to trigger the changes in S and T.

4. Tuple

We query over collections of objects having type ColOf[T], then queries return new objects having type ColOf[Q] [G. Shaw 1989]. Since the result of query is a homogeneous collection of objects we need to determine what this type will be. Often it will be a collection of existing type objects. However, since the properties of existing types may not reflect all relationships desired by a query, some query operations create new relationships between objects.

The concept of a Tuple type is introduced into object-oriented database systems to be used as types for some query results, such as Project, or Ojoin. The Project and Ojoin can create new relationship not explicitly defined in properties of the one of Joined object types.

Tuple is an abstract parameterized type, parameterized by each attribute name and type, $Tuple[< A_1, T_1 >, \ldots, < A_n, T_n >]$. The system defined type **Tuple** is used as a template for other parameterized tuple types. The type **Tuple** has already all properties and operations necessary as a tuple, so all other tuple types can be generated automatically by a generation function, TupleType, with some parameters.

TupleType inherits from ObjectType and defines the generation function of tuple types. Tuple type provides two different kinds of operations , one for generating new tuples and the other for maintaining tuple definitions. The initialization routine of TupleType actually takes the role of the generation function by copying properties and operations of template type **Tuple** into a new Tuple type.

TupleType is similar to the type Type in the sense that it has to store and maintain attribute lists which are similar to the list of properties in types. The scheme for accessing attribute lists is to hash by name of all attributes into a table. We use a double hashing scheme as is used by type Type to store properties definitions. The attribute list is composed three tables, one is the table of attribute names, another is the table attribute types and the other is the list of pointers , in offset, to each of the attribute names.

5. Iterator

Iterators are available in most programming languages to permit users to iterate over arbitrary types of data in a convenient and efficient way.[the book]. Obviously an iteration may go through a collection completely from the first to the last.

We specify the iterator with a pointer and four iterating operations. First, Next, Last,and Previous. The pointer has the current position of iteration and the exploring operations to move the pointer one step(element) back or forward, move to the first element and move it to the last one. To abstract the above iteration methods, we need to specify them in an object. Simply we put them in collection types; each collection has a pointer in it for iteration and iterating operations may be called like methods of a collection. Given the abstraction, users can iterate through a collection :

```
curelms = First(persons);
while(curelms! = Last(persons))
        {
        ; do work with curelms.
        curelms = Next(persons)
        }
end.
```

Although the above scheme performs iteration for collections, it is inconvenient and insufficient. First, we may not be able to do *Nested iteration* upon a collection, because collections have one pointer. Second of all, as is collection types, the index types(B-tree, List) and built-in primitive table types(Bytes, UIDBytes) support iteration methods. They may have different implementations for the representations of operations defined above. For example, the *Next* operation of B-tree moves pointer by following links between nodes, while that of bytes moves the pointer sequentially. Even though the same implementation of iteration operations can be used for two different kinds of aggregation types(e.g. Bytes and UIDBytes), they cannot share operations unless they are related by subtyping.

As an alternative, an *Iterator* type was created, that has the iteration methods and pointer. Iterators can be associated with a collection through a property relationship. Each time an iteration loop is made, an iterator object is created from a proper iterator type. For an example a iteration loop for persons objects is represented as follows:

```
myiterator = CreateInstance(IteratorType, persons)
curelms = First(myiterator);
```

```
while(curelms! = Last(myiterator))
        {
        curelms = Next(myiterator)
        ;1 do work with curelms.
        }
end.
```

6. Future works

6.1. Query optimization

The current implementation of query operations does not include query optimization scheme, since it is still under development. However, without query optimization strategy we still can adopt *Find* operation for processing simple queries, which are performed on one collection without joining several collections. After the query optimization is developed, *Find* can be used properly for Select and Ojoin.

6.2. More indexing mechanism

Currently, we have two types for indexing, List and B-tree. The List supports operations independent from the structure of the Btree as a super type of B-tree. The List type is able to behave as a supertype for any other possible indexing scheme which might be developed in the future. We might need other indexing scheme instead of or in addition to B-tree for the reason of efficiency and convenience.

6.3. More Iterator Types

Since the implementation of Iterator depends on the internal structure of Aggregate objects, we need one Iterator type for one aggregate type. Even though we created one Iterator type for Collection type, other Iterator types will be required to be made for aggregate types.

7. Acknowledgement

I would like to thank my advisor Stanley Zdonik for providing motivation and encouragement to work for this project. I wish many thanks to Page Elmore for helping me to solve a lot of hard problems and for giving constructive advice.

REFERENCES

[W.Kim , 1988] . Kim *A Model of Queries for Object-Oriented Databases.* Tech. Rep. ACA-ST-365-88, MCC, 1988

[M.J.Carey *et al.*, 1988] .J. Carey, et al. *A Data Model and Query Language for EXODUS.* SIGMOD proceedings, pp.413-423, ACM, June 1988.

[G.M. Shaw *et al.*, 1989] .M. Shaw and S.B. Zdonik. *A Query Algebra for Object-oriented Databases.* Brown University Technical Report CS-89-19, Mar. 1989

[T.Bloom *et al.*, 1985] . Bloom and S.B. Zdonik. *Issues in the Design of Object-oriented Database.* Programming Languages, Technical Rpoort No. CS-87-19, Brown Univ. 1987.

[R. Sedgewick , 1988] . Sedgewick *Algorithms, Second Edition.* Addison-Wesley Publishing Company, 1988

[J.D. Ullman , 1988] .D. Ullman *Principles of Database Systems Second Edition.* Computer Science Press, 1982

[B. Liskov *et al.*, 1986] . Liskov, J. Guttag *Abstraction and Specification in Programming Development.* Computer Science Press, 1986

[S.B. Zdonik *et al.*, 1986] .B. Zdonik and P. Wegner *Language and methodology for object-oriented Database Environment.* Nineteenth Annual International conference on systems sciences, 1986

[W.D.Wong , 1989] .D. Wong *A Query Processor For An Object Oriented Database.* Master's Thesis in Brown University, 1989

[B.P. Elmore *et al.*, 1990] .P. Elmore and S.B. Zdonik *(untitled)* . unpublished paper, 1990

ObjectType TupleType(Type) :

This is a subtype of Type type and generates new Tuple types and initializes these new types.

LOCAL PROPERTIES

INTERNAL PROPERTIES

AttrDefList attributeDefs
This property contains the Definitions of attributes for a tuple type.

Integer numAttrs
This property indicates how many attributes a tuple type has.

PUBLIC PROPERTIES

LOCAL OPERATIONS

INTERNAL OPERATIONS

Boolean ITupleType(name:String)
This method generates and initializes a tuple type with the name of it. It is called by CreateInstance on Type.

Boolean DTupleType()
This method undoes the changes made in ITupleType and is called by DeleteMe on Object.

Boolean AddAttribute(name:String,valueClass:Type)
This method adds an attribute called "name" to the type. valueClass is the Type of its attribute.

Boolean RemoveAttribute(attrName:String,valueClass:Type *)
This method removes the property called propName, returning the removed property. It protects against attempts to remove inherited properties.

PUBLIC OPERATIONS

ENBoolean GetAttribute(attrName:String,attrType:Type *)
This method gets the Type of the attribute by attribute name "attrName".

END

TupleType Tuple (Object) :

Tuple type behaves as a prototype of all other tuple types created by TupleType. TupleType created tuple types by copying all operation of properties of Tuple.

LOCAL PROPERTIES

INTERNAL PROPERTIES

UIDBytes attrValues : TS

This property has values of attributes , which are defined in attrDefList. The order of attribute values are dependent on the order of definitions in attrDefList.

Integer curPosition

This property is used to indicate the current attribute value in the attrValues Property.

Integer numAttrs

This property is used to indicate how many attributes the tuple has.

PUBLIC PROPERTIES

LOCAL OPERATIONS

INTERNAL OPERATIONS

Boolean ITuple(attrList:UIDBytes)

This method initializes the property attrValues and put values of each attributes in it.

Boolean DTuple()

This method undoes the initialization made in ITuple.

Object SetAttrValue(attrName:String,value:Object)

This method attempts to set value of an attribute, whose name is "attrName".

PUBLIC OPERATIONS

Object GetAttrValue(attrName:String)

This method find the value of the attrbute, whose name is "attrName", and return this value.

Boolean FirstAttr(attrValue:Object *, pos: Integer *)

This method finds the first attribute value in the list, and returns the attribute value and it's position.

Boolean NextAttr(attrValue:Object *, pos: Integer *)

This method finds the next attribute value in the list depending on the value of currentPos Property.

END

ObjectType AttrDefList (Object) :
This type is used for storing the attribute definition list of Tuple types.

LOCAL PROPERTIES

UIDBytes attrTypes : TS
This property contains the array of types of attributes in a tuple type.

Bytes members :TS
This property contains the offsets of names of attributes in attrNames ' property.

Integer numPosition : TS
This property indicates how many positions can be used in attrTypes.

Integer nextFreePos : TS
This property indicates which position in attrTypes is the next one available.

Bytes attrNames : TS
This property contains string names of attributes.

Integer nextNamesPos : TS
This property indicates which position in attrNames is the next one available.

Integer sizeOfNames : TS
This property indicates how many bytes can be used in attrNames.

Bytes hashTable : TS
This property contains a hash table of positions for types in members and names of attrNames.

Integer currentPos : TS
This property contains the identity of the current position marker – only used and set by First and Next.

INTERNAL PROPERTIES PUBLIC PROPERTIES

Integer numMembers : TS
This property contains the number of attributes.

LOCAL OPERATIONS

INTERNAL OPERATIONS

Boolean IAttrDefList(approxMaxSize:Integer),
The approxMaxSize indicates the guessed size of the list over time.

Boolean GrowMembers(newSize:Integer);
This method grows the internal lists to the new size.

Boolean Add(attrType:Type,name:String)
This method attempts to add an attribute with its name and type at the position given. It will add it at a new position if the one given is taken, and will not add the attribute if there is a name conflict.

Boolean Delete(attrName:String)
This method deletes the attribute with name "attrName".

PUBLIC OPERATIONS

Boolean GetAttrNames(name:String,pos:Integer *)
This method gets the name of an attribute by a particular position.

Boolean Find(type:Type * ,name:String,pos:Integer*)
This method finds an attribute value by attribute name and returns the Type, its position.

Boolean First(name:String,attrType:Type *)
This method finds the first object in the list, and returns the name, and the Type.

Boolean Next(name:String,attrType:Type *,pos:Integer *)
This method finds the next object (relative to the property currentPos) and returns the name ,the Type of the attribute and it's position in the list.

END

ObjectType Iterator(Object) :

This type provide a pointer into a collection. Collections have their own pointer into themselves that moves when you call First, Next etc. An Iterator is used to give you a "personal" handle on a Collection.

LOCAL PROPERTIES

INTERNAL PROPERTIES

Integer curPosition : TS
This property has the pointer to an object.

Object colObj : TS
This property has the collection which is iterated over by self iterator.

PUBLIC PROPERTIES

LOCAL OPERATIONS

Boolean IIterator(itCol:Collection)
This method initializes an iterator. It checks the type of itCol.

Boolean DIterator()
This method undoes the changes made in IIterator.

INTERNAL OPERATIONS

PUBLIC OPERATIONS

Object First()
This method finds the first element in the collection and return it.

Object Last()
This method finds the last element in the collection and return it.

Object Prev()
This method finds the previous object in the collection depending on the value of the currentPos property.

Object Next()
This method finds the next object in the collection depending on the value of the currentPos property.

END

ObjectType ColType(Type) :

This is a subtype of Type type and is a type for all collections. It generates new collection types and initializest them. Collection types are generated by getting copys of all the properties of operations of Type Collection, which roles as a prototype of all Collection types.

LOCAL PROPERTIES

INTERNAL PROPERTIES

Type Memtype : RO

This property contains the type of members which can be inserted to the instances of a Collection type. Actually only this property characterize each Collection types.

PUBLIC PROPERTIES

LOCAL OPERATIONS

INTERNAL OPERATIONS

Boolean IColType(memType:Type)

This method initializes Collection types. It makes the name of a Collection type and initialize the property and operation list.

Boolean DColType()

This method undoes the changes made in IColType and is called by DeleteMe on Object.

Object CreateInstance(seg:Integer,name:String)

PUBLIC OPERATIONS

END

ColType Collection(object:Object) :

This is the super type of every collection type which are created for every user defined type and some system built types. Collection is not an abstract type and roles as a prototype for generating set types.

LOCAL PROPERTIES

INTERNAL PROPERTIES

UIDBytes reps

This property stores pointers to the indexes a Collection has and some informations about indexes, such as the types of indexes, the type of keyvalus being used, and the UID's of indexes.

UIDBytes uidTable

It is implemented by UIDBytes table. Each entry of reps has the UID of objects which is inserted into the Collection

Type memType

This property contains the Type of members which can be inserted into the Colleciton.

Collection super

This property is a collection of super collections from which the self is derived.

PUBLIC PROPERTIES

Integer numMems

This property indicates how many elements a collection has.

LOCAL OPERATIONS

INTERNAL OPERATIONS

Boolean ICollection(extsize:Integer,growsize:Integer)

This operation initialize all the properties of a new Collection. 'extsize' is the initial size of uidTable size and 'growsize' is the size by which the uidTable is grown whenever the uidTable is full.

Boolean DCollection()

This method deletes any objects created by ICollection.

Boolean AddIndex(keyprop:ROProperty,dynStat:Integer,rangeExact:Integer)

This method add an index to the Collection. Index is a data structure helping the search of data in database, such as a B-Tree

Boolean Insert(obj:Object)

This method insert an object into the uidTable and into every index the self collection has.

Boolean Delete(obj:Object)

This method delete an object from the collectin. It deletes the UID of the object from the uidTable and deletes from each indices which the collection has.

PUBLIC OPERATIONS

Object Select(op,params)
This method apply boolean operation 'op' with 'params' to each object in the collection and selects the objects which satisfies it.

Object Find(prop:PropertyType,value:Object)
This method selects objects from the collection using the indices. It chooses a proper index depending on the 'prop'(key type), and find an objects from that index by the value 'value'.

Object Image(op:OperationType,params:Bytes)
This method takes the *Image* of each objects by the operation 'op'. This method applies 'op' to each objects of the collection and takes the result. The output of this method is a collection of all the results.

Object Ojoin(jcol:Collection,op:OperationType,params:Bytes)
This method accomplish *OJoin* operation by applying operation 'op' to self and jcol. This method applies 'op' to every possible pair of self and jcol, and make a set of objects which satisfies 'op' with params.

Object Union(ucol:Collectio)
This method make a collection which is the union of self and ucol.

Object Intersect(icol:Collection)
This method make a collection which is the intersection of self and ucol.

Boolean ObjExist(obj:Object)
This method checks whether obj belongs to the self collection.

Boolean IsEmpty()
This method checks whether the self collection is empty or not.

Object First()
This method retrieves the first element of the set. The uidTable has a variable indicating pointer to a current objects which is supposed to be retrieved by enumaration operations, such as First, Next,Last,and Prev.

Object Next()
This method retrieves the Next element of the current element which pointed by the 'currentPos' pointer of uidTable.

Object Last()
This method retrieves the Last element of the collection.

Object Prev()
This method retrieves the Previous element of the current element which pointed by the 'currentPos' pointer of uidTable.

END

ObjectType List(Object) :

This type behaves as the super type of the Btree and Hash Table, both are indices. This has all the operations which is common to Btree and Hash table.

LOCAL PROPERTIES

INTERNAL PROPERTIES

Integer numMems
    This property contains the number of the members of the index(list).

Type memType
    This property contains the type of members which can belongs to the index.

PUBLIC PROPERTIES

LOCAL OPERATIONS

ENObject IntKey(keyval:Object)
    This method transform the keyval of string Type into a form which can be stored inside an index.

INTERNAL OPERATIONS

Boolean IList()
    This property initializes two properties,numMems and memType of the index.

Boolean DList()
    This method deletes any objects created by IList.

Boolean Add(obj:Object,keyval;Object)
    This method add the object 'obj' with the key value 'keyval' into a list.

Boolean Merge(alist:List)
    This method merges alist into self list.

Boolean Delete(obj:Object,keyval:Object)
    This method delete the object 'obj' with the key value 'keyval' from the self list.

Boolean Diff(dlist:List)
    This method is just the opposite of merge. This method deletes all the objects which belongs to dlist.

PUBLIC OPERATIONS

List Select(keyval:Object)
    This method selects objects which has 'keyval' as a keyvalue in the self list.

Boolean Exist(obj:Object,keyval:Object)
    This method checks the existence of objects of which uid is 'obj', and keyvalue is 'keyval'.

Boolean IsEmpty()

This method checks whether the self list is empty or not.

Object First()

This method retrieves the first element, depending on the keyvalue, of an index. All indices has a pointer to a current object.

Object Next()

This method retrieves the Next element from a index, by the key value.

Object Last()

This method retrieves the Last element from a index, by the key value.

Object Prev()

This method retrieves the previous element of the current object from a index, by the key value.

END

ObjectType BTree(List) :

This type is implementation of the B-Tree. B-Tree is a balanced-tree in external storage in which one node is one physical disk page.

LOCAL PROPERTIES

INTERNAL PROPERTIES

Object members

This property contains actual Btree table of which each nodes are Segments which is defined in belows. Actually members property has UID of root Segment. Segments are related by Hierarchical pointers, from parent node to child nodes.

Bytes strings

This property is used only when the type of key, 'keytype', is String. This property stores all the strings values in it. This is necessary because 'String' is neither are independent objects, nor store values in UID itself.

PUBLIC PROPERTIES

LOCAL OPERATIONS

Boolean IBtree(keytype:Type,memtype:Type)

This method make a new Segment to initialize members property and also initialize strings property.

Boolean DBtree()

This method delete all the objects made in IBtree. In case of members property, it deletes all the child segments which are spawned by the root Segment.

INTERNAL OPERATIONS

Boolean Add(obj:Object,keyval:Object)

This method add an object 'obj' with keyvalue 'keyval' to a Btree.

Boolean Del(obj:Object,keyval:Object)

This method delete an object 'obj' with keyvalue 'keyval' from a Btree.

Boolean Split(page:Integer,parent:Object)

This method is called when a segment gets full, which means no more space in a segment. Then this method splits the segment into halfs and store one half into a new segment.

PUBLIC OPERATIONS

List Select(keyval:Object)

This method retrieves objects which has keyval as keyvalue from a Btree.

Boolean Exist(obj:Object)

This method checks whether the obj is in a Btree(TRUE) or not(FALSE).

Object First()

This method returns First element, which has the smallest key value in a Btree.

Object Last()

This method returns Last element, which has the least key value in a Btree.

Object Next()

Btree has pointer to a current object. It increases whenever Next is called and decreases whenever Last is called. This method retrieves the next object,which has the next bigger keyvalue than the current object.

Object Prev()

This method retrieves the previous object, which has the next smaller keyvalue than the current object.

END

ObjectType Segment(Object) :

This type is made to be used as one page for Btree and Hash Table. Btree and Hash Table both has a set of segments as the value of members property.

LOCAL PROPERTIES

INTERNAL PROPERTIES

UIDBytes members

This property contains objects and its keyvalue.

Object mode

This property indicates to which kind(Btree or Hash Table) of index the segment belongs.

Type memType

This property contains the type of objects.

Type keyType

This property contains the type of key value of objects

Integer nodeLen

This property contains the length of each unit in the members. Each unit contains objects ,keyvalue and pointers to othe segment.

Integer size

This property shows how many units can fit into a segment. It is necessary because the unit size is different for Hash table and Btree.

Integer currentPos

This property has the pointer to an object. It is used for the operation First ,Next,Last and Prev.

Integer numOfElms

This property shows how many elements the segment has.

PUBLIC PROPERTIES

LOCAL OPERATIONS

Integer KeyCmp(inkey:Object,valuekey:Object)

This operation compares inkey and value key. It returns EN_ZERO if two values are equal, returns negative integer if inkey is smaller, and returns positive integer if valuekey is smaller.

Object ExtUid(strings:Bytes,node:Integer)

This operation is used only when the type of keyvalue is String. This operation transforms the keyvalue in node 'node' to an external form of string.

Integer Naddr(node:Integer)

This operation computes the address of the node 'node'.

Boolean NdEmpty(node:Integer)

This operation checks whether node 'node' is empty.

INTERNAL OPERATIONS

Boolean ISegment(keytype:Type,memtype:Type,mode:Type)
This operation initialize the properties, keyType, memType,and mode of segment.

Boolean DSegment
This oepration deletes all the objects made by ISegment.

Boolean ChnItem(node:Integer,item:Integer,valu:Objecte)
This operation change the value of the item 'item' of the node 'node'. For example, node = 3 and item = keyval, then this operation put 'value' in the keyvalue part of node 3.

Object GetItem(node:Integer,item:Integer)
This operation gets the value of the 'item' of the node 'node'.

Boolean Compact(node:Integer)
This operation is used by DelAt operation. When the value of a node is deleted, then this operation shift left all the values in the segment to fill up this node.

Boolean DelAt(node:Integer)
This operation delets values of the node 'node' from the segment.

Boolean AddAt(node:Integer,keyval:Object,obj;Object)
This operation add object'obj' with keyvalue 'keyval'.

Integer Del(strings:Bytes,keyval:Object,obj:Object,mode:Type, rescol:Collection)
This operation deletes object 'obj' from the segment. Before delete the objects, this operation finds where the object is.

PUBLIC OPERATIONS

Boolean IsFull()
This operation checks whether the segment is full.

Integer Find(strings,keyval,obj,mode,rescol)
This operation finds the objects with keyvalue 'keyval'. And it stores all the found objects into rescol.

Object First()
This operation selects the first object, which has the least keyvalue in the segment.

Object Next()
This operation selects the next objects to the current object. Segment Type has the property currentPos, which has the pointer to the current object.

Object Last()
This operation selects the last object, which has the biggest keyvalue in the segment.

Object Prev()

This operation selects the previous objects to the currentobject. Segment Type has the property currentPos, which has the pointer to the current object.

END