NAME

index – SDL index attribute types

SYNOPSIS

```
interface index_obj {
public:
    typedef long     keytype; // can be any simple type
    typedef string   valtype; // can be any simple type
                              // including Ref<X> or,
                              // equivalently, X.

    attribute index<keytype,valtype> idx;
    // attribute maps keytype-values to
    // valtype-values through an index.
};
```

DESCRIPTION

The SDL language and the C++ binding to SDL support indices through a simple interface to the underlying Shore Storage Manager (SSM) index facility. Indices are declared as attributes of SDL interfaces, using the template-style notation `index<keytype,valtype>` for the type of the attribute. Keys can be simple types or structured data types. Comparisons on keys are done on byte-by-byte basis, so it is inadvisable to use structured data types for keys (see "ORDERING", below). In general, it is best to use primitive SDL types such as *long, string, char.*

Values have any of the types that keys can have, with the addition of references (*Ref<type>*).

Index operations are performed through member-function calls on index attributes. The index facility allows users to insert (key,value) pairs into an index, to retrieve the set of (key,value) pairs stored in the index for some range of keys, and to delete specific (key,value) pairs. Keys and values within a particular index will each range over some fixed type; typically, the key will be a numeric type or string, and the value will be a reference to some SDL object type.

ORDERING

The underlying Shore Storage Manager support for indices orders keys based on bitwise binary lexical ordering. This ordering works well for the primitive integral and floating point types on the Sparc architecture machines supported by the Beta release, and for strings. Care must be taken if composed types such as *structs* are used as keys; the ordering used may be problematic. There is no way to specify a user-defined ordering. Similarly, uniqueness of (key,value) pairs is dependent on equality of values inserted for equal keys; this comparison for equality is based on strict bitwise equality. If a structure is used as a key or value type, care must be taken that padding in the structure layout (due to alignment) does not cause uninitialized data to be stored in the index. Such uninitialized data may make it impossible for correct comparisons to be done.

LANGUAGE BINDING

Indices in SDL are declared as attributes in objects with a declaration of the form

```
interface a {
public:
    attribute index<keytype,valtype> varname;
};
```

The `index<keytype,valtype>` declarator may be used in other contexts, e.g. as an element of structs and arrays, but not as a member of a union.

In the C++ language binding, an interface attribute looks like a C++ template class instantiation, with visible fields and member functions as listed below.

```
template <class Key, class Val>
class Index
{

  public:
    // externally visibly names for the type parameters
    typedef Key KeyType;
    typedef Val ValType;

    // initialize an index and set type of index.
    shrc init(IndexKind kind) const;

    // insert a (key,value) pair
    shrc insert(const Key &key,const Val &elt) const;

    // remove a (key,value) pair
    shrc remove(const Key &key,const Val &elt) const;

    // remove all (key,value) pairs with a given key
    // the second parameter , nrm, returns the
    // number of elements removed.
    shrc remove(const Key &key, int &nrm) const;

    // find  a single (key,value) pair with a given key
    shrc find(Key key, Val & elt,bool &found) const;
};
```

If an object type (that is, the name of an SDL interface type) is used as the value type parameter in the declaration of an index in SDL, the object is always inserted by reference; that is, a reference to the object is stored as the value, not a copy of its contents. In other words, the declarations `index<keytype,Person>` and `index<keytype,ref<Person> >` are equivalent, and result in the language binding `Index<keytype, Ref<Person> >` .

Similarly, since the language binding for the `string` type is the class `sdl_string,` the declaration `index<string,Person>` results in the language binding `Index<sdl_string,Ref<Person> >`.

The member functions for index attributes are all declared `const` because they do not modify the object containing the attribute; the underlying index is modified, of course, but the object containing the attribute is not.

After an object containing an index attribute is created, the index must be initialized using the **init** method of the index attribute. Allowable values for the single parameter to the **init** member function, which determines the type of SSM index used, are the elements of the enumeration type IndexKind, shown below.

```
enum IndexKind { LHash, BTree, UniqueBTree, RTree, RDTree };
```

Once initialized, an index cannot be removed from an object; the index remains until the object is destroyed, even if the index contains no entries.

The following example illustrates the creation of an index and insertion of values into the index. Here, we assume the existence of an array of persistent pointers (Refs) and we create an index that will associate each element of the array with its position in the array. In the following examples, a Shore transaction is assumed to be active, and the macro `SH_DO` is used to handling possible errors with index calls.

See **transaction(oc)** for information about the SH_DO macro, transactions, and error handling.

```
// SDL language declaration
module index_vars {
    interface Person {
    public:
        attribute string    name;
        attribute long      age;
    };

    interface  IndexObj {
    public:
        attribute index<string, Person> name_index;
        attribute index<long, Person>     age_index;
    }
}

// Assume an instance of IndexObj has
// been created, and we have a valid
// reference to in w:
//
Ref<IndexObj>   w;
shrc            rc;

// Code fragment to initialize:
rc= w->name_index.init(UniqueBTree);
rc= w->age_index.init(BTree);

// Code fragment to insert:
rc = w->name_index.insert(p->name, p);
rc = w->age_index.insert(p->age, p);

// Code fragment to remove all entries
// when you don’t know the value
int num;

rc = w->name_index.remove("daffy", num);

// Code fragment to remove a (key,value) pair
Ref<Person> p;
bool        found=false;

rc = w->age_index.remove(34, p, found);
```

If there is only one (key,value) pair in an index for some key, the value associated with that key may be found using the **find** member function of the index attribute. If more than one (key,value) pair exists for the particular key (the index cannot be a UniqueBTree in this case), the result is undefined.

```
// Code fragment to find an entry
Ref<Person> p;
bool        found=false;

rc = w->name_index.find("donald", p, found);
if(rc==0 && found && p) {
      // go ahead and dereference p
      ...
```

```
    }
```

Index lookups that require more than one (key,value) pair to be returned are described below.

SCANNING AN INDEX

An *IndexScanIter* class template is provided to implement retrieval of the set of (key,value) pairs associated with some range of keys. This template implements a simple iteration primitive that allows the programmer to iterate over the sequence of (key,value) pairs that are the result of a particular index lookup operation.

```
    //
    // an enumeration defined in ShoreApp.h
    //
    enum CompareOp {
        // only the identifiers are given; the
        // values are not to be implied from this
        // pseudo-declaration:
        eqOp, gtOp, geOp, ltOp, leOp,

        // for boundary comparisons:
        gtNegInf, geNegInf,
        ltPosInf, lePosInf
    };

    template <class Key, class Val>
    class IndexScanIter
    {
    public:
        bool     eof;        // scanned the last entry?
        //    copies of key and value of last entry scanned
        Key     &cur_key;
        Val     &cur_val;

        //constructors...
        // ... for an iterator over the entire index--
        IndexScanIter(const Index<Key,Val> idx);

        // ... for an iterator over the part of the index
        //        that lies between l and u, inclusive
        IndexScanIter(const Index<Key,Val> &idx,Key l, Key u) :

        // Alternatively,
        // set upper and lower bounds and conditions
        // before using the iterator:
        //
        SetLB(Key b) ;
        SetUB(Key b) ;
        SetLowerCond(CompareOp o);
        SetUpperCond(CompareOp o);

        // move the cursor -- must be done once
        // to make the cursor valid.
        shrc next();
        // clean up
        shrc close();

        ˜IndexScanIter(); // destroys copies and closes
    }
```

The iterator template class contains public "cursor" data fields that indicate the current (key,value) pair in the set of such pairs resulting from the lookup and a **next** member function, which is used to move the cursor to the next element of the set. The class is initialized through parameters to its constructors; range bounds may also be separately specified by **SetUB** and **SetLB** member functions to set the upper and lower bounds of the retrieve range, respectively. Note that range queries using upper and lower bounds are meaningful only for b-tree indices; hash indices do not support retrieves where the upper and lower bounds differ. If a lower bound is not set, the index scan will begin at the least element contained in the index; if an upper bound is not set, the scan will terminate with the greatest element in the index. If neither lower or upper bounds are set, the entire index will be scanned. The *IndexScanIter* class must be used in conjunction with an index attribute of an SDL object, and the type parameter of the iterator class must match the type of the index attribute. Care must be taken when the index attribute is declared with a reference, as in `index<long,Person>.` The corresponding use of *IndexScanIter* is `IndexScanIter<long,Ref<Person> >` **and you must be sure to place a space between the two occurrences of '>', lest the C++ compiler parse them as a right-shift token.**

An implementation of the now-obsolete `index_iter` template class is retained for backward compatibility, but it is not documented here.

The following examples illustrate the scanning of an index. The two examples use different idioms for the scan.

```
shrc                                                 rc;
// scan of entire index
IndexScanIter<long,Ref<Person> >        iter(w->age_index);

while( !(rc = iter.next()) && !iter.eof) {
      p = iter.cur_val;
      if(p) {
            ...
      }
}

shrc                                                 rc;
// scan of part of index-- from beginning through "c"
IndexScanIter<sdl_string,Ref<Person> >       iter(w->name_index);
iter.SetLowerCond(geNegInf);
iter.SetUB("c");
iter.SetUpperCond(leOp);

for (rc = iter.next(); rc == RCOK && !iter.eof; rc = iter.next() ) {
      p = iter.cur_val;
      if(p) {
            ...
      }
}
```

RESTRICTIONS

The template `Index<k, v>` cannot be instantiated as a transient data structure. It is only valid when used as an attribute of an Shore object whose type is defined as an SDL interface.

BUGS

Only the two index types **BTree** and **UniqueBTree** are supported for applications.

VERSION

This manual page applies to Version 1.1 of the Shore software.

SPONSORSHIP

The Shore project is sponsored by the Advanced Research Project Agency, ARPA order number 018 (formerly 8230), monitored by the U.S. Army Research Laboratory under contract DAAB07-91-C-Q518.

COPYRIGHT

SEE ALSO

intro(sdl), string(cxxlb), errors(oc), transaction(oc) and **Shore Data Language Reference Manual**