NAME

sort_file − Class ss_m Sorting Methods

SYNOPSIS

```
#include <sm_vas.h>  // which includes sm.h

static rc_t
ss_m::sort_file(
    const stid_t&           fid,  // input file
    const stid_t&           sorted_fid,  // output file (given on input)
    int                     nvids,// array size for vids
    const vid_t*            vids, // array for vids for tmp files
    sort_keys_t&            info, // describes sort
    smsize_t                min_rec_size,// for estimating space use
    int                     run_size,    // # of pages for each run
    int                     temp_space   // # of pages for scratch
                            );
```

DESCRIPTION

The Storage Manager method **ss_m::sort_file** is a sorting method used to sort an existing file in the order of some specified keys. More than one key may be used, and the keys' types may be fundamental or user-defined. The objects being sorted may require marshalling when read from disk to memory, and the keys may be located in the marshalled copy. The result of the sort can be a sorted copy of the input file, or it may be a file ready for bulk-loading an index to the original file. For all this generality, the sort uses callbacks to the server for such things as key comparisons, key derivation, object marshalling and unmarshalling, and for producing the final form of keys for output, when the sort key is not the key to be loaded into the index (for example consider R-trees: a polygon is the index key, but the records are sorted on the polygon's Hilbert value).

ARGUMENTS TO SORT_FILE

Before you call **sort_file,** you must create an output file into which **sort_file** will write the results. *Fid* and *sorted_fid* identify the input and output files.

To control the amount of the buffer pool consumed by the sort, you can specify the number of buffer-pool page frames to be used for reading runs and producing output files. The value is given in pages, in the argument *run_size.* Since at all times, one page is fixed for output, and the rest are for reading the input in runs, the real run size for the sort is *(run_size - 1).*

The sort uses temporary files when the input file contains more than one run. These files are spread across as many volumes as the caller provides in the array *vids. Nvids* is the length, in entries, of the array. (The only reason to use more than one volume is if you have temporary volumes on separate spindles.)

For other temporary space needed, **sort_file** respects a limit on the number of pages given in *temp_space.*

The caller can make the sort more efficient by giving an accurate estimate of the minimum record size in *min_rec_size.*

Most of the information about the keys for the sort is given in the *class sort_keys_t* , which is given in the argument *info* to **sort_file.** This class is described in detail in another section, below.

CALLBACKS

CALLBACK ARGUMENT TYPES

Several callback functions are used, and several of the callback functions use the data types *key_cookie_t* and *key_location_t.* A key cookie is an opaque value that the server passes through **sort_file** to the callback functions:

```
typedef void * key_cookie_t;

/*
 * key_location_t: used in callback functions
 * to describe offset and length of a key in a record (object)
 */
struct key_location_t {
    smsize_t            _off;
    smsize_t            _length;

    key_location_t() : _off(0), _length(0) {}
    key_location_t(const key_location_t &old) :
        _off(old._off), _length(old._length) {}
};
```

A *key_location_t* structure holds the length of a key and its offset from the start of the record (object) in which it resides.

The following data type is meant to be a base type for any factory that generates dynamically-allocated "objects" (space). It provides a generic interface for freeing the allocated space.

```
struct factory_t {
   /* users can derive their write own factories
    * that inherit from this
    */
public:
   virtual      void freefunc(const void *, smsize_t)=0;
   virtual      void* allocfunc(smsize_t)=0;
   virtual      NORET ~factory_t();
   // none: causes no delete - used for statically allocated space
   static factory_t*    none;

   // cpp_vector - simply calls delete[]
   static factory_t*    cpp_vector;

   void factory_t::freefunc(vec_t&v); // calls freefunc for each
                              // vector element.
};
```

When a callback function returns something it allocated, it passes back to the storage manager the factory used to allocate (so the storage maanger can use it to free the object). Similarly, if the callback function needs to free space allocated by the storage mangager, it invokes the free function on the *factory_t* associated with the object being freed.

Another data type used by the callback functions is:

```
class object_t {
        object_t(const object_t&o);
        object_t(const void *hdr, smsize_t hdrlen, factory_t&hf,
                        const void *body, smsize_t bodylen, factory_t&bf);
        ~object_t();

        bool      is_in_buffer_pool() const;
        smsize_t  hdr_size() const;
        smsize_t  body_size() const;
        smsize_t  contig_body_size() const; // pinned amt
                // if is_in_buffer_pool() and is a large object,
```

```
                // contig_body_size() != body_size()
                  // else they are equal.

        const void *hdr(smsize_t offset) const;
        const void *body(smsize_t offset) const; // not valid ptr
                 // in buffer pool and large object

        void      freespace();
        bool      is_valid() const; // false after freespace() called
        void      assert_nobuffers() const; // should work after freespace()

        w_rc_t    copy_out(bool in_hdr, smsize_t offset,
                           smsize_t length, vec_t&dest) const;

};
```

An *object_t* describes an entire storage manager record (object). The object might be in the buffer pool or it might have been copied into the heap (dynamically-allocated memory). The public methods of *object_t* allow one to inspect or copy portions of the object without dealing with such details of its location.

The following data type assists the manipulation of a *key,* whether it is somewhere in an *object* or it is derived.

```
class skey_t {
public:
       // construct a key that's in an object
        skey_t(const object_t& o, smsize_t offset,
                        smsize_t len, bool in_hdr);

       // construct a derived key
        skey_t(void *buf, smsize_t offset,
                        smsize_t len,  // KEY len, NOT NECESSARILY BUF len
                        factory_t& f);
        ~skey_t();

        smsize_t  size() const;
        bool      is_valid() const;
        bool      is_in_obj() const;
        bool      is_in_hdr() const;

        void      freespace();
        void      assert_nobuffers()const;

        smsize_t  contig_length() const; // pinned amt or key length
       // If is_in_obj() and is in object's body and the object
        // is large, contig_length() != size() possibly.
        w_rc_t    copy_out(vec_t&dest) const;

        const void *ptr(smsize_t offset) const;  // key
               // valid ptr ONLY if not a key in a large object
               // In that case, you MUST use copy_out to inspect
                 // the key.
};
```

MARSHAL/UNMARSHAL OBJECT CALLBACK FUNCTIONS

```
typedef w_rc_t (*MOF)(
        const rid_t&    rid,  // record id
        const object_t& obj_in,
        key_cookie_t    cookie,  // type info
                                 // func must allocate obj_out,
        object_t*&      obj_out  // SM will free mem, delete
        );
```

When an object is first encountered by the sort code, if a marshal function is provided, it is called to produce an in-memory version of the entire object (presumably byte-swapped, pointer-swizzled, etc). This copy of the object is passed in to the create-key callback function.

When a large file is being sorted, and the input file is largely randomly ordered, the last pass (when the final output is generated) can either

a) pin the original object to copy it (saving only keys and record-ids during the merge phases), which might result in unacceptably random I/O, or

b) write complete copies of the object during the merge phases, causing more I/O, but with a sequential pattern.

When there is only a single run (the entire sort is in memory), this is not an issue. When option *b)* is in effect, if the object is marshalled, the sort must issue a call to another function to convert the object back to disk form:

```
typedef w_rc_t (*UMOF)(
        const rid_t&    rid,  // orig record id of object in buffer
        const object_t& obj_in,
        key_cookie_t    cookie,  // type info
                                 // func must allocate obj_out,
        object_t*&      obj_out // SM will copy to disk, free mem, delete
        );
```

This call is avoided when possible, such as in the case where the objects are large and a shallow copy is done (the old file is being destroyed). In this case, only the metadata are carried along from phase to phase in temporary files.

CREATE-SORT-KEY CALLBACK FUNCTION

```
typedef w_rc_t (*CSKF)(
        const rid_t&    rid,
        const object_t& in_obj,
        key_cookie_t    cookie,  // type info
        factory_t&      internal,
        skey_t*&        out
        );
```

This function is called for each key for each object, so the key can be located or derived. This function is where disk-to-memory transformations are made. It is also where arbitrary record-to-key mappings are made.

In the case of a derived key, the function must allocate the space, derive the key, write it into the allocated space, and construct an *skey_t* to describe it.

In the case of a key that can be compared in disk-resident form, the function must simply construct an *skey_t* using the given object, and provide the proper offset, length, and Boolean flag to indicate whether the key is in the header or body of the object.

The *rid* might be superfluous.

Such a function might also be provided for the purpose of generating the output key for index bulk-loading.

KEY-COMPARISON CALLBACK FUNCTION

```
typedef int (*CF) (uint4_t , const void *,
                   uint4_t , const void *);
```

Each time two keys are compared, this function is called. The comparison is necessarily based solely on the keys themselves; no metadata are passed in. This function must cope with whatever endian-ness it is dealt: the server (caller of **sort_file**) must properly determine, *a priori,* whether the comparison function will be given adequately aligned, byte-swapped keys. (All the default comparison functions assume they are passed properly aligned, properly swapped values.)

CONTROLLING THE SORT _FILE BEHAVIOR

The storage manager uses the values stored in the *sort_keys_t* to determine what callbacks are needed to locate or derive keys, and what kind of output to produce.

```
class sort_keys_t {
public:
    sort_keys_t(int nkeys);
    sort_keys_t(const sort_keys_t &old);
    ~sort_keys_t();

    int     nkeys() const { return _nkeys; }

    // Control output: copy of input file, or <key,OID> pairs for index?

    //  -- to produce a copy of the input file:
    int     set_for_file(bool deepcopy, bool keeporig, bool carry_obj);
    bool    is_for_file() const;

        // These are sensible only when is_for_file()==true:
       // Make temp copies of entire object (if #runs is >1)?
       bool    carry_obj() const;
       int     set_carry_obj(bool value = true);

       // copy entire object or only large object meta-data?
       // !deep_copy is more efficient if we aren't saving the
       // original file.
       bool    deep_copy() const;
       int     set_deep_copy(bool value = true );

       // Save the original file (or destroy it?)
       bool    keep_orig() const;
       int     set_keep_orig(bool value = true );

    //  -- to produce <key,OID> pairs:
    int     set_for_index(); // use sort key
    int     set_for_index( CSKF lfunc, key_cookie_t ck); // derive output key
    CSKF    lexify() const ;
    key_cookie_t     lexify_cookie() const;
    bool    is_for_index() const;

    // Want to guarantee stable sort? use this:
    void    set_stable(bool val);
    bool    is_stable() const;

    // sort-order: up or down?
```

```
bool    is_ascending() const;
int     set_ascending( bool value = true);

// Permit duplicates in output?
bool    is_unique() const ;
int     set_unique( bool value = true);

// Permit duplicate nulls in output?
bool    null_unique() const ;
int     set_null_unique( bool value = true);

// Want to marshal entire object ?
int     set_object_marshal( MOF marshal, UMOF unmarshal, key_cookie_t c);
MOF     marshal_func() const;
UMOF    unmarshal_func() const;
key_cookie_t    marshal_cookie() const;

// How to locate a key:
int set_sortkey_fixed(int key,
            smsize_t off, smsize_t len,
            bool in_header,
            bool aligned,
            bool lexico,
            CF   cfunc
            );
int set_sortkey_derived(int key,
            CSKF gfunc,
            key_cookie_t cookie,
            bool in_header,
            bool aligned,
            bool lexico,
            CF   cfunc
            );

smsize_t offset(int i) const;
smsize_t length(int i) const;
CSKF keyinfo(int i) const;
CF keycmp(int i) const;
key_cookie_t cookie(int i) const;
bool     is_lexico(int i) const;
bool     is_fixed(int i) const;
bool     is_aligned(int i) const;
bool     in_hdr(int i) const;


// Various static functions that can be used to
// avoid callback, or to avoid writing callback functions
// for basic types:
static w_rc_t noCSKF(
    const rid_t& rid,
    const object_t& obj,
    key_cookie_t    cookie,  // type info
    factory_t&      f,
    skey_t*&        out
```

```
        );

    static w_rc_t generic_CSKF(
        const rid_t&    rid,
        const object_t& in_obj,
        key_cookie_t    cookie,  // type info
        factory_t&      internal,
        skey_t*&        out
        );

    static w_rc_t noMOF (
        const rid_t&    rid,  // record id
        const object_t& obj,
        key_cookie_t    cookie,  // type info
        object_t*&      out
        );

    static w_rc_t noUMOF (
        const rid_t&    rid,  // record id
        const object_t& obj,
        key_cookie_t    cookie,  // type info
       object_t*&      out
        );

    static int string_cmp(uint4_t , const void* , uint4_t , const void*);
    static int uint8_cmp(uint4_t , const void* , uint4_t , const void* );
    static int int8_cmp(uint4_t , const void* , uint4_t , const void* );
    static int uint4_cmp(uint4_t , const void* , uint4_t , const void* );
    static int int4_cmp(uint4_t , const void* , uint4_t , const void* );
    static int uint2_cmp(uint4_t , const void* , uint4_t , const void* );
    static int int2_cmp(uint4_t , const void* , uint4_t , const void* );
    static int uint1_cmp(uint4_t , const void* , uint4_t , const void* );
    static int int1_cmp(uint4_t , const void* , uint4_t , const void* );
    static int f4_cmp(uint4_t , const void* , uint4_t , const void* );
    static int f8_cmp(uint4_t , const void* , uint4_t , const void* );

    static w_rc_t f8_lex(const void *, smsize_t , void *);
    static w_rc_t f4_lex(const void *, smsize_t , void *);
    static w_rc_t u8_lex(const void *, smsize_t , void *);
    static w_rc_t i8_lex(const void *, smsize_t , void *);
    static w_rc_t u4_lex(const void *, smsize_t , void *);
    static w_rc_t i4_lex(const void *, smsize_t , void *);
    static w_rc_t u2_lex(const void *, smsize_t , void *);
    static w_rc_t i2_lex(const void *, smsize_t , void *);
    static w_rc_t u1_lex(const void *, smsize_t , void *);
    static w_rc_t i1_lex(const void *, smsize_t , void *);
};
```

--- OUTPUT FOR FILE OR INDEX

When setting up the *sort_keys_t* object, use one of the following two methods:

```
int set_sortkey_fixed(
       int key,
       smsize_t off, // offset from beg of hdr or body
       smsize_t len, // len of key
```

```
        bool in_header, // true if in header
        bool aligned,  // if guaranteed in all cases
        bool lexico, // true if, like strcmp() of strings,
               // chunks of key can be successively compared
        CF   cfunc
        );

int set_sortkey_derived(int key,
        CSKF func,
        key_cookie_t cookie,
        bool in_header,
        bool aligned, // true if CSKF produces key guaranteed to be
               // properly aligned for key comparisons
        bool lexico,
        CF   cfunc
        );
```

Key is an integer identifying the key, with 0 being the most significant key and 4 being the least significant key. (At most 5 keys are supported.) *Off* is the offset of the start of the key from the beginning of the header or body of the object. *Len* is the length, in bytes, of the key. *Aligned* should be set to true if the key is guranteed to be aligned for key-comparison purposes as it sits in the buffer pool. This allows the sort code to avoid excess copies. *Lexico* should be *true* if the key is sortable in arbitrary sub-parts with the given key-comparison function. *In_header* should be *true* if the key appears in the header rather than in the body of the object. *Func* is the callback function that produces a *skey_t* for the key. *Cfunc* is the callback function for comparing keys.

Use **set_sortkey_fixed** if the key is of fixed length and is in a fixed location in the object and either it can be compared disk-resident format or a marshal function is used. Use **set_sortkey_derived** if the key is to be derived or marshalled (rather than the entire object being marshalled).

The return value is 1 if an error is encountered, 0 if not.

The rest of the functions are self-explanatory.

--- SORTING ON MULTIPLE KEYS

Sort_file can sort on several keys (at most 5), but this is not useful when sorting for bulk-loading indices, as multi-key indices are not supported in the Storage Manager.

--- ELIMINATING DUPLICATES

The sort method can keep or eliminate duplicates. When

```
int set_unique(bool value = true);
```

is given the argument *true,* the sort will eliminate duplicate (null- and non-null-) key values.

```
bool unique() const;
```

returns *true* if duplicates are to be eliminated. NB: duplicates are records with common **keys.** Unlike the old sort, this sort implementation compares only keys to find duplicates.

If only duplicate null keys are to be eliminated (desired for preparing to bulk-load an R-tree), use the method

```
int set_null_unique( bool value = true);
```

and use

```
bool null_unique() const;
```

to return the value last given to **set_null_unique.**

NULLS

Missing keys are identified by the CSKF callback functions as having length 0. The sort does not distinguish 0-length keys from null keys.

EXAMPLE: SORT FOR R-TREE BULK LOAD

```
#include <sm_vas.h>

extern ss_m* sm;

class  data_type
{
    // all objects derive from this class
public:
    static int              compare (uint4_t , const void *,
                        uint4_t , const void *);
    smsize_t         length() const;
    const data_type* field(int i, smsize_t &offset) const;
    static data_type &      typeA;
    static data_type &      typeB;
    CF               comparefuncptr()const;

};

struct my_cookie {
    int                     field_no;
    const data_type* type;
};

w_rc_t
myCSKF(
    const rid_t&        ,  // record id -notused
    const object_t&  obj_in,
    key_cookie_t     k,  // type info
    factory_t&              ,  // not used
    skey_t*&         key
)
{
    my_cookie *      c = (my_cookie *)k;
    smsize_t  offset = 0;
    const data_type *d = c->type->field(c->field_no, offset);
    key = new skey_t(obj_in, offset, d->length(), false);
    return RCOK;
}


int
main()
{
    /* arbitrary numbers for the sake of an example: */
    smsize_t  min_rec_sz = 300;
    int              runsize = 10;

    sort_keys_t info(3);
    int              nkeys =3;
```

```
my_cookie types[nkeys];

types[0].field_no = 0;
types[0].type = &data_type::typeA;
types[1].field_no = 3;
types[1].type = &data_type::typeB;
types[2].field_no = 10;
types[2].type = &data_type::typeA;

/*
 * shallow copy large objects and destroy orig file,
 * copy objects as we write runs to temp files
 */
if(info.set_for_file(false, false, true)) {
   /* handle error */
}

/*
 * Sort on derived key.
 */
for(int k=0; k<nkeys; k++) {
   if( info.set_sortkey_derived(1,
       myCSKF,
       &types[k], // type info
       false, // is in body
       true,     // produces key aligned for comparisons
       false, // no chunk comparisons
       types[k].type->comparefuncptr()
   )) {
   /* handle error */
   }
}

if(info.set_ascending(false)) { // want descending
   /* handle error */
}
// allow duplicates
if(info.set_unique(false)) {
   /* handle error */
}
// allow multiple nulls
if(info.set_null_unique(true)) {
   /* handle error */
}
w_rc_t    rc;
vid_t     vid = 1; // volume on which to create output file
                   // and temp files.
stid_t    fid;   // (= ...) file to sort
stid_t           ofid;

/* create output file for results */
rc =  sm->create_file(vid, ofid, ss_m::t_load_file);
if(rc) {
   /* handle error */
```

```
    }
    rc = sm->sort_file( fid,
       ofid,
       1,
       &vid,
       info          , // sort_keys_t&
       min_rec_sz    ,
       runsize              , // run sz
       runsize*ss_m::page_sz // # pages for scratch
       );


    if(rc) {
       /* handle error */
    }

    return 0;
}
```

EXAMPLE: SORT FILE

```
#include <sm_vas.h>

extern ss_m *sm;

extern factory_t* four_byte_aligned_factory;  // elsewhere in my program

class OBJECT
{
    /* my object type */

public:

    /* byte-swapping functions */
    static  w_rc_t hton(
       const rid_t&  rid,
       const object_t& obj_in,
       key_cookie_t    cookie,
       object_t*&    obj_out
       );
    static  w_rc_t ntoh(
       const rid_t&  rid,
       const object_t& obj_in,
       key_cookie_t    cookie,
       object_t*&    obj_out
       );

};


nbox_t universe;

w_rc_t
hilbertCSKF(
    const rid_t&        ,  // record id - not used
    const object_t&  obj_in,
```

```
    key_cookie_t     cookie,  // offset into body
    factory_t&              ,      // sm internal - not used
    skey_t*&         key
)
{
    nbox_t           box(2);
    smsize_t         length = smsize_t(box.klen());
    smsize_t         offset = smsize_t(cookie);

    int              hvalue=0; /* result value */

    {   /* materialize the box and compute its Hilbert value */
       char *tmp = new char[length];
       if(!tmp) return RC(ss_m::eOUTOFMEMORY);
       vec_t bxvec(tmp,length);
       rc_t rc = obj_in.copy_out(false/* not in hdr */, offset, length, bxvec);
       if(rc) return rc;

       box.bytes2box(tmp, length);
       delete[] tmp;

        hvalue = box.hvalue(universe);
    }

    /* allocate space for the result */
    void *c = four_byte_aligned_factory->allocfunc(sizeof(hvalue));
    memcpy(c, &hvalue, sizeof(hvalue));

    key = new skey_t(c, 0, sizeof(hvalue), *four_byte_aligned_factory);
    return RCOK;
}

w_rc_t
originalboxCSKF(
    const rid_t&        , // not used
    const object_t&  obj_in,
    key_cookie_t     cookie,  // offset into body
    factory_t&              ,
    skey_t*&         out
)
{
    smsize_t         offset = smsize_t(cookie);
    smsize_t         length;
    {
       nbox_t              box(2);
       length = smsize_t(box.klen());
    }
    out = new skey_t(obj_in, offset, length, false);
    return RCOK;
}

int
main()
{
```

```
sort_keys_t info(1);

/* arbitrary numbers for the sake of an example: */
smsize_t  offset = 3;
smsize_t  min_rec_sz = 300;
int              runsize = 10;

CSKF resultfunc = originalboxCSKF;

if(info.set_for_index(resultfunc, key_cookie_t(offset))) {
   /* handle error */
}

/*
 * Sort on derived key.
 */
if( info.set_sortkey_derived(1,
   hilbertCSKF,
   (void *)&offset, // into body
   false, // in body
   true,  // hilbert produces key aligned for comparisons
   false, // no chunk comparisons
   sort_keys_t::int4_cmp
   )) {
   /* handle error */
}

info.set_stable(true);

if(info.set_object_marshal(OBJECT::ntoh, OBJECT::hton, 0)) {
   /* handle error */
}
if(info.set_ascending(true)) {
   /* handle error */
}
// no duplicates
if(info.set_unique(true)) {
   /* handle error */
}
// no duplicate nulls (no nulls at all in this case)
if(info.set_null_unique(true)) {
   /* handle error */
}
w_rc_t    rc;
vid_t     vid = 1; // volume on which to create output file
                   // and temp files.
stid_t    fid; // = ...;  file to sort
stid_t           ofid;

/* create output file for results */
rc =  sm->create_file(vid, ofid, ss_m::t_load_file);
if(rc) {
   /* handle error */
}
```

```
        rc = sm->sort_file( fid,
           ofid,
           1,
           &vid,
           info          , // sort_keys_t&
           min_rec_sz    ,
           runsize              , // run sz
           runsize*ss_m::page_sz // # pages for scratch
           );


        if(rc) {
           /* handle error */
        }

        /* Now bulk-load the index of choice */

        return 0;
    }
```

ERRORS

RCOK is returned if there is no error.

VERSION

This manual page applies to Version 2.0 of the Shore Storage Manager.

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. Further funding for this work was provided by DARPA through Rome Research Laboratory Contract No. F30602-97-2-0247.

COPYRIGHT

SEE ALSO

sort_file(ssm)