NAME

pin_i, pin, repin, unpin – Class for Pinning Records

SYNOPSIS

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

class pin_i : public smlevel_top {
public:
    enum flags_t {
      pin_empty         = 0x0,
      pin_rec_pinned          = 0x01,
      pin_hdr_only            = 0x02,
      pin_separate_data = 0x04,
      pin_lg_data_pinned      = 0x08  // large data page is pinned
    };

    NORET         pin_i();
    NORET         ~pin_i();

    /* Logical-ID version */
    rc_t          pin(
      const lvid_t&         lvid,
      const serial_t&       lrid,
      smsize_t        start,
      lock_mode_t     lmode = SH);

    /* Physical-ID version */
    rc_t        pin(
        const rid_t         rid,
        smsize_t            start,
        lock_mode_t         lmode = SH);


    void          unpin();
    void                set_ref_bit(int value);
    rc_t                repin(lock_mode_t lmode = SH);
    rc_t          next_bytes(bool& eof);
    bool          pinned() const;
    bool          pinned_all() const;
    bool          up_to_date() const;

    // methods for accessing the record
    smsize_t            start_byte() const;
    smsize_t            length() const;
    smsize_t            hdr_size() const;
    smsize_t            body_size() const;
    const char*         hdr() const;
    const char*         body();
    bool          is_small() const;
    const record_t*     rec() const;
    const serial_t&     serial_no() const;
    const lvid_t& lvid() const;
    const rid_t&  rid() const;
```

```
    // methods for changing the record
    rc_t                update_rec(smsize_t start, const vec_t& data);
    rc_t                update_rec_hdr(smsize_t start, const vec_t& hdr);
    rc_t                append_rec(const vec_t& data);
    rc_t                truncate_rec(smsize_t amount);

    // miscellaneous
    const char*   hdr_page_data();
    lpid_t        page_containing(
      smsize_t        offset,
      smsize_t&       start_byte) const;
    static void   pin_stats(
      u_long&              pins,
      u_long&              unpins,
      bool            reset);

private:
    // these methods are disabled
    pin_i(const pin_i&);
    pin_i& operator=(const pin_i&);
};
```

DESCRIPTION

Class **pin_i** supports pinning records in the buffer pool and provides a variety of methods for accessing information about a record and its contents. A **pin_i** object is basically a handle to a pinned record. The _i suffix in a class name indicates that a class is an iterator. Class **pin_i** is an iterator since it is used to iterate over all by bytes in a record's body.

CONSTRUCTORS and DESTRUCTORS

pin_i()

The **pin_i** constructor simply initializes a pin_i object.

~pin_i()

If a record is pinned, **~pin_i** un-pins it.

PINNING

pin(lvid, lrid, start, lmode)

The **pin** method pins a range of bytes of a record. For small records (those that fit on one page), the entire record body will be pinned. For large records, only one page of the body will be pinned at a time. For both small and large records, the record header is always pinned as well. The first two parameters, *lvid* and *lrid* specify the **logical ID** of the record to be pinned.

The *start* parameter specifies a byte offset into the record body corresponding to a region of the body to pin. However, the pin operation will always adjust the starting location of the pin to reflect the beginning of the page containing the byte indicated by the *start* parameter. The true starting location and size of the pinned region are available from the **start_byte** and **length** methods, respectively. For example, *start=0* will always pin the first page of the record body, as will *start=10* (assuming the record is at least 10 bytes long). In both cases, **start_byte** will return 0 and length will either the entire record, if small, or approximately the length of a page, if large.

The *lmode* parameter specifies how the record should initially be locked (ie. the lock mode). The options are **SH** (share/read lock) and **EX** (exclusive/write lock). **EX** the pinned record will be eventually updated (through update_rec, unpdate_rec_hdr, append_rec, or truncate_rec). Using EX in these cases will improve performance and reduce the risk of deadlock, but is not necessary for correctness.

unpin()

The **unpin** method unpins the current record (assuming one is pinned). The pin object can then be used to pin another record. The destructor automatically calls **unpin.**

set_ref_bit(value)

The **set_ref_bit** sets the reference bit *value* to use for the buffer frame containing the currently pinned body page when the page is unpinned. A *value* of 0 is a "hate" hint indicating that the frame can be reused as soon as necessary. By default, a *value* of 1 is used indicating the page will be cached until at least 1 sweep of the buffer clock hand has passed. Higher values cause the page to remain cached longer.

repin(lmode)

The **repin** method repins the previously pinned record, locking it in the mode specified by *lmode* (see **pin** for further discussion of *lmode.* The repin method has a number of uses. First, when the previously pinned record needs to be repinned, it is more efficient to call **repin** than to call **pin** with the ID of the record. Second, it can be used to repin the record after some other operation has modified the page containing the record. See the **RESTRICTIONS** section for further information on this use of **repin.** Third, repin can be used to upgrade the lock held on the currently pinned record. However, this is usually unnecessary since all of the methods in class **pin_i** that modify the record will automatically acquire an **EX** mode lock on the record.

next_bytes(eof)

The **next_bytes** method gets the next range of bytes available to be pinned. Parameter *eof* is set to **true** if there are no more bytes to pin. When eof is reached, the previously pinned range remains pinned

pinned()

The **pinned** methods returns **true,** if a record is currently being pinned, and **false** otherwise.

pinned_all()

The **pinned_all** methods returns **true** if the pinned region includes the entire record, otherwise. **false** is returned.

up_to_date()

The **up_to_date** method returns true if a record is pinned and pin no changes have been made to the page containing the record since it was pinned. See the **RESTRICTIONS** section for information on using this method.

ACCESSING THE RECORD

start_byte()

The **start_byte** method returns the offset, from the beginning of the record, where the pinned region starts. Ie. it is the offset of the location pointed to by the **body** method. **Note:** the value returned by **start_byte** may not be the *start* location passed to **pin** since pinned regions are always aligned on page boundaries.

length()

The **length** method returns the length, in bytes, of the pinned region.

hdr_size()

The **hdr_size** method returns the size, in bytes, of the record header.

body_size()

The **body_size** method returns the size, in bytes, of the entire record body (not just the portion pinned).

hdr()

The **hdr** method returns a pointer to the pinned header. Note that the pointer is **const** since the header can only be updated via **update_rec_hdr.**

body()

The **body** method return a pointer to the pinned region of the body. Note that the pointer is **const** since the body can only be updated via the update methods described below.

is_small()

The **is_small** method returns **true** if the record body fits on the same page as the header and thus is pinned in it’s entirety.

serial_no()

lvid()

The **serial_no** and **lvid** methods return the logical ID of the pinned record assuming it was pinned using logical IDs. **Note:** theses IDs are the "snapped" values -- ie. they are the volume ID where the record is located and the record’s serial# on that volume. Therefore, these may be different than the ones passed in to pin the record.

rid()

The **rid** method return the physical ID of the record.

rec()

The **rec** method returns the pointer **record_t** structure that is used internally to access records on pages. Most uses of this structure, such as finding the size of the record, are already provided by other **pin_i** methods. The primary use of this method is debugging or to do things not provided by other **pin_i** methods.

UPDATING A PINNED RECORD

update_rec(start, data)

update_rec_hdr(start, hdr)

append_rec(data)

truncate_rec(amount)

These methods are used to change a pinned record. They correspond to the class **ss_m** methods of the same name describe in **file(ssm).** They can be called on any pinned record regardless of where and how much is pinned. Using these methods when a record is pinned is considerably more efficient than calling the corresponding **ss_m** methods. Also, the **up_to_date** method will return **true** after calling one of these methods, even though they update the record.

OTHER MEMBER FUNCTIONS

hdr_page_data()

The **hdr_page_data** method returns a pointer to beginning of the page containing the pinned record header. This is used by the Shore VAS when sending entire pages of records to the client. A page can be interpreted with the **shore_file_page_t(ssm)** class (undocumented).

page_containing(offset, start_byte)

The **page_containing** returns the page ID of the page containing the *start_byte* offset from the beginning of the record body. This function is not currently supported.

pin_stats(pins, unpins, reset)

The **pin_stats** method return the number of pins and unpins performed. The *pins* parameter will equal the sum of all **pin** and **repin** calls. The *unpins* parameter will equal the sum of all **unpin repin** calls.

CAVEATS

While a pin_i is valid, a page is fixed in the buffer pool. Pages should not be fixed for long periods of time, and a thread that pins multiple pages in the buffer pool runs the risk of exhausting the buffer pool resources.

It is risky to pin and update two records concurrently, that is, to update one record while another record is pinned. If the records are on the same page, updating one record can invalidate the pin of the other record. The method **repin** can be used to re-validate the pin of the second record, however, it is still a risky proposition: read on.

On the other hand, if the two records are on different pages, and if the thread is not observing a protocol to order the pins, (more importantly, all threads in the VAS must observe the same protocol), **latch-latch deadlocks** can occur (pages are latched when they are fixed), if fine-grained (record) locking is in effect. Whereas deadlock detection is performed on locks, latches are much lighter-weight and do not perform deadlock detection.

ERRORS

To Do.

EXAMPLES

To Do.

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

file(ssm), scan_file_i(ssm), intro(ssm),