CS764 Project Report – Linear Hashing in MINIREL

Weiqing Huang

Computer Sciences Department
University of Wisconsin
Madison, WI 73706
whuang@cs.wisc.edu

May 12, 1995

1 Introduction

In this project, we implemented an access method for MINIREL - Linear Hashing. Linear hashing is one widely used dynamic hashing scheme which adjusts gracefully to inserts and deletes. One record can be retrieved in 1.2 disk access on average.[1][3][2]

We provide ordinary index operations on linear hashing: insertion (with file expansion if necessary), deletion (with file contraction if necessary) and an index scan which supports deleting the currently scanned record. Concurrency control and recovery are also supported and because of the special structure of the index, some techniques have been used to improve the concurrency. [4]

Our component is dependent on several other components in MINIREL including Buffer Manager, Space Manager, Lock Manager and Recovery Manager. The query processing components will use our component if linear hashing index is used to access a relation.

2 External Interface

The class **LinearHashingFile** is used to create/open a linear hashing index, destroy the index, insert records to the index and delete records from the index. The class **LinearHashingScan** is used to open an exact match scan on a linear hashing index. The scan can be used to get the next record and delete the current record.

2.1 LinearHashingFile

Here is the public interfaces of LinearHashingFile:

```
class LinearHashingFile : public IndexFile {
public:
    // constructor for the LinearHashingFile class -- create the index
    // name -- name of the index
```

```
    // length, type -- describing the key attribute
    // N -- the number of initial buckets, default = 1
    LinearHashingFile(Status& status, const char* name, const int length,
                      const AttrType type, const int N = 1);

    // constructor for the LinearHashingFile class -- open the index
    // assume the index already exists
    // name -- name of the index
    LinearHashingFile(Status& status, const char * name);

    // destructor
    ~LinearHashingFile();

    // destroy the index
    Status destroyFile(void);

    // insert an entry <value, rid> to the index
    Status insert(const void* value, const RID rid);

    // delete an entry <value, rid> from the index
    Status Delete(const void* value, const RID rid);

    // open an exact match scan with "value" as the key
    IndexFileScan * new_scan(const void* value);

#ifdef DEBUGLH
    void PrintHeader();             // print the header info
    void PrintBucket();             // print all the buckets
    void PrintBucket(int BucNum);   // print bucket "BucNum"
#endif DEBUGLH
};
```

2.1.1 LinearHashingFile(Status& status, const char* name, const int length,const AttrType type, const int N = 1) LinearHashingFile(Status& status, const char * name)

These two are the constructors for **LinearHashingFile**. The first constructor is used to create a new index. *name* is the name of the index to be created. *length* is the length of the key and *type* is the type of the key. *N* is the number of initial buckets. The default value of N is 1. *status* returns the status of the operation. The second constructor is used to open an existing index. It only takes 2 arguments: *status* as the returning status of the operation and *name* as the name of the index to be opened.

2.1.2 ~LinearHashingFile()

The destructor for **LinearHashingFile**. It's used to close the index and does some cleanup such as unpin the header page and release the Intentional Shared lock on root page.

2.1.3 Status destroyFile(void)

Destroy the whole index. After the index being destroyed, no operation can be done on this index. Return OK if no errors. Otherwise, return the error code.

2.1.4 Status insert(const void* value, const RID rid)

Insert to the index a record whose key is *value* and record ID is *rid*. Return OK if no errors. Otherwise, return the error code.

2.1.5 Status Delete(const void* value, const RID rid)

Delete from the index the record whose key is *value* and record ID is *rid*. Return OK if the record has been deleted successfully. Return RECNOTFOUND if no such record is found in the index. Otherwise, return the error code.

2.1.6 IndexFileScan * new_scan(const void* value)

Open a new (exact match) scan on the index. It will search all the records with key equal to *value*. If the operation fails, return NULL. In our interfaces, we require that when a scan is opened, no update operation is allowed except *delete_current()*.

2.1.7 void PrintHeader()
void PrintBucket()
void PrintBucket(int BucNum)

Print out the index for debugging purpose. *PrintHeader()* prints the header information.*PrintBucket()* prints all the buckets. *PrintBucket(BucNum)* prints bucket *BucNum*. These methods are included when DEBUGLH flag is set during compiling time.

2.2 LinearHashingScan

Here is the public interfaces of **LinearHashingScan**. Notice that the constructor for **LinearHashingScan** is not public. A scan can only be created by calling **LinearHashingFile::new_scan**.

```
class LinearHashingScan : public IndexFileScan {
public:
    // get the next record
    Status get_next(RID & rid);

    // delete the record currently scanned
    Status delete_current();

    // destructor
    ~LinearHashingScan();
};
```

2.2.1 ~LinearHashingScan()

The destructor for **LinearHashingScan**. It's used to close the scan and does some cleanup.

2.2.2 Status get_next(RID& rid)

Get the next record in the scan. Set *rid* to the record ID of the next record. Return OK if the next record has been found successfully. Return NOMORERECS if all records have been scanned. Otherwise return the error code.

2.2.3 Status delete_current()

Delete the currently scanned record. It can only be called after *get_next(rid)* has been called. Otherwise, the result is unpredictable. Return OK if the record has been deleted successfully. Return NOMORERECS if the scan has not started yet. Otherwise return the error code.

2.3 Downward interfaces

The downward interfaces we used in our component include:

- Status BufMgr::newPage(PageId, Page*&, int = 1)
- Status BufMgr::freePage(PageId)
- Status BufMgr::pinPage(PageId, Page*&)
- Status BufMgr::unpinPage(PageId,DIRTY_FLAG)
- Status DB::add_file_entry(const char*, PageId);
- Status DB::get_file_entry(const char*, PageId);
- Status DB::delete_file_entry(const char*, PageId);
- Status RecoveryMgr::WriteUpdateLog(int,PageId,int,void*,void*,Page*)
- Status lock_manager_def::lock_page(page_def,lock_type)
- Status lock_manager_def::unlock_page(page_def)
- Status lock_manager_def::upgrade_lock(page_def,lock_type)
- the constructors and destructors for the above classes

3 Internal Design

3.1 Algorithm

In linear hashing, we have a family of hash functions $h_0, h_1, h_2, ...$ with the property that each function's range is twice that of the predecessor. We keep track of two values: *next* and *level*. Each record is first hashed using hash function h_{level}. If the value returned is less than *next*, then the function $h_{level+1}$ is used to compute the hash value. For insertion, the record is inserted to the corresponding bucket. Overflow page is allocated if necessary. For deletion, the corresponding bucket is searched. If the record is found, then it's deleted. If an empty overflow page occurs, it's deallocated.

A split is triggered when an overflow page is allocated. The *next* bucket is split into two buckets and the records in that bucket are redistributed to two buckets. The value of *next* is incremented by 1 after the split.

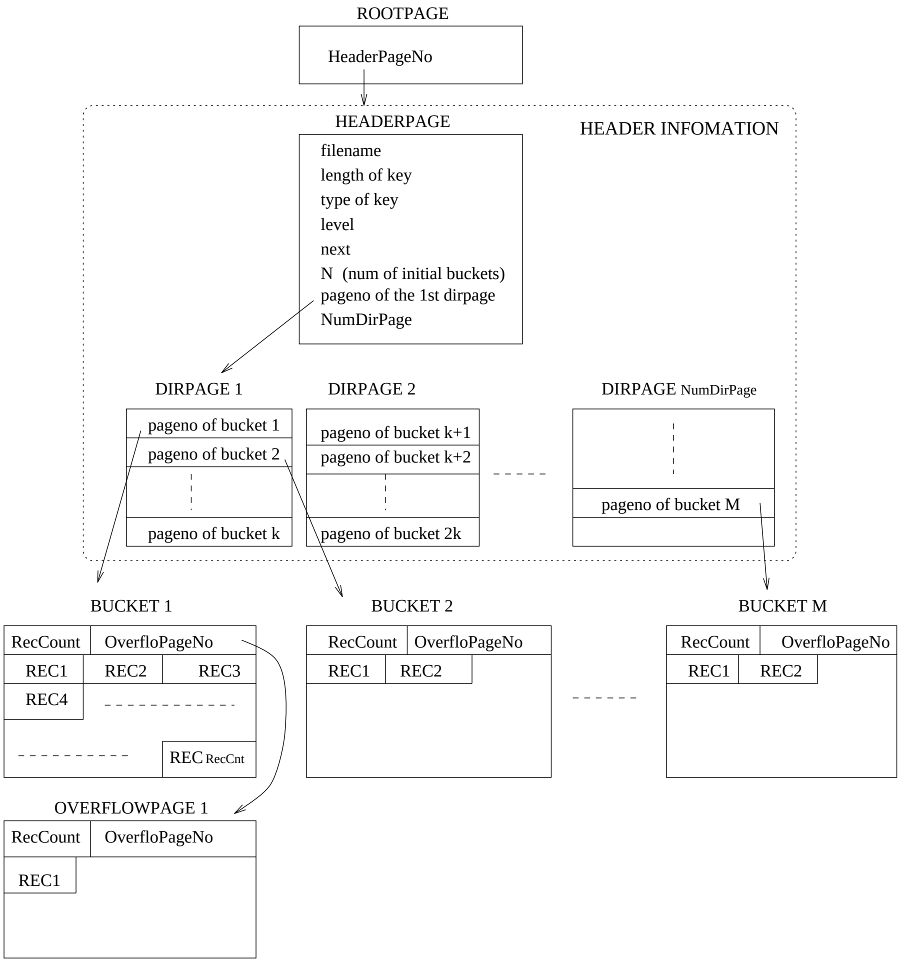

Figure 1: Data Structures of Linear Hashing

If the number of original buckets has been doubled, we start a new round with *level* incremented by 1 and *next* set to 0.

A merge is triggered when an overflow page is deallocated. The last bucket is merged with its image bucket. If The value of *next* is not equal to 1, then it's decremented by 1 after the merge. Otherwise, the value of *level* is decremented by 1 and *next* is set to the last bucket.

3.2 Data Structure

Figure 1 shows an overview of the structure of linear hashing. Since Buffer Manager only supports the handling of a page, the basic storage unit in linear hashing is a page. However, there are several different kinds of pages in linear hashing:

BucketPage (see class *Bucket*)

BucketPage is used to store records. Each BucketPage contains *RecCnt* as the count of how many records are in it, *OverflowPageNo* as the PageId of the overflow page (after it). If no overflow page exists, the number is set to -1. The remaining space is slotted to store the records. (We assume all records are of the same length.) Records are stored one after another. As records are deleted, the resulting hole will be compacted. See the description of deletion for details.

DirPage
: DirPage is used to store the PageId's of the primary pages of buckets. The internal structure of DirPage is very simple – array of PageId's. The PageId's of all the DirPage's are consecutive. See the description of directory handling for details.

HeaderPage (see class *LHHeaderPage*)
: HeaderPage is used to store header information about the index. The information includes:

- FileName: The index's name.
- length: The length of the key
- type: The AttrType of the key.
- level: The current level of linear hashing.
- next: The number of the bucket to be split next.
- N: The number of initial buckets.
- first_dir_pageno: The PageId of the first DirPage.
- NumDirPage: The number of DirPage's.

RootPage (see class *LHRootPage*)
: RootPage is the root of the whole index. It only contains the PageId of the HeaderPage. When the index is created, its PageId is given to *db* to associate it with the name of the index. When the index is opened, *db* looks up the index's name and returns the PageId of the RootPage. Given the RootPage, we can access the HeaderPage, and therefore all the pages in the index. One might have noticed that the HeaderPage can be considered as the root of the index and RootPage is not needed at all (because it's only used to find the HeaderPage). Indeed RootPage is used only for the purpose of concurrency control. See the description of concurrency control for details.

3.3 Create/Open index

When the index is created, the user is required to supply the name of the index, the length and AttrType of the key and the number of initial buckets. (Its default value is 1.) Then a new page is created (by Buffer Manager) as the RootPage and its PageId is given to db to set the file entry for the index. Then HeaderPage, DirPages and initial buckets are created and initialized.

When the index is opened, the user is required to supply the name of the index. Then the name is passed to db to get the PageId of RootPage. The RootPage is read into the buffer pool. The page number of HeaderPage is on the RootPage and from header page we can access all the pages in the index.

3.4 Destroy index

When an index is destroyed, the whole index is traversed from top to the bottom and all pages are destroyed. After that, the *db* will be told to delete the file entry and a *destroyed* flag in the index object is set to true so that any subsequent operations on this index through this index object will be denied.

3.5 Directory handling

Every bucket's primary page's PageId is stored in some DirPage. The PageId's of all the DirPage's are consecutive thanks to the method *BufMgr::newPage* which can allocate a run of pages. When the index is created, the number of the DirPage's initially needed is computed and a run of that number of pages is allocated. The PageId of the first page of this run is stored in the header page. As the file gets expanded, the directory may become full. Then its capacity is doubled by allocating a new run of pages which has twice number of pages of the old directory. All the information in the old directory is copied to the new directory and the old directory is destroyed. Conversely, if only half of the directory is used, then the capacity is reduced by one half in the similar way. The directory looks like a dynamic array in this way. The advantage of this design is that we only need to pin the DirPage that's used and only the PageId of the first DirPage needs to be stored in the header page. Since the expansion and contraction of directory rarely happen, the overhead is very limited.

3.6 Insert records

When a record is inserted, it's first hashed to a bucket number. The primary page of that bucket is brought in and the overflow page number on that page is checked. If there is an overflow page, it's brought in and the overflow page is checked again until the last page is brought in. Since all bucket pages are compact, the record will be inserted to the last slot on the last page. However, if the last page is full, then a new page is allocated to store the new record and a split is triggered.

When a split is triggered, two new bucket pages are allocated. All the records in the *next* bucket are redistributed to the 2 new buckets and overflow pages are allocated if necessary. Then the old bucket is deallocated and the corresponding header information is updated.

3.7 Delete records

When a record is deleted, it's first hashed to a bucket number. Then all the pages in that bucket is searched for this record. If it's not found, RECNOTFOUND is returned. If it's found, then the last record on the last page of this bucket is moved to the deleted record's slot to keep all pages compact. If the last page is an overflow page and it's empty as the result of the deletion, a merge is triggered.

When a merge is triggered, all the records in the last bucket is moved to the end of the image bucket of the last bucket. Overflow pages are allocated if necessary. Then the last bucket is deallocated and the corresponding header information is updated.

3.8 Open an exact match scan

When a scan on some **LinearHashingFile** object is opened, the key to be scanned is passed to the constructor for **LinearHashingScan**. A new object of **LinearHashingScan** is returned. The constructor for **LinearHashingScan** stores the **LinearHashingFile** object and the key. It then initializes its private member *first_scan* to 1.

3.9 Get the next record satisfies the scan

If it's the first time the function is called ($first\_scan = 1$), the *key* is hashed to a bucket number and that bucket will be scanned. *current_pageno* is set to the PageId of the primary page of this bucket. *current_slotno* is set to -1. Then all the pages are scanned until a matched record is found. During the scan, the values

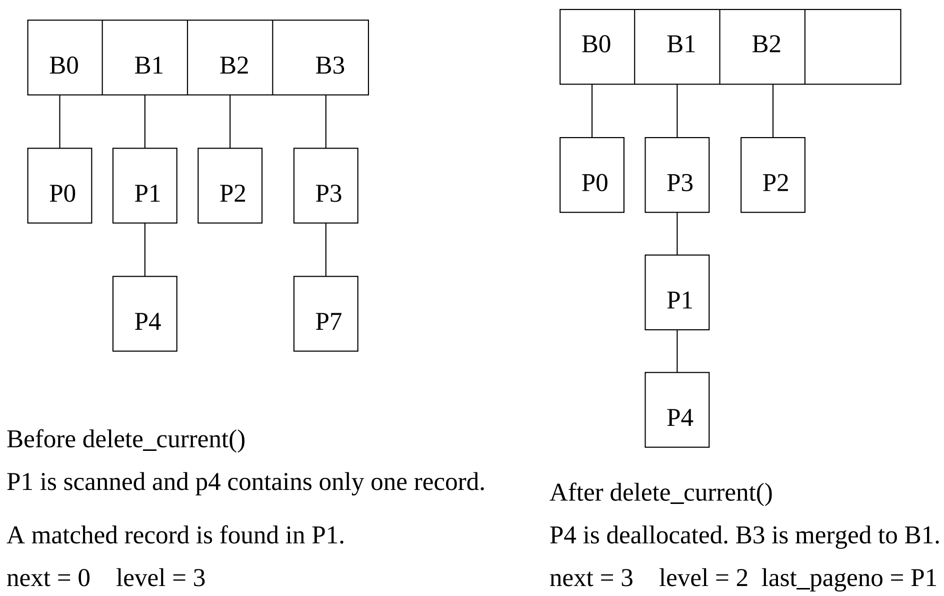

Figure 2: Case 1 in delete_current()

of *current_pageno* and *current_slotno* are updated. Also the PageId of the previously scanned page (if existing) is stored in case some *delete_current()* call later deallocates the current page. If the end of the last page is reached and no matched record is found, NOMORERECS is returned. Another special case when NOMORERECS can be returned is that the page number of the overflow page that's going to be read is equal to *last_pageno*, a private member of the class **LinearHashingScan**. See the description of *delete_current()* for details.

3.10 Delete the currently scanned record

Since the position of the record to be deleted is known, we can just move the last record on the last page to the current slot and decrement *current_slotno* by 1. If an overflow page is deallocated because of the deletion, a merge is triggered. If the bucket to be merged (the last bucket) is neither the bucket we are scanning nor its image, then the normal merge operation described previously can be used here without any modification. Otherwise, if we still use the same merge operation, the values of *current_pageno* and *current_slotno* could be changed due to records movement. In order to minimize the cost and complexity, we designed a special scheme which is illustrated in the following two cases:

- Case 1: the image of the scanned bucket is the last bucket. Since a merge is triggered only when an overflow page is deallocated and we know it's triggered this time after a currently scanned record (in the scanned bucket) is deleted, we can deduce that the scanned bucket consists of some number of **full** pages. So we can just concatenate all the pages of its image bucket (i.e. the last bucket) to its last page and only the *OverflowPageNo* on the last page needs to be changed to the PageId of the first page of its image bucket. If its image bucket has only one empty page, nothing needs to be changed. The whole process is illustrated in Figure 2.
- Case 2: the scanned bucket is the last bucket. Again, (as in case 1,) we know that the scanned bucket consists of some number of **full** pages. We can concatenate all the pages of the image bucket of the

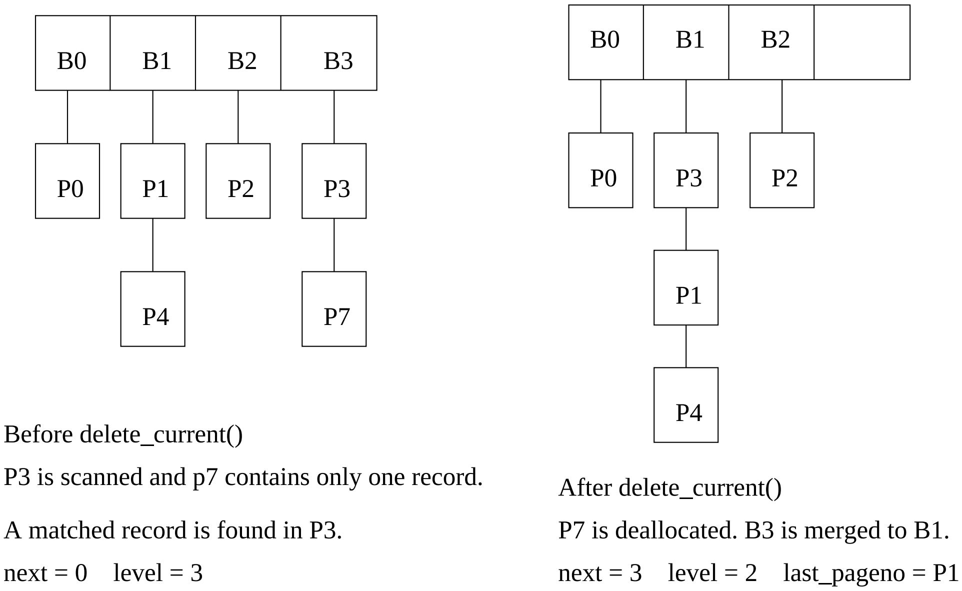

Figure 3: Case 2 in delete_current()

scanned bucket to the last page of the scanned bucket as in case 1. But this time the scanned bucket is to be destroyed, Hence we change the PageId of the primary page of the image bucket (on some DirPage) to the PageId of the scanned bucket to keep the whole structure consistent. The whole process is illustrated in Figure 3.

Note that after these operations, all bucket pages are still compact. Besides, the vales of *current_pageno* and *current_slotno* are unchanged. One drawback of this scheme is that the continuing scan after the merge will then scan all the pages concatenated to the last page of the scanned bucket where no matched records can be found since they belonged to the image bucket of the scanned bucket before the merge. In order to improve the performance, we set a member of the **LinearHashingScan** object, *last_pageno*, to the PageId of the first page concatenated to the scanned bucket. When the scan reads this page, it knows that it's the end of the scan and NOMORERECS is returned. (See Figure 2 and Figure 3.) The above reasoning is based on the fact that when a scan is opened, no update operation is allowed except *delete_current()*.

3.11 Concurrency Control

In MINIREL, two phase locking is used. For special file structure, e.g. linear hashing, special protocol (at index level) is needed to achieve high concurrency.

To make life easier, we consider the whole linear hashing index as a sort of 2-level "tree" structure. The upper level contains *all* the header information including the directory. (See Figure 1.) Useful header information includes *level*, *next* and the whole directory. The information will not be changed by any reader, but could be changed by some updaters. The lower level is individual bucket. Within one bucket, if overflow pages exist, it can be viewed as a sub-tree. For different operations, different protocols are used:

Opening index: when the index is opened, the root page is first locked in Intentional Shared mode to prevent the index from being destroyed. (See the discussion about destroying index for details.) Then a Shared lock on the header page is required before the header page is read to get the header information. After that, the lock on the header page is released. The Intentional Shared lock on the root page is not released until the close of the index.

Creating index: when the index is created, the function *DB::add_file_entry* will be called to add the new file entry. Since *DB* will hold the Exclusive lock on the file entry until the end of transaction, no other transactions can access this index during the transaction. Hence no locking (even for any subsequent operations) is needed at all. For uniformity purpose, the root page is still locked in Intentional Shared mode.

Reader: In linear hashing, reader is equal to the *get_next()* method of **LinearHashingScan**. Reader first requires a Shared lock on header page to read *all* the header information. If it's successful , the header page is read and the PageId of the primary page of the bucket it wants to scan is found. Then a Shared lock on that bucket's primary page is required before the Shared lock on the header page is released. If it's successful, the primary page of the bucket is read in and the page is searched for matched record. Before reading any overflow page (if existing), a Shared lock on that page must be required before the Shared lock on the previous page can be *possibly* released. For each scan, there is a flag called *read* which indicates whether there has been any matched record on the currently scanned page. After a page has been scanned, if the flag is true, the Shared lock cannot be released (because of two phase locking protocol). Otherwise it's released.

Updater: In linear hashing, updater includes *insert*, *delete* and *delete_current*. Updaters will first proceed down the index as if they were readers (using the above protocol for readers) until they get to the page they are going to modify. An Exclusive lock on that page is required. If the update is safe, i.e. it will not trigger any split or merge and hence it will not modify any header information, then the update will proceed and the Exclusive lock is held until the end of the transaction. On the other hand, if the update is not safe, the updater is re-tried and this time an Exclusive lock on the header page is required before it goes down to the bucket to make sure anyone else cannot read any piece of header information during the update. Then it will only require Shared lock on each page of the bucket before reading it. Again, (Read) lock coupling is used when proceeding down the chain of one bucket. If it's found out that the current page is going to be modified, an Exclusive lock is required on the page. It should be pointed out that if the updater finds out that the update is safe while it's holding an Exclusive lock on the header page, the Exclusive lock is released right away to get higher concurrency.

Destroyer: Destroying an index is different from updating it. If someone opens an index, (s)he expects some operation can be done on this index. Therefore the index cannot be destroyed before all other users have closed the index. If we only had a header page without the preceding root page, we would not be able to tell a destroyer from an updater given the locking modes we have. That's the reason why we introduce the root page. Since it's locked in Intentional Read mode when an index is opened, no destroy is allowed before all other users have closed the index. For updater however, only Exclusive (or Shared) lock on header page is required.

3.12 Recovery

Since physical logging is used, it's straightforward to implement writing log. Basically, after every modification, an update log needs to be written. The method *RecoveryMgr::WriteUpdateLog* is used to accomplish this.

4 Accomplishments

Before I started this project, I read the code of the extensible hashing component in the old MINIREL. Since linear hashing's data structures are very different from extensible hashing's, I designed and implemented the system almost from the scratch.

The linear hashing component currently consists of approximately 2300 lines of fully documented C++ code. The single user version has been extensively tested and is working stably. It's also purified and no purify errors were reported in this component. I also finished writing update logs and controlling concurrent accesses to the index. Since the Recovery Manager and Lock Manager have not finished when the report was written, no mutil-user test has been run on this component. Currently I just set all the methods of those two components to empty functions.

For me the difficult parts in this project include extremely rare cases handling, e.g. merging in scan, and locking.

5 Testing

I wrote a program "driver" to test the public interfaces of linear hashing. It tested creating/opening an index, inserts, deletes, destroying the index, opening a new scan, getting the next record and deleting the currently scanned record. It also demonstrates how to use the public interfaces provided by my component.

In order to test some rare cases, I also wrote a program "test" to make sure my program works well in those cases. Also purify was used to detect memory errors and no errors were found in my component.

6 Retrospective

The condition to trigger a split or merge in the current design is mainly associated with allocation or deallocation of overflow pages. I tested the performance and found out that 40% of the pages are overflow pages. This means on average, a record access needs 1.2 disk I/O. The result is the same as that in [3]. However, we could also try to use the overall capacity of the index as the trigger condition. I wish I had time to implement this.

The current design of the *page* class is very elegant in the sense that it eliminates C++ derived-class hierarchy and it's used very conveniently by access methods, e.g. any object (bucket page, etc) can be casted to a *Page*. But when I was adding code to write update logs, it's really a pain to find out the PageId of each "page" since it's not associated with its *Page* object.

There is a method *lock_manager_def::unlockall()* which can unlock all pages locked by a process. It can be used in deadlock situation to release all the locks a process is holding. Hence at access method level, if a deadlock is detected, it's not necessary to release all the locks it has required. I think it's better if we also have a similar method (of BufMgr) to unpin all the pages a process pinned.

One last point, since the whole projet cannot be divided into parallel developments, we should have started the project earlier, especially for lower level groups, to make sure the whole project can work at the end of the semester.

7 Appendix

In the directory */afs/cs.wisc.edu/u/grad/whuang/cs764* you should be able to find my code for linear hashing.

References

[1] Litwin, W. Linear Hashing : A New Tool for File and Table Addressing. IEEE 1980.

[2] Larson, P.-A. Linear Hashing with Separators - A Dynamic Hashing Scheme Achieving One-Access Retrieval. ACM Transactions on Database Systems, Vol. 13, No. 3. September 1988.

[3] Ramakrishnan, R. Relational Databases: Design and Implementation. Course Notes of CS564. Chapter 4. Nov. 1994.

[4] Mohan,C. ARIES/LHS: A Concurrency Control and Recovery Method Using Write-Ahead Logging for Linear Hashing with Separators. IBM Research Report, IBM Almaden Research Center. Nov. 1990.