Lab2 report

521120910245 严冬

1 BufferPool

The `BufferPool` is responsible for managing the reading and writing of pages into memory from disk, as well as handling page locking for concurrency control.

The overall structure of the code is as follows:

1. **Class Definition**: The code defines a class named `BufferPool`.
2. **Class Members**: The class has several instance variables, including `maxNumPages` (the maximum number of pages that can be cached in the buffer pool), `pageMap` (a map that stores the pages in the buffer pool), and `LRUCount` (a map that keeps track of the least recently used pages for eviction purposes).
3. **Constructor**: The constructor `BufferPool(int numPages)` initializes the `maxNumPages` variable and creates empty `HashMap` instances for `pageMap` and `LRUCount`.
4. **Page Retrieval**: The `getPage` method is responsible for retrieving a page from the buffer pool or loading it from disk if it's not already in the buffer pool. If the buffer pool is full, it evicts the least recently used page before loading the requested page.
5. **Transaction Management**: The code includes methods for releasing locks associated with transactions (`releasePage`, `transactionComplete`), checking if a transaction holds a lock on a page (`holdsLock`), and committing or aborting transactions (`transactionComplete` with a boolean argument).
6. **Tuple Operations**: The `insertTuple` and `deleteTuple` methods handle adding and removing tuples from the database, respectively. They acquire necessary locks, update the affected pages, and handle dirty pages by marking them as dirty and adding them to the buffer pool.
7. **Page Flushing**: The `flushAllPages` method flushes all dirty pages to disk, while `flushPage` flushes a specific page to disk.
8. **Page Eviction**: The `evictPage` method is responsible for evicting the least recently used page from the buffer pool when it becomes full. It uses the `LRUCount` map to determine the least recently used page and flushes it to disk before removing it from the buffer pool.
9. **LRU Update**: The `LRUUpdate` method updates the `LRUCount` map for a given page ID, essentially marking it as the most recently used page.
10. **Helper Methods**: The code includes helper methods such as `handleDirtyPages`, which handles marking and adding dirty pages to the buffer pool after tuple operations, and `discardPage`, which removes a specific page from the buffer pool.

1. `BufferPool(int numPages)`: This constructor initializes the `maxNumPages` variable with the provided `numPages` value and creates new `HashMap` instances for `pageMap` and `LRUCount`.
2. `getPage(TransactionId tid, PageId pid, Permissions perm)`: This method first checks if the requested page is already in the `pageMap`. If not, it evicts pages from the buffer pool until there is enough space to load the requested page from disk. It then adds the loaded page to the `pageMap` and updates the `LRUCount` for the page. Finally, it returns the requested page.
3. `insertTuple(TransactionId tid, int tableId, Tuple t)`: This method inserts a new tuple into the specified table. It calls the `insertTuple` method of the corresponding `DatabaseFile` object, which returns a list of dirty pages. The `handleDirtyPages` method is then called to handle these dirty pages.
4. `deleteTuple(TransactionId tid, Tuple t)`: This method deletes a tuple from the database. It calls the `deleteTuple` method of the corresponding `DatabaseFile` object, which returns a list of dirty pages. The `handleDirtyPages` method is then called to handle these dirty pages.
5. `handleDirtyPages(TransactionId tid, List<Page> dirtypages)`: This method iterates over the list of dirty pages. For each dirty page, if the page is not already in the `pageMap` and the buffer pool is full, it evicts a page

from the buffer pool. It then marks the page as dirty with the given transaction ID, adds it to the `pageMap`, and updates the `LRUCount` for the page.
6. `evictPage()`: This method finds the least recently used page in the `LRUCount` map and evicts it from the buffer pool. It first flushes the page to disk using the `flushPage` method, then removes the page from the `pageMap` and `LRUCount` maps.

2 BTreeFile

The `BTreeFile` class is responsible for managing a B+ tree file, which is a file format used to store tuples (rows) of a database table in a sorted order for efficient indexing and searching.

The overall structure of the code is as follows:

1. **Class Definition**: The code defines a class named `BTreeFile` that implements the `DbFile` interface.
2. **Class Members**: The class has several instance variables, including `f` (the file that stores the on-disk backing store for the B+ tree), `td` (the tuple descriptor of tuples in the file), `tableid` (a unique identifier for the table), and `keyField` (the index of the field that the B+ tree is keyed on).
3. **Constructor**: The constructor `BTreeFile(File f, int key, TupleDesc td)` initializes the instance variables based on the provided parameters.
4. **Page Management**: The class provides methods for reading pages from disk (`readPage`), writing pages to disk (`writePage`), and managing the number of pages in the file (`numPages`).
5. **Leaf Page Management**: The class includes methods for finding and manipulating leaf pages, such as `findLeafPage`, `splitLeafPage`, `mergeLeafPages`, and `stealFromLeafPage`. These methods are responsible for handling operations like splitting and merging leaf pages, as well as redistributing tuples between leaf pages.
6. **Internal Page Management**: The class also includes methods for managing internal pages, such as `splitInternalPage`, `mergeInternalPages`, `stealFromLeftInternalPage`, and `stealFromRightInternalPage`. These methods handle operations like splitting and merging internal pages, as well as redistributing entries between internal pages.
7. **Page Eviction and Creation**: The class provides methods for evicting pages from the buffer pool (`evictPage`), getting empty page numbers (`getEmptyPageNo`), creating new empty pages (`getEmptyPage`), and marking pages as empty (`setEmptyPage`).
8. **Tuple Operations**: The class includes methods for inserting and deleting tuples, such as `insertTuple` and `deleteTuple`. These methods handle the necessary locking, page splitting and merging, and updating of parent pointers.
9. **Iterator Support**: The class provides methods for creating iterators over the tuples in the B+ tree file, either for all tuples (`iterator`) or for tuples that match a specific index predicate (`indexIterator`).
10. **Helper Classes**: The code includes two helper classes, `BTreeFileIterator` and `BTreeSearchIterator`, which implement iterators for traversing the tuples in the B+ tree file.

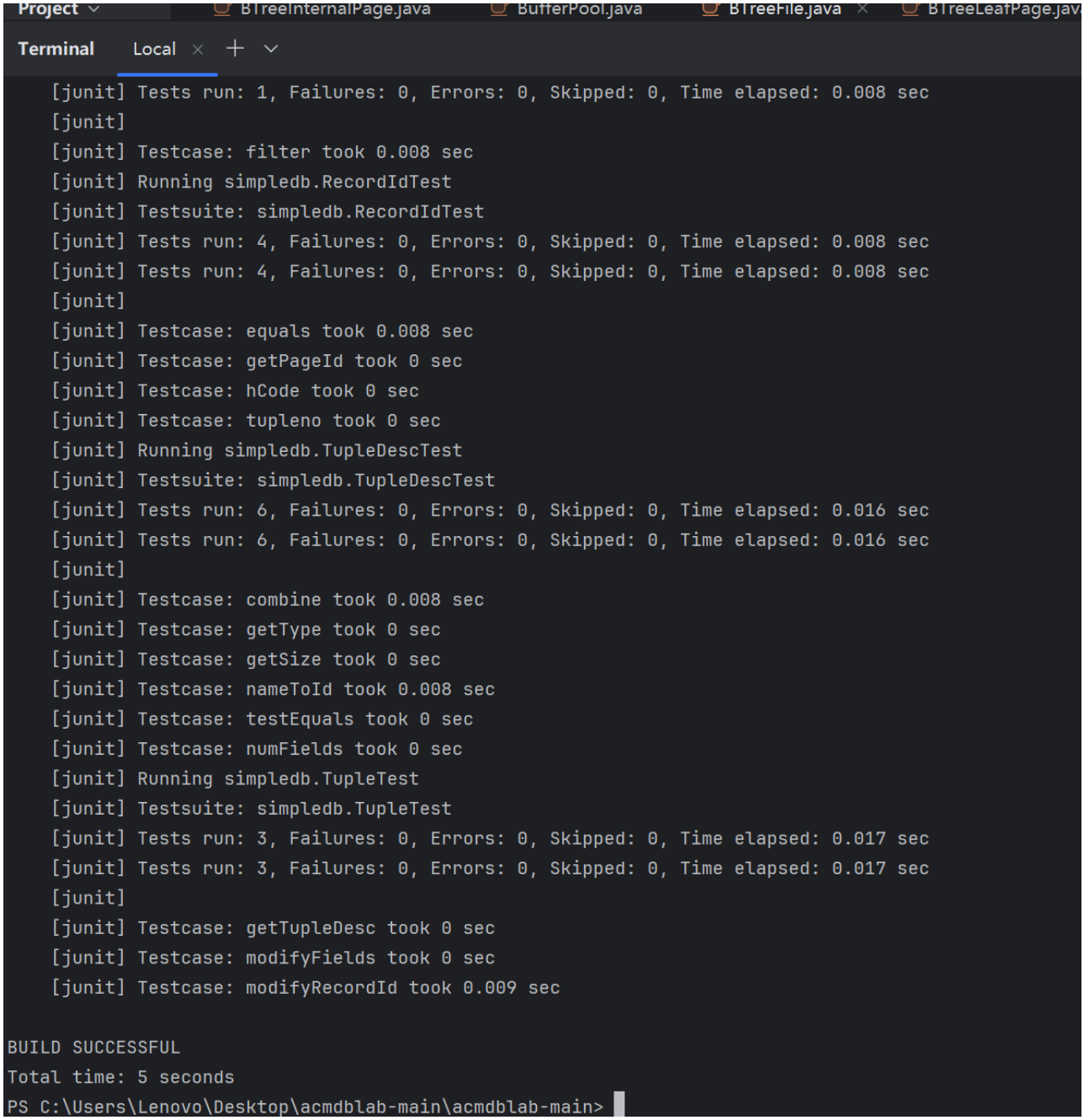
1. `findLeafPage`: This method recursively finds and locks the leaf page in the B+ tree corresponding to the given key field. It locks internal nodes along the path with `READ_ONLY` permission and locks the leaf node with the specified permission.
2. `splitLeafPage`: This method splits a leaf page when it becomes full by creating a new page on the right, moving half of the tuples to the new page, and copying the middle key up into the parent page. It recursively splits the parent page if

necessary.

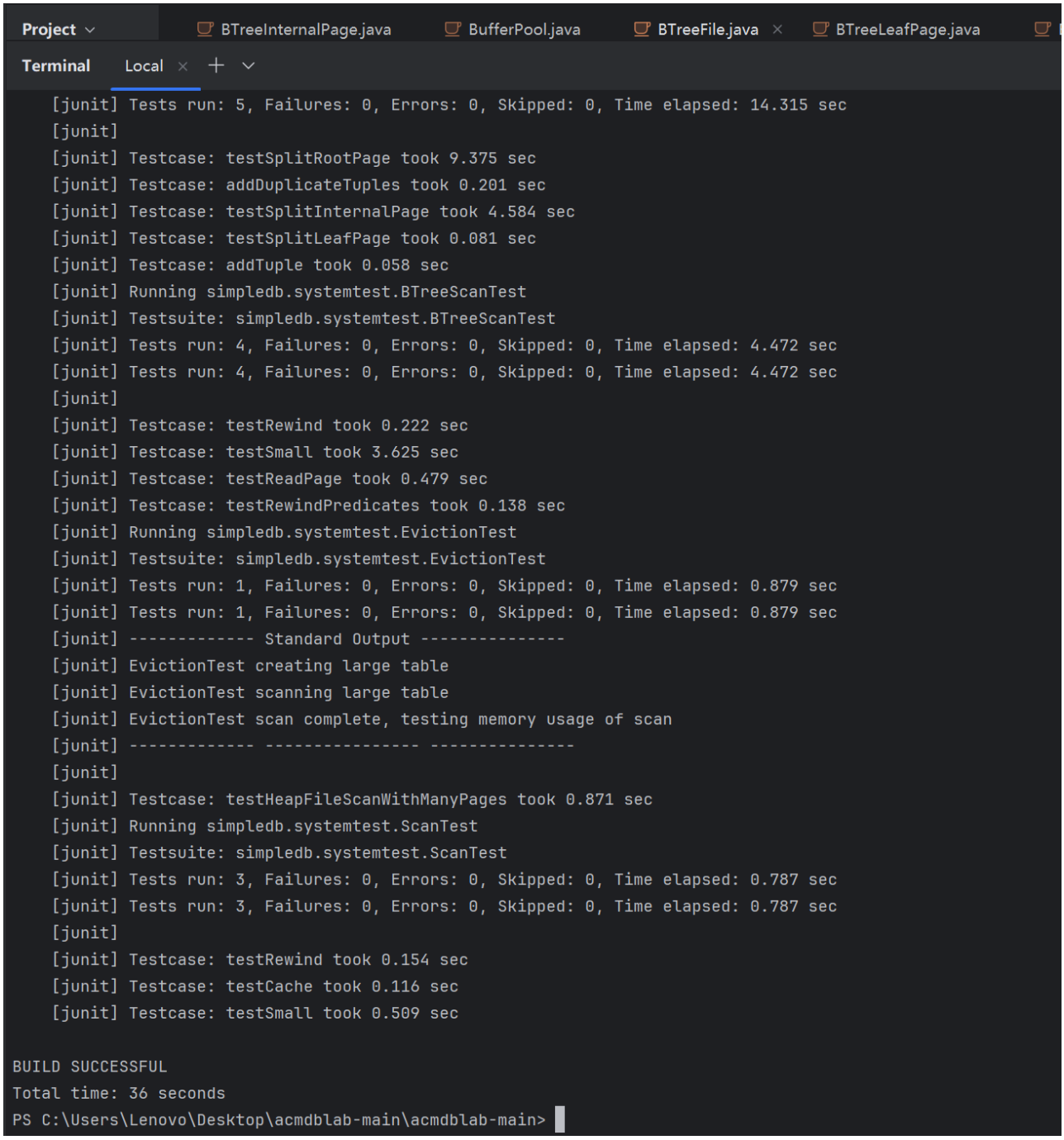
3. `splitInternalPage`: This method splits an internal page when it becomes full by creating a new page, moving half of the entries to the new page, and "pulling down" the corresponding key from the parent entry.
4. `handleMinOccupancyPage`: This method handles the case when a page becomes less than half full due to deletions. It either redistributes tuples/entries from a sibling page or merges with a sibling page.
5. `stealFromLeafPage`: This method redistributes tuples from a sibling leaf page to the given leaf page to ensure that both pages are at least half full.
6. `stealFromLeftInternalPage` and `stealFromRightInternalPage`: These methods redistribute entries from a sibling internal page to the given internal page to ensure that both pages are at least half full.
7. `mergeLeafPages`: This method merges two leaf pages by moving all tuples from the right page to the left page, updating sibling pointers, and making the right page available for reuse.
8. `mergeInternalPages`: This method merges two internal pages by moving all entries from the right page to the left page, updating parent pointers, and making the right page available for reuse.
9. `deleteParentEntry`: This method encapsulates the process of deleting an entry from a parent node, handling the case when the parent becomes empty or falls below minimum occupancy.
10. `getPage`: This method retrieves a page from the buffer pool or loads it from disk if it's not already in the buffer pool, handling dirty pages and locking appropriately.
11. `getRootPtrPage`: This method gets a read lock on the root pointer page and creates the root pointer page and root page if necessary.
12. `getEmptyPageNo`: This method gets the page number of the first empty page in the B+ tree file, creating a new page if none of the existing pages are empty.
13. `getEmptyPage`: This method encapsulates the process of creating a new empty page, reusing old pages if possible, and returning a clean copy locked with read-write permission.
14. `setEmptyPage`: This method marks a page in the B+ tree file as empty by updating the corresponding header page and marking the corresponding slot as empty.

3 Test result

3.1 ant test



3.2 ant systemtest



all passed successful