WU Hoi Fai 20702563

The design of the Information Retrieval System:
1. Indexer
2. Spider

The design principle is to make the index operating at word-level, and the spider operating at page-level.

The spider is the class who deal with web crawling, including which page to be indexed, registering new url and extracting children url. The indexer is the class who deal with building indexes for each page. In simple words, spider extract information from pages, indexer stores these information in an organised way.

Imported libraries:
1. jdbm
2. htmlparser
3. Porter.java from lab3

InvertedIndex's methods:
1. constructor
2. void addPage
3. void addWord
4. void removePage
5. boolean pageUpToDate

1. constructor(RecordManager rm, String stopwords_file)
// Initialize minimum data structures and objects for indexer use
- Use the record manager that the Spider passed in. Both classes share the same record manager.
- Load HTree objects from the database file or creating a new instance if first run.
- Load the stopwords into a hashset for later O(1) checking.

2. void addPage(long pageID, String text, String title, long textsize, long lastModified)
// Add a new page to the index.
- The spider will extract all required information from the page and pass to this method.
- Analyze the text (the HTML file of the page), remove non-alphanumeric characters and break it into word tokens.
- Initialize PageMetadata to store all page information including title, size, last modification time.
- Loop word by word, filter stopwords and use Porter to do stemming.
- for each word,
    - call the private helper function addWord(wordID, pageID, position) which update the inverted index.
    - add the wordID to the pagedata.wordIDList .
- put the pagedata into the forward index with pageID as the key.

3. void addWord(long wordID, long pageID, long position)
// Add a word of a page to the index
- load or create the postings list for this particular wordID.
- load or create the posting for this particular pageID.
- add the updated posting to the postings list.

4. void removePage(long pageID)

// Remove an existing page from the index
- load the wordIDlist of pageID from the PageMetadata object correspond to pageID
- return if there is no such pageID
- for each wordID in the list:
    - delete the corresponding Posting of this pageID from the postings list of that wordID
- remove the PageMetadata object of this pageID

5. boolean pageUpToDate(long pageID, long lastModified)
// Returns true if the index of a page is up to date
- if index of pageID not yet exist, returns false
- if the lastModified date stored in index is smaller than provided, returns false
- otherwise, return true

Database schemas related to indexer:

word <==> wordID translation is handled by a private nested class WordIDTranslator:
1. HTree wordToID: WORD_TO_ID(key: String word, value: Long wordID)
2. HTree IDToWord: ID_TO_WORD(key: Long wordID, value: String word)
- choosing Htree instead of Btree is because sequential access is rare, sorting is also not required

forward index:
1. Htree pageIDToMetadata: PAGEID_METADATA(key: Long pageID, value: PageMetadata)
PageMetadata is a java class implemented Serializable interface, with the following fields:
- String title
- Long size
- Long lastModified
- LinkedList<Long> wordIDList
- choosing Htree instead of Btree is again because of rare sequential access and no need for sorting

backward index:
1. Htree wordIDToPostings: WORDID_POSTINGS(key: Long wordID, value: Long postings_recid)
2. Btree postings: unnamed(key: Long pageID, value: Posting)
Posting is a java class implemented Serializable interface, with the following fields:
- Long pageID
- Long frequency
- LinkedList<Long> positions
- To access a posting, first get its postings list from HTree using key wordID, load the Btree object with the value recid, then get it from Btree with the key pageID.
- The 2 layer design considered relatively small number of words and relatively large number of pages. Later when searching the keywords, the access pattern is mainly random, instead of sequential (unless someone is doing a word range search which is absurd). The postings on the other hand, is often sequentially accessed to return all pages contains the requested word. At the same time, addWord and removePage would frequently access the Postings list in a random manner. Btree is suitable for a achieving quick sequential access and random access at the same time.
- Note that for a small set of pages (e.g. 30), storing each postings list as a Btree would cost high overhead, causing the program running slower than other classmates who implemented the posting list as a simple LinkedList. Yet my design should work better in larger scale.

Spider’s methods:

1. constructor(String starting_url, String dbname, String stopwords_file)

- Initialize required data structure for spider use
- load or create the db file
- add the starting url to the processing queue

2. int crawlPage()
//process an URL from the processing queue.
- dequeue the URL from the processing queue.
- load or assign the page with a pageID.
- call getMetadata(url) to retrieve title, size and lastModified date.
- check whether it is first visited in this execution, if yes, enqueue its child links to the processing queue.
- check whether the index of this page requires update, if no, return 2.
- remove the page from the index
- add the page to the index, with the latest information retrieved
- if any errors and exceptions is thrown during the adding, rollback the changes, and return 3
- if the page is added successfully, return 0 if it is the first visit of this page, otherwise return 1
// to distinguish whether a page is visited multiple times in an execution, is to count the correct number of pages crawled in an execution.

3. void crawlPages(long n)
// crawl and index at most n pages
- while it has not yet indexed n different pages and the processing queue is not empty
- crawl a page
- if that page is indexed successfully, save the changes to disk
- if that page is first visited in this execution, add # of crawled page by 1.

Data schemas related to spider:

URL <==> pageID is managed the private nested class LinkManager
1. Htree URLToPageID: URL_TO_PAGEID(key: String url, value: Long pageID)
2. Htree pageIDToURL: PAGEID_TO_URL(key: Long pageID, value: String url)

Parent ==> Children is managed by LinkManager
1. Htree parentToChildren: PARENT_TO_CHILDREN(key: Long parentID, value: Long pageID)

Cyclic check is also provided by LinkManager
1. Htree cyclicCheck: unnamed(key: Long pageID, value: Boolean true)
- This table will be removed at the end of each execution, i.e. it is intended to be volatile.
- The reason for not implementing it as java.util.HashSet<Long> is to avoid memory overflow when the number of pages is getting large.