US 20110040762A1

(19) United States

(12) Patent Application Publication

Flatland et al.

(10) Pub. No.: US 2011/0040762 A1

(43) Pub. Date: Feb. 17, 2011

(54) SEGMENTING POSTINGS LIST READER

(75) Inventors: **Steinar Flatland**, Clifton Park, NY (US); **Jeff J. Dalton**, Northampton, MA (US)

Correspondence Address:
HESLIN ROTHENBERG FARLEY & MESITI PC
5 COLUMBIA CIRCLE
ALBANY, NY 12203 (US)

(73) Assignee: **GLOBALSPEC, INC.**, East Greenbush, NY (US)

(21) Appl. No.: **12/854,775**

(22) Filed: **Aug. 11, 2010**

Related U.S. Application Data

(60) Provisional application No. 61/233,427, filed on Aug. 12, 2009, provisional application No. 61/233,420, filed on Aug. 12, 2009, provisional application No. 61/233,411, filed on Aug. 12, 2009.

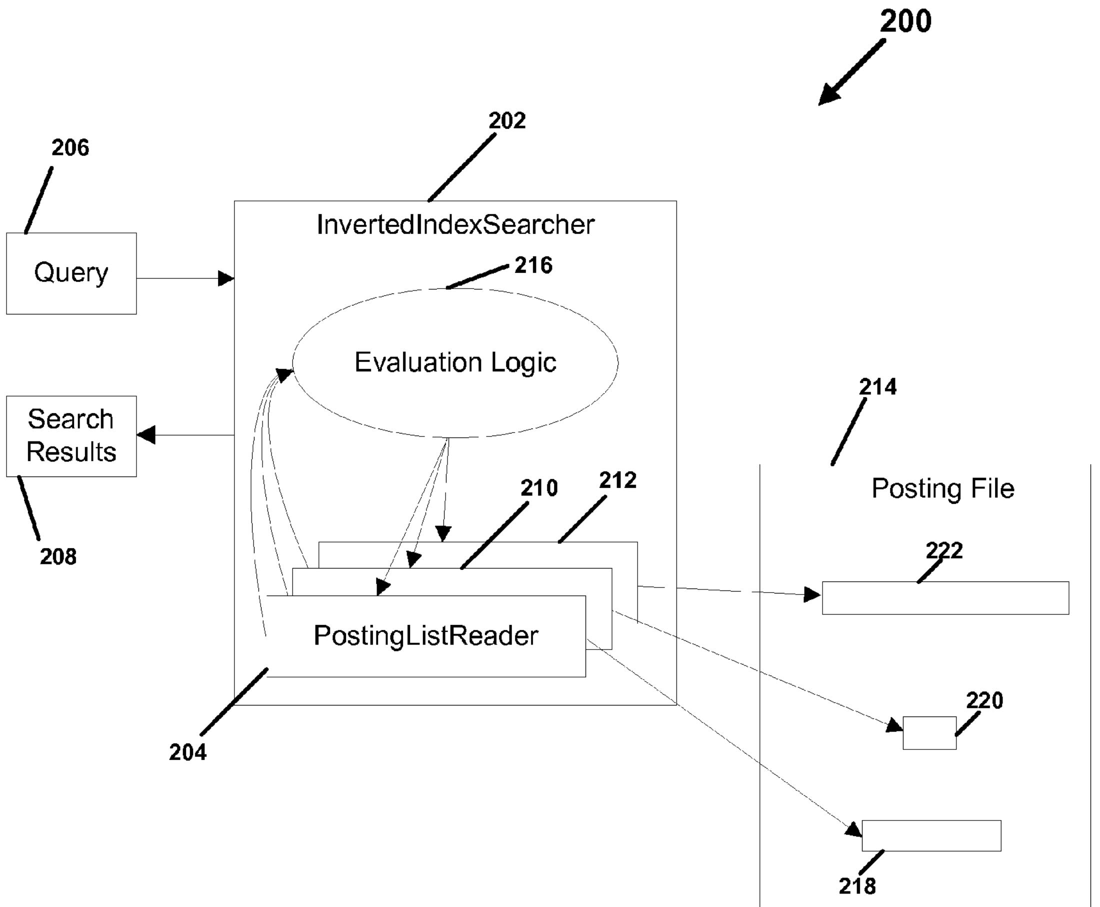
Publication Classification

(51) **Int. Cl.**
G06F 17/30 (2006.01)

(52) **U.S. Cl.** **707/737**; 707/E17.089; 707/749

(57) **ABSTRACT**

A size of a posting list is determined as part of searching an inverted index. The posting list is segmented for reading into a plurality of segments based on the size. For example, the segmenting may be performed if the size is larger than a predetermined size. Finally, each of the plurality of segments is read into memory.



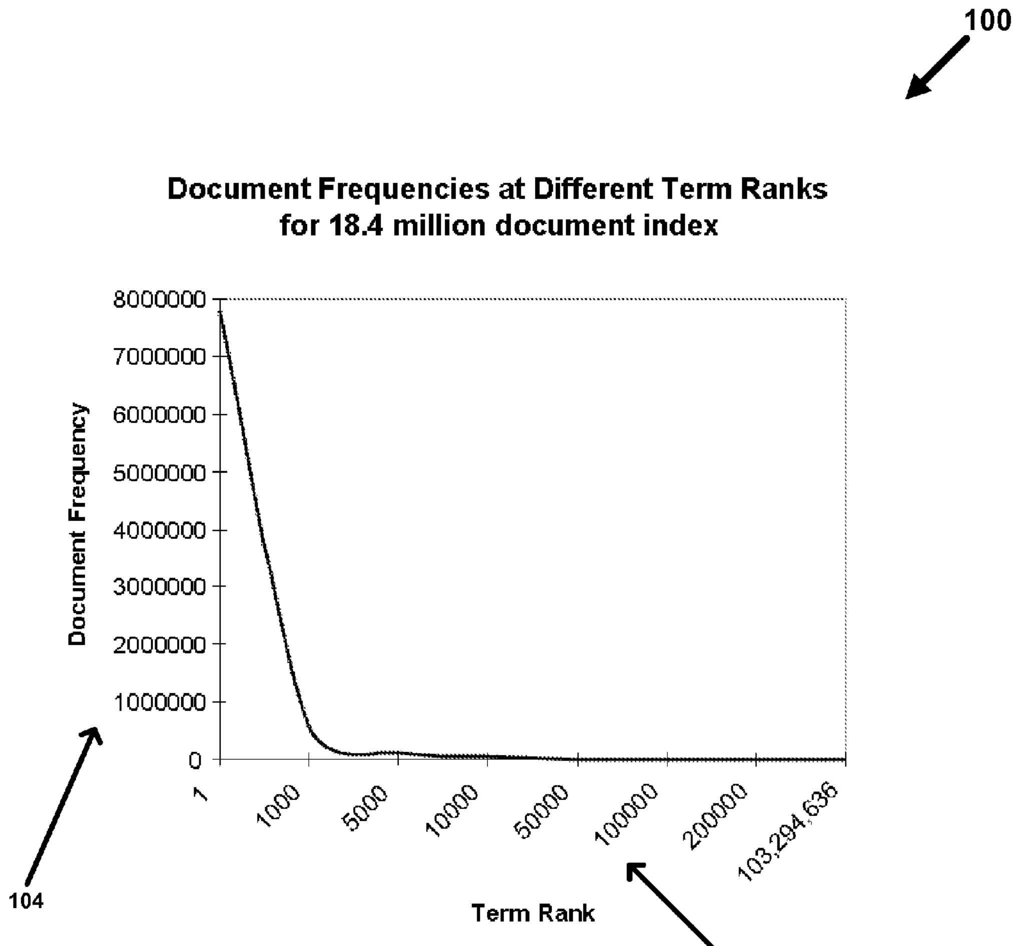

100
Document Frequencies at Different Term Ranks
for 18.4 million document index
Document Frequency
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
1
1000
5000
10000
50000
100000
200000
103,294,636
Term Rank
104
102

FIG. 1

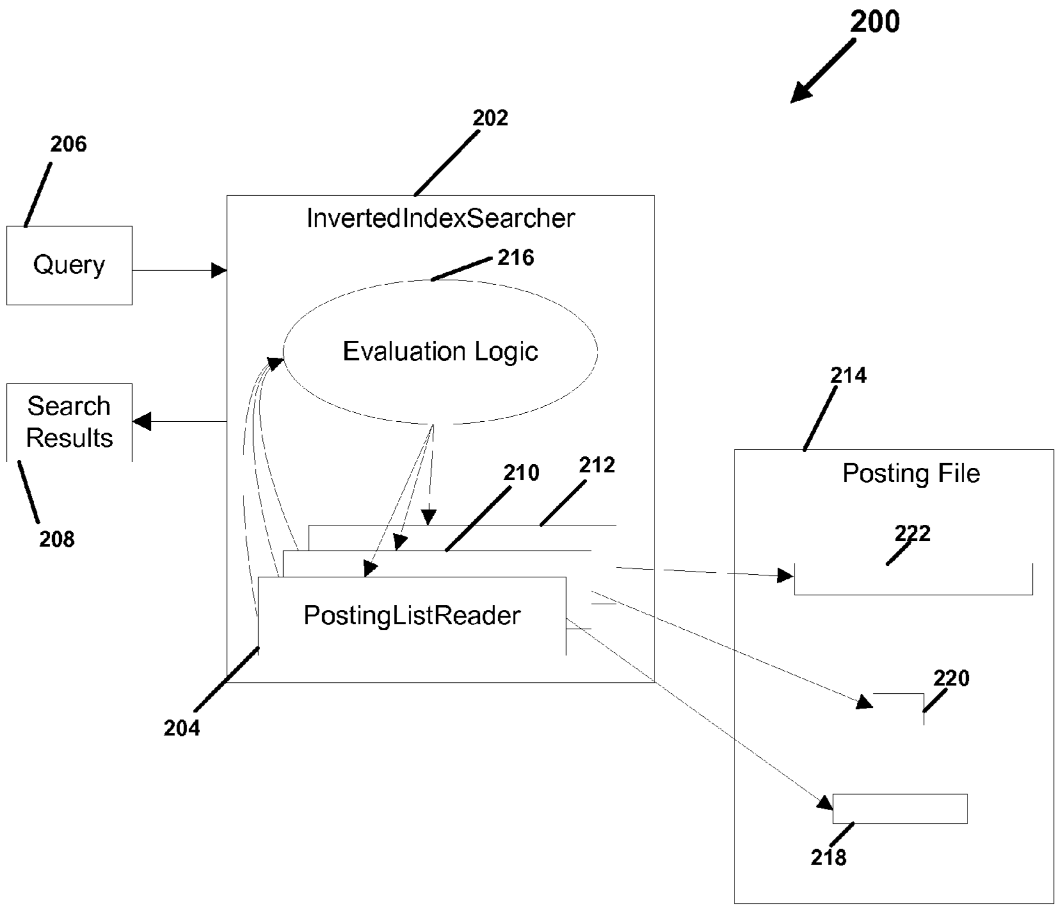

200
202
206
InvertedIndexSearcher
Query
216
Evaluation Logic
Search Results
214
208
210
212
Posting File
222
PostingListReader
220
204
218

FIG. 2

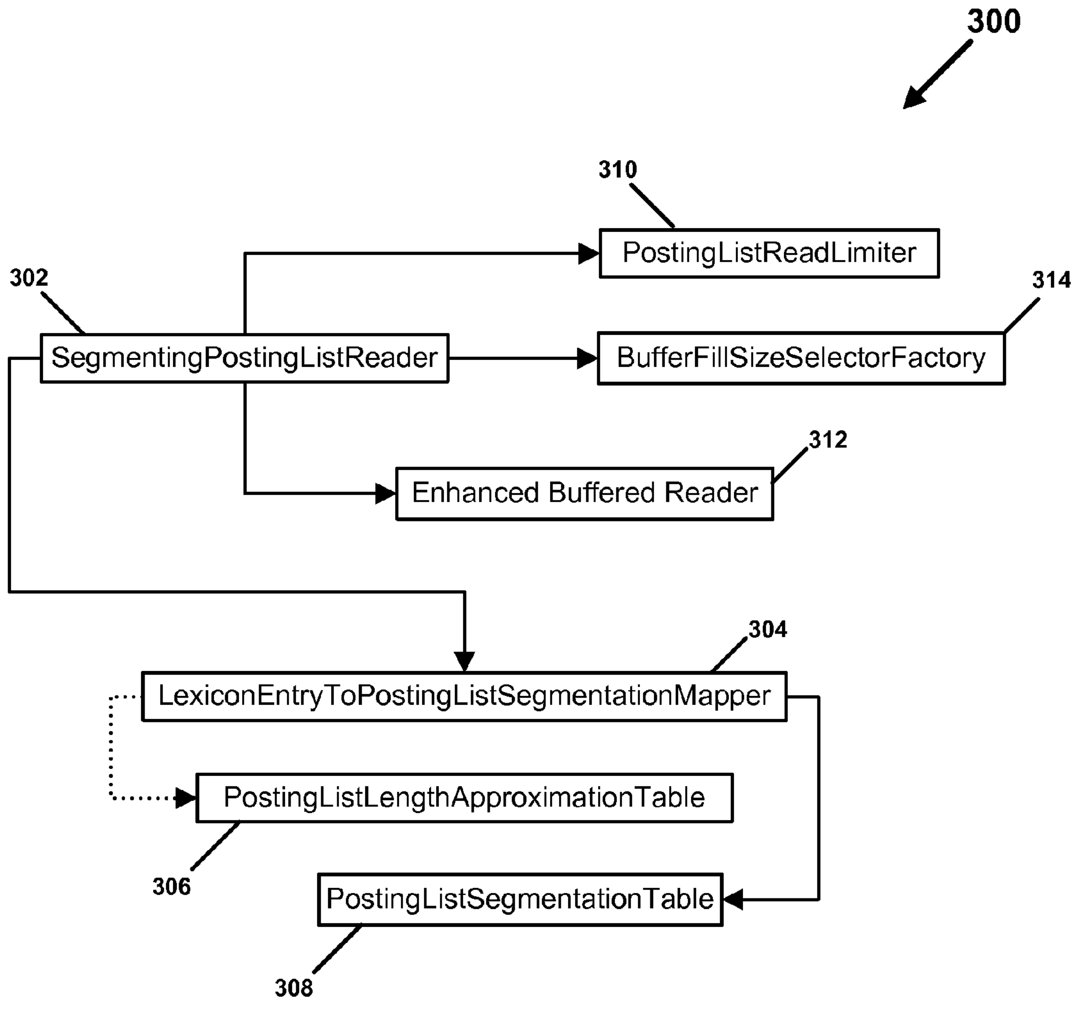

300
310
PostingListReadLimiter
302
SegmentingPostingListReader
314
BufferFillSizeSelectorFactory
312
Enhanced Buffered Reader
304
LexiconEntryToPostingListSegmentationMapper
PostingListLengthApproximationTable
306
PostingListSegmentationTable
308

FIG. 3

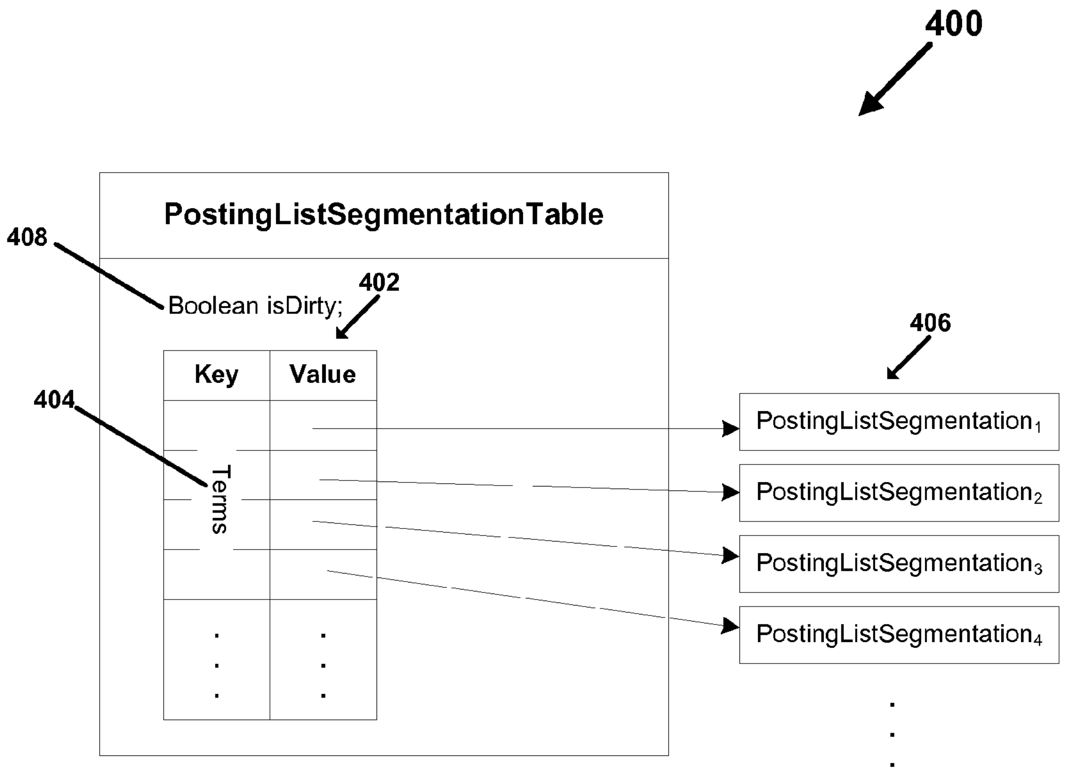

400
PostingListSegmentationTable
408
Boolean isDirty;
402
Key
Value
404
Terms
406
PostingListSegmentation$_1$
PostingListSegmentation$_2$
PostingListSegmentation$_3$
PostingListSegmentation$_4$

FIG. 4

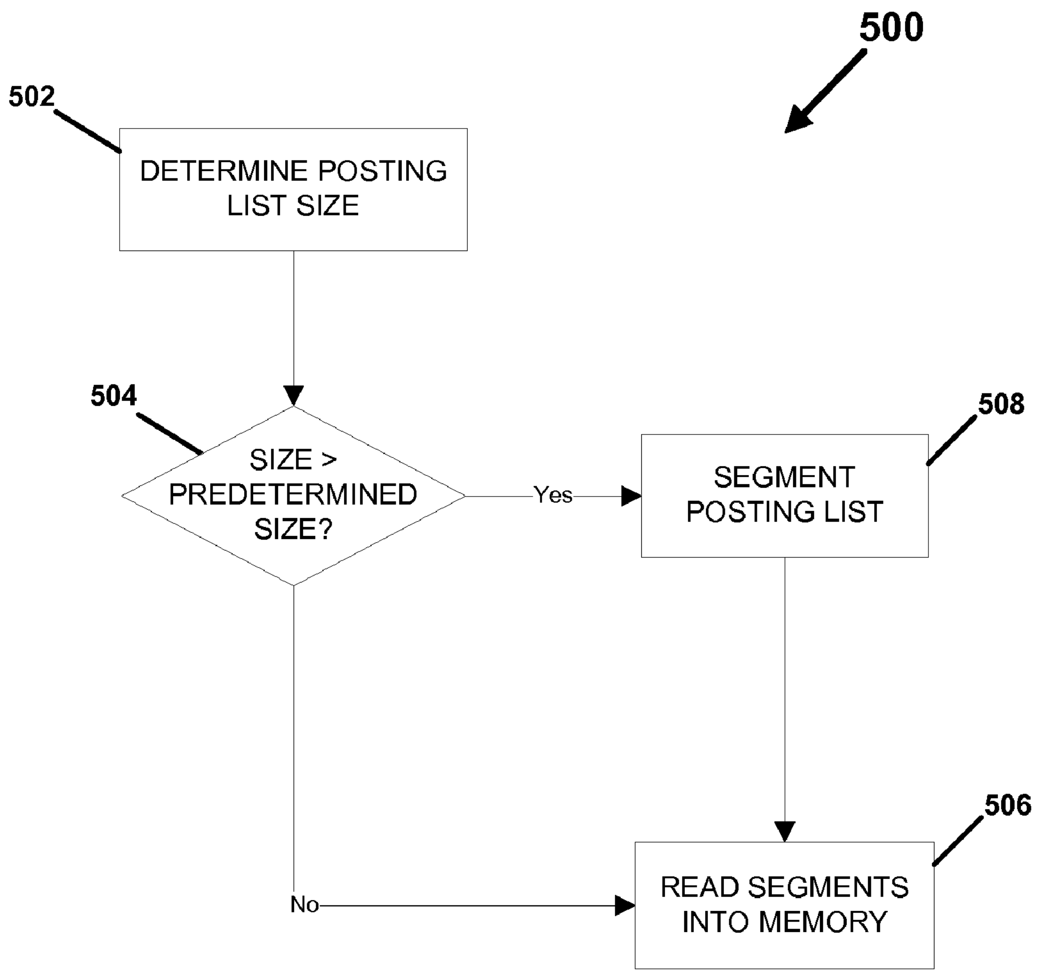

500
502
DETERMINE POSTING LIST SIZE
504
SIZE > PREDETERMINED SIZE?
Yes
508
SEGMENT POSTING LIST
No
506
READ SEGMENTS INTO MEMORY

FIG. 5

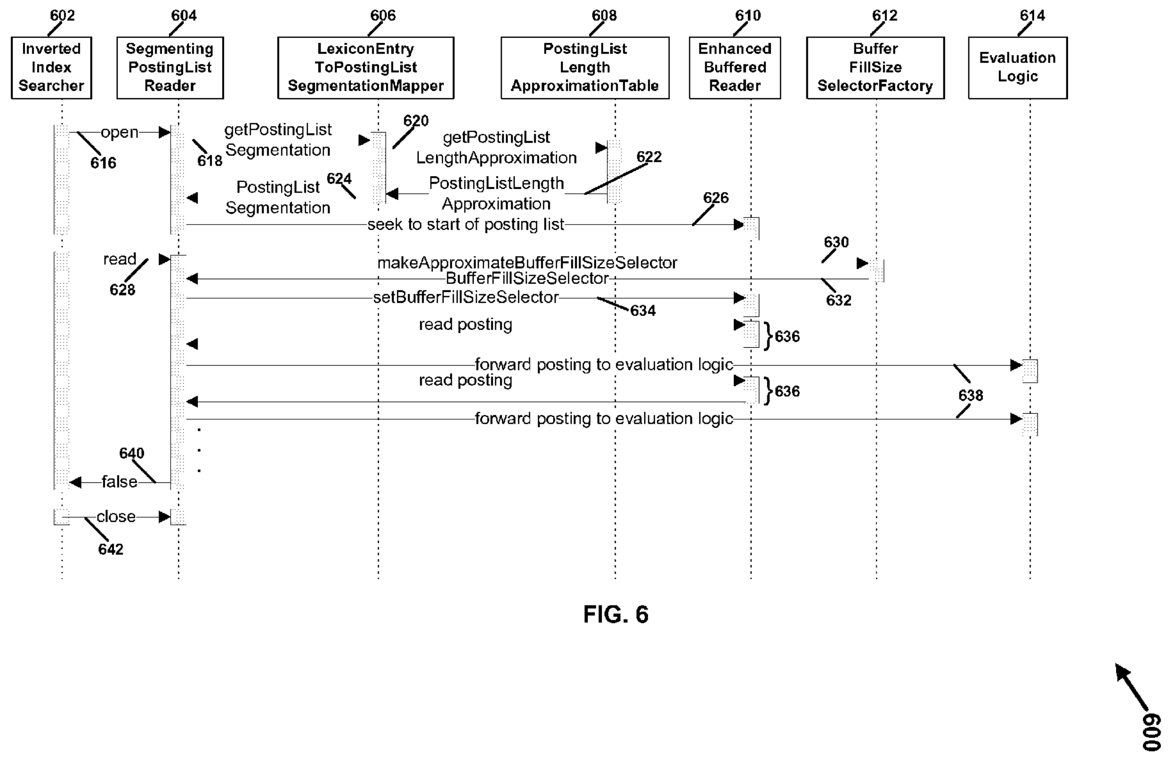

602
Inverted Index Searcher
604
Segmenting PostingList Reader
606
LexiconEntry ToPostingList SegmentationMapper
608
PostingList Length ApproximationTable
610
Enhanced Buffered Reader
612
Buffer FillSize SelectorFactory
614
Evaluation Logic
open
616
618
getPostingList Segmentation
620
getPostingList LengthApproximation
622
PostingList Segmentation
624
PostingListLength Approximation
626
seek to start of posting list
read
628
630
makeApproximateBufferFillSizeSelector
BufferFillSizeSelector
632
setBufferFillSizeSelector
634
read posting
636
forward posting to evaluation logic
read posting
636
638
forward posting to evaluation logic
640
false
close
642
600

FIG. 6

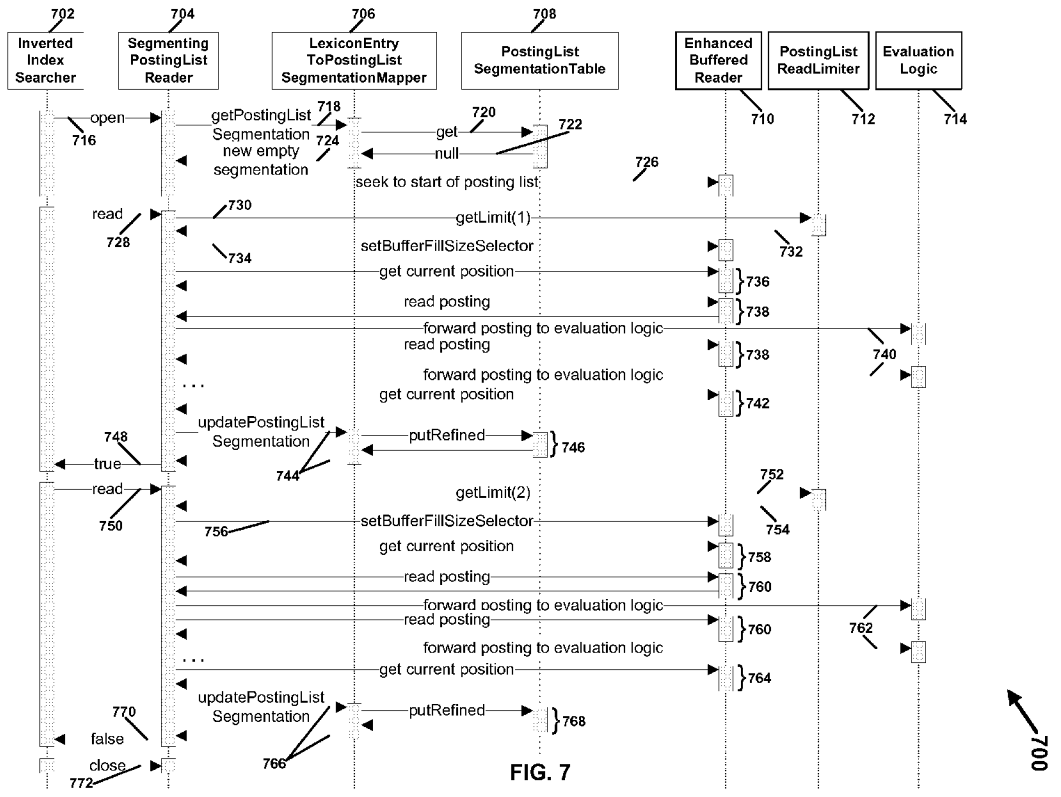

702
Inverted Index Searcher
704
Segmenting PostingList Reader
706
LexiconEntry ToPostingList SegmentationMapper
708
PostingList SegmentationTable
Enhanced Buffered Reader
710
PostingList ReadLimiter
712
Evaluation Logic
714
open
716
getPostingList Segmentation
718
get
720
null
722
new empty segmentation
724
seek to start of posting list
726
read
728
getLimit(1)
730
732
setBufferFillSizeSelector
734
get current position
736
read posting
738
forward posting to evaluation logic
740
. . .
742
updatePostingList Segmentation
744
putRefined
746
true
748
read
750
getLimit(2)
752
754
756
758
760
762
764
766
putRefined
768
false
770
close
772
700

FIG. 7

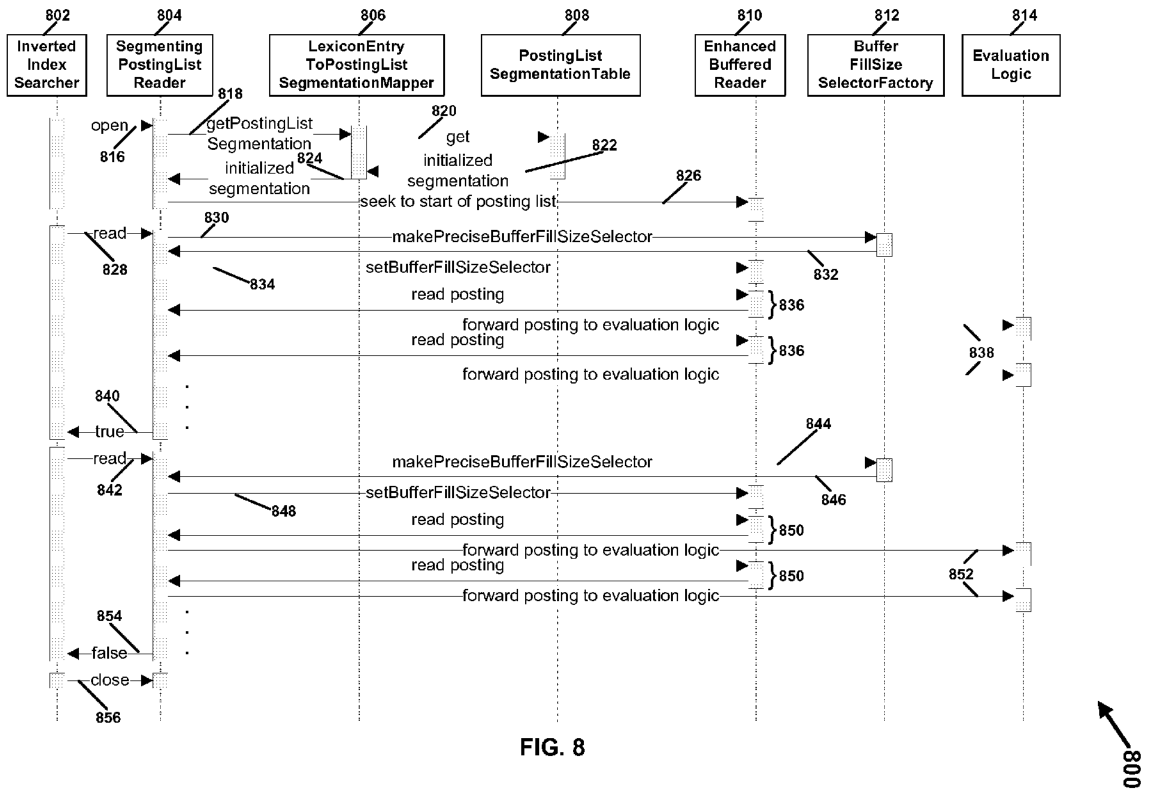

802
Inverted Index Searcher
804
Segmenting PostingList Reader
806
LexiconEntry ToPostingList SegmentationMapper
808
PostingList SegmentationTable
810
Enhanced Buffered Reader
812
Buffer FillSize SelectorFactory
814
Evaluation Logic
open
816
818
getPostingList Segmentation
820
get initialized segmentation
822
824
initialized segmentation
826
seek to start of posting list
read
828
830
makePreciseBufferFillSizeSelector
832
setBufferFillSizeSelector
834
read posting
836
forward posting to evaluation logic
read posting
836
838
forward posting to evaluation logic
840
true
844
read
842
makePreciseBufferFillSizeSelector
846
setBufferFillSizeSelector
848
read posting
850
forward posting to evaluation logic
read posting
850
852
forward posting to evaluation logic
854
false
close
856
800

FIG. 8

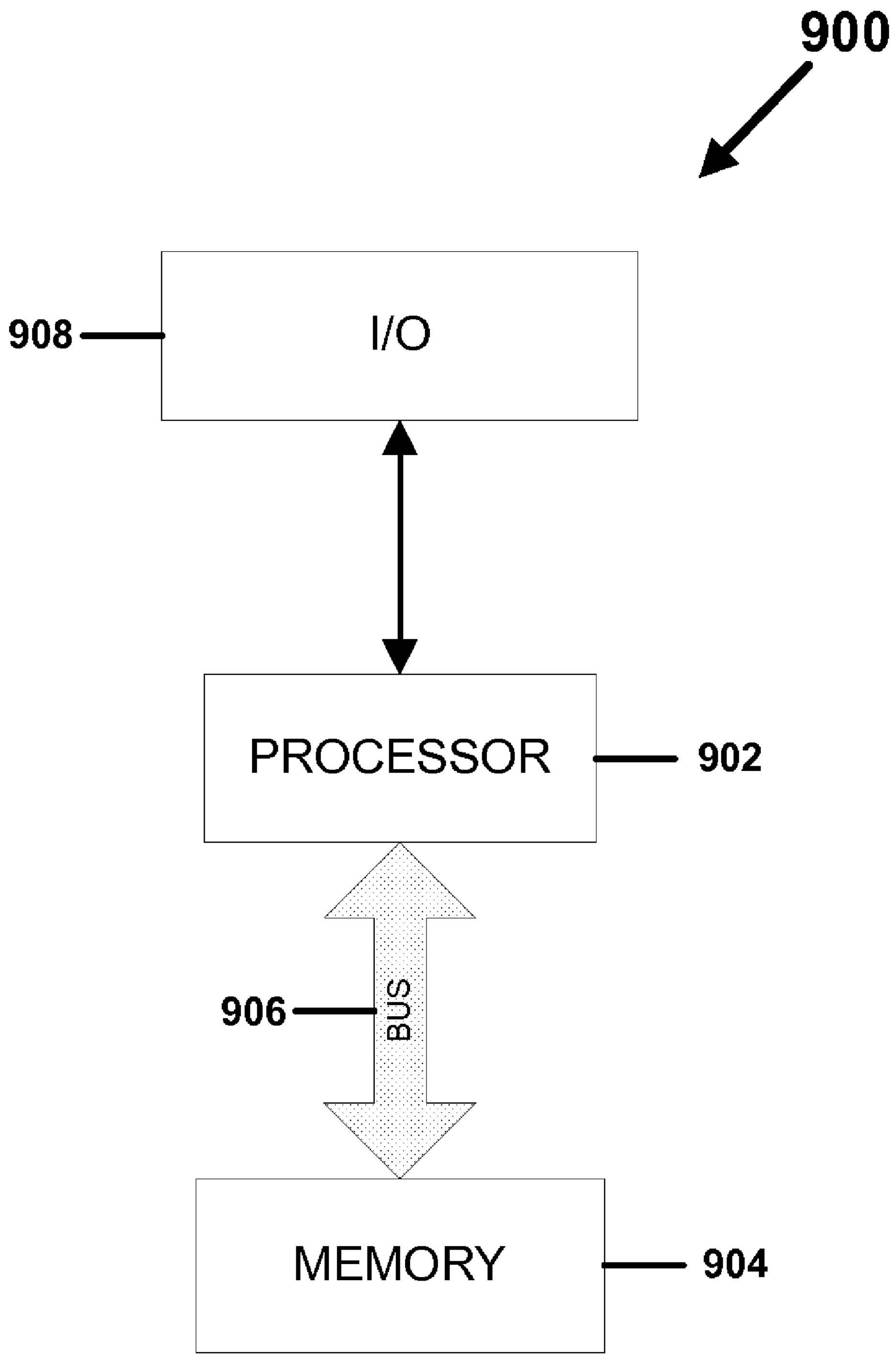


FIG. 9

SEGMENTING POSTINGS LIST READER

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application claims priority under 35 U.S.C. §119 to the following U.S. Provisional Applications, which are herein incorporated by reference in their entirety:

[0002] Provisional Patent Application Ser. No. 61/233,411, by Flatland et al., entitled "ESTIMATION OF POSTINGS LIST LENGTH IN A SEARCH SYSTEM USING AN APPROXIMATION TABLE," filed on Aug. 12, 2009;

[0003] Provisional Patent Application No. 61/233,420, by Flatland et al., entitled "EFFICIENT BUFFERED READING WITH A PLUG IN FOR INPUT BUFFER SIZE DETERMINATION," filed on Aug. 12, 2009; and

[0004] Provisional Patent Application Ser. No. 61/233,427, by Flatland et al., entitled "SEGMENTING POSTINGS LIST READER," filed on Aug. 12, 2009.

[0005] This application contains subject matter which is related to the subject matter of the following applications, each of which is assigned to the same assignee as this application and filed on the same day as this application. Each of the below listed applications is hereby incorporated herein by reference in its entirety:

[0006] U.S. Non-Provisional patent application Ser. No. ______, by Flatland et al., entitled "ESTIMATION OF POSTINGS LIST LENGTH IN A SEARCH SYSTEM USING AN APPROXIMATION TABLE" (Attorney Docket No. 1634.068A); and

[0007] U.S. Non-Provisional patent application Ser. No. ______, by Flatland et al., entitled "EFFICIENT BUFFERED READING WITH A PLUG IN FOR INPUT BUFFER SIZE DETERMINATION" (Attorney Docket No. 1634.069A).

TECHNICAL FIELD

[0008] The present invention generally relates to reading posting lists as part of searching an inverted index. More particularly, the invention relates to segmenting a posting list into a plurality of segments based on the size of the list.

BACKGROUND

[0009] The following definition of Information Retrieval (IR) is from the book *Introduction to Information Retrieval* by Manning, Raghavan and Schutze, Cambridge University Press, 2008:

> **[0010]** Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).

Inverted Index

[0011] An inverted index is a data structure central to the design of numerous modern information retrieval systems. In chapter 5 of *Search Engines: Information Retrieval in Practice* (Addison Wesley, 2010), Croft, Metzler and Strohman observe:

> **[0012]** An inverted index is the computational equivalent of the index found in the back of this textbook . . . . The book index is arranged in alphabetical order by index term. Each index term is followed by a list of pages about the word.

[0013] In a search system implemented using a computer, an inverted index often comprises two related data structures:

> **[0014]** 1. A lexicon contains the distinct set of terms (i.e., with duplicates removed) that occur throughout all the documents of the index. To facilitate rapid searching, terms in the lexicon are usually stored in sorted order. Each term typically includes a document frequency and a pointer into the other major data structure of the inverted index, the posting file. The document frequency is a count of the number of documents in which a term occurs. The document frequency is useful at search time both for prioritizing term processing and as input to scoring algorithms.

> **[0015]** 2. The posting file consists of one posting list per term in the lexicon, recording for each term the set of documents in which the term occurs. Each entry in a posting list is called a posting. The number of postings in a given posting list equals the document frequency of the associated lexicon entry. A posting includes at least a document identifier and may include additional information such as: a count of the number of times the term occurs in the document; a list of term positions within the document where the term occurs; and more generally, scoring information that ascribes some degree of importance (or lack thereof) to the fact that the document contains the term.

[0016] When processing a user's query, a computerized search system needs access to the postings of the terms that describe the user's information need. As part of processing the query, the search system aggregates information from these postings, by document, in an accumulation process that leads to a ranked list of documents to answer the user's query.

[0017] A large inverted index may not fit into a computer's main memory, requiring secondary storage, typically disk storage, to help store the posting file, lexicon, or both. Each separate access to disk may incur seek time on the order of several milliseconds if it is necessary to move the hard drive's read heads, which is very expensive in terms of runtime performance compared to accessing main memory.

[0018] Therefore, it would be helpful to minimize accesses to secondary storage for reading posting lists when searching an inverted index, in order to improve runtime performance.

BRIEF SUMMARY OF INVENTION

[0019] The present invention satisfies the above-noted need by providing a posting list reader that reads a posting list efficiently during inverted index searching by reducing the number of accesses to secondary storage as compared to a traditional buffered reading strategy that repeatedly uses a uniform input buffer size.

[0020] The posting list reader of the present invention will be referred to as a segmenting posting list reader, to distinguish it from posting list readers in general. Further, a posting list segment refers to a sequence of adjacent postings within a posting list. A complete segmentation of a posting list breaks it up into one or more non-overlapping segments that together include all the postings of the list.

[0021] In accordance with the above, it is an object of the present invention to provide a segmenting posting list reader that can determine how many postings to read on each read request.

[0022] It is another object of the present invention to provide a segmenting reader to read short posting lists in a single burst of reading.

[0023] It is still another object of the present invention to provide a segmenting reader that automatically breaks long posting lists into segments according to, for example, a strategy that may vary with the requirements of evaluation logic, posting list organization, or other considerations. Each read request preferably reads the next segment in one burst of reading.

[0024] It is yet another object of the present invention to provide a segmenting reader with support for posting list segments of both exact and approximate size.

[0025] Finally, it is another object of the present invention to provide a segmenting posting list reader that learns, remembers and applies posting list segmentations with only a small amount of up-front configuration.

[0026] The present invention provides, in a first aspect, a method of reading a posting list. The method comprises determining by a processor a size of a posting list as part of searching an inverted index, segmenting the posting list for reading by the processor into a plurality of segments based on the size, and reading by the processor each of the plurality of segments into memory.

[0027] The present invention provides, in a second aspect, a computer system for reading a posting list. The computer system comprises a memory, and a processor in communication with the memory to perform a method. The method comprises determining a size of a posting list as part of searching an inverted index, segmenting the posting list for reading into a plurality of segments based on the size, and reading each of the plurality of segments into memory.

[0028] The present invention provides, in a third aspect, a program product for reading a posting list. The program product comprises a storage medium readable by a processor and storing instructions for execution by the processor for performing a method. The method comprises determining a size of a posting list as part of searching an inverted index, segmenting the posting list for reading into a plurality of segments based on the size, and reading each of the plurality of segments into memory.

[0029] These, and other objects, features and advantages of this invention will become apparent from the following detailed description of the various aspects of the invention taken in conjunction with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

[0030] One or more aspects of the present invention are particularly pointed out and distinctly claimed as examples in the claims at the conclusion of the specification. The foregoing and other objects, features, and advantages of the invention are apparent from the following detailed description taken in conjunction with the accompanying drawings in which:

[0031] FIG. **1** is a graph of term rank versus document frequency;

[0032] FIG. **2** is a block/flow diagram showing aspects of inverted index searching;

[0033] FIG. **3** is one example of a block/flow diagram for a segmenting posting list reader, in accordance with aspects of the present invention;

[0034] FIG. **4** is an instance diagram for a posting list segmentation table and associated objects;

[0035] FIG. **5** is a flow diagram for one example of a method of reading a posting list, in accordance with aspects of the present invention;

[0036] FIG. **6** is sequence diagram for one example of a method of reading a short posting list comprising a single segment;

[0037] FIG. **7** is a sequence diagram for one example of a method of reading and learning the segmentation of a posting list comprising two segments;

[0038] FIG. **8** is a sequence diagram for one example of a method of reading a posting list comprising two segments, taking advantage of segmentation information learned and remembered during an earlier read of the list; and

[0039] FIG. **9** is a block diagram of one example of a computing unit incorporating one or more aspects of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

[0040] Posting lists in a search index are described by Zipf's law, which states that given a corpus of natural language documents, the frequency of any word is inversely proportional to its rank in the frequency table.

[0041] FIG. **1** shows, for an index built from a natural language corpus, a graph **100** of term rank **102** versus document frequency **104**, where document frequency is the number of distinct documents the term occurs in. Another way to think about document frequency is posting list length. The graph shows that most terms have very short posting lists, and only relatively few posting lists are long.

[0042] Observing that queries submitted to a search system are little natural language documents, they too adhere to Zipf's law. It follows that the relatively few long posting lists in a search index are also the most frequently accessed during query processing. An efficient read strategy for long posting lists can help a search system deliver fast query run times. It is convenient that the big posting lists are few. This makes it feasible to craft and hold in memory exact read strategies for these lists.

Inverted Index Searcher and Posting List Reader

[0043] An information retrieval system **200** that searches an inverted index comprises components similar to those labeled InvertedIndexSearcher **202** and PostingListReader **204** in FIG. **2**. An inverted index searcher manages the process of searching an inverted index, and a posting list reader manages the details of reading a posting list from the posting file **214**.

[0044] Inverted index searcher **202** takes a query **206** as input and returns search results **208**. Information contained in the query includes, at a minimum, a term or terms describing the user's information need. The query optionally includes other features such as, for example, Boolean constraints (AND, OR, NOT), term weights, phrase constraints, or proximity restrictions. The query may be expressed literally as submitted by the user, or it may already have been parsed and structured. The search results returned, at a minimum, comprise unique identifiers of the documents matching the query. Often, the search results are returned in order of descending relevance, and each search result may optionally include a variety of other information such as a score, date indexed, document last modified date, a copy of the document as it was indexed, the document's URL if applicable, document title, a "snippet" or keywords in context showing how the query matches the document, and application-specific metadata.

[0045] A given inverted index searcher instance searches a single inverted index. A large scale search engine may have multiple inverted index searcher instances, spread out on different servers in a server cluster. In this case, higher level components, not pictured here, are responsible for broadcasting queries across inverted index search services and integrating the results that come back.

[0046] When inverted index searcher **202** receives query **206**, it forwards it to the evaluation logic **216**, which is the code and associated data structures in the inverted index searcher that executes the query and produces a list of search results. The evaluation logic decides which posting lists to read and dispatches any needed posting list readers. The evaluation logic controls the details of reading, for example, how many posting list readers to use at once, how much of each posting list to read, the order in which lists are read, whether to read a given list all at once, whether to alternate between lists in successive bursts of reading, etc. In the example of FIG. **2**, the evaluation logic has decided to open three posting list readers (**204**, **210**, and **212**) simultaneously over three different posting lists (**218**, **220** and **222**, respectively). As postings are read, the evaluation logic aggregates information in the postings by document and interprets Boolean operators and other advanced search language features to identify matching documents. The end result is a list of search results **208**, often ranked in descending order of relevance, to answer the user's query.

[0047] As it executes a search, an inverted index searcher requires data transfer from the posting file. As previously mentioned, a large search index may require implementing the posting file using secondary storage.

[0048] FIG. **3** is one example of a block/flow diagram **300** for a segmenting posting list reader, in accordance aspects of the present invention. The segmenting posting list reader works together with several other components, pictured in FIG. **3**. Directionality of arrows in FIG. **3** indicates component usage, i.e., an arrow goes from a software component to another component that it uses.

Segmenting Posting List Reader

[0049] The main component in FIG. **3** is the Segmenting Posting List Reader **302** (SPLR) whose purpose is to read posting lists during an inverted index search and make them available to the evaluation logic, in accordance with the efficiencies of the present invention, i.e., reducing the number of reads compared to a conventional reader.

[0050] The SPLR is implemented using several other software components that are introduced here and described in greater detail below. The purpose of the LexiconEntryToPostingListSegmentationMapper **304** is to provide a mapping from each lexicon entry to a segmentation of the associated posting list, thereby determining for each term in the index both the number of bursts of reading to fully read the posting list and the postings that will be read by each successive read request. The LexiconEntryToPostingListSegmentationMapper delegates work optionally to a PostingListLengthApproximationTable **306** and to a PostingListSegmentationTable **308**. A PostingListLengthApproximationTable provides accurate estimates of posting list size, typically in bytes. The PostingListSegmentationTable stores segmentations of the relatively few but frequently accessed posting lists that are larger than a predetermined size. A PostingListReadLimiter **310** helps the SPLR learn segmentations of long posting lists that do not have segmentations in the PostingListSegmentationTable yet, by defining the boundaries between read bursts. An enhanced buffered reader **312** uses configurable predetermined buffer fill size strategies to read from secondary storage more efficiently than a conventional buffered reader. Finally, a BufferFillSizeSelectorFactory **314** manufactures predetermined buffer fill size strategies used to configure an enhanced buffered reader.

[0051] To describe the public interface of the SPLR, it is necessary to first define a LexiconEntry. A LexiconEntry is a record retrieved from the inverted index's lexicon. A LexiconEntry comprises at least three fields: term, document frequency, and posting file start offset. The term is an indexed word or phrase. The document frequency is the length of the term's posting list in number of postings. The posting file start offset is the offset, typically in bytes, in the posting file where the posting list of the term starts. A LexiconEntry consisting of only these 3 fields will be referred to below as a minimal lexicon entry.

[0052] A LexiconEntry may optionally include, for example, a postings file end offset and/or posting list length. A posting file end offset is the offset, typically in bytes, in the posting file where the posting list of the term ends. A posting list length is the length of the posting list of the term, again, typically in bytes. If a lexicon entry has either or both of these fields it will be referred to below as an extended lexicon entry.

[0053] As will become clear, whether a lexicon entry is minimal or extended affects whether a PostingListLengthApproximationTable is required in the implementation of the SPLR.

[0054] The public interface of the SPLR preferably includes the following methods:

[0055] 1. void open (LexiconEntry lexiconEntry)—Prepares the SPLR for reading the posting list indicated by the LexiconEntry. This method has no return value, as indicated by "void."

[0056] 2. Boolean read ( )—Read a burst of postings. The preferred implementation is to forward these postings directly to the evaluation logic via a callback, so it is suggested here that the postings read are not the return value of this method. Because the SPLR automatically decides how many postings to read, the read( )method needs no input parameter such as the number of postings to read. The method returns a Boolean value: whether there are more postings to read, i.e. whether it makes sense for the client to call read( ) again to do another burst of reading.

[0057] 3. void close ( )—Closes the reader, releasing any resources such as memory and/or file handles. This method should leave the SPLR in a state where open( ) can be called again. Making the SPLR reusable in this way facilitates managing resource pools, which is convenient for building the larger search system. The close( ) method has no return value (void).

[0058] A discussion of the various software components, pictured in FIG. **3**, that help implement the SPLR, follows. This, in turn, is followed by some examples of the SPLR in action, illustrated by sequence diagrams, and pseudocode for a proposed SPLR implementation.

PostingListReadLimiter

[0059] The purpose of the PostingListReadLimiter is to give the SPLR a strategy whereby it can learn the complete segmentation of a long posting list.

[0060] The public interface to the PostingListReadLimiter consists of the following method: PostingListReadLimit getLimit (int readSequenceNumber). The getLimit method takes as input a readSequenceNumber, which is an integer greater than or equal to one. A posting list is read using one or more bursts of reading, one burst per segment. The first segment is designated readSequenceNumber **1**, the second as readSequenceNumber **2**, and the readSequenceNumber increases by 1 for each successive burst of reading. The getLimit function returns a PostingListReadLimit that is used by the implementation of the SPLR's read( )method to know when to stop reading during a burst with a given readSequenceNumber.

[0061] The details of how to best define the PostingListReadLimit will vary depending upon the posting list structure of the inverted index and associated evaluation logic.

[0062] In a score sorted index, the postings of each posting list are sorted into descending order by score, so that the evaluation logic gets the postings first with the highest scores, considered the most important. For example, in "Pruned Query Evaluation Using Pre-Computed Impacts," In Proceedings 29th Annual International ACM SIGIR Conference (SIGIR 2006), pp. 372-379, Seattle, Wash., August 2006, incorporated herein by reference in its entirety, V. N. Anh and A. Moffat describe a technique to achieve fast search runtime and a guarantee of search result quality (i.e., relevance) using pruned query evaluation with score-at-a-time processing of an impact-sorted index. In their approach, the postings of each posting list are ordered by descending impact, where impact is a measure of the importance of a term in a document. In their approach, a posting list is read using a sequence of bursts of reading, and within each burst, each posting read contributes the same partial score value toward the score of each document encountered. With a score-sorted posting list organization, to help achieve efficient data access, it is preferable to align the segment boundaries of the present invention with the static score or impact boundaries that are built into the posting list.

[0063] With a score sorted index, the PostingListReadLimit is preferably defined as the minimum impact or score (more generally, the minimum relevance indicator) to read during a burst of reading. To enforce the limit, a burst of reading includes all remaining postings with a score greater than or equal to the minimum score that is the PostingListReadLimit for the current readSequenceNumber. The implementation of PostingListReadLimit getLimit (int readSequenceNumber) in this case is a trivial. The PostingListReadLimiter has as part of its state an array of scores indexed by read sequence number, and the getLimit method simply does an array lookup and returns a score. The array of scores used by the PostingListReadLimiter is preferably configurable through a file or database read by the search system on startup.

[0064] In a document sorted index, another common index organization that is simple and offers good compression characteristics, the postings of each posting list are sorted by document identifier. It is not possible to segment such an index for reading on score boundaries.

[0065] One example strategy to segment a posting list of a document sorted index is to make each successive burst of reading bigger, for example, doubling the size of each successive read. The intuition is to attempt to satisfy the evaluation logic's information need with minimal data transfer, but if the evaluation logic remains unsatisfied, then issue bigger and bigger reads to deliver the needed information with a relatively small number of separate accesses to secondary storage. To implement a strategy like this, the PostingListReadLimit is a minimum number of bytes, for example, to read during a burst. The burst of reading continues until the minimum number of bytes for the readSequenceNumber has been read or until end of list, whichever comes first. The implementation of PostingListReadLimit getLimit (int readSequenceNumber) is straight forward in this case. The PostingListReadLimiter has as part of its state an array of sizes in bytes indexed by read sequence number, and the getLimit method simply does an array lookup and returns a size. The array of sizes used by the PostingListReadLimiter is preferably configurable through a file or database read by the search system on startup.

PostingListSegmentationTable

[0066] A PostingListSegmentationTable is a table of posting list segmentations randomly accessible by term, where a term is an indexed word or phrase. The segmentation information in the table may be complete or incomplete. The SPLR adds segmentation information as it becomes known.

[0067] FIG. **4** shows the structure of a PostingListSegmentationTable **400**, which is preferably held in main memory during search evaluation and also saved to a persistent storage medium for long term storage. The PostingListSegmentationTable includes a hash table **402** keyed on indexed term **404**. Each key is mapped to a PostingListSegmentation object **406**. The PostingListSegmentationTable has a Boolean flag, is Dirty **408**, indicating whether the PostingListSegmentationTable has changed since the last save to persistent storage.

[0068] A PostingListSegmentation object **406** describes a complete or partial segmentation of a posting list. Recall that a posting list segment is a sequence of adjacent postings within a posting list. A complete segmentation of a posting list breaks it up into one or more non-overlapping segments that together include all the postings of the list.

[0069] A PostingListSegmentation object has the following object state:

[0070] 1. Term term—Unique identity of the term whose posting list is being segmented.

[0071] 2. int [ ] postingListSegmentLengths—An array of 0 or more segment lengths in bytes. The length of this array indicates the number of segments that are known. (An array of size 0 is an initial condition, since a posting list generally has at least one item on it.)

[0072] 3. boolean complete—Whether the segmentation is complete.

[0073] 4. boolean approximate—true if the values in postingListSegmentLengths should be treated as approximate sizes; false if the values in postingListSegmentLengths should be treated as exact sizes.

[0074] A PostingListSegmentation also has a convenience method numSegments( ) to return the number of segment lengths that are known. This is the length of the postingListSegmentLengths array.

[0075] A PostingListSegmentationTable includes the following public methods:

[0076] 1. PostingListSegmentation get (Term key)—Given a term, return its segmentation, if any. If the indicated key has no segmentation, return null.

[0077] 2. PostingListSegmentation put (PostingListSegmentation segmentation)—Put the segmentation passed in into the table, keyed on its term. Returns the previous value associated with the key (if any) or null. This method is useful for initially populating the table, for example, if loading it from a persistent data store.

[0078] 3. PostingListSegmentation putRefined (PostingListSegmentation segmentation)—Replaces the PostingListSegmentation associated with the key that is the term of the segmentation passed in. This key will be updated to map to the segmentation parameter if the key currently has no value or if both of the following conditions hold: (a) the current value of the key is incomplete (complete==false) and (b) the current value of the key has a postingListSegmentLengths array that is shorter than the postingListSegmentLengths array in the segmentation that is the parameter to the method. Returns the displaced PostingListSegmentation or null if no value was displaced by this operation. A null return value occurs if a key is set for the first time, or if the put fails because the required conditions do not hold. If this method modifies the PostingListSegmentationTable, it sets the is Dirty flag to true.

[0079] 4. boolean is Dirty ( )—Returns the value of the is Dirty flag, indicating whether the PostingListSegmentationTable has been modified since it was last saved to a persistent storage medium. Mutations caused by the putRefined method cause the is Dirty flag to become true.

[0080] 5. void clearDirty ( )—Set the is Dirty flag to false, to indicate that the PostingListSegmentationTable is unmodified/clean. This method is called when a dirty PostingListSegmentationTable has been successfully saved to a persistent storage medium. The method returns nothing.

[0081] In a search system that is under load, the get( ) and putRefined( )methods may be called concurrently by multiple threads of execution. These methods should be synchronized to avoid erroneous behavior.

PostingListLengthApproximationTable

[0082] A PostingListLengthApproximationTable provides accurate estimates of posting list size, typically in bytes. The main method on a PostingListLengthApproximationTable is:

[0083] PostingListLengthApproximation getPostingListLengthApproximation (documentFrequency)—Returns a PostingListLengthApproximation for a posting list with the indicated document frequency (document frequency is the same thing as posting list length).

[0084] A PostingListLengthApproximation includes the following information: rangeId; average posting list length in bytes for this range; and standard deviation of posting list length in bytes for this range.

[0085] For a detailed discussion of a PostingListLengthApproximationTable refer to U.S. Non-Provisional patent application entitled "ESTIMATION OF POSTINGS LIST LENGTH IN A SEARCH SYSTEM USING AN APPROXIMATION TABLE" (Attorney Docket No. 1634.068A) filed concurrently herewith.

LexiconEntryToPostingListSegmentationMapper

[0086] The purpose of this component is to map a lexicon entry to a PostingListSegmentation. The PostingListSegmentation is useful to the SPLR, representing what is known about how to best break a given posting list into segments for reading.

[0087] The LexiconEntryToPostingListSegmentationMapper delegates work to a PostingListSegmentationTable and optionally to a PostingListLengthApproximationTable as will be spelled out below.

[0088] A LexiconEntryToPostingListSegmentationMapper has the following public methods:

[0089] 1. PostingListSegmentation getPostingListSegmentation (LexiconEntry lexiconEntry)—Given a LexiconEntry, return the PostingListSegmentation to be used by the SPLR to read the posting list.

[0090] 2. PostingListSegmentation updatePostingListSegmentation (PostingListSegmentation segmentation)—Update a posting list segmentation in the PostingListSegmentationTable. (As the SPLR discovers segmentation information, it will call this method to report what it has learned.) This method simply delegates work to the PostingListSegmentationTable's putRefined method.

[0091] Internally, the LexiconEntryToPostingListSegmentationMapper knows how to discriminate between long and short posting lists. A short posting list is one that is short enough to read in its entirety in one burst of reading. A long posting list is one that should be broken into multiple segments and read in pieces. The exact methodology to discriminate between long and short posting lists could vary and is left to the implementer. In one example, the inflection point on the graph of document frequency over term rank (see FIG. **1**) can be used as the dividing line between short and long posting lists.

[0092] As described above, different possible implementations of LexiconEntry include: a minimal lexicon entry that includes just term, document frequency and posting file start offset; and a more extended lexicon entry that adds posting file end offset or posting list length in bytes.

[0093] The implementation of getPostingListSegmentation varies depending upon whether a LexiconEntry is minimal or extended. Examples of Java-like pseudocode for these two scenarios is given below. In the pseudocode below, firstLongDocumentFrequency is the length of the shortest posting list that is considered long as opposed to short per the discussion above.

[0094] Example Pseudocode for getPostingListSegmentation, Minimal Lexicon Entry

```
PostingListSegmentation
    getPostingListSegmentation (LexiconEntry le) {
    if (le.documentFrequency >= firstLongDocumentFrequency) {
        PostingListSegmentation segmentation =
            postingListSegmentationTable.get(le.term);
        if (segmentation == null) {
            // This is a long posting list, but there is no
```

-continued

```
                // segmentation information yet.
                return a new PostingListSegmentation with the
                following state:
                    term = le.term
                    postingListSegmentLengths = trivial empty list
                    complete = false
                    approximate = false
            } else {
                // This is a long posting list, and segmentation info
                // is available; return it.
                return segmentation;
            }
        } else {
            // This is a short posting list. We will approximate its
            // length, because the minimal lexicon entry does not include
            // this information.
            PostingListLengthApproximation la =
                postingListLengthApproximationTable.
                    getPostingListLengthApproximation(le.documentFrequency);
            // Let numPostingListStdDevs be a configured number
            // standard deviations of posting list length
            approximatePostingListLengthBytes =
                la.averagePostingListLengthBytes +
                (numPostingListStdDevs*la.stddevPostingListLengthBytes);
            return a new PostingListSegmentation with the following
            state:
                term = le.term
                postingListSegmentLengths = a list of one item, the
                    approximatePostingListLengthBytes computed above
                complete = true;
                approximate = true;
        }
}
```

[0095] Example Pseudocode for getPostingListSegmentation, Extended Lexicon Entry

```
PostingListSegmentation
    getPostingListSegmentation (LexiconEntry le) {
    if (le.documentFrequency >= firstLongDocumentFrequency) {
        PostingListSegmentation segmentation =
            postingListSegmentationTable.get(le.term);
        if (segmentation == null) {
            // This is a long posting list, but there is no
            // segmentation information yet.
            return a new PostingListSegmentation with the
            following state:
                term = le.term
                postingListSegmentLengths = trivial empty list
                complete = false
                approximate = false
        } else {
            // This is a long posting list, and segmentation info
            // is available; return it.
            return segmentation;
        }
    } else {
        // This is a short posting list, and we know its exact
        // length from the extended lexicon entry.
        return a new PostingListSegmentation with the following
        state:
            term = le.term
            postingListSegmentLengths = a list of one item, the
                exact length of the posting list in bytes, obtained
                from the LexiconEntry le either by subtracting starting
                from ending posting file offsets or simply by using an
                explicit posting list length in bytes from the
                lexicon entry
```

-continued

```
            complete = true;
            approximate = false;
    }
}
```

Enhanced Buffered Reader

[0096] The buffered reader used by the SPLR is an enhanced buffered reader that uses configurable predetermined buffer fill size strategies to read from secondary storage more efficiently than a conventional buffered reader.

[0097] For a detailed discussion of enhanced buffered readers, refer to U.S. Non-Provisional patent application entitled “EFFICIENT BUFFERED READING WITH A PLUG IN FOR INPUT BUFFER SIZE DETERMINATION” (Attorney Docket No. 1634.069A) filed concurrently herewith.

BufferFillSizeSelectorFactory

[0098] The BufferFillSizeSelectorFactory is used to make BufferFillSizeSelector objects for plugging into the enhanced buffered reader. A BufferFillSizeSelector object is a predetermined buffer fill size strategy. More specifically, a BufferFillSizeSelector is an ordered sequence of (fillSize, numTimesToUse) pairs, where fillSize indicates how much of an enhanced buffered reader’s internal input buffer to fill when a buffer fill is needed, and numTimesToUse indicates how many times to use the associated fillSize.

[0099] The object state of the BufferFillSizeSelectorFactory includes maxBufferSize, which is the largest read system call that can be issued, typically in bytes, based on the maximum available input buffer size of the enhanced buffered reader. In one example, a large maxBufferSize (of 20 megabytes or so) is used on a commodity server with an index of 20 million web documents.

[0100] The BufferFillSizeSelectorFactory provides the following public methods:

[0101] 1. BufferFillSizeSelectorFactory (int maxBufferSize)—Constructs a new BufferFillSizeSelectoryFactory. The constructor simply records the maxBufferSize as object state, for future reference.

[0102] 2. BufferFillSizeSelector makePreciseBufferFillSizeSelector (long numBytesToRead)—This method returns a BufferFillSizeSelector to read a precise number of bytes with a minimum number of reads, taking into account the maxBufferSize.

[0103] 3. BufferFillSizeSelector makeApproximateBufferFillSizeSelector (long approximateNumBytesToRead, int supplementalReadSize)—This method returns a BufferFillSizeSelector that will read approximateNumBytesToRead with a minimum number of reads, and will thereafter revert to using buffer fills of the supplementalReadSize if more information is needed.

[0104] In the discussion that follows, let "/" represent the operation of integer division, and "%" represent the operation of integer modulo.

[0105] To implement makePreciseBufferFillSizeSelector, there are two cases to consider, where numBytesToRead is the input to makePreciseBufferFillSizeSelector, and maxBufferSize is the largest read system call that can be issued in bytes:

[0106] Case **1**: maxBufferSize>=numBytesToRead

[0107] Case **2**: maxBufferSize<numBytesToRead

[0108] A discussion of these cases follows.

Case **1**: maxBufferSize>=numBytesToRead

[0109] Build a one-stage predetermined buffer fill size strategy as indicated below in Table I.

TABLE I

Stage	Fill Size	Number of Times to Use
1	numBytesToRead	1

[0110] The above strategy, when installed in an enhanced buffered reader, will read exactly numBytesToRead bytes of data using a single system call.

Case **2**: maxBufferSize<numBytesToRead

[0111] In this case, build a predetermined buffer fill size strategy that generally has two stages, as indicated in Table II. However, the second stage is not necessary when the maxBufferSize evenly divides numBytesToRead.

TABLE II

Stage	Fill Size	Number of Times to Use
1	maxBufferSize	numBytesToRead/maxBufferSize
2	numBytesToRead % maxBufferSize	1

[0112] The above strategy, when installed in an enhanced buffered reader, will read exactly numBytesToRead bytes of data with the minimum possible number of read system calls.

[0113] To implement makeApproximateBufferFillSizeSelector, there are two cases to consider, where approximateNumBytesToRead is input to makeApproximateBufferFillSizeSelector, and maxBufferSize is the largest read system call that can be issued in bytes:

[0114] Case **3**: maxBufferSize>=approximateNumBytesToRead

[0115] Case **4**: maxBufferSize<approximateNumBytesToRead

Also, recall that a supplementalReadSize is provided as input to makeApproximateBufferFillSizeSelector. A discussion of these cases follows.

Case **3**: maxBufferSize>=approximateNumBytesToRead

[0116] Build a two-stage predetermined buffer fill size strategy as indicated below in Table III.

TABLE III

Stage	Fill Size	Number of Times to Use
1	approximateNumBytesToRead	1
2	supplementalReadSize	Repeat as necessary

[0117] The above strategy, when installed in an enhanced buffered reader, will read approximateNumBytesToRead bytes of data using a single read system call and thereafter will perform as many additional system calls of the supplemental read size as necessary.

Case **4**: maxBufferSize<approximateNumBytesToRead

[0118] In this case, build a predetermined buffer fill size strategy that generally has three stages, as indicated below in Table IV. However, the second stage is not necessary when the maxBufferSize evenly divides the approximateNumBytesToRead.

TABLE IV

Stage	Fill Size	Number of Times to Use
1	maxBufferSize	approximateNumBytesToRead/ maxBufferSize
2	approximateNumBytesToRead % maxBufferSize	1
3	supplementalReadSize	Repeat as necessary

[0119] The above strategy, when installed in an enhanced buffered reader, will read approximateNumBytesToRead bytes of data with the minimum possible number of read system calls and thereafter will perform as many additional system calls of the supplemental read size as necessary.

[0120] One example of a method of reading a posting list will now be described with reference to the flow diagram **500** of FIG. **5**. A processor (e.g., as part of a computing unit) is used to determine the size of a posting list as part of an inverted index search, step **502**. Once the size is determined, an inquiry is made as to whether the size is larger than a predetermined size, inquiry **504**. If the size is equal to or smaller than the predetermined size, i.e., not larger than the predetermined size, then the entire posting list is read into memory as a single segment, step **506**. However, if the size is larger than the predetermined size, then the posting list is segmented, step **508**. In one example, the segmenting uses at least one predetermined segment size. In another example, the segmenting uses at least one estimated segment size, and the actual read size is at least the estimated size. The estimated segment size may be stored in a data structure for reuse. The segments, by whatever method of segmentation, are then each read into memory, step **506**.

SPLR Operational Examples

[0121] Having described the SPLR and each of its subcomponents from FIG. **3** in some detail, and providing the basic method above, some examples of how these components work together to read posting lists will now be presented.

[0122] In a first example, the sequence diagram **600** in FIG. **6** shows interactions between an inverted index searcher **602**, SegmentingPostingListReader **604**, LexiconEntryToPostingListSegmentationMapper **606**, PostingListLengthApproximationTable **608**, enhanced buffered reader **610**, BufferFillSizeSelectorFactory **612** and evaluation logic **614** as the inverted index searcher reads a short posting list consisting of only a single segment. In this example, a minimal lexicon entry is being used and therefore a PostingListLengthApproximationTable is also used.

[0123] The inverted index searcher begins a reading session by calling the open method **616** on the SPLR, passing in the lexicon entry of the posting list to read. The SPLR saves a reference to this lexicon entry as part of its state to help control the reading session. The SPLR calls getPostingListSegmentation **618** on the LexiconEntryToPostingListSegmentationMapper, forwarding the lexicon entry. The LexiconEntryToPostingListSegmentationMapper examines the document frequency of the lexicon entry and consults its method of discriminating between long and short posting lists. The LexiconEntryToPostingListSegmentationMapper determines that the posting list to read is short and calls getPostingListLengthApproximation **620** on the PostingListLengthApproximationTable, providing as input the document frequency of the lexicon entry. A PostingListLengthApproximation is returned to the LexiconEntryToPostingListSegmentationMapper **622**, which then builds a complete, approximate PostingListSegmentation, incorporating the term from the lexicon entry and a single posting list segment length equal to the average posting list length in bytes plus the desired number of standard deviations from the PostingListLengthApproximation. The LexiconEntryToPostingListSegmentationMapper returns this newly built PostingListSegmentation to the SPLR **624**, where it becomes part of the SPLR's state to control the reading session. The SPLR finishes the execution of its open( )method by initializing various miscellaneous state variables and finally seeking the enhanced buffered reader to the start of the posting list **626** by passing the posting file start offset of the lexicon entry to the enhanced buffered reader's seek method. At this point, the open method called by the inverted index searcher returns, and the SPLR is ready to accept a read call.

[0124] The inverted index searcher calls the SPLR's read( )method **628**. Based on the state established during the open( )method, the SPLR recognizes that the posting list being read consists of a single segment with an approximate length in bytes. The SPLR forwards the approximate number of bytes to read to the BufferFillSizeSelectorFactory's makeApproximateBufferFillSizeSelector method **630**. A predetermined buffer fill size strategy in the form of a BufferFillSizeSelector object is returned to the SPLR **632**, which it installs in the enhanced buffered reader by calling setBufferFillSizeSelector **634**. The SPLR next uses the enhanced buffered reader to read all of the postings in this relatively short posting list **636**, forwarding each posting to the evaluation logic **638**. Finally, the SPLR's read method returns false to the inverted index searcher **640**, indicating that there are no more postings available to be read, and the inverted index searcher calls close **642** on the SPLR to end the reading session.

[0125] In a second example, the sequence diagram **700** in FIG. **7** shows interactions between an inverted index searcher **702**, SegmentingPostingListReader **704**, LexiconEntryToPostingListSegmentationMapper **706**, PostingListSegmentationTable **708**, enhanced buffered reader **710**, PostingListReadLimiter **712** and evaluation logic **714** as the inverted index searcher reads a posting list consisting of two segments. In this example, the segmentation of the posting list is unknown and has to be learned as the posting list is read.

[0126] The inverted index searcher begins a reading session by calling the open method on the SPLR **716**, passing in the lexicon entry of the posting list to read. The SPLR saves a reference to this lexicon entry as part of its state to help control the reading session. The SPLR calls getPostingListSegmentation on the LexiconEntryToPostingListSegmentationMapper **718**, forwarding the lexicon entry. The LexiconEntryToPostingListSegmentationMapper examines the document frequency of the lexicon entry and consults its method of discriminating between long and short posting lists. The LexiconEntryToPostingListSegmentationMapper determines that the posting list to read is long and calls get( ) on the PostingListSegmentationTable **720**, passing in the term of the lexicon entry as the key for the lookup. The PostingListSegmentationTable consults its hash but finds no mapping from the term to a PostingListSegmentation. In this scenario, the posting list has not been read since the inverted index was deployed, and its segmentation is unknown. The get( ) call returns null to the LexiconEntryToPostingListSegmentationMapper **722**, indicating that no segmentation information is available. In response, the LexiconEntryToPostingListSegmentationMapper creates a new incomplete, precise (i.e. complete=false, approximate=false) PostingListSegementation, incorporating the term from the lexicon entry, and using an empty array of posting list segment lengths. This new empty PostingListSegmentation is returned to the SPLR **724**, where it becomes part of the SPLR's state to control the reading session. The SPLR finishes the execution of its open( )method by initializing various miscellaneous state variables and finally seeking the enhanced buffered reader to the start of the posting list **726** by passing the posting file start offset of the lexicon entry to the enhanced buffered reader's seek method. At this point, the open method called by the inverted index searcher returns, and the SPLR is ready to accept a read call.

[0127] The inverted index searcher calls the SPLR's read( )method **728**. Based on the state established during the open( )method, the SPLR recognizes that the posting list consists of multiple segments, that the segment boundaries are unknown, and the segment boundaries need to be learned. Because this is the first call to read in this session, the SPLR forwards the value 1 (one) to the getLimit method of the PostingListReadLimiter **730**. The PostingListReadLimiter returns a PostingListReadLimit **732**, an indication of how far the SPLR may read during this first read call. With this information, the SPLR is almost ready to read postings. Since the SPLR does not know the size in bytes of the segment it is about to read, it calls setBufferFillSizeSelector **734** to install a default predetermined buffer fill size strategy on the enhanced buffered reader that always buffers several disk blocks worth of data whenever the buffered reader needs more data. This strategy is acceptable for learning a new segmentation, after which a better strategy will be available.

[0128] Before reading any postings, the SPLR is careful to note the current logical position of the enhanced buffered reader in the posting file **736**. Knowing the read start position will allow the SPLR to know the length of the segment later when reading stops. The SPLR now uses the enhanced buffered reader to read postings **738**, forwarding each one to the evaluation logic as soon as it is read **740**, stopping when the PostingListReadLimit is reached or at end of posting list, whichever comes first. In this case, reading stops because the PostingListReadLimit is reached. Once again the SPLR gets the current logical position from the enhanced buffered reader **742**. The difference between this second logical position and the first one that was obtained is the length of the segment just read. The SPLR creates and remembers an updated PostingListSegmentation object that includes the new segment length just learned. The SPLR then passes the updated PostingListSegmentation to the updatePostingListSegmentation method of the LexiconEntryToPostingListSegmentationMapper **744**, to preserve the updated segmentation information for reuse by future read sessions. The LexiconEntryToPostingListSegmentationMapper simply forwards the PostingListSegmentation to the putRefined method of the PostingListSegmentationTable **746**, where the PostingListSegmentation is stored for reuse. Because reading stopped due to the PostingListReadLimit (and not due to end of posting list), there are more postings to read and the SPLR's read method returns true **748** to the inverted index searcher to indicate this fact.

[0129] The inverted index searcher then calls the SPLR's read method a second time **750**. Based on the state of the SPLR after the first read call, the SPLR recognizes that the posting list consists of multiple segments, more postings are available, but the extent of the next segment to read is unknown and has to be learned. Because this is the second call to read in this session, the SPLR forwards the value 2 (two) to the getLimit method of the PostingListReadLimiter **752**. The PostingListReadLimiter returns a PostingListReadLimit **754**, an indication of how far the SPLR may read during this second read call. The SPLR now follows the same steps it used during the first read call, installing a default predetermined buffer fill size strategy on the enhanced buffered reader **756**, noting the read start position by getting the current logical position from the enhanced buffered reader **758**, and reading postings **760** and forwarding each one to the evaluation logic **762**. As before, reading stops when the PostingListReadLimit is reached or at end of posting list, whichever comes first. In this case, reading stops because end of posting list is reached.

[0130] The SPLR then gets the current logical position from the enhanced buffered reader **764**. The difference between this second logical position and the first one that was obtained is the length of the segment just read. The SPLR creates and remembers an updated PostingListSegmentation object that includes both the new segment length just learned and the new knowledge that the segmentation of this posting list is complete (complete=true). The SPLR then passes the updated PostingListSegmentation to the updatePostingListSegmentation method of the LexiconEntryToPostingListSegmentationMapper **766**, to preserve the updated segmentation information for reuse by future read sessions. The LexiconEntryToPostingListSegmentationMapper simply forwards the PostingListSegmentation to the putRefined method of the PostingListSegmentationTable **768**, where the PostingListSegmentation is stored for reuse. Because reading stopped this time due to end of posting list, there are no more postings to read and the SPLR's read method returns false to the inverted index searcher to indicate this fact **770**. Finally, the inverted index searcher calls close to close this read session **772**.

[0131] In a third example, the sequence diagram **800** in FIG. **8** shows interactions between an inverted index searcher **802**, SegmentingPostingListReader **804**, LexiconEntryToPostingListSegmentationMapper **806**, PostingListSegmentationTable **808**, enhanced buffered reader **810**, BufferFillSizeSelectorFactory **812** and evaluation logic **814** as the inverted index searcher reads a posting list consisting of two segments. In this example, the segmentation of the posting list is known. This scenario shows the benefit of learning and reusing posting list segmentations for large, frequently accessed posting lists.

[0132] The inverted index searcher begins a reading session by calling the open method on the SPLR **816**, passing in the lexicon entry of the posting list to read. The SPLR saves a reference to this lexicon entry as part of its state to help control the reading session. The SPLR calls getPostingListSegmentation on the LexiconEntryToPostingListSegmentationMapper **818**, forwarding the lexicon entry. The LexiconEntryToPostingListSegmentationMapper examines the document frequency of the lexicon entry and consults its method of discriminating between long and short posting lists. The LexiconEntryToPostingListSegmentationMapper determines that the posting list to read is long and calls get( ) on the PostingListSegmentationTable **820**, passing in the term of the lexicon entry as the key for the lookup. The PostingListSegmentationTable consults its hash and finds that the term is mapped to a complete, precise (i.e. complete=true, approximate=false) PostingListSegmentation with 2 segments. The get( ) call returns this PostingListSegmentation to the LexiconEntryToPostingListSegmentationMapper **822**, which in turn simply returns it to the SPLR **824**, where it becomes part of the SPLR's state to control the reading session. The SPLR finishes the execution of its open( )method by initializing various miscellaneous state variables and finally seeking the enhanced buffered reader to the start of the posting list **826** by passing the posting file start offset of the lexicon entry to the enhanced buffered reader's seek method. At this point, the open method called by the inverted index searcher returns, and the SPLR is ready to accept a read call.

[0133] The inverted index searcher calls the SPLR's read( )method **828**. Based on the state established during the open( )method, the SPLR recognizes that the posting list being read consists of two segments of known sizes in bytes. The SPLR forwards the exact size in bytes of the first segment to the BufferFillSizeSelectorFactory's makePreciseBufferFillSizeSelector method **830**. A predetermined buffer fill size strategy in the form of a BufferFillSizeSelector object is returned to the SPLR **832**, which it installs in the enhanced buffered reader by calling setBufferFillSizeSelector **834**. The SPLR next uses the enhanced buffered reader to read all of the postings in the first segment of this posting list **836**, forwarding each posting to the evaluation logic **838**. Finally, the SPLR's read method returns true to the inverted index searcher **840**, indicating that there are more postings available to be read.

[0134] The inverted index searcher again calls the SPLR's read( ) method **842**. Based on the state after the first read call, the SPLR recognizes that there is another segment of known size in bytes available to read. The SPLR forwards the exact size in bytes of the second segment to the BufferFillSizeSelectorFactory's makePreciseBufferFillSizeSelector method **844**. A predetermined buffer fill size strategy in the form of a BufferFillSizeSelector object is returned to the SPLR **846**, which it installs in the enhanced buffered reader by calling setBufferFillSizeSelector **848**. The SPLR next uses the enhanced buffered reader to read all of the postings in the second segment of this posting list **850**, forwarding each posting to the evaluation logic **852**. Finally, the SPLR's read method returns false **854** to the inverted index searcher, indicating that there are no more postings available to be read, and the inverted index searcher closes the read session by calling close( ) on the SPLR **856**.

Common Pseudocode for a SPLR Implementation

[0135] The SPLR and its subcomponents, pictured in FIG. **3**, were described above. Note that flexibility in the design of the SPLR allows for several different scenarios, for example:

- **[0136]** 1. Minimal or extended lexicon entry, which implies the presence or absence of the PostingListLengthApproximationTable, which in turn requires differences in implementation of the LexiconEntryToPostingListSegmentationMapper;
- **[0137]** 2. Different posting list organizations (e.g., score sorted or document id sorted);
- **[0138]** 3. Flexibility of discriminating between "short" and "long" posting lists; exact method to be chosen by implementer; and
- **[0139]** 4. Different ways of implementing PostingListReadLimit (e.g., using a sequence of limiting scores or a sequence of limiting read sizes).

[0140] The pseudocode below applies equally to all the scenarios listed above; thus, it is the common pseudocode for a SPLR implementation.

[0141] As a prerequisite to understanding the pseudocode for methods of the SPLR, it is helpful to first understand the data members that are part of its state. The following data members are initialized by sending object references to the SPLR's constructor.

- **[0142]** 1. BufferedReader bufferedReader—This is an enhanced buffered reader that is open over the posting file. It has a large input buffer (perhaps 20 MB for a large scale index on a commodity server);
- **[0143]** 2. LexiconEntryToPostingListSegmentationMapper lexiconEntryToPostingListSegmentationMapper;
- **[0144]** 3. BufferFillSizeSelectorFactory bufferFillSizeSelectorFactory; and
- **[0145]** 4. PostingListReadLimiter postingListReadLimiter.

[0146] The SPLR has additional state that is set up as part of a call to its open (lexiconEntry) method. These data members are documented here.

- **[0147]** 5. LexiconEntry lexiconEntry—Lexicon entry of the posting list to read;
- **[0148]** 6. PostingListSegmentation pls—The most complete segmentation of this posting list currently available;
- **[0149]** 7. int readNum—A count of how many times the read method has been called. This variable is set to 1 throughout the first call to read( ) to 2 throughout the second call to read( ) and so on;
- **[0150]** 8. boolean done—Whether end of posting list has been reached; and
- **[0151]** 9. int numPostingsRead—The number of postings that have been read.

[0152] The SPLR has three public methods:

- **[0153]** 1. void open (LexiconEntry lexiconEntry)
- **[0154]** 2. boolean read( )
- **[0155]** 3. void close( )

[0156] Open( ) should be called first to prepare for reading. Read( ) may be called multiple times. Each call to read( ) reads a segment of postings, and the boolean return value indicates whether there is another segment available. Finally, a well behaved client calls close( ) to signal the end of the reading session.

[0157] The pseudocode below is Java-like. Java operators and Java-like syntax are used, and array indexes start at 0. Example pseudocode for each of the SPLR's public methods follows.

[0158] Example pseudocode for open method

```
public void open(LexiconEntry aLexiconEntry) {
    // Set the SPLR data member, lexiconEntry, based on the
    aLexiconEntry passed in lexiconEntry = aLexiconEntry;
    // Set the SPLR data member, pls by lookup in the
    pls = lexiconEntryToPostingListSegmentationMapper.
        getPostingListSegmentation(lexiconEntry);
    // Initialize various other SPLR data members
    readNum = 0;
    done = false;
    numPostingsRead = 0;
    // Seek enhanced buffered reader to start of posting list
    bufferedReader.seek(lexiconEntry.postingFileStartOffset);
}
```

[0159] Example Pseudocode for read method

```
public boolean read( ) {
// Reminder:   This method returns true if there are more postings to read and false
//             otherwise.
    readNum = readNum + 1;
    if (done) {
        return false; // nothing else to read
    }
    // NOTE: && is logical AND; == is the equality test
    if ( pls.complete && pls.approximate && (pls.numSegments( ) == 1) ) {
        // This is a short posting list for which we have an
        // approximate size in bytes.
        // Any read beyond the first segment is trying to go too far.
```

-continued

```
        if ( readNum > 1 ) {
            done = true; // just to be sure
            return false; // nothing else to read
        }
        // readNum is 1. This is the first read of a 1-segment list.
        readShortPostingList
            (pls.postingListSegmentLengths[readNum−1], lexiconEntry.documentFrequency);
        done = true;
        return false; // nothing else to read
    } else if (! pls.approximate) { // NOTE: ! means logical NOT
        // The segmentation has or will have precise information.
        // A precise segmentation has been or will be learned.
        // The segmentation may or may not be complete at this time.
        if (readNum <= pls.numSegments( )) {
            // Segment size for this read is known.
            readPostingListSegment(pls.postingListSegmentLengths[readNum−1]);
            // If the last segment has been read, set the done flag
            if ( (readNum == pls.numSegments( )) && pls.complete) {
                done = true;
            }
            return (! done); // whether more postings
        }
        // If the program gets here, readNum > pls.numSegments
        // The program is trying to read past known segmentation info.
        if (pls.complete) {
            // There's nothing else to learn.
            done = true;
            return false; // nothing else to read
        }
        // If the program gets here, there ARE more postings.
        // There's segmentation information to be learned.
        PostingListReadLimit readLimit =
            postingListReadLimiter.getLimit(readNum);
        readAndLearnSegmentation(readLimit);
        // pls has been maintained by the call to readAndLearnSegmentation
        // If the last segment has been read, set the done flag
        if ( (readNum == pls.numSegments( )) && pls.complete) {
            done = true;
        }
        return (! done); // whether more postings
    } else {
        // This should never happen, if the
        // LexiconEntryToPostingListSegmentationMapper
        // is building valid PostingListSegmentations.
        Log an error;
        done = true;
        return false; // nothing else to read
    }
}
private void readShortPostingList
    (int approximateNumBytesToRead, int documentFrequency) {
    BufferFillSizeSelector bufferFillSizeSelector =
        bufferFillSizeSelectorFactory.
            makeApproximateBufferFillSizeSelector
                (approximateNumBytesToRead, supplementalReadSize( ));
    bufferedReader.setBufferFillSizeSelector(bufferFillSizeSelector);
    while (numPostingsRead < documentFrequency) {
        use bufferedReader to read posting;
        forward posting to evaluation logic;
        numPostingsRead = numPostingsRead + 1;
    }
}
private int supplementalReadSize( ) {
    // Return the number of bytes to read for the relatively rare case
    // when the approximate covering read size for a short posting list
    // was insufficient. A value like a several kilobytes is fine.
}
private void readPostingListSegment(int numBytesToRead) {
    BufferFillSizeSelector bufferFillSizeSelector =
        bufferFillSizeSelectorFactory.
            makePreciseBufferFillSizeSelector(numBytesToRead);
    bufferedReader.setBufferFillSizeSelector(bufferFillSizeSelector);
    // Get the current logical offset of the buffered reader from start
    // of data.
    long startOffset = bufferedReader.offset( );
    //We want to read to here.
```

-continued

```
    long targetOffset = startOffset + numBytesToRead;
    while (bufferedReader.offset( ) < targetOffset) {
        use bufferedReader to read posting;
        forward posting to evaluation logic;
        numPostingsRead = numPostingsRead + 1;
    }
}
private void readAndLearnSegmentation(PostingListReadLimit readLimit) {
    // A segmentation strategy is being learned here.
    // We do not know a better strategy to use.
    bufferedReader.setBufferFillSizeSelector(getTraditionalBufferingStrategy( ));
    // Get current logical position within posting data
    long startOffset = bufferedReader.offset( );
    while (readLimit has not been exceeded &&
                numPostingsRead < lexiconEntry.documentFrequency) {
        use bufferedReader to read posting;
        forward posting to evaluation logic;
        numPostingsRead = numPostingsRead + 1;
    }
    long endOffset = bufferedReader.offset( );
    long newSegmentLength = endOffset − startOffset;
    PostingListSegmentation newPls =
        a copy of the pls data member, with the following
        adjustments applied:
            1.  A single additional element has been added to the postingListSegmentLengths[ ]
                array: the newSegmentLength learned above
            2.  if the while loop above reached end of posting list, i.e. (numPostingsRead ==
                lexiconEntry.documentFrequency) then set complete = true
    // Save the segmentation we learned for reuse, to read more intelligently next time.
    lexiconEntryToPostingListSegmentationMapper.
        updatePostingListSegmentation(newPls);
    // And don't forget to maintain this object's state.
    pls = newPls;
}
private BufferFillSizeSelector getTraditionalBufferingStrategy( ) {
    // This method returns a buffering strategy that says to buffer
    // several disk blocks whenever data needs to be read from the
    // operating system, until further notice. We use this buffering
    // strategy only while learning a segment boundary for the first
    // time.
}
```

[0160] Example pseudocode for close method

```
public void close( ) {
    // In this basic implementation, close does not need to do anything.
    // The next call to open( ) will fully reset all SPLR state
    // for the next reading session. We assume it is OK to leave the
    // buffered reader open over the posting file between read sessions.
}
```

[0161] As evident in the pseudocode above, the implementation of the SPLR's read method has to handle different cases defined by the combination of the PostingListSegmentation state and the readNum. Recall that the readNum is 1 throughout the first call to read, 2 throughout the second call to read, and so on. The combination of the PostingListSegmentation (pls) state and the readNum defines cases as described in Table V below.

TABLE V

pls.complete	pls.approximate	readNum vs. pls.numSegments	Comments
true	true	>	Short posting list, single approximately sized segment. The client is trying to read too far.
true	true	<=	Short posting list, single approximately sized segment. About to read the first and only segment.
true	false	>	Posting list could be long or short. Its segmentation is complete, and the client is trying to read beyond the end of the list.

TABLE V-continued

pls.complete	pls.approximate	readNum vs. pls.numSegments	Comments
true	false	<=	Posting list could be long or short. Its segmentation is complete, and the size of the next segment to read is known.
false	true	>	Invalid state. Incomplete approximate PostingListSegmentations are never created.
false	true	<=	Invalid state. Incomplete approximate PostingListSegmentations are never created.
false	false	>	The current posting list is long and incompletely segmented, and the next read will learn a new segmentation.
false	false	<=	The current posting list is long and incompletely segmented, but the size of the next segment to read is known.

[0162] The definition of the cases in Table V depends upon how PostingListSegmentation objects are created by the LexiconEntryToPostingListSegmentationMapper. An awareness of this dependency is helpful for understanding and possibly evolving the pseudocode that was presented.

Persistence of PostingListSegmentationTable

[0163] The PostingListSegmentationTable will be updated dynamically as the SPLR's read method is called. When the search service shuts down, the PostingListSegmenationTable is preferably saved to disk or other nonvolatile storage medium. To avoid losing the work of learning segmentations, the PostingListSegmenationTable could also be saved automatically (like every 5 or 10 minutes or so) if it has become dirty.

Performing Index Maintenance

[0164] If the inverted index changes, the PostingListSegmentation table becomes invalid. On any index maintenance, all persistent and in-memory copies of this table must be deleted. The system can then re-learn the up-to-date segmentations.

[0165] The present invention includes the following aspects:

[0166] 1. Learning posting list segmentation strategy dynamically as the search system executes searches.

[0167] 2. Supporting a plug-in (PostingListReadLimiter in the above example) that is used by the posting list reader to determine how to segment large posting lists into pieces. This PostingListReadLimiter can be tailored to fit the posting list organization and query logic.

[0168] 3. Minimizing access to secondary storage via dynamically crafted read strategies for large posting lists, rather than a traditional buffered reader.

[0169] 4. Supporting optional use of a PostingListLengthApproximationTable (as described above) if lexicon entries contain minimal information (to save space in memory) and do not include the length of the posting list in bytes.

[0170] 5. Using an enhanced BufferedReader (as described in section above) to plug application-specific knowledge of good read sizes into the low level I/O system.

[0171] The shortcomings of the prior art are overcome and additional advantages are provided through the provision of a computer program product for efficient reading of posting lists as part of inverted index searching. The computer program product comprises a storage medium readable by a processor and storing instructions for execution by a processor for performing a method. The method includes, for instance, determining by a processor a size of a posting list as part of searching an inverted index, segmenting the posting list by the processor for reading into a plurality of segments based on the size, and reading by the processor each of the plurality of segments into memory.

[0172] Methods and systems relating to one or more aspects of the present invention are also described and claimed herein.

[0173] Additional features and advantages are realized through the techniques of the present invention. Other embodiments and aspects of the invention are described in detail herein and are considered a part of the claimed invention.

[0174] In one aspect of the present invention, an application can be deployed for performing one or more aspects of the present invention. As one example, the deploying of an application comprises providing computer infrastructure operable to perform one or more aspects of the present invention.

[0175] As a further aspect of the present invention, a computing infrastructure can be deployed comprising integrating computer readable code into a computing system, in which the code in combination with the computing system is capable of performing one or more aspects of the present invention.

[0176] As yet a further aspect of the present invention, a process for integrating computing infrastructure comprising integrating computer readable code into a computer system may be provided. The computer system comprises a computer readable medium, in which the computer medium comprises one or more aspects of the present invention. The code in combination with the computer system is capable of performing one or more aspects of the present invention.

[0177] As will be appreciated by one skilled in the art, aspects of the present invention may be embodied as a system, method or computer program product. Accordingly, aspects of the present invention may take the form of an entirely hardware embodiment, an entirely software embodiment (including firmware, resident software, micro-code, etc.) or an embodiment combining software and hardware aspects that may all generally be referred to herein as a "circuit," "module" or "system." Furthermore, aspects of the present invention may take the form of a computer program product embodied in one or more computer readable medium(s) having computer readable program code embodied thereon.

[0178] Any combination of one or more computer readable medium(s) may be utilized. The computer readable medium may be a computer readable storage medium. A computer readable storage medium may be, for example, but not limited to, an electronic, magnetic, optical, or semiconductor system, apparatus, or device, or any suitable combination of the foregoing. More specific examples (a non-exhaustive list) of the computer readable storage medium include the following: an electrical connection having one or more wires, a portable computer diskette, a hard disk, a random access memory (RAM), a read-only memory (ROM), an erasable programmable read-only memory (EPROM or Flash memory), an optical fiber, a portable compact disc read-only memory (CD-ROM), an optical storage device, a magnetic storage device, or any suitable combination of the foregoing. In the context of this document, a computer readable storage medium may be any tangible medium that can contain or store a program for use by or in connection with an instruction execution system, apparatus, or device.

[0179] In one example, a computer program product includes, for instance, one or more computer readable media to store computer readable program code means or logic thereon to provide and facilitate one or more aspects of the present invention. The computer program product can take many different physical forms, for example, disks, platters, flash memory, etc.

[0180] Program code embodied on a computer readable medium may be transmitted using an appropriate medium, including but not limited to wireless, wireline, optical fiber cable, RF, etc., or any suitable combination of the foregoing.

[0181] Computer program code for carrying out operations for aspects of the present invention may be written in any combination of one or more programming languages, including an object oriented programming language, such as Java, Smalltalk, C++ or the like, and conventional procedural programming languages, such as the "C" programming language or similar programming languages. The program code may execute entirely on the user's computer, partly on the user's computer, as a stand-alone software package, partly on the user's computer and partly on a remote computer or entirely on the remote computer or server. In the latter scenario, the remote computer may be connected to the user's computer through any type of network, including a local area network (LAN) or a wide area network (WAN), or the connection may be made to an external computer (for example, through the Internet using an Internet Service Provider).

[0182] Aspects of the present invention are described herein with reference to flowchart illustrations and/or block diagrams of methods, apparatus (systems) and computer program products according to embodiments of the invention. It will be understood that each block of the flowchart illustrations and/or block diagrams, and combinations of blocks in the flowchart illustrations and/or block diagrams, can be implemented by computer program instructions. These computer program instructions may be provided to a processor of a general purpose computer, special purpose computer, or other programmable data processing apparatus to produce a machine, such that the instructions, which execute via the processor of the computer or other programmable data processing apparatus, create means for implementing the functions/acts specified in the flowchart and/or block diagram block or blocks.

[0183] These computer program instructions may also be stored in a computer readable medium that can direct a computer, other programmable data processing apparatus, or other devices to function in a particular manner, such that the instructions stored in the computer readable medium produce an article of manufacture including instructions which implement the function/act specified in the flowchart and/or block diagram block or blocks.

[0184] The computer program instructions may also be loaded onto a computer, other programmable data processing apparatus, or other devices to cause a series of operational steps to be performed on the computer, other programmable apparatus or other devices to produce a computer implemented process such that the instructions which execute on the computer or other programmable apparatus provide processes for implementing the functions/acts specified in the flowchart and/or block diagram block or blocks.

[0185] The flowchart and block diagrams in the figures illustrate the architecture, functionality, and operation of possible implementations of systems, methods and computer program products according to various embodiments of the present invention. In this regard, each block in the flowchart or block diagrams may represent a module, segment, or portion of code, which comprises one or more executable instructions for implementing the specified logical function (s). It should also be noted that, in some alternative implementations, the functions noted in the block may occur out of the order noted in the figures. For example, two blocks shown in succession may, in fact, be executed substantially concurrently, or the blocks may sometimes be executed in the reverse order, depending upon the functionality involved. It will also be noted that each block of the block diagrams and/or flowchart illustration, and combinations of blocks in the block diagrams and/or flowchart illustration, can be implemented by special purpose hardware-based systems that perform the specified functions or acts, or combinations of special purpose hardware and computer instructions.

[0186] A data processing system **900**, as shown in FIG. **9**, may be provided suitable for storing and/or executing program code is usable that includes at least one processor **902** coupled directly or indirectly to memory elements **904** through a system bus **906**. The memory elements include, for instance, local memory employed during actual execution of the program code, bulk storage, and cache memory which provide temporary storage of at least some program code in order to reduce the number of times code must be retrieved from bulk storage during execution.

[0187] Input/Output or I/O devices **908** (including, but not limited to, keyboards, displays, pointing devices, DASD, tape, CDs, DVDs, thumb drives and other memory media, etc.) can be coupled to the system either directly or through intervening I/O controllers. Network adapters may also be coupled to the system to enable the data processing system to become coupled to other data processing systems or remote printers or storage devices through intervening private or public networks. Modems, cable modems, and Ethernet cards are just a few of the available types of network adapters.

[0188] The terminology used herein is for the purpose of describing particular embodiments only and is not intended to be limiting of the invention. As used herein, the singular forms "a", "an" and "the" are intended to include the plural forms as well, unless the context clearly indicates otherwise. It will be further understood that the terms "comprises" and/or "comprising", when used in this specification, specify the presence of stated features, integers, steps, operations, elements, and/or components, but do not preclude the presence or addition of one or more other features, integers, steps, operations, elements, components and/or groups thereof.

[0189] The corresponding structures, materials, acts, and equivalents of all means or step plus function elements in the claims below, if any, are intended to include any structure, material, or act for performing the function in combination with other claimed elements as specifically claimed. The description of the present invention has been presented for purposes of illustration and description, but is not intended to be exhaustive or limited to the invention in the form disclosed. Many modifications and variations will be apparent to those of ordinary skill in the art without departing from the scope and spirit of the invention. The embodiment was chosen and described in order to best explain the principles of the invention and the practical application, and to enable others of ordinary skill in the art to understand the invention for various embodiment with various modifications as are suited to the particular use contemplated.

1. A method of reading a posting list, the method comprising:

determining by a processor a size of a posting list as part of searching an inverted index;

segmenting the posting list for reading by the processor into a plurality of segments based on the size; and

reading by the processor each of the plurality of segments into memory.

2. The method of claim **1**, wherein the segmenting is performed if the size is larger than a predetermined size.

3. The method of claim **2**, further comprising reading by the processor all of the posting list at once if the size is the predetermined size or smaller.

4. The method of claim **1**, wherein the segmenting is performed using at least one predetermined segment size.

5. The method of claim **1**, wherein the segmenting is performed for at least one segment using at least one estimated segment size.

6. The method of claim **5**, wherein at least one actual read size for the at least one segment is greater than or equal to at least one of the at least one estimated segment size, the method further comprising storing by the processor the at least one actual read size in a data structure for reuse.

7. The method of claim **1**, wherein the posting list includes a plurality of relevance indicators for a plurality of postings in the posting list, wherein the posting list is sorted into descending order by relevance, and wherein the segmenting is performed for at least one segment using at least one relevance indicator.

8. The method of claim **7**, wherein the reading comprises reading for the at least one segment until reaching a relevance indicator lower than the at least one relevance indicator, the method further comprising storing by the processor a read size for the at least one segment in a data structure for reuse.

9. The method of claim **7**, wherein the plurality of relevance indicators comprises a plurality of scores.

10. A computer system for reading a posting list, the computer system comprising:

a memory; and

a processor in communication with the memory to perform a method, the method comprising:

determining a size of a posting list as part of searching an inverted index;

segmenting the posting list for reading into a plurality of segments based on the size; and

reading each of the plurality of segments into memory.

11. The system of claim **10**, wherein the segmenting is performed if the size is larger than a predetermined size.

12. The system of claim **10**, further comprising reading all of the posting list at once if the size is the predetermined size or smaller.

13. The system of claim **10**, wherein the segmenting is performed using at least one predetermined segment size.

14. The system of claim **10**, wherein the segmenting is performed for at least one segment using at least one estimated segment size.

15. The system of claim **14**, wherein at least one actual read size for the at least one segment is greater than or equal to at least one of the at least one estimated segment size, the method further comprising storing the at least one actual read size in a data structure for reuse.

16. The system of claim **10**, wherein the posting list includes a plurality of relevance indicators for a plurality of postings in the posting list, wherein the posting list is sorted into descending order by relevance, and wherein the segmenting is performed for at least one segment using at least one relevance indicator.

17. The system of claim **16**, wherein the reading comprises reading for the at least one segment until reaching a relevance indicator lower than the at least one relevance indicator, the method further comprising storing a read size for the at least one segment in a data structure for reuse.

18. The system of claim **16**, wherein the plurality of relevance indicators comprises a plurality of scores.

19. A program product for reading a posting list, the program product comprising:

a storage medium readable by a processor and storing instructions for execution by the processor for performing a method, the method comprising:

determining a size of a posting list as part of searching an inverted index;

segmenting the posting list for reading into a plurality of segments based on the size; and

reading each of the plurality of segments into memory.

20. The program product of claim **19**, wherein the segmenting is performed if the size is larger than a predetermined size.

21. The program product of claim **20**, further comprising reading all of the posting list at once if the size is the predetermined size or smaller.

22. The program product of claim **19**, wherein the segmenting is performed using at least one predetermined segment size.

23. The program product of claim **19**, wherein the segmenting is performed for at least one segment using at least one estimated segment size.

24. The program product of claim **23**, wherein at least one actual read size for the at least one segment is greater than or equal to at least one of the at least one estimated segment size, the method further comprising storing the at least one actual read size in a data structure for reuse.

25. The program product of claim **19**, wherein the posting list includes a plurality of relevance indicators for a plurality of postings in the posting list, wherein the posting list is sorted into descending order by relevance, and wherein the segmenting is performed for at least one segment using at least one relevance indicator.

26. The program product of claim **25**, wherein the reading comprises reading for the at least one segment until reaching a relevance indicator lower than the at least one relevance indicator, the method further comprising storing a read size for the at least one segment in a data structure for reuse.

27. The program product of claim **25**, wherein the plurality of relevance indicators comprises a plurality of scores.

* * * * *