[MS-FSIXDS]:
Index Data Structures

Revision Summary

Date	Revision History	Revision Class	Comments
02/19/2010	1.0	Major	Initial Availability
03/31/2010	1.01	Editorial	Revised and edited the technical content
04/30/2010	1.02	Minor	Updated the technical content
06/07/2010	1.03	Editorial	Revised and edited the technical content
06/29/2010	1.04	Editorial	Changed language and formatting in the technical content.
07/23/2010	1.04	No change	No changes to the meaning, language, or formatting of the technical content.
09/27/2010	1.04	No change	No changes to the meaning, language, or formatting of the technical content.
11/15/2010	1.04	No change	No changes to the meaning, language, or formatting of the technical content.
12/17/2010	1.05	Minor	Clarified the meaning of the technical content.
03/18/2011	1.05	No change	No changes to the meaning, language, or formatting of the technical content.
06/10/2011	1.05	No change	No changes to the meaning, language, or formatting of the technical content.
01/20/2012	1.05	No change	No changes to the meaning, language, or formatting of the technical content.
04/11/2012	1.05	No change	No changes to the meaning, language, or formatting of the technical content.
07/16/2012	1.05	No change	No changes to the meaning, language, or formatting of the technical content.

Table of Contents

1 Introduction

This document specifies the Index Data Structures, as well as the file format and hierarchy of the files that represent the search index in an enterprise search service.

Sections 1.7 and 2 of this specification are normative and can contain the terms MAY, SHOULD, MUST, MUST NOT, and SHOULD NOT as defined in RFC 2119. All other sections and examples in this specification are informative.

1.1 Glossary

The following terms are defined in [MS-GLOS]:

Augmented Backus-Naur Form (ABNF)
little-endian
MD5 hash
schema object
UTF-8

The following terms are defined in [MS-OFCGLOS]:

attribute vector
backup indexer node
Boolean occurrences
content collection
context catalog
datetime
document identifier
document summary
dynamic rank
index partition
index schema
indexing component
item
managed property
master indexer node
position occurrences
property context
property index
query matching component
query matching node
query refinement
search query
search service application
token
token ordinal number

The following terms are specific to this document:

occurrence file: A file that contains information about the position and occurrence of a token in indexed items.

MAY, SHOULD, MUST, SHOULD NOT, MUST NOT: These terms (in all caps) are used as described in [RFC2119]. All statements of optional behavior use either MAY, SHOULD, or SHOULD NOT.

1.2 References

References to Microsoft Open Specifications documentation do not include a publishing year because links are to the latest version of the technical documents, which are updated frequently. References to other documents include a publishing year when one is available.

1.2.1 Normative References

We conduct frequent surveys of the normative references to assure their continued availability. If you have any issue with finding a normative reference, please contact dochelp@microsoft.com. We will assist you in finding the relevant information. Please check the archive site, http://msdn2.microsoft.com/en-us/library/E4BD6494-06AD-4aed-9823-445E921C9624, as an additional source.

[MS-FSCX] Microsoft Corporation, "Configuration (XML-RPC) Protocol Specification".

[MS-FSDQE] Microsoft Corporation, "Distributed Query Execution Protocol Specification".

[MS-FSFIXML] Microsoft Corporation, "FIXML Data Structure".

[MS-FSIPA] Microsoft Corporation, "Index Publication and Activation Protocol Specification".

[MS-FSRFC] Microsoft Corporation, "Remote File Copy Protocol Specification".

[MS-FSRFCO] Microsoft Corporation, "Remote File Copy Orchestration Protocol Specification".

[MS-FSSCFG] Microsoft Corporation, "Search Configuration File Format Specification".

[RFC1950] Deutsch, P., and Gailly, J-L., "ZLIB Compressed Data Format Specification version 3.3", RFC 1950, May 1996, http://www.ietf.org/rfc/rfc1950.txt

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, http://www.rfc-editor.org/rfc/rfc2119.txt

[RFC5234] Crocker, D., Ed., and Overell, P., "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008, http://www.rfc-editor.org/rfc/rfc5234.txt

1.2.2 Informative References

[MS-GLOS] Microsoft Corporation, "Windows Protocols Master Glossary".

[MS-OFCGLOS] Microsoft Corporation, "Microsoft Office Master Glossary".

1.3 Structure Overview (Synopsis)

This document specifies the file sets produced on a **master indexer node** and consumed by one or more instances of a **backup indexer node** or **query matching node** .

These file sets are only copied over the network when the **search service application** is set up to have a query matching node or backup indexer node on a different computer than the master indexer node.

The content of these files is transferred using the remote file copy protocol, as described in [MS-FSRFC]. There are five different file sets. Each file set is associated with the subscription name, as described in the Remote File Copy Orchestration Protocol in [MS-FSRFCO] section 3.2.1. The Remote File Copy Orchestration Protocol in [MS-FSRFCO] determines which files to copy. The Remote File Copy Protocol [MS-FSRFC] performs the actual file transfer of each individual file in the file sets.

These file sets are the following:

- Index
- Dictionary
- State
- Generation
- Counter

The file structures described in section 2 follow this file set hierarchy. All files that belong to one file set are described in the same subsection in section 2.

1.3.1 Index File Set

Consisting of files used by the query matching component for matching purposes, the files in the index file set are part of an index partition.

The master indexer node chooses the set of items to include in an index partition, then generates the index files, and copies the files to query matching nodes and backup indexer nodes.

Multiple, different index file sets can be copied to the same query matching node, and each index file set will be processed by a query matching component. The index file sets are stored on a query matching node, as described in the following diagram.

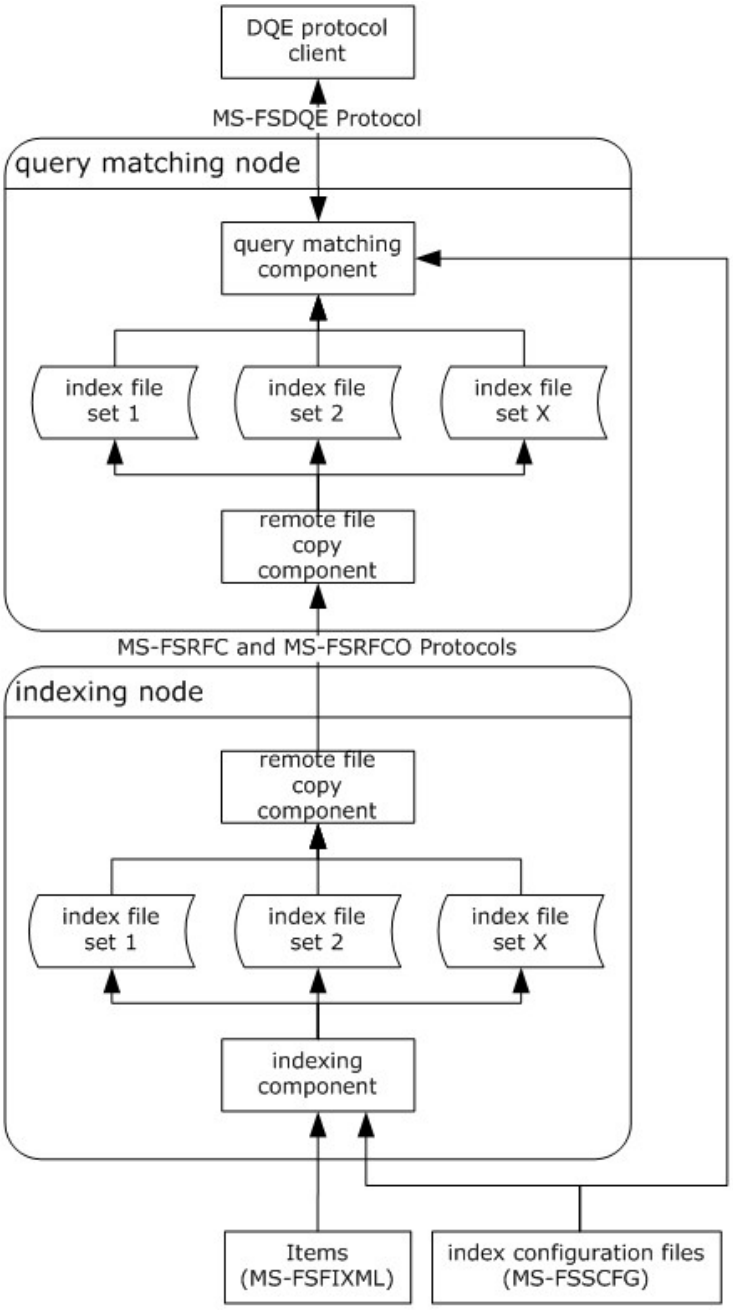


Figure 1: Index file set placement in system

Each separate index file set is stored in a unique path. The path contains a part that describes which index partition number is associated with the index file set. This is the PP part of the path for the index file sets specified in section 2.1. The path also contains a timestamp, as described in the TTTT part of the path.

The index files for one index file set are stored in a directory hierarchy partially governed by the configuration file index.cf, as described in [MS-FSSCFG] section 2.9.

The **query matching component** uses these files for matching **search queries** submitted using the protocol described in [MS-FSDQE].

The index files are classified into index file subsets, using boxes, and hierarchically structured, using lines, as described in the following diagram. For each index file subset the cardinality information about the line pointing into the group of files describes whether there can be only one instance (1..1) or if there can be multiple (1..*) instances of the index file subset. The boxes that are placed on the same vertical position are on the same directory level. A box that is beneath and to the right of another box is one directory level lower.

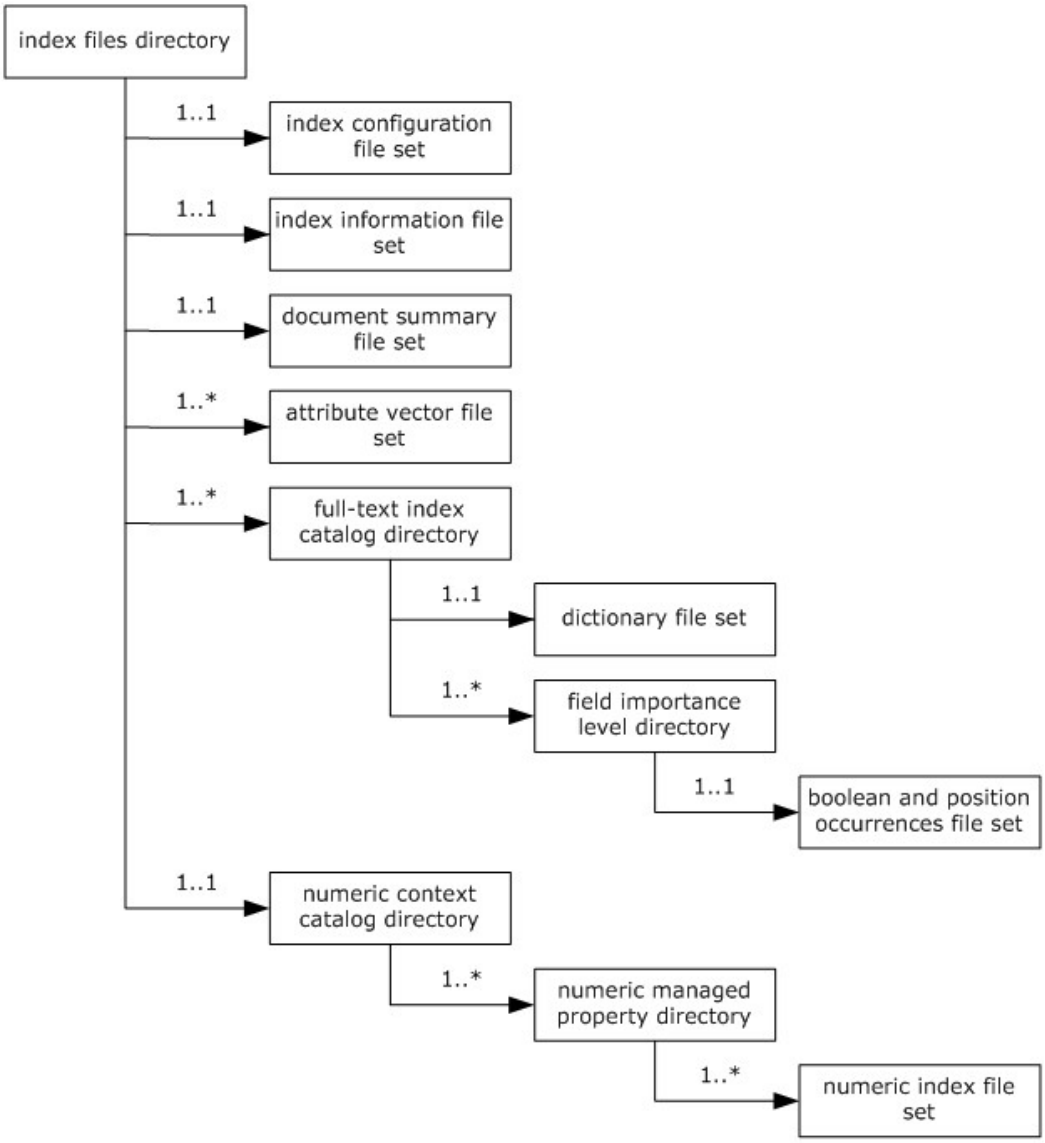


Figure 2: Overview of index files

Each index file subset describes the base for specific features of the query matching component.

- Index configuration and information files: contain information about how the index is structured, which enables query matching components and indexing components to use these files.
- **Document summary** file subset: For each item that matches a query and is to be displayed on the search front-end, parts of item information in these files is processed and sent to the search front end during a document summary request.
- **Attribute vector** file subset: contains attribute vector information used by the query matching component to sort, rank and collapse search results, and also to produce an aggregated data set that will be presented by the search front end.
- Full-text index **context catalog** directory: For each full-text index context catalog described in the index configuration a directory is created. Within this directory a set of dictionary files and sub-directories with **Boolean occurrences** and **position occurrences** files are created. This file set contains the inverted index files for textual keyword lookup.
- Dictionary file subset: contains a dictionary of all **tokens** with extra information for each token, such as the number of items the token appeared in, and the total number of occurrences of the token in the index. This information is used by the query matching component for ranking purposes and to determine which parts of the Boolean occurrence or position occurrences files to read.
- Boolean and position occurrences file subset: contains information about the items in which the token exists. The number of times a token occurs, property index and position information for the token in each item is also contained in these files, and this is used to calculate the rank of each item.
- Numeric context catalog directory: is a directory that contains all the numeric managed property directories.
- Numeric managed property directory: for each numeric managed property, a directory of this type is made, that contains the numeric index file set for the numeric managed property.
- Numeric index file subset: contains information that is used by the query matching component to evaluate which items match the numeric parts of a query. This file set contains the inverted index files for numeric value lookup.

1.3.2 Dictionary File Set

The dictionary file set consists of a subset of the files in an index file set, as described in section 1.3.1. The indexing component can merge dictionary files of one or more index partitions into one global dictionary file set. This global dictionary file set represents the full set of tokens for the index partitions that were included.

This merged dictionary file set can then be used by the query matching component to look up the term frequency for each token and how many occurrences there were in total for that token. These global dictionary count fields are used by the query matching component to calculate the **dynamic rank**.

1.3.3 State File Set

On each indexing node, the indexed items are partitioned into a disjointed set of index partitions. The indexing nodes use the state file set to identify the currently active set, or generation, of index partitions. The state file set consists of the following files:

- An index set generation file that describes which index file sets are currently active.

- An index set stamp file that describes the time when the index set became active.
- Index partition stamp files, one per index partition that describes the time when the index partition was created.
- Index partition valid files, one per index partition that describes whether or not the index partition is valid.

1.3.4 Generation File Set

When a set of changes has occurred to the set of indexed items in an index partition, a new index generation can be created. The set of changes is decided by the indexing component. Each index generation has an associated index file set, but a new index generation does not require a new index file set. If items are removed, for example, the new index generation reuses the index file set of the previous generation and generates an additional exclusion list. The removed items are then excluded by the query matching component using this additional exclusion list.

The generation file set consists of the following files:

- A time stamp file that describes the creation time of the index generation.
- An exclusion list file that describes which items have been submitted for exclusion on the query matching nodes.
- A sorted item file that describes the items contained in the index partition, and whether or not they have been submitted for exclusion on the query matching nodes.

1.3.5 Counter File Set

The counter file set is used by the master indexer to determine when to re-index an index partition. This file set is also used by backup indexer nodes. There is one counter file set per index partition.

The counter file set consists of the following files:

- A counter file that describes how many times the index partition has been indexed.
- A counter file that describes how many times the index partition has been activated.
- A counter file that describes how many times the index partition has been both indexed and activated.
- Time stamp files that describes the time when the preceding counter files were created.

1.4 Relationship to Protocols and Other Structures

The following references describe related protocols and structures:

- [MS-FSRFC] describes the protocol used to copy the files that contain the data structures.
- [MS-FSRFCO] describes the protocol that determines the file sets to copy.
- [MS-FSSCFG] describes the index configuration files that control the layout of the index.
- [MS-FSDQE] describes the protocol for the query matching component that reads the files.
- [MS-FSFIXML] describes the item structure that the indexing component typically uses for input.

1.5 Applicability Statement

The file names, directory structure and file data structures in this specification are applicable only to the following cases:

- Copying these files to a protocol server that uses the protocol, as described in [MS-FSRFC].
- Processing these files on a search server that uses these files and returns results based on this processing. The requests and results use the protocol, as described in [MS-FSDQE].

1.6 Versioning and Localization

None.

1.7 Vendor-Extensible Fields

None.

2 Structures

Each file is specified in each subsection in this section.

In addition, the exact file name and path in the directory hierarchy is specified in the beginning of the description for each file type. This makes it possible to map the specified file in a directory hierarchy back to its section in the document. The path is based on the root directory for the file group. The **PP**, **TT**, **TTTT**, **FQDN** and **NN** notation used in the path specifications for the following sections is specified in the subscription mapping in [MS-FSRFCO] section 3.2.1.

A file name without a directory prefix means that the file is in the root directory of the file set. All files and paths MUST use the uppercase or lowercase format as specified in the file name and path for each type of file.

Some file types can exist in multiple paths. These files have a path specification that includes a variable. This variable is explained in the overview section for the file set.

2.1 Index File Set

Specifies the directory structure, file naming and the internal format of each type of file in each index file set.

The index configuration specifies the exact directory structure and file naming of a specified index.

2.1.1 Common Formats

The index files specified in this section share some common information.

2.1.1.1 Byte Ordering

All multi-byte numeric fields in the search index files MUST be represented in **little-endian** order. This order is used for the information fields that originated from the original items, and also the generated numeric fields such as count and offset values in the index files to the data files.

For string fields in the index and data files, the order of the bytes is specified for each file type in the following sections. For some strings the representation MUST be in ASCII, and for others it MUST be in **UTF-8**.

2.1.1.2 Data Types and Internal Format Conversion

2.1.1.2.1 Internal Text Data Type

Managed properties of type text and Boolean MUST be represented in each full-text index context catalog and attribute vector files as type string.

The content of a managed property of type text MUST be stored in the string attribute vector files directly in the original byte by byte representation.

Managed property content MUST also be stored in the relevant property context catalog files. An additional suffix character can be added to the token for implementation-specific purposes. The indexing component may use this to mark tokens, as specified in [MS-FSDQE] section 2.2.6.1.

Each string in the string attribute vector files MUST be in UTF-8 format and delimited by a zero termination character, 0x00.

The content of a managed property of type Boolean is indexed as the token **true** or **false**. In the attribute vector files, the token string is terminated with the zero termination character, 0x00.

A string in UTF-8 format is specified using the following **Augmented Backus-Naur Form (ABNF)** grammar, as specified in [RFC5234].

```
UTF8-string          = 1*UTF8-char
UTF8-char            = ASCII-char / UTF8-non-ASCII-chars
ASCII-char           = %x21-7E
UTF8-non-ASCII-chars = (%xC0-DF 1UTF8-CONTENT) /
                       (%xE0-EF 2UTF8-CONTENT) /
                       (%xF0-F7 3UTF8-CONTENT) /
                       (%xF8-FB 4UTF8-CONTENT) /
                       (%xFC-FD 5UTF8-CONTENT)
UTF8-CONTENT         = %x80-BF
```

This UTF-8 string ABNF grammar is used in some of the file specifications using ABNF in the following sections, and it MUST be added to the file ABNF to complete it.

2.1.1.2.2 Internal Numeric Data Type

Managed properties of type integer, **float**, decimal, and **datetime** are converted to a specific numeric representation internally in the numeric attribute vector and integer index files.

The table in this section specifies how data type fields in an **item** are converted and stored by the **indexing component** into the internal representation in the numeric attribute vector and integer index files. The indexing component typically reads these data type fields for an item stored as FIXML, as specified in [MS-FSFIXML].

The indexing component and query matching component read the maptransform.xml file, as specified in [MS-FSSCFG] section 2.2, to determine the mapping from external representation to internal representation. The information in the maptransform.xml file and in the following table is used to convert numeric data types to the internal representation.

The schema abstract data model data type is the data type of a managed property, as specified in [MS-FSSCFG] section 1.3.2.1 and described in the following table.

Schema abstract data model data type	Internal type, as specified in maptransform.xml	Conversion explanation
integer	INT	For integer occurrence index files, the integer key fields are specified using 64-bit integers in little-endian order. For positive values, the highest bit MUST be set to 1; for negative values, the highest bit MUST be set to 0. For attribute vector files, the integer is specified using 64-bit signed integers in little-endian order.
float	FLOAT2B	A managed property of type **float** is specified using the **FLOAT2B** data type specified in the maptransform.xml file. In this data type 52 bits are used for the mantissa, 11 bits for the exponent, and 1 for the sign-bit. The numeric base MUST be 2. The following algorithm converts a floating point number *num* to the internal format.

Schema abstract data model data type	Internal type, as specified in maptransform.xml	Conversion explanation

		<pre>SET maxMan = 2^52 - 1
		SET maxExp = 2^10 - 1
		SET manFactor = 10^floor(log10(maxMan)) ; = 10^15
		SET manOffset = 0
		
		SET exp = ceil(log2(abs(num)))
		SET frac = num*2^(-exp)
		IF abs(frac) >=1 THEN
		  SET frac=frac/2
		  SET exp=exp+1
		END IF
		
		IF abs(exp) > maxExp THEN
		  SET frac = sign(frac)
		  SET exp = maxExp * sign(exp)
		ENDIF
		
		SET frac = (frac * manFactor) + manOffset
		SET frac = minimum(frac,maxMan) * sign(frac)
		
		IF frac == 0 THEN
		  SET exp = 0
		  SET frac = 0
		ELSE
		  SET exp = maxExp + (sign(frac) * exp)
		END IF
		
		SET sign = 1 - sign(frac)
		IF frac < 0 THEN
		  SET frac = maxMan + frac
		END IF
		
		; Use shift operation to put the parameters inside an int64
		SET transformed = (sign << 63) | (exp << 52) | frac</pre>
		The transformed value will be an integer of 64 bits.

decimal	DECIMAL_NAV (attribute vectors) DECIMAL (integer index)	The decimal fields MUST be converted to the internal format as follows: For a specific **managed property**, the number of decimal places is specified in the **datatype** reference for the corresponding **field** entry in the maptransform.xml file, as specified in [MS-FSSCFG] section 2.2.3.3. The decimal field is multiplied with 10^x, where x is the **decimal-places** field for the managed property. This field is then converted into a signed **int64** field. Any excessive fractional parts after the multiplication MUST be discarded. For example, with decimal-places set to 3, the original value 3.14159 will be converted to the integer 3141 in the internal format when indexed. The same transformation MUST be performed on the query side.

Schema abstract data model data type	Internal type, as specified in maptransform.xml	Conversion explanation

datetime	INT	The **datetime** fields are specified as a 64-bit, that is, 8 bytes unsigned integer. The integer field represents the number of 100 nanoseconds after the date -29000-01-01T00:00:00.0000000Z.
		The conversion MUST be performed based on the following rules:
		▪ Each second has 10*1000*1000 ticks, so each tick is 100 nanoseconds long.
		▪ Each minute has 0-59 seconds.
		▪ Each hour has 0-59 minutes.
		▪ Each day has 0-23 hours.
		▪ Each day of the year is represented in the range 0-365, as specified in the following day offset table: January 1 is day 0. February 1 is day 31. March 1 is day 60. April 1 is day 91. May 1 is day 121. June 1 is day 152. July 1 is day 182. August 1 is day 213. September 1 is day 244. October 1 is day 274. November 1 is day 305. December 1 is day 335.
		Note that February is assumed to always have 29 days (leap years). Other day of year offsets can be calculated by the day offset into each month. For example, August 10 is day offset 213 + 10-1 = 222.
		▪ Each year is represented as a number between 0 and 58000, where 0 represents the year 29000 BC.
		The number of ticks for each time field is as follows:
		`TICKS_IN_SECOND = 10 * 1000 * 1000`
		`TICKS_IN_MINUTE = TICKS_IN_SECOND * 60`
		`TICKS_IN_HOUR = TICKS_IN_MINUTE * 60`
		`TICKS_IN_DAY = TICKS_IN_HOUR * 24`
		`TICKS_IN_YEAR = TICKS_IN_DAY * 366`
		The internal field is calculated as specified in the following formula:
		`ticks = years after 29000 BC * TICKS_IN_YEAR + day of year * TICKS_IN_DAY + hour of day * TICKS_IN_HOUR + minute of the hour * TICKS_IN_MINUTE + second into the minute * TICKS_IN_SECOND + 100th nanoseconds into the second`
		Positive leap seconds MUST be represented as the closest lower accepted datetime representation. For example, 2008-31-12T23:59:60 is represented as 2008-31-12T23:59:59.

2.1.1.3 Producer Information

The content of each item MUST be split, processed and stored in the appropriate index file by the indexing component. Each item contains the required content based on the index schema, so that all the features in the query matching component have the necessary information ready.

The indexing component can use the content format, as specified in [MS-FSFIXML], as the input source for items. Alternative indexing components can use other input formats.

During processing of the items, the indexing component extracts the following information, and sets up the index files so that the connections between them are as specified for each file type.

- **Token:** Each token is extracted from the text-based content in the item. If using the format for input, as specified in [MS-FSFIXML], the tokens are contained within the **context** elements of text **catalog** elements. Text catalogs are specified as type text in the indexConfig.xml file, by the CT_catalog element, as specified in [MS-FSSCFG] section 2.8.3.3. The tokens are separated by the ASCII character 0x20, except for catalogs that have the **ST_substringRange** field set to greater than zero, as specified in [MS-FSSCFG] section 2.8.4.3.
- **Numeric value:** This is extracted from each section in the item that represents a numeric managed property. For each numeric managed property specified in the **index schema**, the indexConfig.xml file MUST contain a corresponding context element in the catalog of type integer with name "bi1", as specified in [MS-FSSCFG] section 2.8.5.2.1. If using the format for input, as specified in [MS-FSFIXML], the numeric fields are contained in the context elements of the catalog named "bi1".
- **Token identifier:** This is a 32-bit numeric field that maps to a specific token. It is calculated based on the alphabetic order of each token. The alphabetically ordered first token MUST have token identifier value zero, the next token has token identifier value one, and so on.
- **Document identifier:** This is calculated during indexing based on the order of items. The first item indexed has **document identifier** zero, the second item indexed has document identifier one, and so on.
- **Attribute vector data:** This MUST be extracted from the sections of the item that represent the content that will be used for sorting, ranking, field collapsing and for computing aggregated data sets. In the format as specified in [MS-FSFIXML], the attribute vector information for one attribute vector is contained within the **avField** elements of each **attrVec** element.
- **Document summary:** This MUST be extracted from the sections of the item that represent the content to present to the end user. In the format, as specified in [MS-FSFIXML], the document summaries for an item is contained within the **sField** elements of the **summary** element.

2.1.1.4 Consumer Information

Each index file set contains the necessary information so that the query matching component can perform lookup of information based on the following.

The query matching component MUST retrieve the following identifier values in the index files.

- **Token:** Extracted from the text parts of the query.
- **Numeric value:** Extracted from the numeric parts of the query.
- **Token identifier**. Located in the dictionary files.
- **Document identifier:** Located in the **occurrence files**.

- **Attribute vector data:** Located in the attribute vector data files based on the document identifiers in the result set.
- **Document summary:** Located in the document summary files based on the document identifier.

2.1.2 Index Configuration File

The index configuration files MUST always exist for an index file set and are stored in the top level of the directory hierarchy of an index as "PP\index_TTTT\index_data".

The file names and directory structure of the index data files are partially derived from the index configuration files, as specified in [MS-FSSCFG]. The configuration files are the following:

- index.cf
- rank.cf
- summary.cf
- summary.map

These files are included in the index file set that is copied over the network, as specified in [MS-FSRFC] and [MS-FSRFCO].

The indexing component produces these files initially for each index file set when the other index files are generated. If the index schema is updated after system installation, because the system administrator has run an index schema reconfiguration, one or more of these files can change on the configuration service node. In that event, the changed files MUST be copied to all the directories on all the system nodes that have a copy of these files. This copy process is triggered by the configuration service, as specified in [MS-FSCX]. The indexing and query matching components on each node receive the files and copy them to the paths where the previous version of the files were placed.

The consumer of the index configuration files, the query matching component, uses the most current configuration files to be able to read the structure of the index partition and produce query results according to the current configuration.

2.1.3 Index Partition Tuning File

Path and file name for this file MUST be "PP\index_TTTT\index_data\indextune.cf", which MUST exist. Its content is static and is specified using the following ABNF grammar:

```
indextune     = "#" LF
```

2.1.4 Indexed OK Stamp File

The path and file name for this file MUST be "PP\index_TTTT\index_data\IndexedOK".

It contains the number of items in the index partition. The number of items in the index partition is used by the query matching and indexing components to validate the other index partition files.

The file is specified using the following ABNF grammar:

```
indexed-ok     = item-count LF
```

```
item-count     = 1*10DIGIT
```

item-count: The number of items in the index partition. The numeric value MUST be in the range 0 to 4294967295 (0xFFFFFFFF).

2.1.5 Merged Findex Done Stamp File

The path and file name for this file MUST be "PP\index_TTTT\index_data\merged\.findex_done". It specifies that all files in the index partition are complete and consistent.

It MUST be present and MUST be empty, which means a length of size 0.

The file system timestamp for this file specifies when the indexing component successfully finished processing this index partition.

2.1.6 Stamp Text File

Path and file name of this file MUST be "PP\index_TTTT\index_data\stamp.txt".

It specifies that all files in the index partition are complete and consistent, and also specifies when the index partition was generated.

This file MUST be present, and MUST contain the timestamp of the index partition. This is in addition to the file system timestamp of the .findex_done file. The timestamps of these two files depends on the implementation and can be different.

The file is specified using the following ABNF grammar:

```
stamp-txt       = index-timestamp
index-timestamp = 1*10DIGIT
```

index-timestamp: The field specifies the point in time when the process that generates the index finished successfully. The time is represented as the number of seconds after 1970-01-01T00:00:00 UTC.

2.1.7 Attribute Vector Indexing Information File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged\attributevector-indexing.txt".

This file is separate from the attribute vector file set, as it is global for all attribute vector file set instances. See section 1.3 for information about the attribute vectors file set.

This file contains the file size of the largest attribute vector data file that was created during processing of attribute vector information input. The indexing component uses the size information as a hint as to whether or not to load the attribute vector into memory.

This file MUST be present and is specified using the following ABNF grammar:

```
attribute-vector-indexing-txt  = peak-memory-usage
peak-memory-usage              = 1*20DIGIT
```

The peak-memory-usage field is calculated as specified in the following algorithm, where the set of **attribute_vectors** is the list of lines prefixed with the **attributevector** setting within the index.cf file, as specified in [MS-FSSCFG] section 2.9.2.4. The **attr-name** setting of each **attributevector** is specified as the second word on each **attributevector** line. The file name of the attribute vector data file is created by appending the suffix ".dat" to the **attr-name string**. The **attribute_vector.size** field is the file size of each attribute vector data file, as follows.

```
SET maxsize = 0
FOREACH attribute_vector IN attribute_vectors
   IF attribute_vector.size > maxsize THEN
      SET maxsize = attribute_vector.size
   END IF
END FOREACH
```

2.1.8 Attribute Vector Search Information File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged\attributevector.txt".

This file is separate from the attribute vector file set, as it is global for all attribute vector file set instances.

This file represents the total memory usage, in bytes, required to read the attribute vector files into memory. The files that are loaded into memory are the following:

- For single-valued attribute vectors, the .eidx and .sudat files.
- For multi-valued attribute vectors, the .idx, .eidx, .sudat files.

The query matching component reads this file to determine whether it can load all attribute vector information into memory. If it cannot load everything, the search process MUST not begin.

This file MUST be present and is specified using the following ABNF grammar:

```
attribute-vector-txt         = total-required-memory-usage
total-required-memory-usage  = 1*20DIGIT
```

total-required-memory-usage: The sum of the **enum.ramusage** field in each attribute vector information file, as specified in section 2.1.13.4. This value can be calculated as specified in the following algorithm.

See section 2.1.13 for how the set of attribute vectors are determined. This set MUST be input to the attribute_vectors value specified in the following example. The **attribute_vector_info_file** name is generated by appending the suffix ".info" to the end of each attribute vector name.

The **attribute_vector.enum_ramusage** field is the value of the enum.ramusage line in each attribute vector information file, as follows.

```
SET totsize = 0
FOREACH attribute_vector_info_file IN attribute_vectors
   SET totsize = totsize + attribute_vector_info_file.enum_ramusage
END FOREACH
```

2.1.9 Document Identifier Map File

Path and file name of this file MUST be "PP\index_TTTT\index_data\urlmap.txt".

The file specifies the item internal identifier, store identifier and document identifier for all items in the index partition. The format and definition of the item internal identifier, store identifier and document identifier are specified after the following ABNF.

This file MUST NOT exist for indexes with zero items. An index with zero items is a valid index partition. This file MUST exist for indexes with 1 or more items.

The file contains entries of internal name, store identifier and document identifier, as specified in the following ABNF.

The file can contain duplicates of the item internal identifier. The query matching component excludes duplicates and includes only the item with the highest document identifier value.

The file is specified using the following ABNF grammar:

```
urlmap-txt       = 1*2147483647(internal-id "," store-id SP doc-id LF)
internal-id      = docname-checksum "_" collection-name
store-id         = *(ALPHA / DIGIT / "_" / "\" / ".")
doc-id           = 1*10DIGIT
docname-checksum = 32HEXDIG
collection-name  = *(ALPHA / DIGIT / "-")
```

urlmap-txt: The document identifier map file that contains lines of the specified type. Maximum number of items for an index partition is 2^31 - 1.

internal-id: This string uniquely identifies each item in the index.

store-id: A string that specifies the store from which the item was read by the indexing component. The store identifier is a file name.

doc-id: This contains the internal numeric 32-bit document identifier. This document identifier is used as a lookup key in many of the other index files.

docname-checksum: An **MD5 hash** of the original name of the document.

collection-name: A string that specifies the **content collection** to which the item belongs.

2.1.10 Sorted Document Identifier Map File

Path and file name of this file MUST be "PP\index_TTTT\index_data\urlmap_sorted.txt".

The indexing component uses this file to specify which elements exist in an already created index partition. The file is not used by the query matching component.

This file MUST NOT exist for index partitions with zero items. It MUST exist for index partitions with 1 or more items.

This file is the sorted representation of the document identifier map file. The file MUST be sorted in ascending order as specified by each byte of each internal identifier string.

The file is specified using the following ABNF grammar:

```
urlmap-sorted-txt = 1*2147483647((inc-marker / exc-marker)
                    internal-id "," store-id SP doc-id LF)
exc-marker        = "#"
inc-marker        = "."
internal-id       = docname-checksum "_" collection-name
store-id          = *(ALPHA / DIGIT / "_" / "\" / ".")
doc-id            = 1*10DIGIT
docname-checksum  = 32HEXDIG
collection-name   = *(ALPHA / DIGIT / "-")
```

Most of these ABNF fields are in the Document identifier map file ABNF section, as specified in section 2.1.9.

The only extra fields are the exc-marker and inc-marker characters at the beginning of each line. The indexing component uses these fields to specify whether a doc-id is excluded or included in subsequent index updates.

exc-marker: The document MUST be excluded from later index updates.

inc-marker: The document MUST be included in later index updates.

2.1.11 Version Information File

The path and file name of this file MUST be "PP\index_TTTT\index_data\version.txt".

This file specifies the index format version of the index partition. The version MUST contain the value "1.1". The file is specified using the following ABNF grammar:

```
version-txt    = version-string LF ok-string LF
version-string = "1.1"
ok-string      = "Ok"
```

2.1.12 Range Information File

Path and file name of this file MUST be "PP\index_TTTT\index_data\range".

This file enables the indexing component to specify which store identifiers are used as input for the items in the index partition.

This file MUST be present.

The start-range and end-range fields specify the range of items to include in the index partition.

The values are used only by the indexing component.

The file named "range" is specified using the following ABNF grammar:

```
range              = items-in-index SP start-range SP end-range LF
items-in-index     = 1*10DIGIT
start-range        = 1*10DIGIT
end-range          = 1*10DIGIT
```

range: The range file with zero or more entries.

items-in-index: The number of items in the index partition. A zero indicates there are no items in the index.

start-range: The start range field for the items included in the index partition.

end-range: The end range field for the items included in the index partition.

2.1.13 Attribute Vector Files

The attribute vector files are used for **query refinement** and sorting functionality in the query matching component.

Query refinement or sorting features can be enabled for a managed property, as specified in [MS-FSSCFG] section 1.3.2. If one of those features is enabled, an attribute vector file set MUST be generated that contains the query refinement or sort values for each item in the index.

An attribute vector MUST be of the type "query refinement" or the type "sorting".

Query refinement attribute vectors MUST be multi-valued, and always contain zero or more values per item.

Sorting attribute vectors MUST be single-valued, and always contain exactly one value per item.

The set of attribute vectors that are contained in an index partition is specified in the index.cf file, as derived from the indexConfig.xml file. This is specified by the **attributeVector** element, as specified in [MS-FSSCFG] section 2.8.3.30.

Each attribute vector file set consists of the index and data files, and also the attribute vector information file, as specified in section 2.1.13.4. This file specifies for each attribute vector the following information:

- The data type of the attribute vector data file.
- Whether the attribute vector is multi-valued or single-valued.
- The memory requirements for loading various combinations of the attribute vector index and data files into memory.

The files in the attribute vector file set are related as specified in the following figure.

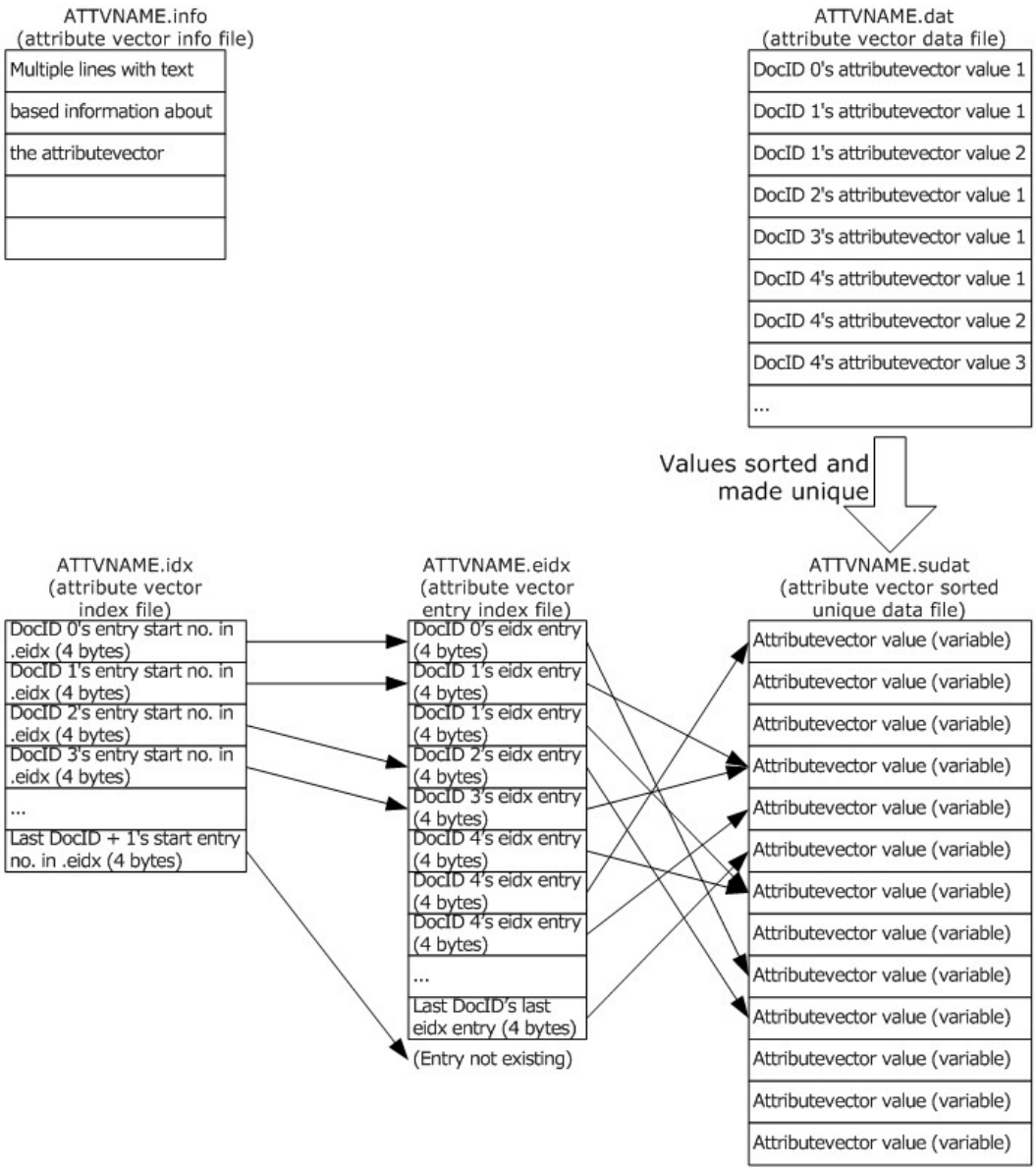


Figure 3: Attribute vector files relationship

The file name prefix of each attribute vector file set is derived from the **name** attribute of the **attributeVector** element in indexConfig.xml setting, as specified in [MS-FSSCFG] section 2.8.3.30. This file name prefix is referred to as ATTVNAME in the following individual attribute vector file specifications. The file name prefix can be ba*, as used in the preceding figure. Each of the attribute vector files has a suffix that specifies the file type. The file type MUST be ".idx", ".eidx", ".sudat", ".dat", or ".info".

The relationship between the files is as follows:

- The attribute vector index file (ba*.idx) MUST be used for lookup in the attribute vector entry index file (ba*.eidx).

- The attribute vector entry index file (ba*.eidx) MUST be used for lookup in the attribute vector sorted unique data file (ba*.sudat).
- The attribute vector sorted unique data file (ba*.sudat) contains the attribute vector fields that will be used by the query matching component. The content of this file is derived from the attribute vector data file (ba*.dat), and contains the unique (non-duplicate) values of this file. The values are sorted in ascending order.
- The attribute vector data file (ba*.dat) is only used for indexing purposes, to generate the attribute vector sorted unique data file (ba*.sudat).

The following sections contain more information about the relationship between these files.

2.1.13.1 Data File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged\ATTVNAME.dat".

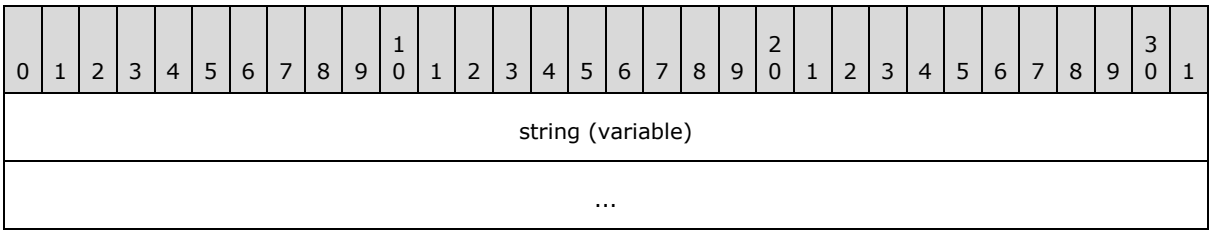
This file contains the attribute vector information for all items for the attribute vector. This is used by the indexing component to enable new index partitions to be created based on existing index partitions. The query matching component does not require this file. Instead it reads the attribute vector information from the sorted unique attribute vector data file, as specified in section 2.1.13.5.

An attribute vector data file MUST be of the type specified by the **type** attribute in the **attributeVector** element, as specified in [MS-FSSCFG] section 2.8.4.14. The type MUST be string, integer, or **float**. The mapping between the managed property data type and the attribute vector data type MUST be as specified in section 2.1.1.2.

There MUST be zero or more attribute vector information entries per item. The number of entries per item is determined in the attribute vector index file, as specified in section 2.1.13.3.

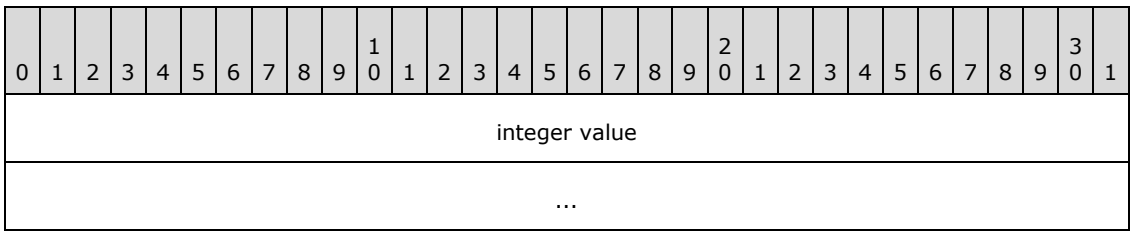
All entries in an attribute vector data file are of type string, **int64** or **float**.

The **string** attribute vector entry is specified as follows.



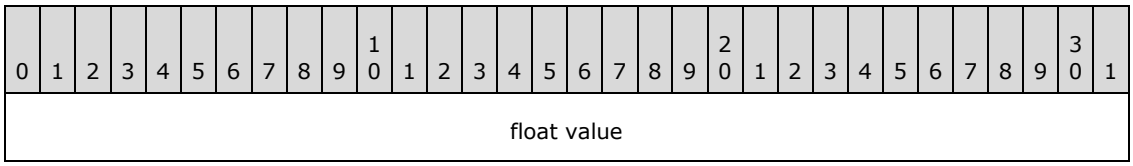
string (variable): The string content for this element, specified in UTF-8 format and terminated by the ASCII character 0x00.

The **int64** attribute vector entry is specified as follows.



integer value (8 bytes): The internal representation of a numeric value. See the conversion explanations specified in section 2.1.1.2 for how to encode or decode the **int64** values.

The **float** attribute vector entry is specified as follows.



float value (4 bytes): The internal representation of a float field. See the conversion explanations specified in section 2.1.1.2 for how to encode or decode the float fields.

2.1.13.2 Entry Index File

The path and file name of this file MUST be "PP\index_TTTT\index_data\merged\ATTVNAME.eidx".

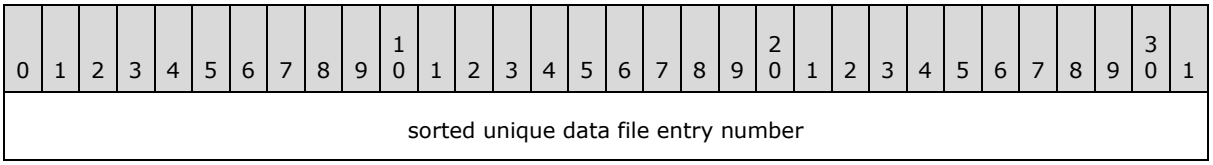
This file enables the query matching component to look up the attribute vector information for a specific item that is stored in the attribute vector sorted unique data file, as specified in section 2.1.13.5.

This file MUST contain one or more entries. The number of entries in this file is the same as the number of entries in the attribute vector sorted unique data file. The entry values in the attribute vector sorted unique data file are specified in section 2.1.13.5.

When the attribute vector that this file represents is single-valued there is no attribute vector index file that references each entry in this file. In such cases, entry number zero in this file is for document identifier zero, entry number one is for document identifier one, and so on.

In the cases where the attribute vector is multi-valued, the attribute vector index file will point to the first entry for each document identifier in this file.

Each entry in the attribute vector entry index file MUST be as specified in the following format.



sorted unique data file entry number (4 bytes): The entry number for the sorted unique data file that belongs to the current entry index element. The attribute vector sorted unique data (.sudat) file consists of one or more entries of fixed or varying size. The query matching component calculates the entry number in that file for each entry when loading that file into memory. For string entries of varying sizes, the delimiter zero specifies the length, so the component can determine where a new entry begins. Each entry in the attribute vector sorted unique data file is therefore implicitly numbered. The first entry in that file has offset number 0, the second string has entry number 1 and so on. This file MUST contain fields that represent the entry number in the corresponding attribute vector sorted unique data file.

2.1.13.3 Index File

The path and file name of this file MUST be "PP\index_TTTT\index_data\merged\ATTVNAME.idx".

This file enables the query matching component to look up how many attribute vector information fields each item has in the attribute vector sorted unique data file, as specified in section 2.1.13.5. Attribute vectors that are single-valued always have one attribute vector information field for each item. Attribute vectors that are multi-valued have one or more attribute vector information fields for each item.

This file only exists for multi-valued attribute vectors. This file is used to calculate how many values each item has and to look up the correct entries in the attribute vector entry index file.

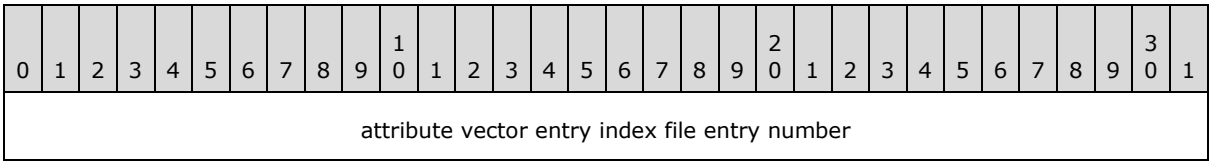
This file contains 32-bit (4-byte) unsigned little-endian order fields that specify the entry number into the attribute vector entry index file where the entries for each item begin.

There MUST be as many entries in this file as there are items in the **index partition** plus one extra entry, which is the last one in this file. The first entry is "0", and the last extra entry contains a value that is one number greater than the total number of entries in the attribute vector entry index file.

The first entry in this file is for document identifier 0, the second entry is for document identifier 1, and so on. The next to last entry in this file is the entry number for the highest numbered document identifier.

The number of entries for an item is calculated by subtracting the entry number for the next document identifier by the entry number of the actual document identifier. The last entry number and the next last entry number in this file are used to calculate the number of attribute vector information fields for the last document identifier. There MUST be zero or more entries per item.

Each entry in the attribute vector index file MUST follow this format.



attribute vector entry index file entry number (4 bytes): The entry number in the attribute vector entry index file for this document identifier. This file contains 32-bit (4 bytes) unsigned little-endian order fields that represent the entry number in the corresponding attribute vector entry index file. Each entry in the attribute vector entry index file is specified using 4 bytes. The last entry in this file contains the total number of entries in the attribute vector entry index file. Therefore the size in bytes of the attribute vector entry index file is 4 bytes multiplied by the value of the last entry number in this file.

The entry numbers in this file increase by zero or more for each consecutive entry.

2.1.13.4 Information File

The path and file name of this file MUST be "PP\index_TTTT\index_data\merged\ATTVNAME.info".

This file specifies metadata about the attribute vector.

This file MUST exist for each ATTVNAME. All elements in this file are specified with ASCII characters. The file is specified using the following ABNF grammar:

```
attrvector-info  = datatype enum-bits enum-maxvalue
                   enum-ramusage format multivalue
                   [offset-bits offset-ramusage]
```

```
                  plain-ramusage [sortsigned]
datatype          = "datatype" SP "=" SP ("string" / "int64" / "float") LF
enum-bits         = "enum.bits" SP "=" SP "32" LF
enum-maxvalue     = "enum.maxvalue" SP "=" SP 1*10DIGIT LF
enum-ramusage     = "enum.ramusage" SP "=" SP 1*20DIGIT LF
format            = "format" SP "=" SP "plain," ["offset,"] "enum" LF
multivalue        = "multivalue" SP "=" SP ("yes" / "no") LF
offset-bits       = "offset.bits" SP "=" SP "32" LF
offset-ramusage   = "offset.ramusage" SP "=" SP 1*20DIGIT LF
plain-ramusage    = "plain.ramusage" SP "=" SP 1*20DIGIT LF
sortsigned        = "sortsigned" SP "=" SP ("yes" / "no") LF
```

Key	Value description
datatype	Specifies the internal data type of the attribute vector. This field is of type string, **int64** or **float**.
enum-bits	Specifies the number of bits used for each entry in the attribute vector entry index file (.eidx file), see section 2.1.13.2. This field MUST contain the value "32".
enum-maxvalue	Specifies the highest numbered entry value in the .eidx file. This is the same as the number of entries in the .sudat file.
enum-ramusage	Specifies the estimated memory usage in bytes required to load the .eidx,.idx and .sudat files into memory. This MUST be calculated as specified in the following formula. Each size is represented in bytes. For numeric type: `enum-ramusage = size in bytes of .idx file + size in bytes of .eidx file + size in bytes of .sudat file` For string type: See the preceding entry in this table for the definition of **the enum-maxvalue** field. enum-ramusage = size in bytes of .idx file plus size, in bytes, of .eidx file plus size, in bytes, of .sudat file plus `((enum-maxvalue + 1) * 4 bytes)`
format	Specifies the file formats present for this attribute vector file set. The format line MUST specify this setting as follows: ▪ To "plain,offset,enum" for attribute vectors that have multiValue = "yes" and datatype = "string". ▪ To "plain,enum" for the attribute vectors that do not have the required settings for the preceding format, which means that it does not have multiValue = "yes" and datatype = "string". This means that if either multiValue is "no", or datatype contains "int64" or "float", then the format MUST be set to "plain,enum". See the "multivalue" and "datatype" setting descriptions in this table.
multivalue	Specifies whether the attribute vector is multivalued. This setting is the same as the **multi** attribute in the **attributeVector** element with the name set to ATTVNAME in the indexconfig.xml file. It MUST be set to "yes" for multi-valued attribute vector, or it MUST be set to "no" for single value attribute vectors. See [MS-FSSCFG] section 2.8.3.30 for information about the multi setting.
offset-bits	The number of bits represented by each offset field. This field MUST contain the value 32.
offset-	Specifies the estimated memory usage in bytes for loading the .idx and .sudat file into

Key	Value description
ramusage	memory. This is only specified for attribute vectors of type string. This field MUST be ignored by the query matching component. This field is calculated as specified in the following formula: offset-ramusage = size in bytes of .idx file + size in bytes of .sudat file
plain-ramusage	Specifies the estimated memory usage in bytes for loading the .eidx,.idx and .dat files into memory. This is calculated as specified in the following formula. For type string: plain-ramusage = size in bytes of .idx file + size in bytes of .dat file + size of .eidx file For type numeric: plain-ramusage = size in bytes of .idx file + size in bytes of .dat file
sortsigned	This setting only exists in the .info file for attribute vectors with data type **int64**. The setting MUST contain the value of the signedValue attribute in the corresponding **attributeVector** element in the indexConfig.xml file. See [MS-FSSCFG] section 2.8.3.30 for information about the **signedValue** setting.

2.1.13.5 Sorted Unique Data File

The path and file name of this file MUST be "PP\index_TTTT\index_data\merged\ATTVNAME.sudat".

This file contains only the unique attribute vector information elements from the attribute vector data file. The elements are stored in sorted order. The query matching component loads the content of this file into memory, and uses this information to calculate aggregated data sets.

This file contains the sorted and unique values extracted from the corresponding .dat file. The values MUST be sorted in ascending order. String fields are sorted by the string of bytes that represent the value, first by the first byte, then by second byte, and so on. Numeric type fields are sorted by the entire multi-byte value that it represents.

Unique means that there are no entries with the same numeric value or string content. This unique processing MUST be performed by the indexing component across all item fields, regardless of whether the item contains multiple fields.

The format of each entry in this file is the same as the entry in the attribute vector data file, as specified in section 2.1.13.1.

2.1.14 Property Context Catalog File

This section contains an overview of the files that are part of a full-text index context catalog.

2.1.14.1 Overview

2.1.14.1.1 Local Terminology

Token identifier: A 32-bit integer field that uniquely specifies a token in a context catalog. The lowest token identifier value MUST be 0. The token identifier 0 MUST be the token with the lowest alphabetical sort order. The next sort ordered token MUST be token identifier 1, and so on.

Token position: The position of a token inside an item. When an item is a collection of tokens, for example the phrase "Here you are", the first position is token position 0, the token position of "Here" is 0, the token position of "you" is 1, and the token position of "are" is 2.

Token ordinal number: Specifies the token identifier value relative to the first token identifier for the file page. Dictionary files in an index partition are divided into file pages, each of which contains attributes for a subset of the token entries in the index partition. The token identifiers of the tokens in one file page are a continuous number range, for example all the integers from 100 to 120. The **token ordinal number** for a token in the page is calculated by subtracting the first token identifier on the page from the specified token identifier. The first token on the page contains token ordinal number 1, the next token contains the token ordinal number 2, and so on.

2.1.14.1.2 Index Configuration

Managed properties of type text contain content that is indexed into text based context catalog files.

The index.cf configuration file uses the **catalogtype** setting to specify which context catalogs MUST exist in the index, as specified in [MS-FSSCFG]. For each context catalog of **catalogtype** text or **textsynthetic**, a separate set of context catalog files MUST be generated.

There is one set of dictionary files for each context catalog of **catalogtype** text or **textsynthetic**. The dictionary files contain all the words found in all the documents in the properties belonging to the context catalog. In addition to the words themselves the dictionary files contain lookup information to the different index files in the property indexes.

The dictionary data files are logically divided into file pages. Each file page specifies properties for a subset of the tokens. Each file page for a file has the same structure, only the tokens within each file page differ. A corresponding dictionary index file is used to determine the file page that specifies properties for a specific token. The dictionary index file specifies where each file page begins. The end of one file page is the beginning of next file page minus one byte.

For each context catalog, the set of **property contexts** included in a **property index** is specified by the **contains** setting for the context catalog, as specified in [MS-FSSCFG] section 2.9.2.1.

Each property index MUST have a separate set of index files, stored in a sub-directory of the context catalog directory. The context catalogs with **catalogtype** text can have multiple property index sub-directories with separate index files. The context catalogs with **catalogtype** set to **textsynthetic** have only one sub-directory with index files. The files that represent a property index are prefixed with "boolocc" or "posocc".

A context catalog of **catalogtype** text can have *p* different property indexes, but at most 8. The property index order number is a number between 1 and *p*. The mapping of property index order number to property index is specified in the **contexts** keyword in the index.cf file, see [MS-FSSCFG] section 2.9.2.1.

In the following sections, the dictionary files that are part of a context catalog file set are stored in a path that contains the *textcatalogname* variable. The occurrence files are stored in a subdirectory of the *textcatalogname* directory. This subdirectory is referred to in the paths with the *textsubindex* variable. For each context catalog file set, the *textcatalogname* variable MUST be replaced with the context catalog name, as specified in the **catalog-def** setting in [MS-FSSCFG] section 2.9.2.1. For each property index in a context catalog, the *textsubindex* variable MUST be replaced with the property index name, as specified in the index setting in [MS-FSSCFG] section 2.9.2.1.

The query matching component uses the context catalog files to find the items that contain the text token specified in the query.

2.1.14.1.3 Context Catalog Files

The context catalog files are related to each other as specified in the following figure.

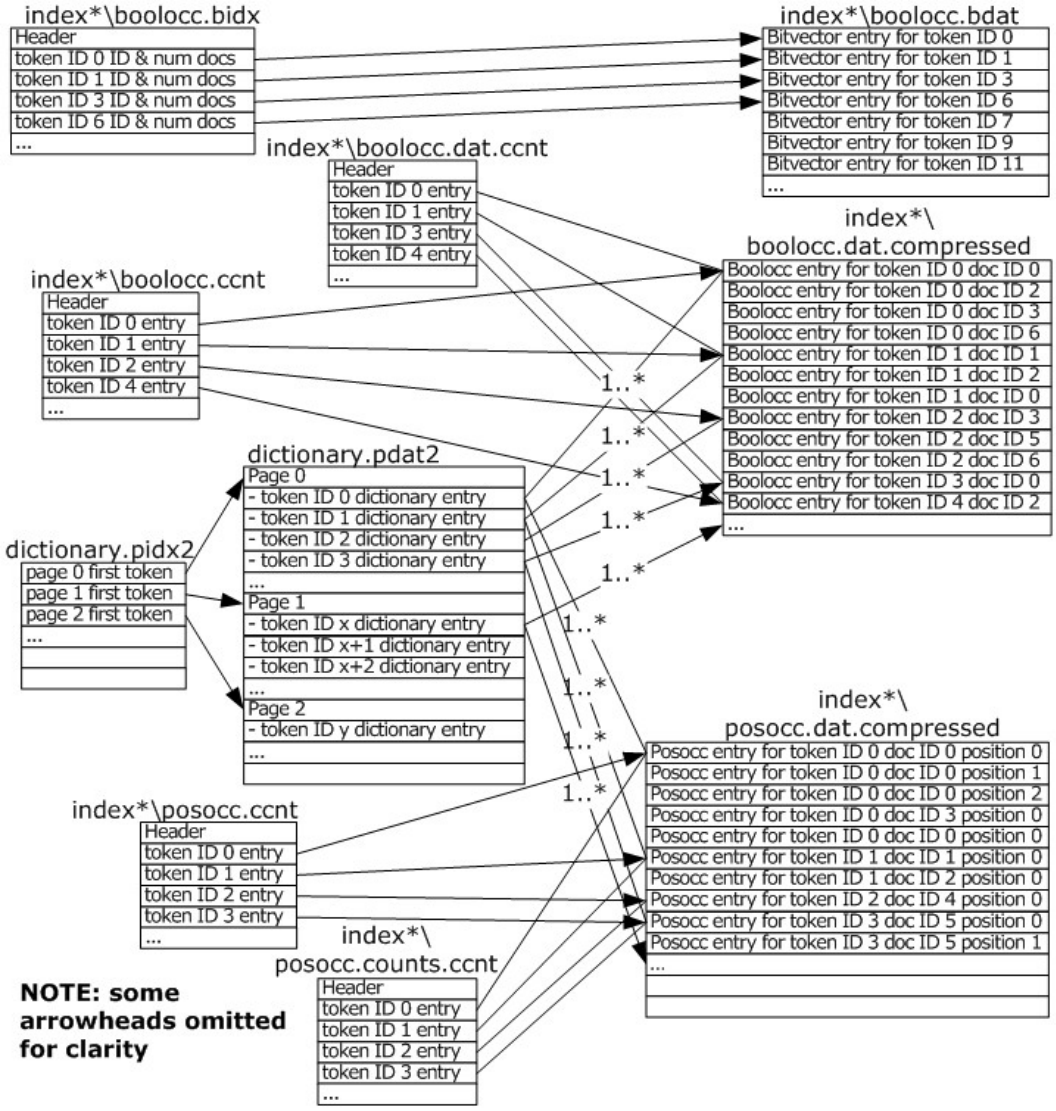


Figure 4: Relationships between context catalog files

2.1.14.1.4 Binary Data Fields

Binary data fields represent information in many fulltext-index catalog files. In a binary data field, the logical unit is the single bit, that is, the logical bit position. This abstraction of how information is represented on disk simplifies the specification of fields. The structures in binary data fields are specified by the logical bit position. Physically, on a disk, the bits are grouped together as unsigned integers named binary data bulks, therefore there is a corresponding physical representation of the bit position. This is represented by two numbers, the binary data bulk number (*i)* and the bit number within that binary data bulk *(b)*, where *b*=0 corresponds to least significant bit. The first binary bulk is assigned *i=0*. The logical position of a bit is specified by one integer only, (*l),* the first

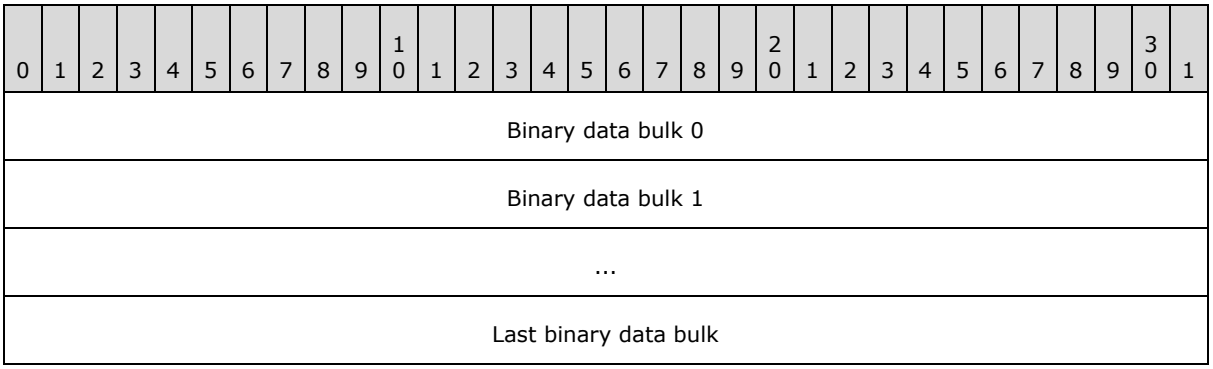
logical position is specified by *l=0.* Because the structures that use binary data fields are specified using logical bit position, the implementer of a protocol MUST use the following mappings:

```
l -> (i,b) and (i,b) -> l
```

Those mappings are used to read and write the information to disk, in addition to using the structures specified in the following sections. This mapping is shown in the following two tables. The first table specifies the layout of the unsigned integers that contain the bits and the second table specifies the mapping between logical position and physical representation. This is specified in the following formula:

```
l = 32i + 31 - b
```

As a consequence of this structure, in general there will be unused bits at the end of a binary data field.



The following table specifies the mapping between logical position and physical representation.

Logical order (l)	Binary data bulk number on file (i)	Bit number inside binary data bulk (b)
0	0	31
1	0	30
…	…	…
31	0	0
32	1	31
33	1	30
…	…	…
63	1	0
64	2	31
…	…	….

2.1.14.1.4.1 Common Algorithms for Decoding Binary Encoded Fields

The common binary encoding schemes used across the different binary data fields are specified in the following subsections.

2.1.14.1.4.2 NextBit Subroutine

This subroutine returns the value of the current bit and steps to next logical bit position in the binary data field.

2.1.14.1.4.3 ONES Subroutine

This subroutine returns the number of consecutive bits that are set to 1 (true).

The following is the algorithm for this subroutine:

```
CALL NextBit returns bit
SET count = 0
WHILE bit
   count = count + 1 ; number of consecutive bits that are set
   CALL NextBit returns bit
END WHILE
```

2.1.14.1.4.4 ReadN Subroutine

This subroutine accepts one parameter, the unsigned integer *n*. It reads *n* bits and calculates the "n bit unsigned integer" value.

Example: *ReadN(2)* on the bit sequence "11" (0x3) returns the bitstring "11" (0x3), while ReadN(1) on the same bit sequence returns the bitstring "1" (0x1).

The following is the algorithm for this subroutine:

```
SET value = 0 ; value is the returned value
FOR i = 0 :i<= n - 1
  CALL NextBit returns bit
  IF bit is set THEN
    SET value = value + 2^(n-1-i)
  END IF
END FOR
```

2.1.14.1.4.5 RICE-S Decoding Subroutine

The subroutine accepts one input parameter, the unsigned integer K. At the end of processing the decoded value MUST be present in the unsigned integer *rice*. This is returned to the calling application.

The following is the algorithm for this subroutine:

```
CALL ONES returns e ; unary coding of e
CALL ReadN with e returns g
CALL ReadN with K returns s
SET rice = (2^e + g - 1) *(2^K) + s ; the decoded value
```

2.1.14.1.4.6 Decode32 Decoding Subroutine

This subroutine accepts no parameters. At the end of processing the value decoded MUST be present in the unsigned integer *value*. This value is returned to the calling application.

The following is the algorithm for this subroutine:

```
CALL ReadN with n=3 returns numberOfNibbles
CALL ReadN with n=numberOfNibbles*4 + 4 returns value
```

2.1.14.1.4.7 RICE-C Decoding Subroutine

This is a variant of the preceding decoding subroutine. This subroutine accepts two parameters, unsigned integer *K* and unsigned integer *Max*. At the end of processing the decoded value MUST be present in the unsigned integer *value*. This value is returned to the calling application.

The following is the algorithm for this subroutine:

```
CALL RICE-S with K returns value
IF value equals 0 THEN
   CALL DECODE32 returns value
   SET value = value + Max
END IF
SET value = value - 1
```

2.1.14.1.4.8 RICE-D Decoding Subroutine

This is another variant of decoding an unsigned integer. This subroutine accepts two parameters, the unsigned integer *K* and unsigned integer *Max*. At the end of processing the decoded value MUST be present in the unsigned integer *value*. This value is returned to the calling application.

The following is the algorithm for this subroutine:

```
CALL NextBit returns bit
IF NOT bit THEN
   SET value to 0
ELSE
   CALL NextBit returns bit
   IF not bit THEN
      SET value to 1
   ELSE
      CALL RICE-C with K,Max returns value
      SET value = value + 2
   END IF
END IF
```

2.1.14.1.4.9 RICE-D0 Decoding Subroutine

This is another variant of the decoding, as specified in section 2.1.14.1.4.8. This subroutine accepts two parameters, the unsigned integers *K* and unsigned integer *Max*. At the end of processing the decoded value MUST be present in the unsigned integer *value*. This value is returned to the calling application.

The following is the algorithm for this subroutine:

```
CALL NextBit returns bit
IF NOT bit THEN
   SET value to 0
ELSE
   CALL RICE-C with K,Max returns value
   SET value = value + 1
END IF
```

2.1.14.1.4.10 RICE-BOOL Decoding Subroutine

This is another variant of decoding an unsigned integer. This subroutine accepts one parameter, unsigned integer K. At end of processing the decoded value MUST be present in the unsigned integer *value*. This value is returned to the calling application.

The following is the algorithm for this subroutine:

```
CALL RICE-S with K returns value
IF value equals 0 THEN
   CALL ReadN with n = 32 returns value
END IF
SET value = value - 1
```

2.1.14.1.4.11 RICE-2 Decoding Subroutine

This is another variant of decoding an unsigned integer. This subroutine accepts three parameters, the unsigned integers *K, Max* and *n*. At the end of processing, the decoded value MUST be present in the unsigned integer *value*. This value is returned to the calling application.

The following is the algorithm for this subroutine:

```
CALL RICE-S with K returns value
IF value is 0 THEN
   ReadN with n returns numberOfNibbles
   CALL ReadN with numberOfNibbles*4+4 returns value
END IF
SET value = value - 1
```

2.1.14.1.4.12 DECODE64-D0 Subroutine

This is a subroutine for decoding an unsigned integer. This subroutine accepts no parameters. At the end of processing, the decoded value MUST be present in variable *value.* This value is returned to the calling application.

The following is the algorithm for this subroutine:

```
IF NOT NextBit THEN
   SET value to 0
ELSE
   CALL ReadN with n=4 returns numberOfNibbles
   CALL ReadN with numberOfNibbles*4+4 returns value
END IF
```

2.1.14.1.4.13 DECODE64-D Subroutine

This is a subroutine for decoding an unsigned integer. This subroutine accepts no parameters. When processing is finished the decoded value MUST be present in the unsigned integer *value.* This value is returned to the calling application.

The following is the algorithm for this subroutine:

```
IF NOT NextBit THEN
   SET value to 0
ELSE
   IF NOT NextBit THEN
      SET value to 1
   ELSE
      CALL ReadN with n=4 returns numberOfNibbles
      CALL ReadN with numberOfNibbles*4+4 returns value
   END IF
END IF
```

2.1.14.2 Boolean Occurrences

This set of files specifies information that is associated with Boolean occurrences of tokens in items. A Boolean occurrence is a structure that stores information about which items contain a specific token. The files are the bit-vector data file, the bit-vector index file, the file that contains the counts of compressed occurrences, the data compressed sizes file, and the binary data file, as specified in the following sections.

2.1.14.2.1 Bit-vector Data File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged*textcatalogname**textsubindex*\boolocc.bdat".

This file enables the query matching component to determine which document identifiers exist for a token identifier.

This file contains the bit vectors for a subset of tokens represented in the dictionary. The Boolean occurrences bit vector index file contains token identifiers for the tokens in this file, as specified in section 2.1.14.2.2.

The format of this file MUST be specified using the following ABNF grammar:

```
file       = *(bit-vector)
bit-vector = 1*(%x00-ffff)
```

file: The file consists of *n* number of bit-vector fields, where *n* equals the number of bit vector entries in the Boolean occurrences bit vector index file, as specified in section 2.1.14.2.2. Each bit vector represents a token.

bit-vector (variable): A sequence of unsigned integers, that is, 32 bits little-endian. Each bit represents a document identifier. Bit 0 is for document identifier 0, the first item in the index. Bit 1 is for the second item in the index and so on. Bit number 0 MUST be the lowest or rightmost bit in the first unsigned 32-bit integer. Bit number 1 is the next lowest bit in the first unsigned integer. Bit number 32 is the lowest bit in the second unsigned integer, and so on.

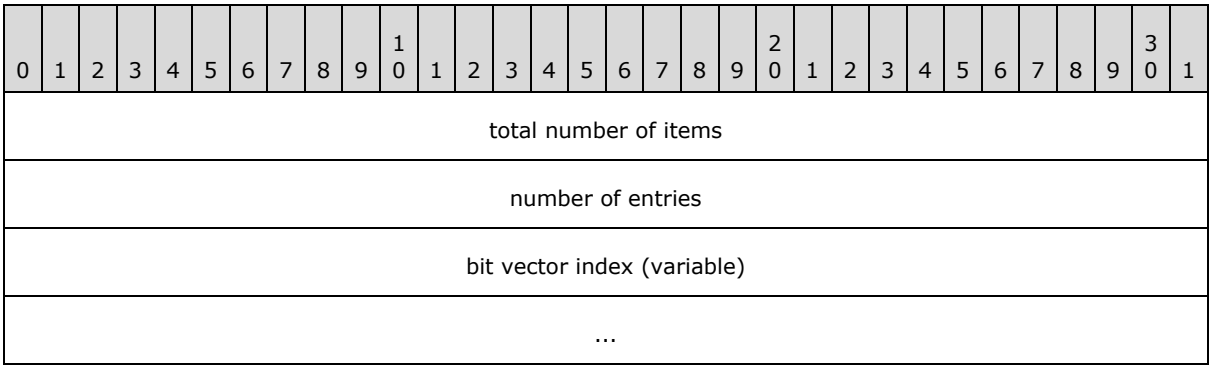
When a bit is set, the current token MUST exist in the bit that represents the item. Each bit vector contains at least as many bits as there are items in the index. The number of bits MUST be rounded up so the total number of bits is aligned on the 32-bit length by using padding. The bits not belonging to the actual document identifier MUST be set to zero and MUST be ignored.

2.1.14.2.2 Bit-vector Index File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged*textcatalogname**textsubindex*\boolocc.bidx".

This file enables the query matching component to determine the offset at which to begin reading the Boolean occurrences bit-vector data file to retrieve the bit-vector for the specified token identifier.

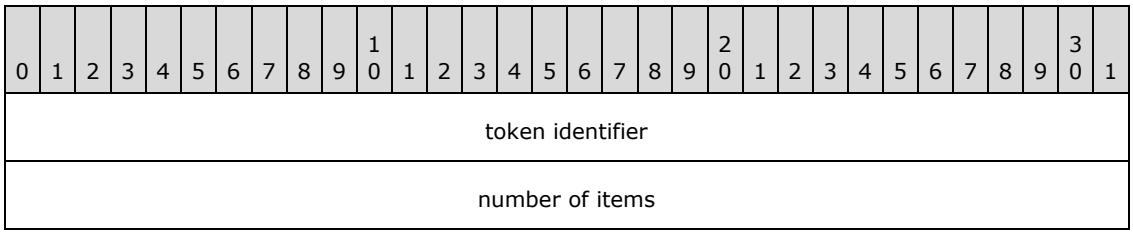
This file contains token identifiers that reference the bit vectors in the bit-vector data file, as specified in section 2.1.14.2.1. The query matching component locates the token identifier for the specified token in this index file to read a bit vector for a specific token. The count number of the entry is associated with the token identifier. The first token identifier in the index file has count number 0, the next has count number 1, and so on. The offset in number of bytes into the Boolean occurrences bit vector data file is the count number multiplied by the size of the bit vector, as specified in section 2.1.14.2.1. This is shown in the following table.



total number of items (4 bytes): This is the number of **items** in the index partition. It is specified as an unsigned 32-bit integer field in little-endian order.

number of entries (4 bytes): This is the number of bit vector index entries following this field. It is specified as an unsigned 32-bit field in little-endian order.

bit vector index (variable): This field consists of as many bit vector index entries as there are bit vectors in the Boolean occurrences bit vector data file, as specified in section 2.1.14.2. A bit vector index entry is specified in the following table.



token identifier(4 bytes): This is the token identifier for the token, which bit vector MUST be present in the Boolean occurrences bit vector data file, as specified in section 2.1.14.2.1. It is an unsigned 32-bit integer field in little-endian order.

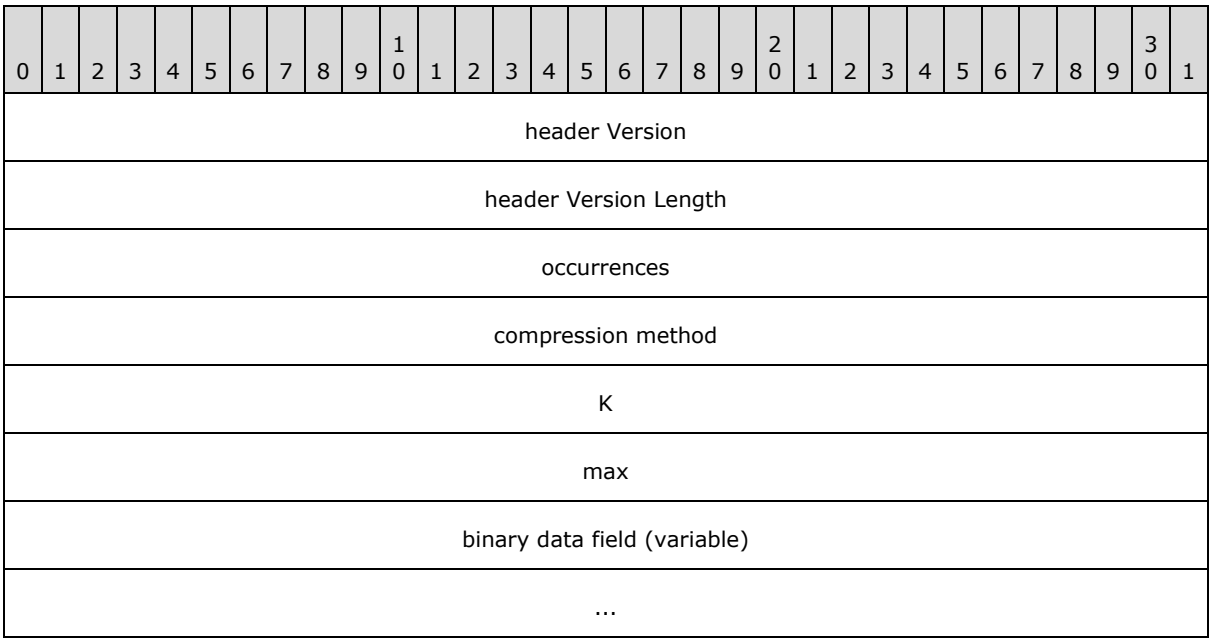
number of items (4 bytes): This is the number of bits set to 1 in the bit vector data file for the current bit vector entry. This value represents the number of items that contains the token. It is an unsigned 32-bit integer field in little-endian order.

2.1.14.2.3 Compressed Occurrence Counts File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged*textcatalogname**textsubindex*\boolocc.ccnt".

This file enables the query matching component to determine for each token how many items contain the token.

The 4-byte fields in the header of this file are unsigned 32-bit integer values in little-endian order. The file is specified as follows.



header Version (4 bytes): The header version for this file. This field MUST contain the value 1.

header Version Length (4 bytes): The header version length for this file. This field MUST contain the value 16.

occurrences (4 bytes): The number of count numbers that are encoded in the binary data field in this file.

compression method (4 bytes): This specifies the encoding algorithm used on the count numbers encoded in the binary data field. This field MUST contain the value 8.

K (4 bytes): A compression parameter. This field MUST contain the value 2.

max (4 bytes): A compression parameter. This field MUST contain the value 1020.

binary data field (variable): This represents a binary data field, as specified in section 2.1.14.1.4. This field contains one or more entries representing an encoded count value for each token. The tokens are sorted by the token identifier, beginning with token identifier 0. The decoded value is the number of items in which the token exists. Each **count** value is decoded as specified in the following algorithm.

```
CALL RICE-D with k=2, Max=1020 returns count
```

2.1.14.2.4 Data Compressed Sizes File

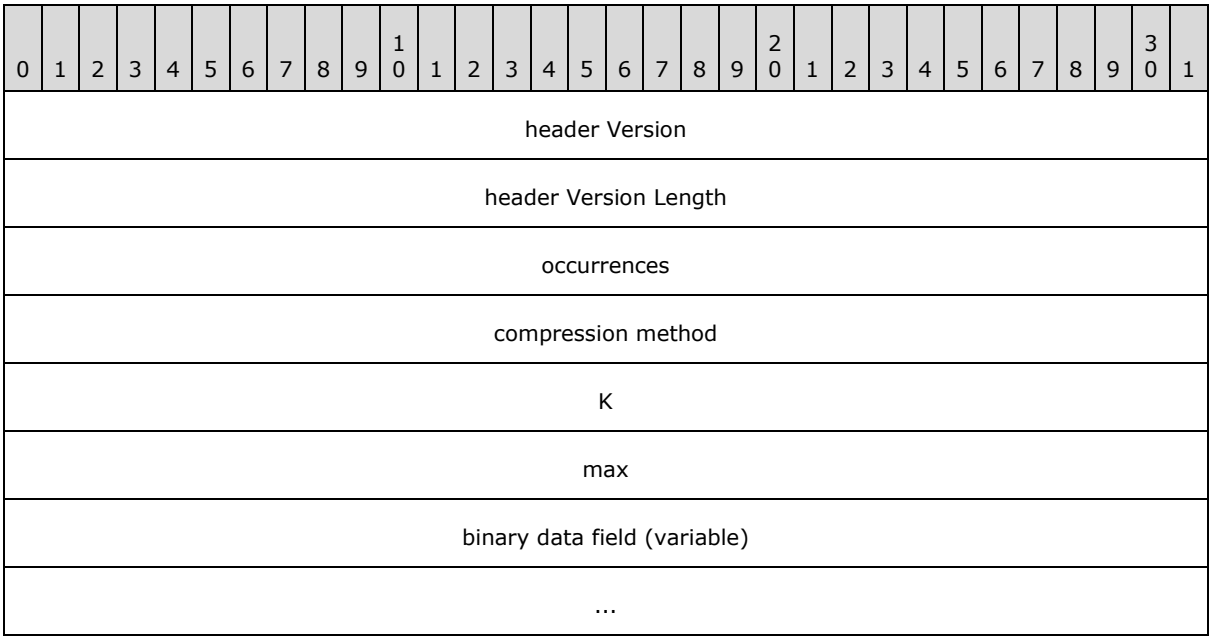
Path and file name of this file MUST be "PP\index_TTTT\index_data\merged*textcatalogname**textsubindex*\boolocc.dat.ccnt".

This file contains for each token, the size of the region in the Boolean occurrences data file, which is used to described occurrences for each token, given in number of bits.

The value for a given token MUST equal to the number of bits consumed by the algorithm, as specified in section 2.1.14.2.5.1, the inner loop.

The 4-byte fields in the header of this file are unsigned 32-bit integers in little-endian order.

The file is specified as follows.



header Version (4 bytes): The header version for this file. This field contains the value 1.

header Version Length (4 bytes): The header version length for this file. This field MUST contain the value 16.

occurrences(4 bytes): The number count numbers in this file. This number is the number of count numbers that are encoded in the binary data field.

compression method (4 bytes): A field that specifies the encoding algorithm used on the count numbers encoded in the binary data field. This field MUST contain the value 7.

K (4 bytes): A compression parameter. This field MUST contain the value 7.

max (4 bytes): A compression parameter. This field MUST contain the value 524160.

binary data field (variable): This is a binary data field, as specified in section 2.1.14.1.4. This field MUST contain a count value for each token. The count value MUST be sorted by token identifier, beginning with token identifier 0. Each count value is the number of bits used to store the token-identifier entry for this token, as specified in section 2.1.14.2.5.1. Each count value is decoded using the following algorithm:

```
CALL RICE-D0 with k=7, Max=524160 returns count
```

2.1.14.2.5 Binary Data File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged*textcatalogname**textsubindex*\boolocc.dat.compressed".

This file enables the query matching component to determine which document identifiers contain a token. Along with each document identifier there MUST also be associated information that the query matching component uses to calculate the dynamic rank. This is shown in the following table.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
header Version																															
header Length																															
binary data field (variable)																															
...																															

header Version (4 bytes): This field specifies the header version for this file. This field MUST contain the value 1 as an unsigned 32-bit field in little-endian order.

header Length (4 bytes): This field specifies the header version length for this file. This field MUST contain the value 0 as an unsigned 32-bit field in little-endian order.

binary data field (variable): See section 2.1.14.2.5.1.

2.1.14.2.5.1 Binary Data Field

This field specifies which document identifiers contain a token. Along with each document identifier entry, additional information MUST also be present, as specified in the **contextMapBoost**, **extNumOccBoost**, **firstOccBoost** and **numOccBoost** fields.

For a general description of a binary data field, see section 2.1.14.1.4. For an example of how to decode the values from this type of file, see the Boolean occurrences binary data file in section 3.7.

The information in this field MUST be as specified in this section.

In the following algorithm, the *number-of-tokens-in-the-dictionary-paged-data-file* parameter MUST be equal to the **token count** field in the dictionary paged data file (section 2.1.14.4.3). The *number-of-items-where-tokenId-is-present* parameter is located in the Boolean occurrences compressed occurrences counts file, in the binary field of the current **tokenId** field. How to retrieve an **item** count value for a specified token identifier is specified in section 2.1.14.4.1.

```
FOR tokenId = 1 : tokenId <= number-of-tokens-in-the-dictionary-paged-data-file
  SET previousDocId = 0
  FOR occNumber = 1 : tokenId <= number-of-items-where-tokenId-is-present
    CALL ReadN with n=4 returns flags
    CALL NextBit returns newEntry
    IF flags bit 1 is set THEN  ; bit 1 is the leftmost bit in flags
      CALL ReadN with 8 returns contextmapBoost[tokenId][occNumber]
    ELSE
      contextmapBoost[tokenId][occNumber] = contextmapBoost[tokenId][occNumber-1]
    END IF
    IF flags bit 2 is set THEN
      CALL ReadN with 8 returns extNumOccsBoost[tokenId][occNumber]
    ELSE
      extNumOccsBoost[tokenId][occNumber]= extNumOccsBoost[tokenId][occNumber-1]
    END IF
    IF flags bit 3 is set THEN
      CALL ReadN with 8 returns firstOccBoost[tokenId][occNumber]
    ELSE
      firstOccBoost[tokenId][occNumber]= firstOccBoost[tokenId][occNumber-1]
    END IF
    IF flags bit 4 is set THEN
      CALL ReadN with 8 returns numOccBoost[tokenId][occNumber]
    ELSE
      numOccBoost[tokenId][occNumber]= numOccBoost[tokenId][occNumber-1]
    END IF
    CALL RICE-BOOL with 6 returns rice
    IF newEntry THEN
      ; rice represent the item number
      SET docId[tokenId][occNumber] = rice
    ELSE
      ; rice represents the difference from previous item number decoded
      SET docId[tokenId][occNumber] = previousDocId + rice
    END IF  SET previousDocId = docId[tokenId][occNumber]   END FOR
END FOR
```

Flags (4 bits): This field contains four bits that represent four Boolean flags. The first bit, or leftmost bit, is bit 1. If bit 1 is set, then the **contextMapBoost** field MUST be present. If bit 2 is set, then the **extNumOccBoost** field MUST be present. If bit 3 is set, then the **firstOccBoost** field MUST be present. If bit 4 is set, then the **numOccBoost** field MUST be present.

newEntry (1 bit): A flag that is used when decoding the rest of the values. This field MUST be set to TRUE (1) if the current item-entry is the first for the current token-identifier-entry.

contextMapBoost (integer array): This is an 8-bit unsigned field that specifies the property context instances that contain the specified token in the current item. For example, let *n* be a number between 0 and 7. Then if bit *n* is set, the token is located in property context numbered *n*. For the element named **contextMapBoost**[*i*][*j*], the subscript *i* is between 1 and *number-of-tokens* in the dictionary paged data file. The subscript *j* is between *1* and *number–of-items* containing the token identifier *j*. The two maximum numbers for *i* and *j* are located in the dictionary paged data file.

extNumOccsBoost (integer array): This specifies how many times the current token exists inside an **external context**. If token exists more than 255 times in an external context, this value is set to 255. For example, the element **extNumOccsBoos**t[*i*][*j*] has a subscript *i* that is between 1 and the number of tokens in the dictionary paged data file. The subscript *j* is between *1* and the number of items containing the token specified by token identifier *i*. The two subscripts use the token and the items associated with the token to locate the occurrence count within the **extNumOccsBoost** array. The maximum values for *i* and *j* are located in the dictionary paged data file.

firstOccBoost (integer array): This two-dimensional array contains the position of the first occurrence of this token for the current item. If value is greater than 255, this value is set to 255. For example, the element **firstOccBoost**[*i*][*j*] has a subscript *i* that is between 1 and the number of tokens in the dictionary paged data file. The subscript *j* MUST contain a value between 1 and the number of items that contain the token specified by token identifier *i*. The maximum values for *i* and *j* are located in the dictionary paged data file.

numOccBoost (integer array): This two-dimensional array contains the number of occurrences the token was located in the current item. If this exceeds 255, then it MUST be set to 255. For example, the element **numOccBoost**[*i*][*j*] has a subscript *i* that is between 1 and the number of tokens in the dictionary paged data file. The subscript *j* MUST contain a value between *1* and the number of items that contain the token specified by token identifier *i*. The maximum values for *i* and *j* are located in the dictionary paged data file.

rice (4 bytes): If the **newEntry** field is **true**, this field MUST be set to the current item identifier. If the **newEntry** field is **false**, this field MUST contain the difference between the current document identifier and the previous document identifier. For each token entry, the **previousDocId** field MUST contain the document identifier of the previous item entry.

docId: For `docId[tokenId][occNumber]`, the occNumber field specifies for which occurrence number this document identifier entry is. For Boolean occurrences, there MUST be only one occurrence per item. The **tokenId** field specifies the token identifier that is associated with this document identifier entry. The document identifier at each **tokenID** and **occNumber** position MUST specify the document identifier that contains **tokenId**.

2.1.14.3 Position Occurrences Files

This section specifies the position occurrences files. A position occurrence is a structure that stores information about the location of each token in an indexed item. These files contain components that are similar but not identical to the Boolean occurrences files. The position occurrence files are the compressed sizes file, the compressed occurrence counts file, and the binary data file, as specified in the following sections.

2.1.14.3.1 Compressed Sizes File

The path and file name of this file MUST be "PP\index_TTTT\index_data\merged*textcatalogname**textsubindex*\posocc.ccnt".

This file enables the query matching component to determine how many bits each position data section uses for each token identifier in the position occurrences binary data file.

The 4-byte fields in the header of this file are unsigned 32-bit integers. The file MUST be specified as follows.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
header Version																															
header Version Length																															
occurrences																															
compression method																															
K																															
max																															
binary data field (variable)																															
...																															

header Version (4 bytes): The header version for this file. This field MUST contain the value 1.

header Version Length (4 bytes): The header version length for this file. This field MUST contain the value 16.

occurrences(4 bytes): The number of count numbers in this file, which is the same as the number of count numbers that are encoded in the binary data field.

compression method (4 bytes): A field that specifies the encoding algorithm used on the count numbers encoded in the binary data field. This field MUST contain the value 12.

K (4 bytes): A compression parameter. This field MUST contain the value 6.

max (4 bytes): A compression parameter. This field MUST contain the value 524160.

binary data field (variable): This represents a binary data field, (section 2.1.14.1.4). This field MUST contain the count value for each token. The count value for the token is sorted by token identifier, beginning with token identifier 0. One count MUST equal the length of the section of posocc.data.compressed file that specifies the positions for current token identifier. Each count value is decoded using the following algorithm:

```
CALL RICE-D0 with k=6, Max=524160 returns count
```

2.1.14.3.2 Compressed Occurrence Counts File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged*textcatalogname**textsubindex*\posocc.counts.ccnt".

This file contains the number of occurrences for each token identifier.

The 4-byte fields in the header of this file are unsigned 32-bit integers in little-endian order. The file MUST be as specified in the following table.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
header Version																															
header Version Length																															
occurrences																															
compression method																															
K																															
max																															
binary data field (variable)																															
...																															

header Version (4 bytes): The header version for this file. This field MUST contain the value 1.

header Version Length (4 bytes): The header version length for this file. This field MUST contain the value 16.

occurrences (4 bytes): The number of count numbers in this file. This is the number of count numbers that are encoded in the binary data fields.

compression method (4 bytes): A field that specifies the encoding algorithm used on the count numbers encoded in the binary data fields. This field MUST contain the value 8.

K (4 bytes): A compression parameter. This field MUST contain the value 2.

max (4 bytes): A compression parameter. This field MUST contain the value 1020.

binary data field (variable): This represents a binary data field (section 2.1.14.1.4). This field contains a count value for each token. This value is sorted by the token identifier, beginning with token identifier 0. One count value MUST equal the number of total number of occurrences of the token. Each value is decoded as specified in the following algorithm:

```
CALL RICE-D with k=2, Max=1020 returns count
```

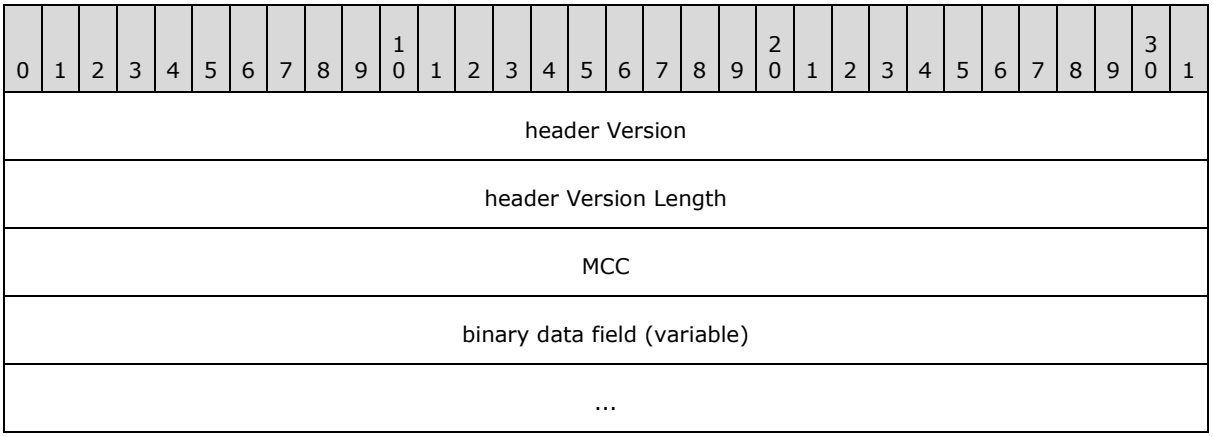
2.1.14.3.3 Binary Data File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged*textcatalogname**textsubindex*\posocc.dat.compressed".

This file enables the query matching component to determine which positions are associated with a token identifier for each document identifier. Information about the property index in which the token was included is associated with each position.

How to find the offset into this file for a specific token is specified in the dictionary paged data file in section 2.1.14.4.3.

The 4-byte fields in the header of this file are unsigned 32-bit integers in little-endian order. The file MUST be specified as follows.



header Version (4 bytes): The header version for this file. This field MUST be set to 1. It is specified as an unsigned 32-bit integer field in little-endian order.

header Version Length (4 bytes): The length of the header version for this file. This is specified as a 32-bit unsigned integer field in little-endian order. The field MUST be set to 4.

MCC (4 bytes): This field MUST be ignored.

binary data field (variable): See section 2.1.14.3.3.1.

2.1.14.3.3.1 Binary Data Field

This represents a binary data field, as specified in section 2.1.14.1.4. For each token, the field MUST contain a list of items. For each item, the field contains the position entries inside that item. For an item, there are one or more tokens at each position. The first token(s) in the item has position 0; the second token(s) has position 1, and so on. A position entry specifies a token position in the item. The context value MUST also be determined for each position entry. If there is no context value specified directly for a position entry, the context value MUST be computed by using the context value from a previous position entry for the same item that specified a context value. The context value of the current position entry MUST be set to the same value as the context value of the previous entry.

Assume that there are *n* tokens in the dictionary. Then the binary data field is decoded as specified by the following algorithm. See the dictionary paged index file section 2.1.14.4.4 for how to find *n*.

```
; Loop over all the tokens in the dictionary

FOR tokenNr = 1 : tokenNr <= n  ; First document id
    SET docIdx = 1 ; An index integer for document ids for a token
    CALL RICE-BOOL with K=22 returns docId[tokenNr][docIdx]
    REPEAT ; A REPEAT-UNTIL loop over the document ids
        SET posIdx = 1 ; First position index number
        ; First position value
        CALL RICE-BOOL with K=8 returns position[tokenNr][docIdx][posIdx]
        CALL NextBit returns contextPresent
```

```
        IF contextPresent THEN
          CALL ReadN with n = 3 returns context[tokenNr][docIdx][posIdx]
        END IF
          CALL NextBit returns posEntryPresent
        WHILE posEntryPresent ; while loop over positions
            SET posIdx = posIdx + 1
            CALL RICE-BOOL with K=4 returns diffPosition
            SET position[tokenNr][docIdx][posIdx] = \
              position[tokenNr][docIdx][posIdx-1]+1+diffPosition
            CALL NextBit returns contextPresent
            ; If context present read it
            ; if not present, use previous
            IF contextPresent THEN
                CALL ReadN with n = 3 returns context[tokenNr][docIdx][posIdx]
            ELSE
                SET context[tokenNr][docIdx][posIdx] = \
                   context[tokenNr][docIdx][posIdx-1]
            END IF
            CALL NextBit returns posEntryPresent
        END WHILE ; end loop over positions
        CALL NextBit returns nextDocIDPresent
        IF nextDocIDPresent THEN
            CALL RICE-BOOL with K=7 returns diffDocIdEntry
            SET docIdx += 1
            SET docid[tokenNr][docIdx] = \
                diffDocIdEntry + 1 + docid[tokenNr][docIdx]
        END IF
    UNTIL nextDocIDPresent ; end of REPEAT-UNTIL loop over document ids
END FOR ; end for loop over tokens
```

docid (integer array): After processing, this two-dimensional array contains all the document identifiers present in current context index. For example, in the element `docid[i][j]`. The index variable *i* is the token identifier**.** The index variable *j* is an item index that begins at 1 and increases for each new item encountered for the current token in the file. The `docid[tokenId]` array contains all the document identifiers for token identifier **tokenId**. Each decoded document identifier is an unsigned 32-bit field in little-endian order.

position (integer array): After processing each document identifier present in the current context index, this field contains all the positions for the current token. In the element `position[i][j]`, the index variable *i* is the token identifier. The index variable *j* is a document identifier index that begins at 1 and increases for each new document identifier encountered for the current token in the file. The elements in `position[tokenId][docIdx][]` contain all the positions for token where token identifier is equal to *tokenId* in the item with `document identifier docid[tokenId][docIdx]`. Each decoded position is an unsigned 32-bit field in little-endian order.

context (integer array): After processing each document identifier present in current context index, this field contains all the full-text context indexes for the current token. The array element `context[tokenId][docIdx][]` contains all the full-text context indexes for the token which token identifier is equal to **tokenId** in the item with `document identifier docid[tokenId][docIdx]`. Each decoded context value is an unsigned 32-bit field in little-endian order.

2.1.14.4 Dictionary Files

This section specifies the dictionary files. Dictionary files contain information about tokens in addition to the occurrences of tokens within items. The dictionary files contain all the words found in all the documents associated with the properties in the context catalog. The dictionary files are the paged count data file, the paged count index file, the paged data file, paged index file, sorted hash file, token number count index file, the token number index file, and the warmup file, as specified in the following sections.

2.1.14.4.1 Paged Count Data File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged*textcatalogname*\dictionary.pcdat".

This file enables the query matching component to look up occurrence information for a token. The information contains item occurrence and token occurrence information.

The following variables are used for the purposes of this specification as placeholder names for the various types of summations that occur while managing search application indexes:

item-occurrence-number: Specifies how many items contain the current token.

item-occurrence-accumulated-number: Specifies the sum of the **item-occurrence numbers**, over all the tokens from token identifier zero up to and including the current token identifier.

token-occurrence-number: Specifies how many times a token occurs in all the indexed items.

token-occurrence-accumulated-number: Specifies the sum of the **token-occurrence numbers**, over all the tokens from token identifier zero up to and including the current token identifier.

This file contains file pages that are identically structured. Each file page contains **token-occurrence-accumulated-number** tokens and **item-occurrence-accumulated-number** tokens or a subset of all the tokens. How to locate the file page that contains information for a specified token is specified in section 2.1.14.4.3. Let *n* be assigned to be the number of property indexes and the constant `K=(number-tokens)-1`, where **number-tokens** are number of tokens described in the current page.

The page is specified using the following ABNF grammar. The UTF-8 string ABNF in section 2.1.1.2.1 MUST be added to this file ABNF on a new line before the word-"string rule" to complete it.

```
page                        = accumulated-numbers-before accumulated-numbers-last
                              number-tokens first-wordid diff-accumulated-numbers
                              acc-word-lengths word-strings
accumulated-numbers-before = 1*context-property-numbers
accumulated-numbers-last   = 1*context-property-numbers
context-property-numbers   = token-accumulated-number item-accumulated-number
token-accumulated-number   = 8OCTET
item-accumulated-number    = 8OCTET
number-tokens              = 4OCTET
first-wordid               = 4OCTET
diff-accumulated-numbers   = *diff-contexts
diff-contexts              = *diff-token-numbers
diff-token-numbers         = diff-token-accumulated-number
                              diff-item-accumulated-number
```

```
diff-token-accumulated-number = 4OCTET
diff-item-accumulated-number  = 4OCTET
acc-word-lengths           = *acc-word-length
acc-word-length            = 2*OCTET
word-strings               = *word-string
word-string                = UTF8-string %x00
```

accumulated-numbers-before: This contains *n* occurrences of context-property-numbers, one for each property index. The context-property-numbers MUST be valid for the first token identifier at current page. The *n* instances of context-property-numbers are ordered by the property index ordering number.

accumulated-numbers-last: This contains *n* occurrences of context-property-numbers, one for each property index. The occurrences MUST be ordered by the property index ordering number. The context-property-numbers MUST be valid for the first token identifier at the next page. If the context property number is the last word in the dictionary, then the token identifier contains 16 bytes which MUST be ignored. This is because the accumulated count numbers are specified as up to, but not including the current token.

token-accumulated-number: This contains the token occurrence accumulated number for the current token and for the current property index**.**

item-accumulated-number: This number MUST contain the item occurrence accumulated number for the current token and for the current property index.

diff-accumulated-numbers: This field MUST contain *K-1* occurrences of diff-token-numbers. The first occurrence of diff-contexts contains information about the token with token ordinal number 1, the next occurrence of diff-contexts contains information about the token with token ordinal number 2, and so on until the last token on the page.

number-tokens: This MUST be the number of tokens contained in this file page, as specified in section 2.1.14.4.3.

first-wordid: This is the token identifier.

diff-contexts: This field MUST contain *n* occurrences of the **diff-token-numbers** field sorted by the property index ordering number.

diff-token-accumulated-number: This MUST contain the *token-occurrence-accumulated-number* for current token minus *token-occurrence-accumulated-number* for the token with token ordinal number 1 and the same property index.

diff-item-accumulated-number: This MUST contain the *item-occurrence-accumulated-number* for the current token minus *item occurrence accumulated number* for token with token ordinal number 1 and the same property index.

acc-word-lengths: This rule MUST expand to the number of tokens on the page – 1. The first item is for the first token, the second item is for the second token, and so on. The last token on the page has no entry.

acc-word-length: The accumulated length of strings on this page.

word-strings: This rule MUST expand to the number of tokens on the page, one for each token.

word-string: A null terminated UTF-8 string that represents a token entry.

2.1.14.4.2 Paged Count Index File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged*textcatalogname*\dictionary.pcidx".

This file enables the query matching component to locate the file page for a specific token in the dictionary.pcdat file.

The file MUST contain *n* number of tokens in alphabetical order. The tokens represent a subset of all the tokens found in the dictionary paged data file, see section 2.1.14.4.3. This file is used to find the file page in the dictionary paged data file (section 2.1.14.4.3) where the information for a token is stored. The corresponding file page in the dictionary paged data file MUST contain all the entries for the token identifiers in the range r to s-1, where r and s represent two consecutive tokens in this file, the dictionary paged count index file. The size of one file page is 4096 bytes.

The algorithm for finding the corresponding file page for a specified token is specified as follows.

The first token in the dictionary paged count index file is for file page 0, the next token in the index file is for file page 1, and so on. Using the searched for token, scan through this file and find the last token in this file that occurs before or is the same as the sought token in alphabetical order, and note its number. This number is the file page number for the requested token. The following is an example:

Let the index contain the tokens "alpha, gamma, upsilon" assigned the numbers 0,1 and 2. The token to search for is "roma". Because the dictionary is in alphabetical order, the last token in the dictionary before "roma" is "gamma". The assigned number to "gamma" is 1, and therefore the file page number for "roma" is 1.

The ABNF grammar for this file, assuming there are a total of *p* pages, is specified as follows:

```
file   = 1*token
token  = UTF8-string %x00
```

file: The file MUST consist of *p* number of tokens.

token: The string content of the token. The token MUST consist of UTF-8 characters, delimited by the ASCII character 0x00.

2.1.14.4.3 Paged Data File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged*textcatalogname*\dictionary.pdat2".

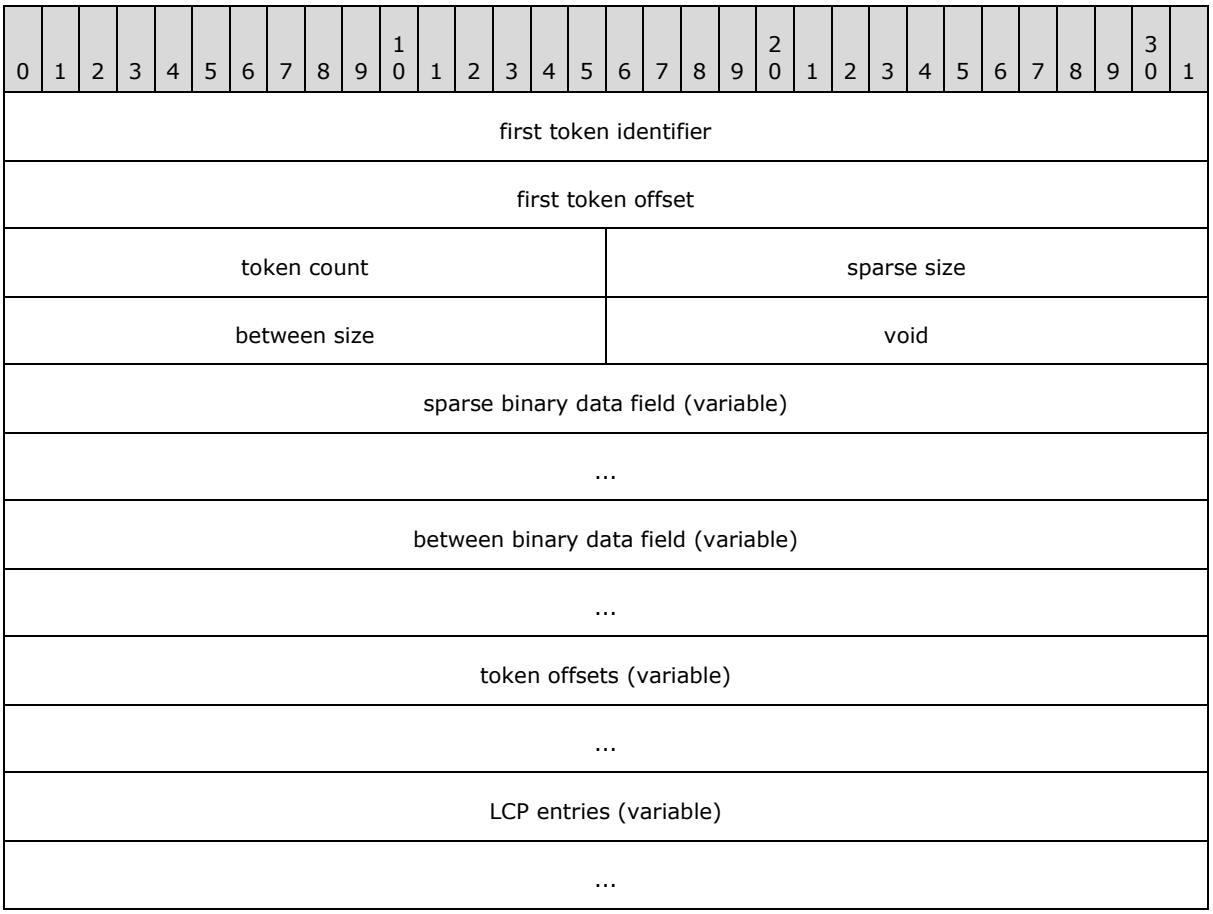
This file enables the query matching component to determine what tokens are available, in how many items the token occurred, how many total token occurrences were counted over all the **items**, and where the item occurrence information for each token is stored in the Boolean occurrence files and the position occurrence files.

There is one dictionary file set for each context catalog, as specified in **catalog-def** field in the index.cf file.

This file consists of a number of file pages, in consecutive order. File page number 0 is first, then file page 1, and so on. The size of one file page is 4096 bytes. Each file page, except the last page, contains information about 512 consecutive tokens in the dictionary.shash, see section 2.1.14.4.5. In the following assume that the file page number is *g*, where *g* is a non-negative integer. This file

page contains information about tokens that have token identifiers `first=512*g` to `last=512*(g+1)-1`.

Each file page in this file is specified as follows.



first token identifier (4 bytes): This is an integer that contains the token identifier for the first token specified for this file page. This token identifier MUST belong to the corresponding lookup token in the dictionary paged index file, as specified in section 2.1.14.4.4.

first token offset (4 bytes): This field MUST be ignored.

token count (2 bytes): This integer MUST be the number of tokens contained in this file page.

sparse size (2 bytes): This integer specifies the length of the sparse binary data field, in units of 4-byte integers. Therefore, the length of the sparse binary field in bytes MUST be this value multiplied by 4.

between size (2 bytes): This integer specifies the length of the **between binary data field** field in 4-byte integers. Therefore, the length of the **between binary field** field in units of bytes is this value multiplied by 4. The size of the **padding region** field after the **between binary data field** field MUST be included in this **between size** field.

void (2 bytes): This field MUST be ignored.

sparse binary data field (variable): Contains binary encoded information about tokens. This is specified in section 2.1.14.4.3.1.

between binary data field (variable): Contains binary encoded information about tokens. This is specified in section 2.1.14.4.3.2.

token offsets (variable): Contains binary encoded information about tokens. This is specified in section 2.1.14.4.3.3.

LCP entries (variable): Contains binary encoded information about tokens. This is specified in section 2.1.14.4.3.4.

2.1.14.4.3.1 Sparse Binary Data Field

This variable-length field represents a binary data field, as specified in section 2.1.14.1.4. Encoded in this binary field is *p* number of sparse entries. Each sparse entry represents one token in the dictionary. Only a subset of the token with token identifiers in the range of first to last MUST be represented. This subset consists of the first token at the file page followed by every 16th token thereafter, for example, the tokens which token ordinal numbers are 1, 17, 33, and so on. The syntax of the field MUST be as specified in the following algorithm.

```
FOR i = 1 : i <= number of property indexes
  CALL DECODE64-D returns accnumDocsSparse[1][i]
  CALL DECODE64-D0 returns boolOccOffsetSparse[1][i]
  IF posPresent THEN
    CALL DECODE64-D0 returns posOccOffsetSparse[1][i]
  END IF
END FOR

FOR e = 1 : e <= number of sparse entries
  FOR i = 1 : i <= number of property indexes
    CALL NextBit returns bit
    IF bit is set THEN ; If not set, then there is no information for this entry
      CALL NextBit returns bit
      IF NOT bit is set THEN ; Numbers are rice encoded
        CALL RICE-2 with K=3, Max=8184, n=3 returns delta
        SET accnumDocsSparse[e][i] += delta
        CALL RICE-2 with K=9, Max=2096640, n=3 returns delta
        SET boolOccOffsetSparse[e][i] += delta
        IF posPresent THEN
          CALL RICE-2 with K=9, Max=2096640, n=3 returns delta
          SET posOccOffsetSparse[e][i] += delta
        END IF
      ELSE ; Numbers are coded in another way
        CALL DECODE64-D returns delta
        SET accnumDocsSparse[e][i] += delta
        CALL DECODE64-D0 returns delta
        SET boolOccOffsetSparse[e][i] += delta
        IF posPresent THEN
          CALL DECODE64-D0 returns delta
          SET posOccOffsetSparse[e][i] += delta
        END IF
      END IF
    END IF
  END FOR
  CALL RICE-2 with K=10, Max=2096128, n=3 returns betweenOffset[e]
END FOR
```

The example calls subroutines that are specified in section 2.1.14.1.4. It requires various input values that MUST be determined as follows:

number of property indexes: This field is the number of property indexes specified in the dictionary paged index file (section 2.1.14.4.4).

number of sparse entries: Let a=token count (section 2.1.14.4.3). Then the value *ceil(a/16)-1* MUST be equal to number of sparse entries. The function *ceil* rounds decimal values up to the closest integer value, for example, *ceil(4.3)=5*.

posPresent: This value MUST be **true** if and only if the **flags** field for the dictionary paged index file (section 2.1.14.4.4) is equal to the value 0xB1.

The fields are defined as follows.

accnumDocsSparse (integer array): This is a two-dimensional array that MUST contain the item occurrence accumulated numbers for the current token in the current property index. Take an element `accnumDocsSparse[i][j]`. The index variable *i* is in the range 1 to "token count". The index variable *j* is in the range 1 to number of property indexes. The index variable *j* has the same ordering properties as the property index ordering number.

boolOccOffsetSparse (integer array): An offset value that is used when retrieving information from the Boolean occurrences binary data file for the current token and current property index. The offset into the Boolean occurrences binary data file for current property index MUST be offset = `boolOccOffsetSparse + 64` in units of bits. The offset points to the beginning of the field "token identifier entry" in the Boolean occurrences binary data file. Take an element `boolOccOffsetSparse[i][j]`. The index variable *i* is in the range 1 to "token count". The index variable *j* is in the range 1 to "number of property indexes". The index variable *j* has the same ordering properties as the property index ordering number.

posOccOffsetSparse(integer array): Offset in number of bits into the position occurrence compressed data file for current token current property index**.** The component uses the offset **posOccOffsetSparse**, specified in bits, to compute the corresponding token entry in the position occurrence compressed data file. For example, in the element `posOccOffsetSparse[i][j]`, the index variable *i* is in the range 1 to "token count". The index variable *j* is in the range 1 to "number of property indexes". The index variable *j* has the same ordering properties as the property index ordering number.

betweenOffset (integer array): This field MUST contain the number of bits in the between region that are used to store the information about the 16 tokens before the current token in the dictionary. So if the current token has token identifier 32, this number is the number of bits used to store information about tokens with token identifiers 16,17…,31. For example the element `betweenOffset[i]` has a subscript variable i that MUST be between 1 and the value contained in the token count field.

padding bits region: This region is used for padding bits after the sparse binary data field. The number of bits in this region MUST be specified so that the following between binary data field begins on a 32-bit aligned boundary. The bits in this region MUST be set to 0.

2.1.14.4.3.2 Between Binary Data Field

This is a **binary data field**, as specified in section 2.1.14.1.4. Encoded in the binary field contains *b* number of **between binary data field** entries**.** Each between entry MUST represent one token in the dictionary. All the **tokens** are specified in consecutive order.

The algorithm requires the following input parameters:

number of tokens in file page: This field MUST contain the token count (section 2.1.14.1.4).

number of property indexes: This field MUST contain the number of property indexes value as specified in the dictionary paged index file (section 2.1.14.4.4).

posPresent: This value MUST be **true** if and only if the **flags** field for the dictionary paged index file (section 2.1.14.4.4) contains the value 0xB1.

The syntax of the field MUST be as specified in the following algorithm.

```
FOR e = 1 : e <= number of tokens in file page
  FOR i = 1 : i <=  number of property indexes
    CALL NextBit returns bit
    IF bit is set THEN ; If not set, then there is no information for this entry
      CALL NextBit returns bit
      IF NOT bit is set THEN
        CALL RICE-2 with K=7, Max=524160, n=4 returns boolOccLength[e][i]
        IF posPresent THEN
          CALL RICE-2 with K=6, Max=262080, n=4 returns posOccLength[e][i]
        END IF
      ELSE
        CALL RICE-D with K=3, Max=8184 returns deltaDoc
        CALL RICE-2 with K=7, Max=524160, n=4 returns boolOccLength[e][i]
        IF posPresent THEN
          CALL RICE-2 with K=6, Max=262080, n=3 returns poslOccLength[e][i]
        END IF
      END IF
    END IF
  END FOR
  CALL RICE-2 with K=3, Max=8184, n=3 returns normDocCount
END FOR
```

deltaDoc (integer): This field MUST contain the value 1.

boolOccLength(integer array): This field MUST equal the number of bits used in the Boolean occurrences binary data file to list data relevant for the current token in the current property index. Take an element `boolOccLength[i][j]`. The index variable *i* MUST be between 1 and the value contained in the number of tokens in file page field. The index variable *j* MUST be in the range 1 to "number of property indexes". The index variable *j* MUST have the same ordering properties as the property index ordering number.

posOccLength(**integer array):** This field MUST equal the number of bits used in the Boolean occurrences binary data file to list data relevant for the current token in the current property index. Given an element `posOccLength[i][j]`, the index variable *i* is in the range 1 to token count (in the header of current file). The index variable *j* is in the range 1 to "number of property indexes". The index variable *j* has the same ordering properties as the property index ordering number.

padding bits region: This region is used for padding bits after the between binary data field. The number of bits in this region MUST be specified so that the following token offsets region begins on a 32-bit aligned boundary. The bits in this region MUST be set to 0.

2.1.14.4.3.3 Token Offsets

The token offsets region (this region) and the LCP entries region (section 2.1.14.4.3.4) determines the full **string** content for each token.

The first token in the file page MUST have the full string content specified in the dictionary paged index file (section 2.1.14.4.4). It MUST be specified as token ordinal number 1.

The following formula MUST be used to determine the byte offset in the current page at which the token offsets region begins:

```
offset = 16 + sparse size + between size
```

This offset is relative to the beginning of the file page.

The variables on the right side of the preceding formula MUST be determined as specified for the **sparse size** field and **between size** field (section 2.1.14.1.4).

This region consists of token `count - 2` entries and has a length of `(token count - 2) * 2` bytes.

If *token count* is less than two, this region MUST NOT be present, and therefore has a size of 0.

Each offset entry is specified as a 2-byte unsigned integer field in little-endian order.

Token ordinal number 1 MUST NOT have an entry in the token offset region. The first token offset is for token ordinal number 2. The next token offset is for token ordinal number 3, and so on up to the last token ordinal number for the page. Each token offset is specified as a 2 byte unsigned integer field.

The token offsets are stored as represented in the following table.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
token offset for token ordinal number 2																token offset for token ordinal number 3															
token offset for token ordinal number 4																...															
token offset for last token ordinal number																															

The token offsets entries in this region are used to calculate the beginning offset for each LCP entry (section 2.1.14.4.3.4).

The following algorithm MUST be used to find the LCP offset for token ordinal number T:

```
SET tokens region start offset = byte offset in the current file page to the start of this
token offsets region
SET the offset of token T = The value of token offset element number T in the token offset
region.
SET the LCP entry offset of token T = tokens region start offset + the offset of token T
```

To determine the full string content for all tokens in the page, look up the LCP entry offset for all token ordinal numbers. The token identifier for a token is computed by comparing the full string content of each token ordinal number with the string content of the sought token.

2.1.14.4.3.4 LCP Entries

The following algorithm is used to find the LCP start offset:

```
IF token count <= 2 THEN
  SET LCPs start offset = start of token offset field
ELSE
SET LCPs start offset = start of token offset field + (token count - 2) * 2
END IF
```

The following is an explanation of the Longest Common Prefix (LCP) algorithm, and how it is used in this file.

Define an LCP binary tree in which each node in the tree consist of an integer "prefix", and a 0x00 terminated string where the "prefix" first letters have been removed. The top node in the tree has "prefix=0". The child nodes have "prefix" set to the number of the common prefix letter with their parent. For example, if the string value "ABCD" represents a parent node , and the string value "ABGH" represents one child node, then the "prefix" value for the child node is set to 2, and the string value is set to "GH".

The LCP entries region MUST specify the string values for the tokens with token ordinal numbers 2 to the last token ordinal number for the page. The string value for token with token ordinal number 1 MUST be located in the dictionary.idx2 file, as specified in section 2.1.14.4.4. The consecutive LCP bulk fields together define the LCP tree. The format of the **LCP-data** field is specified using the following ABNF grammar. The description of how the different nodes in the tree map to the list of LCP entries follows that. The following algorithm MUST be implemented to read the LCP entries:

```
FOR i = 1 : i < token count   prefix[i] = Read one byte as unsigned integer   LCP[i] = Read
null terminated stringEND FOR
```

prefix (1 byte): An unsigned integer. The first occurrence is for token ordinal number 2, the next is for token ordinal number 3, and so on.

LCP (null terminated string): A null terminated UTF-8 string, as specified in section 2.1.1.2.1. The first element is for token ordinal number 2, the next for token ordinal number 3, and so on.

The tree structure MUST be mapped to the list of LCP entries as follows. Let (n1 be the token ordinal number for the top node of the tree. The number n1 MUST be computed using the following algorithm:

```
SET n1 to 1
WHILE n1 < token count
   SET n1 = n1 * 2
END WHILE
SET n1 = n1 / 2
```

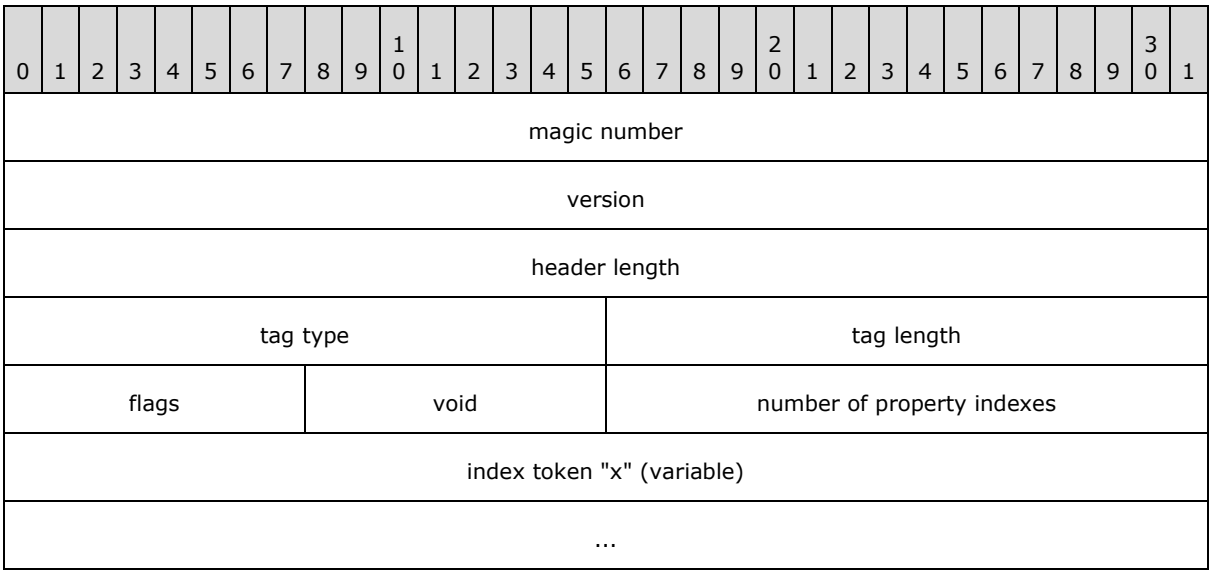
The LCP entries that correspond to different generations of child nodes are computed using the following algorithm:

```
; The two child nodes of the top-node are computed as follows:
SET first-generation-child-1 = n1 + n1/2
SET first-generation-child-2 = n1 - n1/2
; The four child nodes of child nodes of top node are computed as follows:
SET second-generation-child-1 = n1 + n1/2 + n1/4
SET second-generation-child-2 = n1 + n1/2 - n1/4
SET second-generation-child-3 = n1 - n1/2 + n1/4
SET second-generation-child-4 = n1 - n1/2 - n1/4; and so on.
```

2.1.14.4.4 Paged Index File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged*textcatalogname*\dictionary.pidx2".

The query matching component uses this file to locate information for a specified token in the dictionary paged data file stored. The file is specified as follows.



magic number (4 bytes): This is a unsigned 32-bit integer in little-endian order and MUST contain the value 1157702663.

version (4 bytes): The header version for this file. This field MUST contain the value 2.

header length (4 bytes): This integer MUST contain the value 8.

tag type (2 bytes): This integer MUST contain the value 1.

tag length (2 bytes): This integer MUST contain the value 4.

flags (1 byte): This field MUST contain the value 0x1B or value 0x9. If and only if the value is 0x1B then the sparse binary data field and between binary data field in the dictionary paged data file (section 2.1.14.4.3) MUST include position occurrences information. All position occurrences files MUST be present if and only if the value is 0x1B. These are the position occurrences compressed sizes file (section 2.1.14.3.1), the position occurrences compressed occurrence counts file (section 2.1.14.3.2), and the position occurrences binary data file (section 2.1.14.3.3).

void (1 byte): This field MUST be ignored.

number of property indexes (2 bytes): This field MUST contain the number of property indexes in the full-text index context catalog.

index token "x" (variable): This field MUST contain a string in UTF-8 format that represents the token for this entry. The string MUST be terminated with the value 0x00. There MUST be *n* number of index token fields. The tokens are in alphabetical order. The total number of tokens in the index token region is computed from the number of index token strings in this file. The tokens represents a subset of the tokens found in the dictionary sorted hash file, see section 2.1.14.4.5. This file is used to find the file page for a specified token in the dictionary paged data file, see section 2.1.14.4.3.

The algorithm for finding the corresponding file page for a specified token is as follows.

Let the first token in this file be token number 0, the next token number 1, and so on. Find the first token in this index file that occurs alphabetically after the token to retrieve, and store that token number. This number is the file page number for the token to retrieve. For example, let the index contain the tokens "alpha, gamma, upsilon", and assign them numbers 0, 1 and 2. The token to search for is "roma". Because the dictionary is in alphabetical order, the first word that occurs before "roma" is "gamma". The number that is assigned to "gamma" is 1, and therefore the file page number for "roma" is 1.

2.1.14.4.5 Sorted Hash File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged*textcatalogname*\dictionary.shash".

This file enables the indexing component to specify which tokens exist in an index partition.

This file specifies all the keys for the dictionary for current context catalog. It MUST be specified using the following ABNF grammar:

```
dict-shash    = token-count LF *(token-entry LF)
token-count   = 0*11SP 1*12DIGIT
token-entry   = occ-count doc-count token
occ-count     = 1*DIGIT SP
doc-count     = 1*DIGIT SP
token         = UTF8-string
```

dict-shash: This is the dictionary sorted hash file.

token-count: The total number of tokens in the file. The number is prefixed with spaces, so that the total first line length in bytes MUST be 12.

token-entry: A line that specifies a token with associated information. The token-entries MUST be ordered alphabetically by the contained token.

occ-count: The total number of times the token occurred in the index partition. This field MUST be ignored by the query matching component.

doc-count: The number of items that contain the token.

token: A token that was part of an item. This token is represented in UTF-8 format, as specified by the UTF-8 **string** ABNF grammar in section 2.1.1.2.1.

2.1.14.4.6 Token Number Count Index File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged*textcatalogname*\dictionary.wncidx".

This file enables the query matching component to determine which token identifier exists at the beginning of each file page in the dictionary paged count data file.

This file MUST contain in consecutive order, one token identifier for each file page except the first page, as specified in section 2.1.14.4.3. The first token identifier for the first token in each file page MUST be written here. Assume that the dictionary paged data file contains *p* pages. The ABNF grammar for this file is specified as follows:

```
file                = *token-identifier
token-identifier    = 4OCTET
```

file: the file MUST consist of *p-1* token-identifier rules.

token-identifier (4 bytes): Occurrence number *n* of this field MUST equal the unsigned 32-bit integer token identifier of the first token in page number *n+1* in the dictionary paged count data file.

2.1.14.4.7 Token Number Index File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged*textcatalogname*\dictionary.wnidx2".

This file enables the query matching component to determine which token identifier exists at the beginning of each file page in the dictionary paged data file.

This file MUST contain in consecutive order, one token identifier for each file page, except the first, in the dictionary paged data file, for more information see section 2.1.14.1.2. The first token identifier for the first token represented at each file page MUST be written here.

The file is specified using the following ABNF:

```
file              = *token-identifier
token-identifier  = 4OCTET
```

For information about each element in the preceding ABNF, see the ABNF element explanations in section 2.1.14.4.6.

2.1.14.4.8 Warmup File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged*textcatalogname*\dictionary.warmup".

This file enables the query matching component to load into memory information in the dictionary paged data file, as specified in section 2.1.14.4.3 for the tokens specified in this file.

The indexing component MUST generate the token entries in this file for a subset of the tokens in the corresponding dictionary sorted hash file, as specified in section 2.1.14.4.5.

The tokens in this file are sorted in the same order as the corresponding tokens in the dictionary sorted hash file.

The file is specified using the following ABNF format:

```
dictionary-warmup = *token LF
```

```
token              = UTF8-string
```

token: A token that is part of an item. The token MUST be represented as a UTF-8 encoded string.

2.1.15 Integer Occurrence Index Files

2.1.15.1 Overview

The integer index files contain the index and information used by the query matching component for numeric key to item identifiers lookups. A numeric managed property is a managed property of the type **int**, decimal, **float** or **datetime**. There can be zero or more managed properties for these numeric types.

For each numeric managed property, a numeric index directory MUST be made within the numeric context catalog directory. The name of each numeric managed property subdirectory MUST be the same as the string for each property index name, as specified in the ABNF in [MS-FSSCFG] section 2.9.1. The property index name MUST be prefixed with "bidx". The name of the corresponding numeric managed property is the same as the string that follows the "bidx" prefix. In the following integer occurrence file set specification sections, the *fieldname* variable name that is used in the path specification contains the numeric managed property name.

Each integer index file set MUST contain the numeric fields for each property index in the context catalog that have **type** set to "integer" in the index.cf file, as specified in [MS-FSSCFG] section 2.9 .

The numeric fields are used as keys when retrieving entries from the index files. The values that are found by this key-value lookup process are in some cases file offsets, but in other cases the field is a document identifier.

The integer keys in the different integer index files MUST be 8 bytes. The internal integer representation of the value of a numeric field depends on the data type of the managed property. This conversion is specified in section 2.1.1.2.2.

In the specification of the following integer index files, the integer keys are referred to often as low and high integer key in the individual entry types. This clarifies the purpose of the integer key in the specific entry. Low integer key means it will be used for integer range lookups and the low key is the initial value. High integer key is the end value for integer range lookups.

The integer occurrences sparse sparse index file, sparse index file, index file, and data files are related as specified in the following figure.

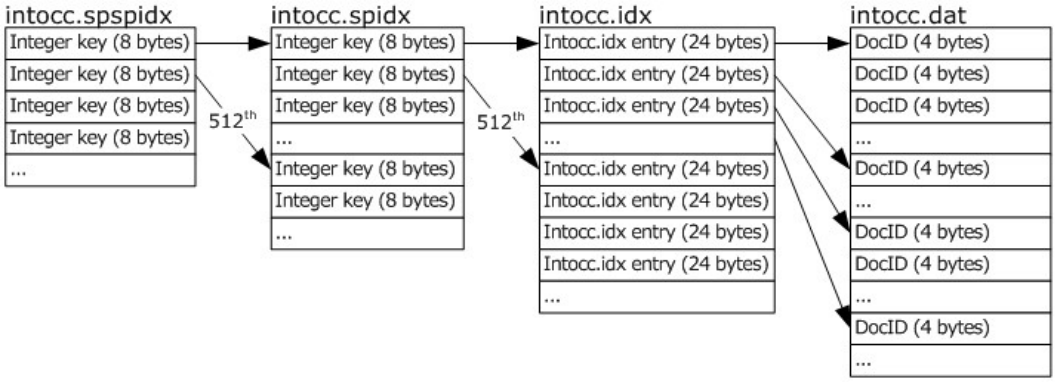


Figure 5: Integer index files relationship

The various integer occurrences bit-vector index files and bit-vector data file are related as specified in the following figure.

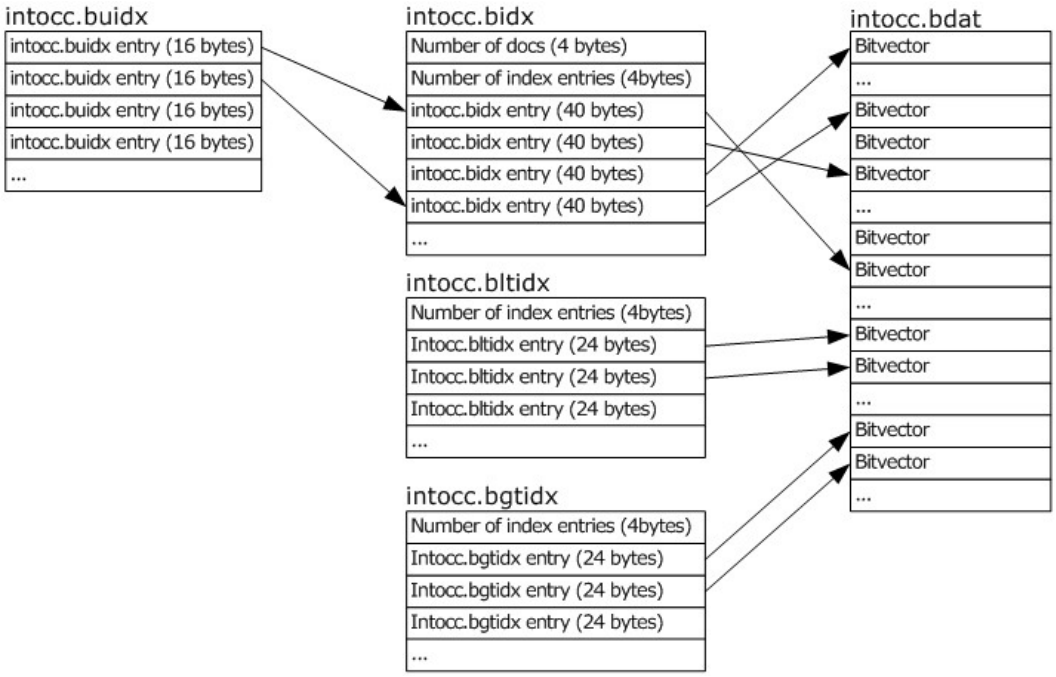


Figure 6: Integer bit-vector index files relationship

A bit-vector is written as specified in the following procedure.

First, a calculation of a **bitmapfactor** value is performed as follows:

1. Calculate the initial **bitmapfactor** value by finding the lowest power of two that is greater than the number of items in the index. For example, with 30089 items, the **bitmapfactor** is first set to 32768.

2. Divide the **bitmapfactor** by 32 to make a **bitmapfactor** of one represent the same amount of disk space usage as one document identifier element (32 bytes) in the intocc.dat file.

3. If the **bitmapfactor** is less than 4096, set it to 4096.

When generating the index, for every new integer key added to intocc.idx, the indexing component MUST determine whether the current total occurrence count, that is the number of items that contain the integer key, is greater than the **bitmapfactor**. If it is greater, then the component writes a bit-vector to intocc.bdat. An intocc.bidx entry MUST also be written with the low integer key set to the low integer key used at the beginning of this round of bit-vector storing. The high integer key is set to the integer key which occurrences caused the occurrence count to add up to more than the current total **bitmapfactor**.

All document identifiers that are associated with integer keys are stored in the intocc.dat file.

2.1.15.2 Bit-vector Data File

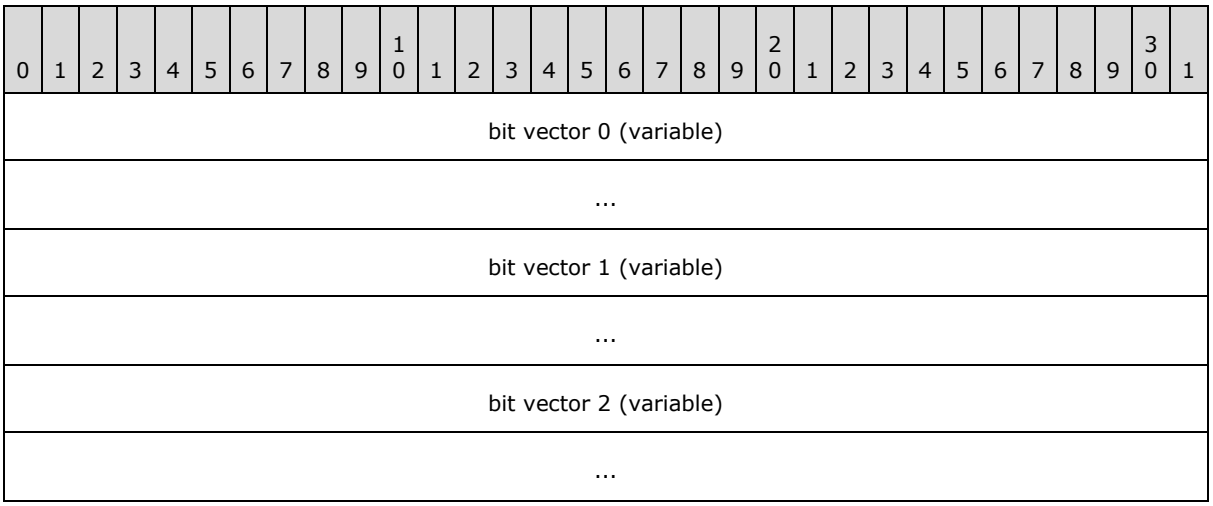
Path and file name of this file MUST be "PP\index_TTTT\index_data\merged\bi1\bidx*fieldname*\intocc.bdat".

This file enables the query matching component to determine which items contain the specified integer value.

This file contains zero or more bit-vectors. Each bit-vector specifies which document identifiers have an integer value in the range between the low and high integer keys inclusively. The low and high integer keys are specified in an entry in the integer occurrence bit-vector index file, as specified in section 2.1.15.4. Each entry in that file points to a bit-vector in this file. The entries in the integer occurrence bit-vector less than index file (section 2.1.15.5) and in the integer occurrence bit-vector greater than index file (section 2.1.15.3) also points to bit-vectors in this file.

The total number of bit-vectors MUST be equal to the sum of the **number of index entries** fields in the header of the integer occurrence bit-vector index file, the bit-vector greater than index file, and the bit-vector less than index files.

The format of this file is specified as follows.



bit-vector (variable): A sequence of bits, in which each bit represents an item identity. Bit number 0 is for item 0, bit 1 for item 1 and so on. Each bit-vector contains at least as many bits as there is **items** in the index. The number of bits MUST be rounded up so the total number of bits is aligned on 32-bit length by using padding. The bits that are not associated with document identifiers MUST be set to zero and MUST be ignored.

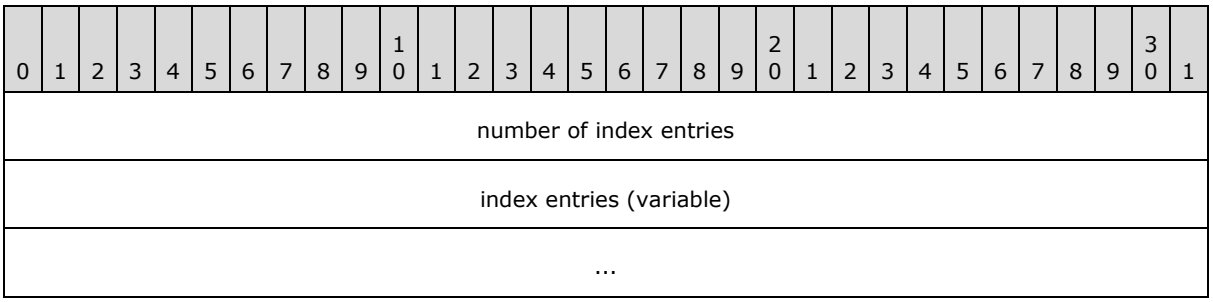
2.1.15.3 Bit-Vector Greater Than Index File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged\bi1\bidx*fieldname*\intocc.bgtidx".

This file enables the query matching component to determine which items have integer values greater than the specified value.

The query matching component uses this file for efficient integer range query evaluation.

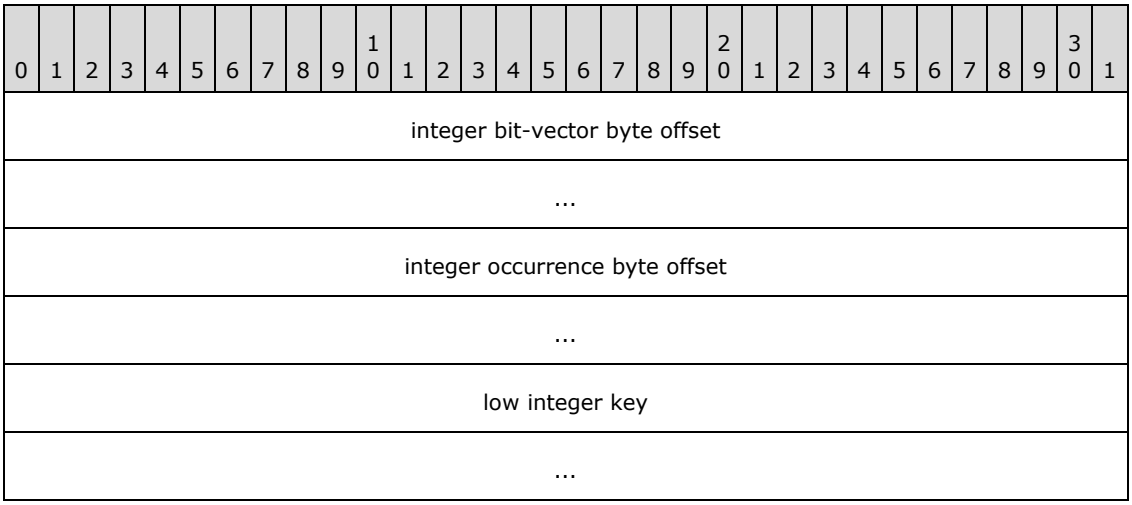
This file contains a header followed by zero or more index entries. The file is specified as follows.



number of index entries (4 bytes): The number of index entries following the header as a unsigned integer field in little-endian order. A value of zero means there are no index entries.

index entries (variable): Zero or more index entries. Each entry is 24 bytes long.

Each index entry MUST be specified as follows.



integer bit-vector byte offset (8 bytes): The initial byte offset for a bit-vector in the integer occurrences bit-vector data file, as specified in section 2.1.15.2. The bit-vector represents the set of document identifiers with numeric values that are greater than the low integer key in this entry. This is an unsigned 64-bit field in little-endian order.

integer occurrence byte offset (8 bytes): The byte offset for the first occurrence of the low integer key in the integer occurrences data file, as specified in section 2.1.15.7. This is an unsigned 64-bit field in little-endian order.

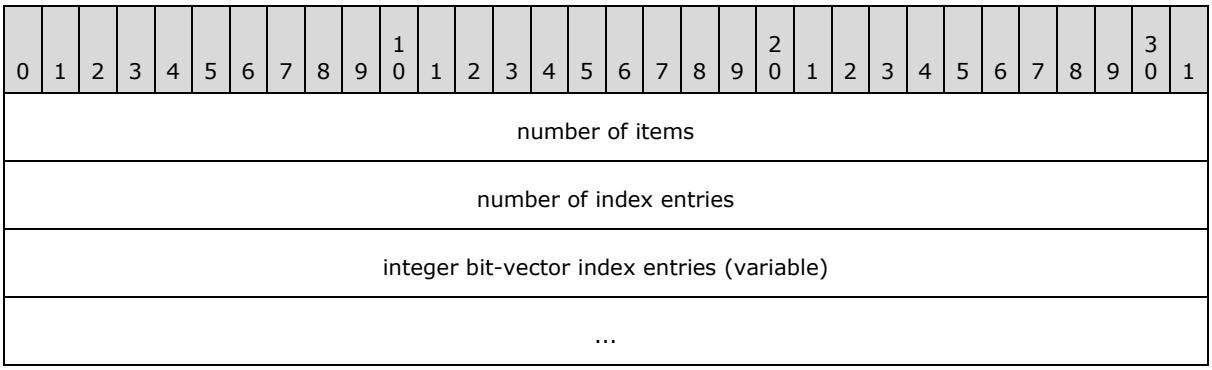
low integer key (8 bytes): The low integer key for this entry. This is of the data type for this numeric managed property, as specified in section 2.1.1.2.2.

2.1.15.4 Bit-vector Index File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged\bi1\bidx*fieldname*\intocc.bidx".

This file enables the query matching component to determine where in the integer occurrence data file and integer occurrence bitvector data file to begin reading the occurrences for the specified integer value.

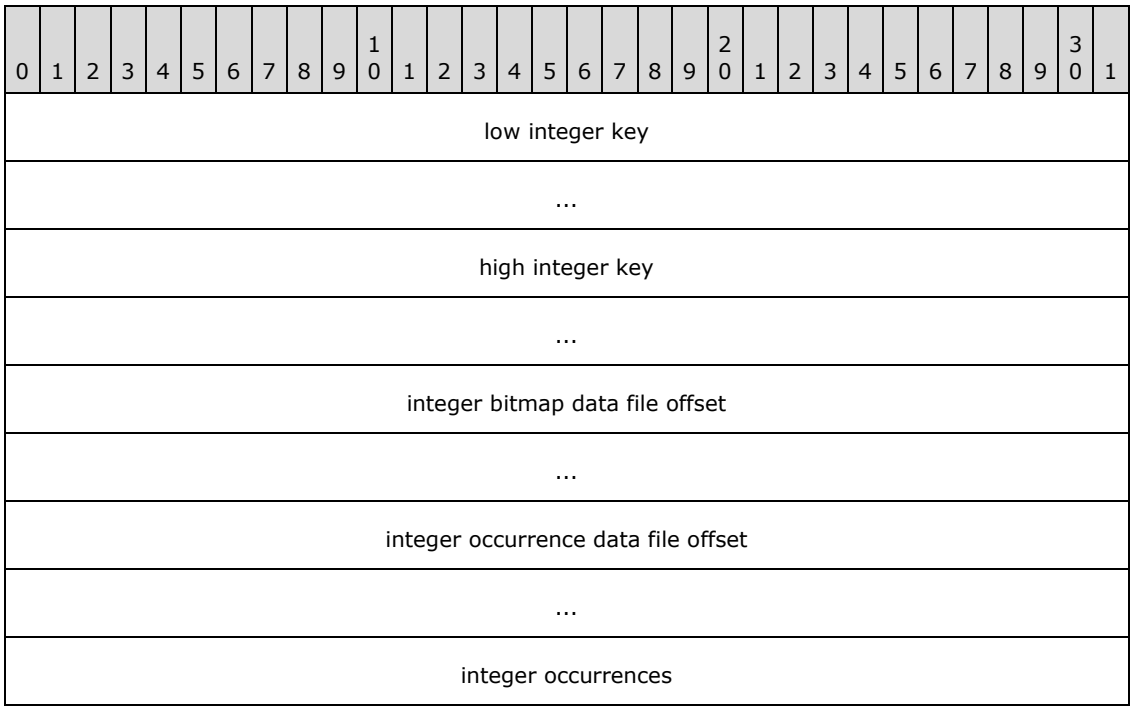
This file contains a header followed by zero or more integer occurrence bit-vector index entries. The file is specified as follows.



number of items (4 bytes): The number of items in the index partition as an unsigned 32-bit field in little-endian order. This field contains the value in the document summary quantity count file (section 2.1.16.5).

number of index entries (4 bytes): The number of integer bit-vector index entries that occur after the header in this file. This is an unsigned 32-bit field in little-endian order. A value of zero means there are no bit-vectors.

integer bit-vector index entries (variable): Zero or more bit-vector index entries. Each entry is 40 bytes long. The entries are sorted in ascending order by low integer key, and then descending order by high integer key. For example, for entries with the same low integer key, the high integer keys MUST be sorted in decreasing order. The format of each entry is specified as follows.



empty value

low integer key (8 bytes): The lowest integer key used by the items in the bit-vector to which this entry points. This field is a signed 64-bit field in little-endian order, and is converted based on the data type for this numeric managed property, as specified in section 2.1.1.2.

high integer key (8 bytes): The highest integer key used by the items in the bit-vector to which this entry points. This field is a signed 64-bit field in little-endian order, and is converted based on the data type for this numeric managed property, as specified in section 2.1.1.2.

integer bitmap data file offset (8 bytes): The byte offset into the integer occurrence bit-vector data file (section 2.1.15.2) where the bit-vector for the range from the low to the high integer key is stored. This field is an unsigned 64-bit field in little-endian order.

integer occurrence data file offset (8 bytes): The offset into the integer occurrence data file (section 2.1.15.7) where the low integer key occurrence entries begin. This field is an unsigned 64-bit field in little-endian order. The offset value is the document identifier element number in the integer occurrence data file. Each document identifier element is 4 bytes long, so the integer occurrence data file offset value 1 in this file MUST be byte offset 4 in the integer occurrence data file, and so on. The document identifier entries in the integer occurrence data file is sorted in ascending order. Therefore the document identifier entries that are between each file offset referenced from this file are also in ascending order.

integer occurrences (4 bytes): The total number of items that have integer values in the low to high integer key range. This is an unsigned 32-bit field in little-endian order. This field contains the number of items for the range from the low to the high integer key in this entry. This is the same as the number of bits set to 1 in the corresponding bit-vector in the integer occurrence bit-vector data file (section 2.1.15.2). It is also the same as the number of document identifier entries for this low to high range in the integer occurrence data file (section 2.1.15.7).

empty value (4 bytes): An empty value, used to enforce 64-bit alignment of consecutive integer bit-vector index entries. This field MUST be zero, and MUST be ignored. This field is specified as an unsigned 32-bit field in little-endian order.

2.1.15.5 Bit-vector Less than Index File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged\bi1\bidx*fieldname*\intocc.bltidx".

This file enables the query matching component to determine which items have integer values less than the specified value.

This file contains a header followed by zero or more index entries. It is specified as follows.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
number of index entries																															
index entries (variable)																															

...

number of index entries (4 bytes): The number of index entries following the header. A value of zero means there are no index entries. This field is an unsigned 32-bit integer field in little-endian order.

index entries (variable): Zero or more index entries. Each entry MUST be 24 bytes long, and is specified as follows.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
integer bit-vector byte offset																															
...																															
integer occurrence byte offset																															
...																															
high integer key																															
...																															

integer bit-vector byte offset (8 bytes): The beginning byte offset for a bit-vector in the integer occurrences bit-vector data file that represents the set of document identifiers with numeric values less than the high integer key in this entry. This field is an unsigned 64-bit field in little-endian order.

integer occurrence byte offset (8 bytes): The byte offset into the intocc.dat file for the first occurrence of the lowest value that is greater than the high integer key. This field is specified as an unsigned 64-bit field in little-endian order.

high integer key (8 bytes): The high integer key for this entry, formatted as the data type for this numeric managed property as specified in section 2.1.1.2.

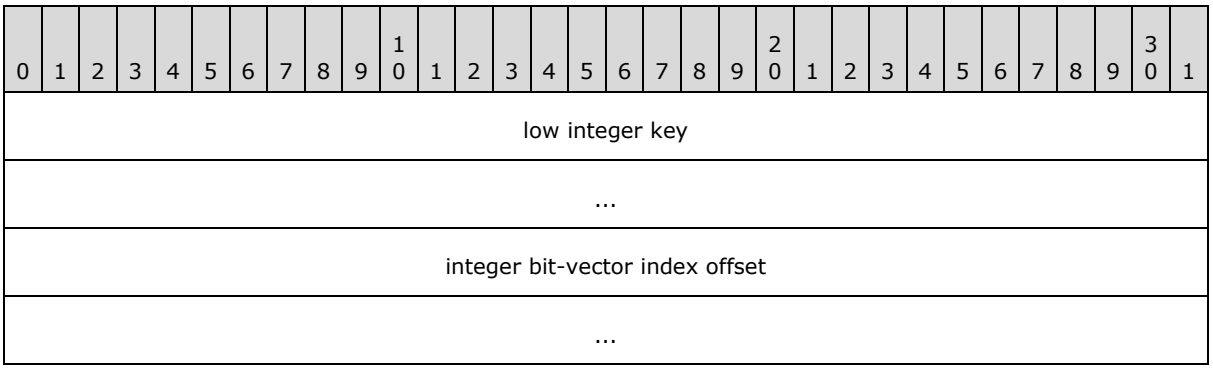
2.1.15.6 Bit-vector Unique Index File

The path and file name of this file MUST be "PP\index_TTTT\index_data\merged\bi1\bidx*fieldname*\intocc.buidx".

This file enables the query matching component to determine where the index entries for each integer key begin in the bit-vector index file.

The entries in this file are sorted in ascending order based on the low integer key. Each unique integer key is specified only once.

The file contains zero or more entries that are specified as follows.



low integer key (8 bytes): The low integer key for this entry, formatted in the data type for this numeric managed property, as specified in section 2.1.1.2.

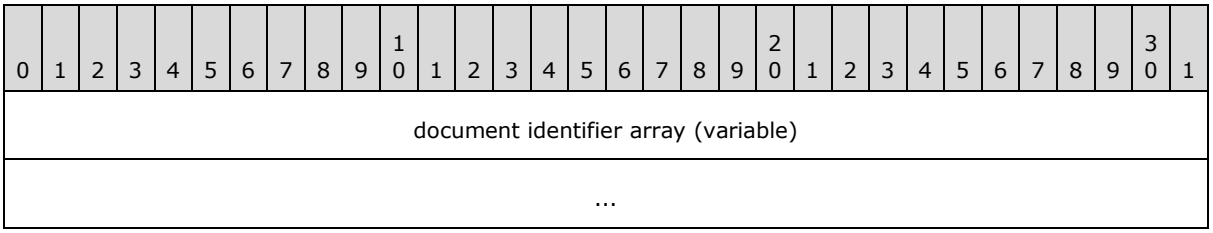
integer bit-vector index offset (8 bytes): The entry number initial offset for the bit-vector index entry in the integer occurrences bit-vector index file with the low integer key that is equal to the low integer key in this entry. This field is an unsigned 64-bit field in little-endian order.

2.1.15.7 Data File

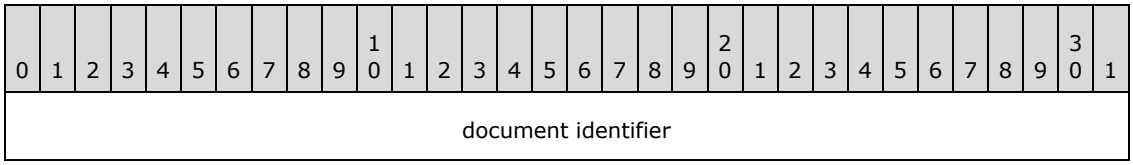
Path and file name of this file MUST be "PP\index_TTTT\index_data\merged\bi1\bidx*fieldname*\intocc.dat".

This file enables the query matching component to determine which items contain the specified integer value. This is used in combination with the other integer index files to find the correct set of items for a range of integer values.

This file contains sequences of one or more 32-bit document identifier values for each 64-bit numeric key that is represented for this field. For each numeric key, the offset value in the file MUST specify the byte offset into this file for the set of document identifiers that has that numeric key. The format of the file is specified as follows.



document identifier array (variable): An array of one or more document identifiers. Each document identifier is as shown in the following table.



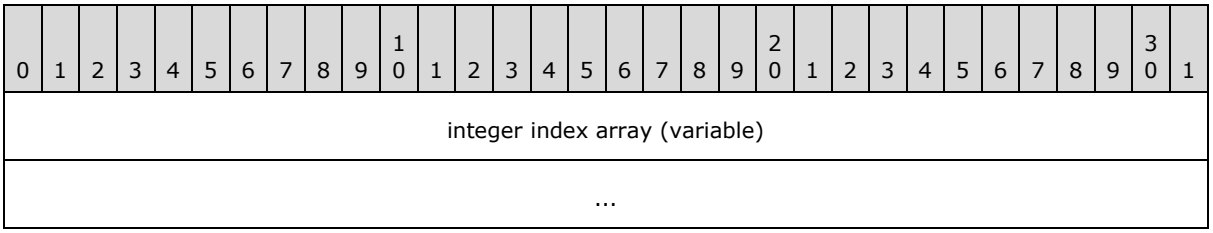
document identifier (4 bytes): An document identifier value that uniquely represents one of the **items** in the index structure. Each document identifier is an unsigned 32-bit field in little-endian order.

2.1.15.8 Index File

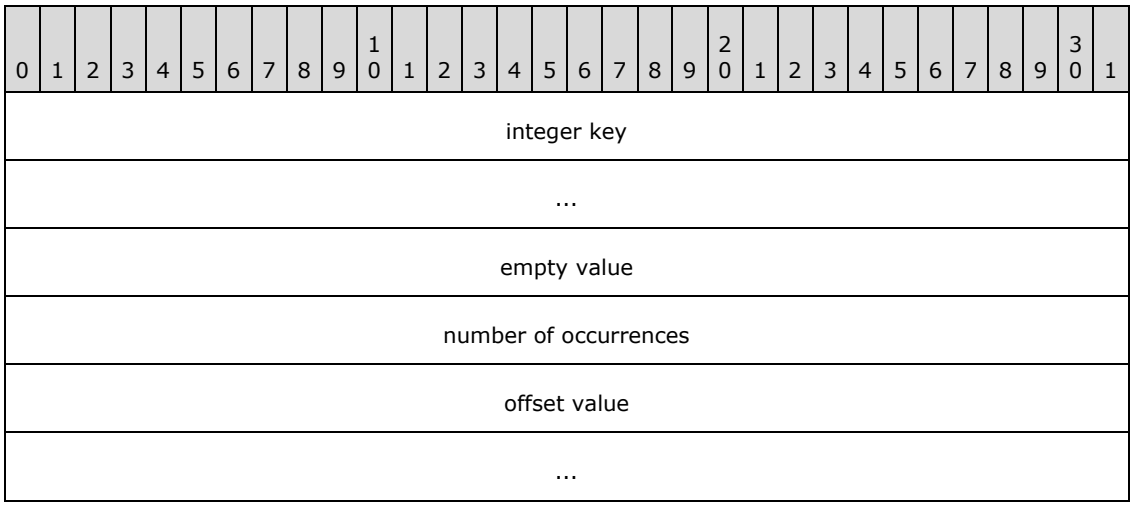
Path and file name of this file MUST be "PP\index_TTTT\index_data\merged\bi1\bidx*fieldname*\intocc.idx".

This file enables the query matching component to specify the offset at which to begin reading the integer occurrences data file for the specified occurrence information, as specified in section 2.1.15.7.

This file contains sets of integer key, number of occurrences and offset values. The file format is specified as follows.



integer index array (variable): An array of integer index entries. An integer index entry is specified in the following table.



integer key (8 bytes): The numeric key for this entry as an 8 byte integer. All numeric type values supported in the **schema object** model will be converted into an 8 byte integer field during indexing. The conversion process from the supported data types to the internal representation is specified in section 2.1.1.2.

empty value (4 bytes): An empty value, that is used to ensure proper alignment for the 64-bit offset value. This field MUST be zero, and MUST be ignored.

number of occurrences (4 bytes): The total number of **items** in which this integer value occurs. Multiple occurrences of a numeric value in one item are counted as only one occurrence. This field is specified as an unsigned 32-bit field in little-endian order.

offset value (8 bytes): This value multiplied by 4 MUST be the byte offset for the initial position in the integer occurrences data file for the sequence of document identifiers that contains

integers equal to the integer key field. This field is an unsigned 64-bit field in little-endian order.

2.1.15.9 Limits File

The path and file name of this file MUST be "PP\index_TTTT\index_data\merged\bi1\bidx*fieldname*\intocc.limits".

This file enables the query matching component to determine the maximum and minimum values for the numeric managed property that is specified in the fieldname.

This file includes the minimum and maximum numeric values for this field, represented as decimals in human-readable ASCII text.

The minimum and maximum values MUST be equivalent to the first and last integer key entries in the intocc.idx file for the same field, with the difference that the numbers are represented as ASCII decimals in the intocc.limits file.

The format of this file is specified using the following ABNF grammar:

```
intocc-limits   = numeric-value64 ":" numeric-value64 LF
                    ; minimum 0, maximum 2^64-1
numeric-value64 = 1*20DIGIT
```

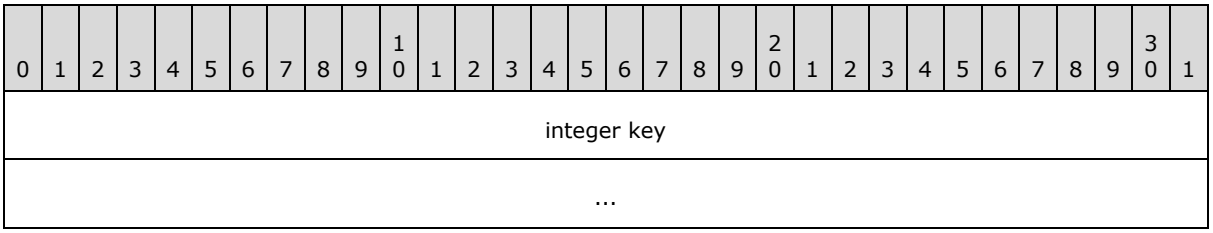
2.1.15.10 Sparse Index File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged\bi1\bidx*fieldname*\intocc.spidx".

This file enables the query matching component to look up every 512th entry in the integer occurrence index file, as specified in section 2.1.15.8.

This file includes one integer key for every 512th integer key entry in the integer occurrences index file. The first integer key in that file MUST be the first entry in this file. Each integer key is 8 bytes.

Each element in this file MUST be specified as follows.



integer key (8 bytes): The integer key that is the next 512th integer key in the integer occurrences index file. There is one or more of these elements in this file.

For more details on the relationships between the integer occurrences index file, the sparse index file, and the sparse sparse index file, see the integer occurrences sparse sparse index file in section 2.1.15.11.

2.1.15.11 Sparse Sparse Index File

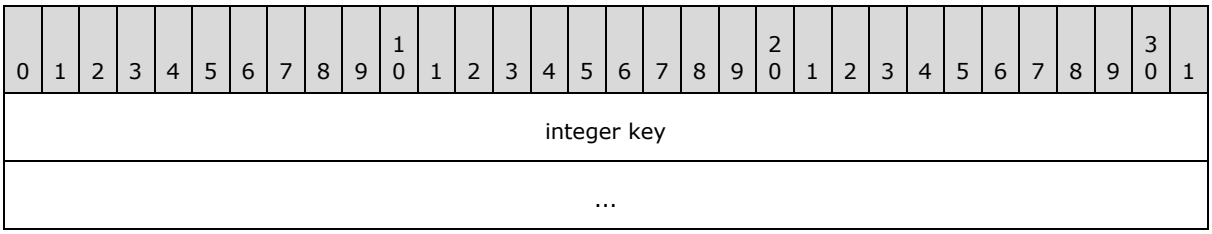
Path and file name of this file MUST be "PP\index_TTTT\index_data\merged\bi1\bidx*fieldname*\intocc.spspidx".

This file enables the query matching component to look up every 512th entry in the integer occurrence sparse index file, as specified in section 2.1.15.10.

This file includes one integer key for every 512th integer key entry in the integer occurrences sparse index file. The first integer key in that file MUST be the first entry in this file. Each integer key is 8 bytes.

The query matching component uses this file to perform fast lookups of a specific integer key in the integer occurrences sparse index file, then perform lookups in the integer occurrences index file (section 2.1.15.8), and finally find the set of **items** for the sought integer key in the integer occurrences data file (section2.1.15.7).

Each element in this file MUST be specified as follows.



integer key (8 bytes): The integer key that is the next 512th integer key in the integer occurrences sparse index file. There is one or more of these elements in this file.

2.1.16 Document Summary Files

2.1.16.1 Overview

The document summary files contain sets of field values for each item in the index.

There is only one set of docsum.idx, docsum.overflow, and docsum.dat files for each index partition, and they are stored directly within the merged directory.

The document summary index and data files are related as specified in the following diagram.

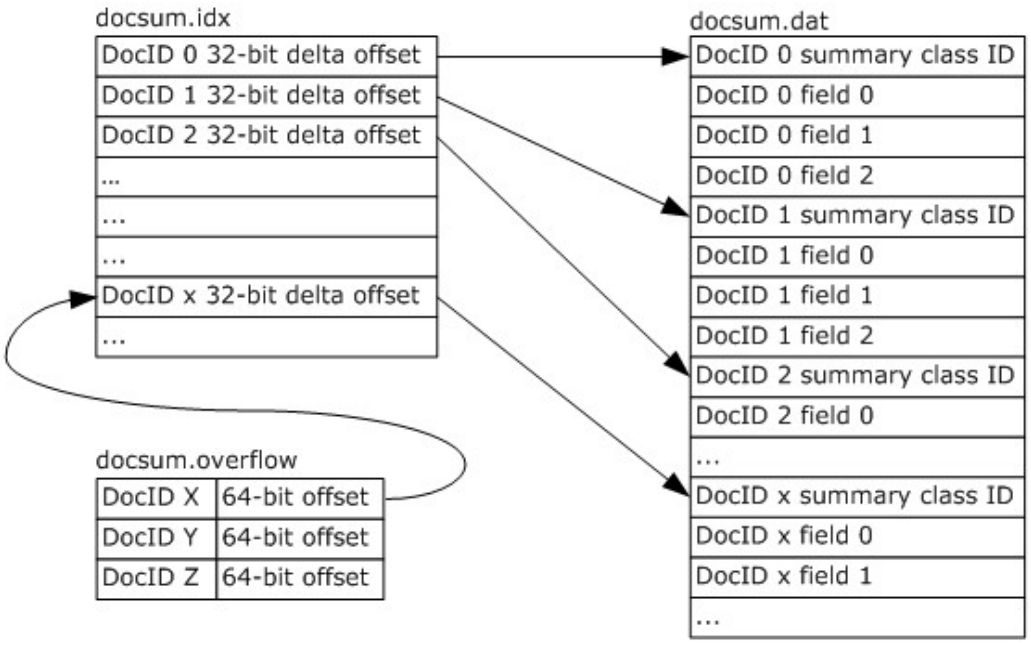


Figure 7: Document summary files relationship

The document summary data file contains different sets of document summaries. The set type is determined by the document summary class identifier. The format of each document summary set is controlled by the definitions for one of the document summary classes in the summary.cf file, as specified in [MS-FSSCFG] section 2.18.

2.1.16.2 Data File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged\docsum.dat".

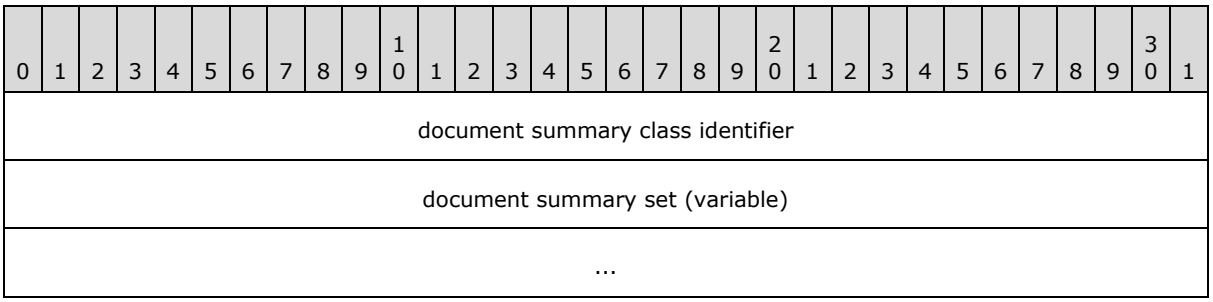
This file enables the query matching component to return to the search front-end the document summaries for one or more requested document identifiers. The document summaries are requested using the protocol specified in [MS-FSDQE].

This file contains the result details view data sets for all document identifiers. The sets are stored item by item; the set associated with document identifier 0 is processed first, then the set associated with document identifier 1, and so on.

The query matching component uses the document summary index file entry for a document identifier to perform the following tasks:

- Find the offset in the document summary data file. The offset is the byte position at which to begin reading the document summary for the specified summary.
- Calculate the length of the document summary set associated with an item. The length is calculated by looking up the offset of the next document identifier, and subtracting the offset for the current document identifier. The difference is the size of the current document identifier.

Each **docsum.dat** element is specified as follows.



document summary class identifier (4 bytes): The summary identifier for this document summary set. This number is an unsigned integer field in little-endian format. The query matching component uses the document summary class identifier to determine how to decode the document summary set. The content of each field in the set is then extracted as specified in the corresponding document summary class identifier in the summary.cf file, as specified in [MS-FSSCFG] section 2.18.

document summary set (variable): The set of document summaries for this document identifier. The entries within the set can be of the following types: **string**, **data**, **longstring**, as specified in following table.

Type	Description
string	A 2-byte length field, followed by a variable length field representing the **string** content. The length field MUST be an unsigned 2-byte field in little-endian order. The **string** field MUST NOT be zero terminated and MUST be in UTF-8 encoding.
data	A 2-byte length field, followed by a variable length field containing the information. The length field MUST be an unsigned 2-byte field in little-endian order. The information in the variable length field MUST be stored untransformed, as represented in the original item that was indexed. Each byte is represented using any byte value from 0 to 255 inclusively, not only ASCII or UTF-8 characters.
longstring	Two 4-byte fields, followed by a variable length compressed information buffer. The two length fields are stored as 4 byte unsigned fields in little-endian order. The first length parameter is computed as the length of the compressed buffer section that follows plus the 4 bytes. In addition, bit 32 of this length field is set to 1, OR the initially calculated length with 0x8000000. The second length parameter contains the original length of the **string** that was compressed. When decompressed the original **string** MUST have a length equivalent to the value of this parameter. In the compressed information buffer each byte field is represented using any value from 0 to 255 inclusively. The compression MUST use the zlib format, as specified in [RFC1950]. The "deflate" compression method MUST be used. When decompressed, the output is a **string** in UTF-8 format that contains the text parts of an item.

The document summary elements in this file MUST be in the same order as the **field** elements for the corresponding document summary class identifier in the summary.cf file, as specified in [MS-FSSCFG] section 2.18.2.

2.1.16.3 Index File

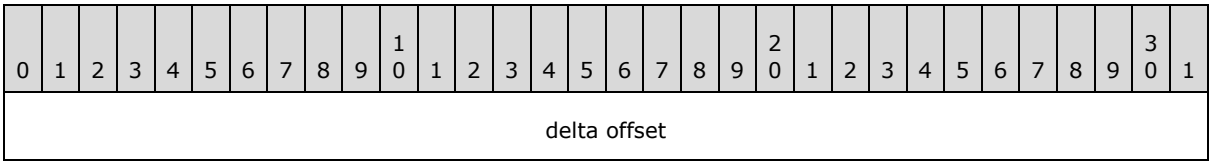
Path and file name of this file MUST be "PP\index_TTTT\index_data\merged\docsum.idx".

This file enables the query matching component to retrieve the offset for the specified item from the document summary data file. The offsets are also used to calculate the size in bytes of each document summary data file's entries.

This file contains the 32-bit offset pointers into the docsum.dat file for all document identifiers. It contains as many entries as there are items in the index partition, plus one entry that specifies the offset just beyond the last entry in the docsum.dat file. Each entry is a 32-bit unsigned field in little-endian order.

The first entry in this file points to the offset (in bytes) for document identifier 0 in docsum.dat, which is at offset 0. The second entry points to the offset for document identifier 1, and so on.

The last entry in this file points to the offset value one byte beyond the end of the docsum.dat file. This last entry and the entry before it MUST be used by the query matching component to calculate the size of the last document summary data file entry. The size in number of bytes in the docsum.dat file MUST be equivalent to the last offset value in the docsum.idx, plus the last offset value in the docsum.overflow file, if the overflow file contains any entries. Each docsum.idx element is specified as follows.



delta offset (4 bytes): The delta offset value to use for lookups in docsum.dat. This is used directly, or in addition to the base offset specified in the docsum.overflow file. The field is an unsigned 32-bit little-endian field. For document identifiers which identifier is lower than the first element in docsum.overflow, the delta offset is used directly for lookup into docsum.dat.

2.1.16.4 Overflow File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged\docsum.overflow".

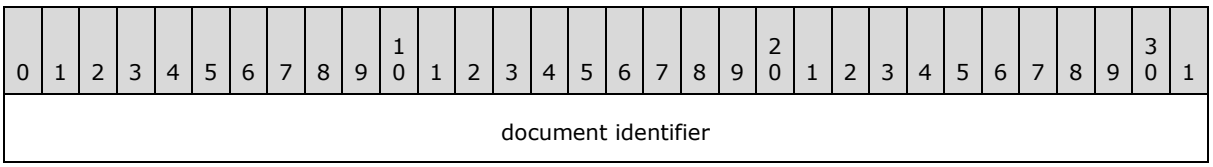
This file enables the query matching component to calculate the correct offset of the document summary of an item when the document summary index file is larger than 2^32 bytes.

This file contains pairs of 64-bit document identifiers and 64-bit offset values.

In case the document summary data file, as specified in section 2.1.16.2, is smaller than 2^32 bytes, this file has no elements and MUST therefore be of size 0.

For every entry added to the document summary index file (section 2.1.16.3) the protocol server MUST determine whether the delta offset value of the new entry causes an overflow. If it will cause an overflow, then an entry MUST be added to the overflow file. The entry in the document summary index file contains the delta value that MUST be added to the offset value for the corresponding entry in this file.

Each entry in this file is specified as follows.



...
offset value
...

document identifier (8 bytes): The document identifier that marks the first document identifier where it is necessary to add the 64-bit offset specified in the same element in this file with the same document identifier and all following document identifiers delta offset in the document summary index file. The document identifier field MUST be specified as an unsigned 64-bit integer field in little-endian order, and range from 0 to 2147483647 (2^31-1).

offset value (8 bytes): This is the base offset in the calculation specified in the preceding document identifier field. This is an unsigned 64-bit integer field in little-endian order.

2.1.16.5 Quantity Count File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged\docsum.qcnt".

This file enables the query matching component to determine the number of items in the index partition.

This file specifies the number of **items** in the index. The number of **items** is written as a numeric ASCII value terminated with the linefeed character 0x0A, and is computed as (size of docsum.idx / 4) - 1.

The file is specified as in the following ABNF grammar:

```
docsum-qcnt    = docs-in-index LF
docs-in-index  = 1*10DIGIT
```

2.1.17 Unique Identity Data File

Path and file name of this file MUST be "PP\index_TTTT\index_data\merged\uniqueid.dat".

This file enables the query matching component to use the item internal identifier to determine which document identifier the specified item contains.

This file contains mapping tables from item internal identifiers to document identifiers. It MUST be as follows. The header occurs first, followed by the mappings pages, which are the main content of the file.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
string "Version"																															
...																														version	
...																														header size	

...	item count
...	pages count
...	page boundary entries
...	
page boundary entry (20 bytes)	
...	

string "Version" (7 bytes): The string "Version" occupies the first 7 bytes of the header, using both uppercase and lowercase characters. This string MUST be in ASCII format.

version (4 bytes): The version field MUST be zero as an unsigned 32-bit integer in little-endian order.

header size (4 bytes): The header size specifies the length in bytes of the header of this file. It is calculated as follows:

```
7+4*4+page_count*20+4+collection_count*4+sum_of_all_collection_lengths.
```

The **page_count** is specified in the **pages count** entry. The **collection_count** is specified in the **collection count** entry. The sum_of_all_collection_lengths is the sum of the collection string length values in the collection string entries section of the header. The total size in bytes of the file specified in this section MUST be `Header size + (16384 * pages count)` bytes. This is an unsigned 32-bit integer in little-endian order.

item count (4 bytes): The number of items in the index, and thereby also specifying the total number of mappings in this file. This is an unsigned 32-bit integer in little-endian order.

pages count (4 bytes): The number of pages. This is an unsigned 32-bit integer in little-endian order.

page boundary entries (variable): This field contains the page boundary entries. The number of **page boundary entry** elements is the same as the **pages count** value specified.

page boundary entry (20 bytes): This specifies the last item mapping on each page. Each **page boundary entry** consists of an item internal identifier and a collection identifier. This field is as specified in the following table.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
item internal identifier (16 bytes)																															
...																															
collection identifier																															

collection count
collection string entries (variable)
...
collection string entry (variable)
...

item internal identifier (16 bytes): A 16 bytes long byte sequence that specifies an item in the index partition. The field is represented in binary format in this field. It contains the same value as the ASCII string representation of the docname-checksum part of an item internal identifier in the document identifier map file, as specified in section 2.1.9.

collection identifier (4 bytes): The collection identifier for the page boundary entry. The collection identifier is the string number in the **collection string entries** table. String number 0 is collection identifier 0, string number 1 is collection identifier 1, and so on. This is an unsigned 32-bit integer in little-endian order.

collection count (4 bytes): The number of collections specified in the **collection string entries** section. This is an unsigned 32-bit integer in little-endian order.

collection string entries (variable): The collection string entries. The number of entries is stored in the **collection count** field.

collection string entry (variable): Each collection string entry contains a **collection string length** field and a **collection string content** field, as specified in the following table.

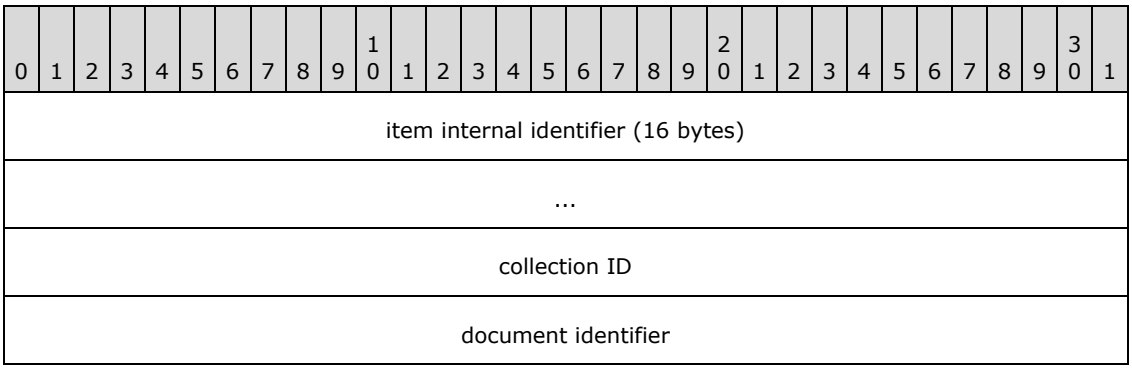
0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
collection string length																															
collection string content (variable)																															
...																															

collection string length (4 bytes): The length of the **collection string content** array following the length. This is an unsigned 32-bit integer in little-endian order.

collection string content (variable): This MUST be equivalent to the collection part of the internal identifier for the item, including the underscore and the text following it. The underscore MUST be included in the string content as the first byte. The string is specified in ASCII format.

The preceding specifies the fields in the header of the uniqueid.dat file. The description of the content in each mapping page in this file follows.

Each mapping page is 16384 bytes. Each page contains the same number of item map entries, except for the last page which can contain fewer entries. An item map entry is specified as follows.



item internal identifier (16 bytes): The 16-bit value for an item internal identifier. This value is represented in hexadecimal in this field. This hexadecimal value represents the same value as the docname-checksum value for an item internal identifier in the document identifier map file.

collection ID (4 bytes): This specifies the collection name with which the item is associated. The collection identifier is the string number in the collection string entries section in the header of this file. This is an unsigned 32-bit integer in little-endian order.

document identifier (4 bytes): The document identifier, as used in the other index files. This is an unsigned 32-bit integer in little-endian order.

An item map entry is 24 bytes. A page is 16,384 bytes. The remaining bytes in a page that are not item map entries MUST be filled with the ASCII character 0x23, the number sign (#).

2.1.18 Duplicates Data File

The path and file name of this file MUST be "PP\index_TTTT\index_data\merged\duplicates.dat".

This file enables the query matching component to determine which document identifiers have duplicates in the index partition. The query matching component discards these duplicates from all search results.

This file can be of size zero, which means there are no duplicate **items** in the index partition.

When the file has entries, each entry specifies a document identifier that has a duplicate.

The file is specified using the following ABNF grammar:

```
duplicates-dat-file  = *(doc-id LF)
doc-id          = 1*10DIGIT
```

doc-id (variable): This is a document identifier that is a duplicate in the index partition. The doc-id text string MUST be equal to the corresponding text string specified after the space character in the urlmap.txt file.

2.1.19 Duplicates Text File

The path and file name of this file MUST be "PP\index_TTTT\index_data\merged\duplicates.txt".

This file enables the query matching component to determine which items with the specified internal identification have duplicates in the index partition. The query matching component discards these duplicates from all search results.

This file MUST contain zero or more entries that specify the duplicate items in the index partition. Each item is associated with an internal identifier. In addition a store identifier string MUST be specified for each internal identifier. See the **internal-id** and **store-id** fields for the specification of these two string values.

The file is specified using the following ABNF grammar;

```
duplicates-text-file = *(duplicate-key LF)
duplicate-key        = internal-id "," store-id
internal-id          = 32HEXDIG "_" collection-name
store-id             = 1*(ALPHA / DIGIT / "_" / "\" / ".")
collection-name      = 1*(ALPHA / DIGIT / "-")
```

duplicate-key: This is a string that specifies an item in the index partition that is a duplicate of another item in the index.

internal-id: This string uniquely identifies each item in the index.

store-id: This string specifies the store from which the item was read by the indexing component. The store can be a full path and file name.

collection-name: This string specifies the content collection to which the item belongs.

2.2 Dictionary File Set

2.2.1 Index Configuration File

Path and file name of this file MUST be "FQDN.normalized.TT\index.cf".

This file is specified in the same manner as the index file set index configuration file, as specified in section 2.1.2.

2.2.2 Index Partition Tuning File

The path and file name of this file MUST be "FQDN.normalized.TT\indextune.cf".

This file is specified in the same manner as the index file set index partition read tuning file, as specified in section 2.1.3.

2.2.3 Stamp Text File

The path and file name of this file MUST be "FQDN.normalized.TT\stamp.txt".

This file is specified in the same manner as the index file set Stamp text file , as specified in section 2.1.6.

2.2.4 Version Information File

The path and file name of this file MUST be "FQDN.normalized.TT\version.txt".

This file is specified in the same manner as the index file set Version information file, as specified in section 2.1.11.

2.2.5 Merged Fusion Dictionary Counts Done Stamp File

The path and file name of this file MUST be "FQDN.normalized.TT\merged\.fusiondictcounts_done".

This file specifies that all files in this file set are complete and consistent.

This file MUST be present, MUST be 1 byte in size, and MUST contain the byte value 0x64, that is, the ASCII character "d".

The file system timestamp for this file specifies when the file set generator finished processing successfully.

2.2.6 Dictionary Paged Count Data File

Path and file name of this file MUST be "FQDN.normalized.TT\merged*textcatalogname*\dictionary.pcdat".

This file is specified in the same manner as the index file set Dictionary paged count data file, as specified in section 2.1.14.4.1.

The tokens included in this file represent the set of tokens for all index partitions that exist on one indexing node.

2.2.7 Dictionary Paged Count Index File

The path and file name of this file MUST be "FQDN.normalized.TT\merged*textcatalogname*\dictionary.pcidx".

This file is specified in the same manner as the index file set Dictionary paged count index file, as specified in section 2.1.14.4.2.

2.2.8 Dictionary Token number Count Index File

The path and file name of this file MUST be "FQDN.normalized.TT\merged*textcatalogname*\dictionary.wncidx".

This file MUST be specified the same way as the index file set Dictionary token number count index file, as specified in section 2.1.14.4.6.

2.3 State File Set

2.3.1 Index Set Generation File

This file specifies the most recent generation of index files. Path and file name of this file MUST be "state\indexsetgeneration".

This file is specified using the following ABNF grammar:

```
indexsetgeneration = generation
generation         = 1*DIGIT
```

generation: This is an unsigned integer field that specifies the currently active generation of the index set. This is a unique value for each new generation. The field is encoded as an ASCII string.

2.3.2 Index Set Stamp File

This file specifies the timestamp for the most recent generation of index files. Path and file name of this file MUST be "state\stamp.txt".

This file is specified using the following ABNF grammar:

```
stamp-txt = timestamp
timestamp = 1*DIGIT
```

timestamp: This is the timestamp for the current generation. This field specifies the point in time when this generation of the index partitions set was made available to the query matching component. The timestamp is specified as the number of seconds after 1970-01-01T00:00:00UTC. The field is encoded as an ASCII string.

2.3.3 Index Partition Stamp File

The path and file name of this file MUST be "state\PP\stamp.txt".

This file specifies the timestamp for index partition **PP**.

This file is specified using the following ABNF grammar:

```
stamp-txt = timestamp
timestamp = 1*DIGIT
```

timestamp: This is the timestamp that specifies when this index partition was generated. The timestamp is specified as the number of seconds after 1970-01-01T00:00:00 UTC. The field is encoded as an ASCII string.

2.3.4 Index Partition Index Valid File

File path MUST be "state\PP\index_valid".

This file specifies the timestamp for index partition **PP** when the indexing component verified that the index was valid. The timestamp in this file can be different from the timestamp in section 2.3.3.

This file is specified using the following ABNF grammar:

```
index-valid  = timestamp-ns
timestamp-ns = 1*DIGIT
```

timestamp-ns: This is the timestamp for when this index partition was generated. It represents the number of nanoseconds after 1970-01-01T00:00:00UTC. The field is encoded as an ASCII string.

2.4 Generation File Set

2.4.1 Stamp File

The path and file name of this file MUST be "PP\index_TTTT\NN\stamp.txt".

This field specifies the time when this generation of index partitions was made available to the query matching component.

This file is specified using the following ABNF grammar:

```
stamp-txt = timestamp
timestamp = 1*DIGIT
```

timestamp: This is the timestamp for the index partition generation. This field represents the time when this generation of the index partitions set was made available to the query matching component. It MUST be specified as the number of seconds after 1970-01-01T00:00:00UTC. The field is encoded as an ASCII string.

2.4.2 Sorted Document Identifier Map File

The path and file name of this file MUST be "PP\index_TTTT\NN\urlmap_sorted.txt".

This file MUST be specified the same way as the index file set Sorted document identifier map file, as specified in section 2.1.10.

This file MUST be present.

The entries in this file specify all items in the index partition, and indicate for each item whether it will be excluded or included in search results by the query matching component.

For each generation, that is, the **NN** value in the path, this file specifies the state of excluded and included items for that generation.

2.4.3 Exclusion Listed File

Path and file name of this file MUST be "PP\index_TTTT\NN\exclusionlisted.txt".

This file enables the indexing component to maintain a record of which items have been excluded from search results by the query matching component.

This file MUST only be present if there are items to be excluded from search results.

When present, this file contains the internal identifiers for the items that have been excluded by the query matching component that reads the files in this index partition. The excluded items MUST not be returned in any query results.

For each generation, that is, the **NN** field in the path, this file reflects the excluded items for that generation.

This file is specified using the following ABNF grammar:

```
exclusionlisted-txt = 1*2147483647(internal-id LF)
internal-id       = docname-checksum "_" collection-name
docname-checksum  = 32HEXDIG
collection-name   = *(ALPHA / DIGIT / "-")
```

docname-checksum: This is an MD5 hash of the original name of the document.

collection-name: This is a string that specifies the content collection to which the item belongs.

2.5 Counter File Set

2.5.1 Activated Counter File

Path and file name of this file MUST be "PP\activated_counter\counter".

This file is specified using the following ABNF grammar:

```
counter-file = count-value
count-value = 1*DIGIT
```

count-value: This is an unsigned integer that specifies how many times this index partition was activated for use, as specified in [MS-FSIPA]. The field is encoded as an ASCII string.

2.5.2 Activated Counter Stamp File

The path and file name of this file MUST be "PP\activated_counter\stamp.txt".

This file is specified using the following ABNF grammar:

```
indexsetgeneration = timestamp
timestamp          = 1*DIGIT
```

timestamp: This is a string that specifies the time of creation for the Activated Counter file, as specified in section 2.5.1. The timestamp represents the number of seconds after 1970-01-01T00:00:00UTC. The field is encoded as an ASCII string.

2.5.3 Activated Indexed Counter File

Path and file name of this file MUST be "PP\activated_indexed_counter\counter".

This file is specified using the following ABNF grammar:

```
counter-file = count-value
count-value = 1*DIGIT
```

count-value: This is an unsigned integer that specifies how many times this index partition was both indexed and activated for use, as specified in [MS-FSIPA]. The field is encoded as an ASCII string.

2.5.4 Activated Indexed Counter Stamp File

The path and file name of this file MUST be "PP\activated_indexed_counter\stamp.txt".

This file is specified using the following ABNF grammar:

```
indexsetgeneration = timestamp
timestamp          = 1*DIGIT
```

timestamp: This is a string that specifies the time of creation for the index counter file, as specified in section 2.5.5. The timestamp represents the number of seconds after 1970-01-01T00:00:00UTC. The field is encoded as an ASCII string.

2.5.5 Index Counter File

The path and file name of this file MUST be "PP\index_counter\counter".

This file is specified using the following ABNF grammar:

```
counter-file = count-value
count-value = 1*DIGIT
```

count-value: This is an unsigned integer that specifies how many times this index partition has been indexed. The field is encoded as an ASCII string.

2.5.6 Index Counter Stamp File

The path and file name of this file MUST be "PP\index_counter\stamp.txt".

This file is specified using the following ABNF grammar:

```
indexsetgeneration = timestamp
timestamp          = 1*DIGIT
```

timestamp: This is a string that specifies the time of creation for the index counter file, as specified in section 2.5.5. The timestamp represents the number of seconds after 1970-01-01T00:00:00UTC. The field is encoded as an ASCII string.

3 Structure Examples

3.1 Full Index Directory Structure

The following lists all the files present in the file data index, where the top directory, here symbolized by "." is the directory PP\index_TTTT\index_data.

.\index.cf

.\IndexedOK

.\indextune.cf

.\merged\.findex_done

.\merged\anchortext\complete\boolocc.bdat

.\merged\anchortext\complete\boolocc.bidx

.\merged\anchortext\complete\boolocc.ccnt

.\merged\anchortext\complete\boolocc.dat.ccnt

.\merged\anchortext\complete\boolocc.dat.compressed

.\merged\anchortext\dictionary.pcdat

.\merged\anchortext\dictionary.pcidx

.\merged\anchortext\dictionary.pdat2

.\merged\anchortext\dictionary.pidx2

.\merged\anchortext\dictionary.shash

.\merged\anchortext\dictionary.wncidx

.\merged\anchortext\dictionary.wnidx2

.\merged\assocqueries\complete\boolocc.bdat

.\merged\assocqueries\complete\boolocc.bidx

.\merged\assocqueries\complete\boolocc.ccnt

.\merged\assocqueries\complete\boolocc.dat.ccnt

.\merged\assocqueries\complete\boolocc.dat.compressed

.\merged\assocqueries\dictionary.pcdat

.\merged\assocqueries\dictionary.pcidx

.\merged\assocqueries\dictionary.pdat2

.\merged\assocqueries\dictionary.pidx2

.\merged\assocqueries\dictionary.shash

.\merged\assocqueries\dictionary.wncidx

.\merged\assocqueries\dictionary.wnidx2

.\merged\attributevector-indexing.txt

.\merged\attributevector.txt

.\merged\batvcrawltime.dat

.\merged\batvcrawltime.eidx

.\merged\batvcrawltime.info

.\merged\batvcrawltime.sudat

.\merged\batvcreated.dat

.\merged\batvcreated.eidx

.\merged\batvcreated.info

.\merged\batvcreated.sudat

.\merged\batvdocdatetime.dat

.\merged\batvdocdatetime.eidx

.\merged\batvdocdatetime.info

.\merged\batvdocdatetime.sudat

.\merged\batvdocrank.dat

.\merged\batvdocrank.eidx

.\merged\batvdocrank.info

.\merged\batvdocrank.sudat

.\merged\batvlastmodifiedtime.dat

.\merged\batvlastmodifiedtime.eidx

.\merged\batvlastmodifiedtime.info

.\merged\batvlastmodifiedtime.sudat

.\merged\batvprocessingtime.dat

.\merged\batvprocessingtime.eidx

.\merged\batvprocessingtime.info

.\merged\batvprocessingtime.sudat

.\merged\batvsiterank.dat

.\merged\batvsiterank.eidx

.\merged\batvsiterank.info

.\merged\batvsiterank.sudat

.\merged\batvsize.dat

.\merged\batvsize.eidx

.\merged\batvsize.info

.\merged\batvsize.sudat

.\merged\batvtitle.dat

.\merged\batvtitle.eidx

.\merged\batvtitle.info

.\merged\batvtitle.sudat

.\merged\batvurldepthrank.dat

.\merged\batvurldepthrank.eidx

.\merged\batvurldepthrank.info

.\merged\batvurldepthrank.sudat

.\merged\bavnauthor.dat

.\merged\bavnauthor.eidx

.\merged\bavnauthor.idx

.\merged\bavnauthor.info

.\merged\bavnauthor.sudat

.\merged\bavncompanies.dat

.\merged\bavncompanies.eidx

.\merged\bavncompanies.idx

.\merged\bavncompanies.info

.\merged\bavncompanies.sudat

.\merged\bavnconcepts.dat

.\merged\bavnconcepts.eidx

.\merged\bavnconcepts.idx

.\merged\bavnconcepts.info

.\merged\bavnconcepts.sudat

.\merged\bavndocdatetime.dat

.\merged\bavndocdatetime.eidx

.\merged\bavndocdatetime.idx

.\merged\bavndocdatetime.info

.\merged\bavndocdatetime.sudat

.\merged\bavnemails.dat

.\merged\bavnemails.eidx

.\merged\bavnemails.idx

.\merged\bavnemails.info

.\merged\bavnemails.sudat

.\merged\bavnformat.dat

.\merged\bavnformat.eidx

.\merged\bavnformat.idx

.\merged\bavnformat.info

.\merged\bavnformat.sudat

.\merged\bavnlanguages.dat

.\merged\bavnlanguages.eidx

.\merged\bavnlanguages.idx

.\merged\bavnlanguages.info

.\merged\bavnlanguages.sudat

.\merged\bavnlocations.dat

.\merged\bavnlocations.eidx

.\merged\bavnlocations.idx

.\merged\bavnlocations.info

.\merged\bavnlocations.sudat

.\merged\bavnpersonnames.dat

.\merged\bavnpersonnames.eidx

.\merged\bavnpersonnames.idx

.\merged\bavnpersonnames.info

.\merged\bavnpersonnames.sudat

.\merged\bavnsize.dat

.\merged\bavnsize.eidx

.\merged\bavnsize.idx

.\merged\bavnsize.info

.\merged\bavnsize.sudat

.\merged\bcatcontent\bidxcontentlvl1\boolocc.bdat

.\merged\bcatcontent\bidxcontentlvl1\boolocc.bidx

.\merged\bcatcontent\bidxcontentlvl1\boolocc.ccnt

.\merged\bcatcontent\bidxcontentlvl1\boolocc.dat.ccnt

.\merged\bcatcontent\bidxcontentlvl1\boolocc.dat.compressed

.\merged\bcatcontent\bidxcontentlvl1\posocc.ccnt

.\merged\bcatcontent\bidxcontentlvl1\posocc.counts.ccnt

.\merged\bcatcontent\bidxcontentlvl1\posocc.dat.compressed

.\merged\bcatcontent\bidxcontentlvl2\boolocc.bdat

.\merged\bcatcontent\bidxcontentlvl2\boolocc.bidx

.\merged\bcatcontent\bidxcontentlvl2\boolocc.ccnt

.\merged\bcatcontent\bidxcontentlvl2\boolocc.dat.ccnt

.\merged\bcatcontent\bidxcontentlvl2\boolocc.dat.compressed

.\merged\bcatcontent\bidxcontentlvl2\posocc.ccnt

.\merged\bcatcontent\bidxcontentlvl2\posocc.counts.ccnt

.\merged\bcatcontent\bidxcontentlvl2\posocc.dat.compressed

.\merged\bcatcontent\bidxcontentlvl3\boolocc.bdat

.\merged\bcatcontent\bidxcontentlvl3\boolocc.bidx

.\merged\bcatcontent\bidxcontentlvl3\boolocc.ccnt

.\merged\bcatcontent\bidxcontentlvl3\boolocc.dat.ccnt

.\merged\bcatcontent\bidxcontentlvl3\boolocc.dat.compressed

.\merged\bcatcontent\bidxcontentlvl3\posocc.ccnt

.\merged\bcatcontent\bidxcontentlvl3\posocc.counts.ccnt

.\merged\bcatcontent\bidxcontentlvl3\posocc.dat.compressed

.\merged\bcatcontent\bidxcontentlvl4\boolocc.bdat

.\merged\bcatcontent\bidxcontentlvl4\boolocc.bidx

.\merged\bcatcontent\bidxcontentlvl4\boolocc.ccnt

.\merged\bcatcontent\bidxcontentlvl4\boolocc.dat.ccnt

.\merged\bcatcontent\bidxcontentlvl4\boolocc.dat.compressed

.\merged\bcatcontent\bidxcontentlvl4\posocc.ccnt

.\merged\bcatcontent\bidxcontentlvl4\posocc.counts.ccnt

.\merged\bcatcontent\bidxcontentlvl4\posocc.dat.compressed

.\merged\bcatcontent\dictionary.pcdat

.\merged\bcatcontent\dictionary.pcidx

.\merged\bcatcontent\dictionary.pdat2

.\merged\bcatcontent\dictionary.pidx2

.\merged\bcatcontent\dictionary.shash

.\merged\bcatcontent\dictionary.wncidx

.\merged\bcatcontent\dictionary.wnidx2

.\merged\bi1\bidxcrawltime\intocc.bdat

.\merged\bi1\bidxcrawltime\intocc.bgtidx

.\merged\bi1\bidxcrawltime\intocc.bidx

.\merged\bi1\bidxcrawltime\intocc.bltidx

.\merged\bi1\bidxcrawltime\intocc.buidx

.\merged\bi1\bidxcrawltime\intocc.dat

.\merged\bi1\bidxcrawltime\intocc.idx

.\merged\bi1\bidxcrawltime\intocc.limits

.\merged\bi1\bidxcrawltime\intocc.spidx

.\merged\bi1\bidxcrawltime\intocc.spspidx

.\merged\bi1\bidxcreated\intocc.bdat

.\merged\bi1\bidxcreated\intocc.bgtidx

.\merged\bi1\bidxcreated\intocc.bidx

.\merged\bi1\bidxcreated\intocc.bltidx

.\merged\bi1\bidxcreated\intocc.buidx

.\merged\bi1\bidxcreated\intocc.dat

.\merged\bi1\bidxcreated\intocc.idx

.\merged\bi1\bidxcreated\intocc.limits

.\merged\bi1\bidxcreated\intocc.spidx

.\merged\bi1\bidxcreated\intocc.spspidx

.\merged\bi1\bidxdocdatetime\intocc.bdat

.\merged\bi1\bidxdocdatetime\intocc.bgtidx

.\merged\bi1\bidxdocdatetime\intocc.bidx

.\merged\bi1\bidxdocdatetime\intocc.bltidx

.\merged\bi1\bidxdocdatetime\intocc.buidx

.\merged\bi1\bidxdocdatetime\intocc.dat

.\merged\bi1\bidxdocdatetime\intocc.idx

.\merged\bi1\bidxdocdatetime\intocc.limits

.\merged\bi1\bidxdocdatetime\intocc.spidx

.\merged\bi1\bidxdocdatetime\intocc.spspidx

.\merged\bi1\bidxdocrank\intocc.bdat

.\merged\bi1\bidxdocrank\intocc.bgtidx

.\merged\bi1\bidxdocrank\intocc.bidx

.\merged\bi1\bidxdocrank\intocc.bltidx

.\merged\bi1\bidxdocrank\intocc.buidx

.\merged\bi1\bidxdocrank\intocc.dat

.\merged\bi1\bidxdocrank\intocc.idx

.\merged\bi1\bidxdocrank\intocc.limits

.\merged\bi1\bidxdocrank\intocc.spidx

.\merged\bi1\bidxdocrank\intocc.spspidx

.\merged\bi1\bidxlastmodifiedtime\intocc.bdat

.\merged\bi1\bidxlastmodifiedtime\intocc.bgtidx

.\merged\bi1\bidxlastmodifiedtime\intocc.bidx

.\merged\bi1\bidxlastmodifiedtime\intocc.bltidx

.\merged\bi1\bidxlastmodifiedtime\intocc.buidx

.\merged\bi1\bidxlastmodifiedtime\intocc.dat

.\merged\bi1\bidxlastmodifiedtime\intocc.idx

.\merged\bi1\bidxlastmodifiedtime\intocc.limits

.\merged\bi1\bidxlastmodifiedtime\intocc.spidx

.\merged\bi1\bidxlastmodifiedtime\intocc.spspidx

.\merged\bi1\bidxprocessingtime\intocc.bdat

.\merged\bi1\bidxprocessingtime\intocc.bgtidx

.\merged\bi1\bidxprocessingtime\intocc.bidx

.\merged\bi1\bidxprocessingtime\intocc.bltidx

.\merged\bi1\bidxprocessingtime\intocc.buidx

.\merged\bi1\bidxprocessingtime\intocc.dat

.\merged\bi1\bidxprocessingtime\intocc.idx

.\merged\bi1\bidxprocessingtime\intocc.limits

.\merged\bi1\bidxprocessingtime\intocc.spidx

.\merged\bi1\bidxprocessingtime\intocc.spspidx

.\merged\bi1\bidxsiterank\intocc.bdat

.\merged\bi1\bidxsiterank\intocc.bgtidx

.\merged\bi1\bidxsiterank\intocc.bidx

.\merged\bi1\bidxsiterank\intocc.bltidx

.\merged\bi1\bidxsiterank\intocc.buidx

.\merged\bi1\bidxsiterank\intocc.dat

.\merged\bi1\bidxsiterank\intocc.idx

.\merged\bi1\bidxsiterank\intocc.limits

.\merged\bi1\bidxsiterank\intocc.spidx

.\merged\bi1\bidxsiterank\intocc.spspidx

.\merged\bi1\bidxsize\intocc.bdat

.\merged\bi1\bidxsize\intocc.bgtidx

.\merged\bi1\bidxsize\intocc.bidx

.\merged\bi1\bidxsize\intocc.bltidx

.\merged\bi1\bidxsize\intocc.buidx

.\merged\bi1\bidxsize\intocc.dat

.\merged\bi1\bidxsize\intocc.idx

.\merged\bi1\bidxsize\intocc.limits

.\merged\bi1\bidxsize\intocc.spidx

.\merged\bi1\bidxsize\intocc.spspidx

.\merged\bi1\bidxurldepthrank\intocc.bdat

.\merged\bi1\bidxurldepthrank\intocc.bgtidx

.\merged\bi1\bidxurldepthrank\intocc.bidx

.\merged\bi1\bidxurldepthrank\intocc.bltidx

.\merged\bi1\bidxurldepthrank\intocc.buidx

.\merged\bi1\bidxurldepthrank\intocc.dat

.\merged\bi1\bidxurldepthrank\intocc.idx

.\merged\bi1\bidxurldepthrank\intocc.limits

.\merged\bi1\bidxurldepthrank\intocc.spidx

.\merged\bi1\bidxurldepthrank\intocc.spspidx

.\merged\docsum.dat

.\merged\docsum.idx

.\merged\docsum.overflow

.\merged\docsum.qcnt

.\merged\msynthcat\all\boolocc.bdat

.\merged\msynthcat\all\boolocc.bidx

.\merged\msynthcat\all\boolocc.ccnt

.\merged\msynthcat\all\boolocc.dat.ccnt

.\merged\msynthcat\all\boolocc.dat.compressed

.\merged\msynthcat\all\posocc.ccnt

.\merged\msynthcat\all\posocc.counts.ccnt

.\merged\msynthcat\all\posocc.dat.compressed

.\merged\msynthcat\dictionary.pcdat

.\merged\msynthcat\dictionary.pcidx

.\merged\msynthcat\dictionary.pdat2

.\merged\msynthcat\dictionary.pidx2

.\merged\msynthcat\dictionary.shash

.\merged\msynthcat\dictionary.wncidx

.\merged\msynthcat\dictionary.wnidx2

.\merged\[a]_bscpxml\all\boolocc.bdat

.\merged\[a]_bscpxml\all\boolocc.bidx

.\merged\[a]_bscpxml\all\boolocc.ccnt

.\merged\[a]_bscpxml\all\boolocc.dat.ccnt

.\merged\[a]_bscpxml\all\boolocc.dat.compressed

.\merged\[a]_bscpxml\all\posocc.ccnt

.\merged\[a]_bscpxml\all\posocc.counts.ccnt

.\merged\[a]_bscpxml\all\posocc.dat.compressed

.\merged\[a]_bscpxml\dictionary.pcdat

.\merged\[a]_bscpxml\dictionary.pcidx

.\merged\[a]_bscpxml\dictionary.pdat2

.\merged\[a]_bscpxml\dictionary.pidx2

.\merged\[a]_bscpxml\dictionary.shash

.\merged\[a]_bscpxml\dictionary.wncidx

.\merged\[a]_bscpxml\dictionary.wnidx2

.\merged\[c]_bscpxml\all\boolocc.bdat

.\merged\[c]_bscpxml\all\boolocc.bidx

.\merged\[c]_bscpxml\all\boolocc.ccnt

.\merged\[c]_bscpxml\all\boolocc.dat.ccnt

.\merged\[c]_bscpxml\all\boolocc.dat.compressed

.\merged\[c]_bscpxml\all\posocc.ccnt

.\merged\[c]_bscpxml\all\posocc.counts.ccnt

.\merged\[c]_bscpxml\all\posocc.dat.compressed

.\merged\[c]_bscpxml\dictionary.pcdat

.\merged\[c]_bscpxml\dictionary.pcidx

.\merged\[c]_bscpxml\dictionary.pdat2

.\merged\[c]_bscpxml\dictionary.pidx2

.\merged\[c]_bscpxml\dictionary.shash

.\merged\[c]_bscpxml\dictionary.wncidx

.\merged\[c]_bscpxml\dictionary.wnidx2

.\merged\[n]_bscpxml\datetime\boolocc.bdat

.\merged\[n]_bscpxml\datetime\boolocc.bidx

.\merged\[n]_bscpxml\datetime\boolocc.ccnt

.\merged\[n]_bscpxml\datetime\boolocc.dat.ccnt

.\merged\[n]_bscpxml\datetime\boolocc.dat.compressed

.\merged\[n]_bscpxml\datetime\posocc.ccnt

.\merged\[n]_bscpxml\datetime\posocc.counts.ccnt

.\merged\[n]_bscpxml\datetime\posocc.dat.compressed

.\merged\[n]_bscpxml\dictionary.pcdat

.\merged\[n]_bscpxml\dictionary.pcidx

.\merged\[n]_bscpxml\dictionary.pdat2

.\merged\[n]_bscpxml\dictionary.pidx2

.\merged\[n]_bscpxml\dictionary.shash

.\merged\[n]_bscpxml\dictionary.wncidx

.\merged\[n]_bscpxml\dictionary.wnidx2

.\merged\[n]_bscpxml\double\boolocc.bdat

.\merged\[n]_bscpxml\double\boolocc.bidx

.\merged\[n]_bscpxml\double\boolocc.ccnt

.\merged\[n]_bscpxml\double\boolocc.dat.ccnt

.\merged\[n]_bscpxml\double\boolocc.dat.compressed

.\merged\[n]_bscpxml\double\posocc.ccnt

.\merged\[n]_bscpxml\double\posocc.counts.ccnt

.\merged\[n]_bscpxml\double\posocc.dat.compressed

.\merged\[n]_bscpxml\float\boolocc.bdat

.\merged\[n]_bscpxml\float\boolocc.bidx

.\merged\[n]_bscpxml\float\boolocc.ccnt

.\merged\[n]_bscpxml\float\boolocc.dat.ccnt

.\merged\[n]_bscpxml\float\boolocc.dat.compressed

.\merged\[n]_bscpxml\float\posocc.ccnt

.\merged\[n]_bscpxml\float\posocc.counts.ccnt

.\merged\[n]_bscpxml\float\posocc.dat.compressed

.\merged\[n]_bscpxml\int32\boolocc.bdat

.\merged\[n]_bscpxml\int32\boolocc.bidx

.\merged\[n]_bscpxml\int32\boolocc.ccnt

.\merged\[n]_bscpxml\int32\boolocc.dat.ccnt

.\merged\[n]_bscpxml\int32\boolocc.dat.compressed

.\merged\[n]_bscpxml\int32\posocc.ccnt

.\merged\[n]_bscpxml\int32\posocc.counts.ccnt

.\merged\[n]_bscpxml\int32\posocc.dat.compressed

.\merged\[s]_bscpxml\all\boolocc.bdat

.\merged\[s]_bscpxml\all\boolocc.bidx

.\merged\[s]_bscpxml\all\boolocc.ccnt

.\merged\[s]_bscpxml\all\boolocc.dat.ccnt

.\merged\[s]_bscpxml\all\boolocc.dat.compressed

.\merged\[s]_bscpxml\all\posocc.ccnt

.\merged\[s]_bscpxml\all\posocc.counts.ccnt

.\merged\[s]_bscpxml\all\posocc.dat.compressed

.\merged\[s]_bscpxml\dictionary.pcdat

.\merged\[s]_bscpxml\dictionary.pcidx

.\merged\[s]_bscpxml\dictionary.pdat2

.\merged\[s]_bscpxml\dictionary.pidx2

.\merged\[s]_bscpxml\dictionary.shash

.\merged\[s]_bscpxml\dictionary.wncidx

.\merged\[s]_bscpxml\dictionary.wnidx2

.\range

.\rank.cf

.\stamp.txt

.\summary.cf

.\summary.map

.\urlmap.txt

.\version.txt

3.2 URL Map File

The following table shows the content of this file when two items are indexed. For information about the format description, see section 2.1.10.

Content	Explanation
d4f345bff288a95c0c8cc2dc456cb4dc_sp 0	internalid = d4f345bff288a95c0c8cc2dc456cb4dc collection-name = "sp" doc-id = 0
13ba8e5cd93d36f2df09ebafd2b77e88_sp 1	internalid = 13ba8e5cd93d36f2df09ebafd2b77e88 collection-name = "sp" doc-id = 1

3.3 Attribute Vector Data File

The attribute vector data file format is described in section 2.1.13.1. The file is a sortable field attribute vector that contains the dates for indexing two items, as shown in the following table.

Content (hexadecimal)	Explanation
802e 9b80 0ba5 1588 00d3 5a88 0ba5 1588	Ticks calculated as unsigned 64-bit integer (first 8 bytes): 9805925232490000000. In seconds: 980592523249. Dividing by (24x366x60x60) gives the number of years after -1/1/29000:31009 years, therefore year 2009. Proceeding gives 156 days which means 5. June, 12 hours and 49 minutes. The second date time (00d3 5a88 0ba5 1588) is converted in the same manner.

3.4 Attribute Vector Enum File

This is a binary file, in which 8 bytes and 8 bytes form unsigned 64-bit integer numbers, as shown in the following table. For more information about the file format description, see section 2.1.13.2.

Content	Explanation
00000000 01000000	Entry 0 in sorted unique data file is entry number 0 in byte-sorted order. Entry 1 in sorted unique data file is entry number 1 in byte-sorted order.

3.5 Boolean Occurrences Bit-vector File

This is a binary file; for information about the file description, see section 2.1.14.2.1. There are only two items. For each token in the dictionary, there is a bit-vector 32 bits in length. Only the two first bits are used. If the file is written as a series of 4-byte unsigned integers, there is one decimal value for each token in the dictionary. The value is 1 if the token exists in item 1, 2 if the token exists in item 2, and 3 if the token exists in both items.

```
03
03
01
01
01
01
01
01
01
02
03
02
02
01
01
03
02
02
02
02
01
01
02
02
03
03
02
02
02
02
03
01
01
```

3.6 Boolean Occurrences Bit Compressed Count File

The following file format is described in section 2.1.14.2.3. First there are six unsigned 32-bit integers, followed by compressed count values, one for each token in the dictionary. All the values are annotated with hexadecimal values, and the unsigned integer decimal value for the first six uncompressed unsigned integers. For the compressed count values, a bit sequence is used, as shown in the leftmost column. The decompressed value of the bit sequence is shown following the "Count value" text.

```
0x1 0x0 0x0 0x0  Header Version 1
0x10 0x0 0x0 0x0  Header Version Length 16
0x21 0x0 0x0 0x0  Nr Occurrences 33
0x8 0x0 0x0 0x0  Compression Method 8
0x2 0x0 0x0 0x0  K 2
0xfc 0x3 0x0 0x0  Max 1020
11001 Count value 2
11001 Count value 2
10 Count value 1
10 Count value 1
10 Count value 1
10 Count value 1
10 Count value 1
10 Count value 1
```

```
10 Count value 1
10 Count value 1
11001 Count value 2
10 Count value 1
10 Count value 1
10 Count value 1
10 Count value 1
11001 Count value 2
10 Count value 1
10 Count value 1
10 Count value 1
10 Count value 1
10 Count value 1
10 Count value 1
10 Count value 1
10 Count value 1
11001 Count value 2
11001 Count value 2
10 Count value 1
10 Count value 1
10 Count value 1
10 Count value 1
11001 Count value 2
10 Count value 1
10 Count value 1
0 Count value 0
0 Count value 0
0 Count value 0
0 Count value 0
0 Count value 0
0 Count value 0
0 Count value 0
0 Count value 0
0 Count value 0
```

3.7 Boolean Occurrences Compressed Data File

The following file format is described in section 2.1.14.2.5. The information consumed is on the left, followed by the explanation on the right and sometimes a value that is processed using the information on the left. For the header entries, that is, the entries prefixed with 0x, the information in the left column is written in hexadecimal. For the compressed information, the information is written as a bit sequence.

There are two items in the following example files. Each token has a set of entries in this file, one for each item where it is present. The beginning of a new token is seen where the Boolean *New Entry* is set to **true**.

```
0x1 0x0 0x0 0x0  Header version1
0x0 0x0 0x0 0x0  Header length 0
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 context map
00000010 first occurrence pos
00000001 number of occurrences
```

```
0000001 If New Entry:doc-id else difference to previous:0
0 New Entry ?
1 contexts map present
0 external context count present
0 first occurrence present
0 number occurrences present
00000000 context map
0000010 If New Entry:doc-id else difference to previous:1
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 context map
00000010 first occurrence pos
00000001 number of occurrences
0000001 If New Entry:doc-id else difference to previous:0
0 New Entry ?
1 contexts map present
0 external context count present
0 first occurrence present
0 number occurrences present
00000000 context map
0000010 If New Entry:doc-id else difference to previous:1
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 context map
00000011 first occurrence pos
00000001 number of occurrences
0000001 If New Entry:doc-id else difference to previous:0
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 context map
00000011 first occurrence pos
00000001 number of occurrences
0000001 If New Entry:doc-id else difference to previous:0
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 context map
00000100 first occurrence pos
00000001 number of occurrences
0000001 If New Entry:doc-id else difference to previous:0
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 context map
00000100 first occurrence pos
00000001 number of occurrences
```

```
0000001 If New Entry:doc-id else difference to previous:0
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 context map
00000100 first occurrence pos
00000001 number of occurrences
0000001 If New Entry:doc-id else difference to previous:0
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 context map
00000100 first occurrence pos
00000001 number of occurrences
0000001 If New Entry:doc-id else difference to previous:0
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00001000 context map
11111111 first occurrence pos
00000001 number of occurrences
0000001 If New Entry:doc-id else difference to previous:0
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00001000 context map
11111111 first occurrence pos
00000001 number of occurrences
0000010 If New Entry:doc-id else difference to previous:1
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00001000 context map
11111111 first occurrence pos
00000001 number of occurrences
0000001 If New Entry:doc-id else difference to previous:0
0 New Entry ?
0 contexts map present
0 external context count present
0 first occurrence present
0 number occurrences present
0000010 If New Entry:doc-id else difference to previous:1
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 context map
00000010 first occurrence pos
```

```
00000001 number of occurrences
0000010 If New Entry:doc-id else difference to previous:1
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 context map
00000010 first occurrence pos
00000001 number of occurrences
0000010 If New Entry:doc-id else difference to previous:1
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 context map
00000001 first occurrence pos
00000001 number of occurrences
0000001 If New Entry:doc-id else difference to previous:0
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 context map
00000001 first occurrence pos
00000001 number of occurrences
0000001 If New Entry:doc-id else difference to previous:0
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00001000 context map
11111111 first occurrence pos
00000001 number of occurrences
0000001 If New Entry:doc-id else difference to previous:0
0 New Entry ?
0 contexts map present
0 external context count present
0 first occurrence present
0 number occurrences present
0000010 If New Entry:doc-id else difference to previous:1
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 context map
00000100 first occurrence pos
00000001 number of occurrences
0000010 If New Entry:doc-id else difference to previous:1
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 context map
```

```
00000100 first occurrence pos
00000001 number of occurrences
0000010 If New Entry:doc-id else difference to previous:1
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 context map
00000100 first occurrence pos
00000001 number of occurrences
0000010 If New Entry:doc-id else difference to previous:1
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 context map
00000100 first occurrence pos
00000001 number of occurrences
0000010 If New Entry:doc-id else difference to previous:1
1 New Entry ?
0 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 first occurrence pos
00000001 number of occurrences
0000001 If New Entry:doc-id else difference to previous:0
1 New Entry ?
0 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 first occurrence pos
00000001 number of occurrences
0000001 If New Entry:doc-id else difference to previous:0
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 context map
00000011 first occurrence pos
00000001 number of occurrences
0000010 If New Entry:doc-id else difference to previous:1
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 context map
00000011 first occurrence pos
00000001 number of occurrences
0000010 If New Entry:doc-id else difference to previous:1
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
```

```
1 number occurrences present
00001000 context map
11111111 first occurrence pos
00000001 number of occurrences
0000001 If New Entry:doc-id else difference to previous:0
0 New Entry ?
0 contexts map present
0 external context count present
0 first occurrence present
0 number occurrences present
0000010 If New Entry:doc-id else difference to previous:1
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00001000 context map
11111111 first occurrence pos
00000001 number of occurrences
0000001 If New Entry:doc-id else difference to previous:0
0 New Entry ?
0 contexts map present
0 external context count present
0 first occurrence present
0 number occurrences present
0000010 If New Entry:doc-id else difference to previous:1
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 context map
00000001 first occurrence pos
00000001 number of occurrences
0000010 If New Entry:doc-id else difference to previous:1
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 context map
00000001 first occurrence pos
00000001 number of occurrences
0000010 If New Entry:doc-id else difference to previous:1
1 New Entry ?
0 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 first occurrence pos
00000001 number of occurrences
0000010 If New Entry:doc-id else difference to previous:1
1 New Entry ?
0 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 first occurrence pos
00000001 number of occurrences
```

```
0000010 If New Entry:doc-id else difference to previous:1
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00001000 context map
11111111 first occurrence pos
00000001 number of occurrences
0000001 If New Entry:doc-id else difference to previous:0
0 New Entry ?
0 contexts map present
0 external context count present
0 first occurrence present
0 number occurrences present
0000010 If New Entry:doc-id else difference to previous:1
1 New Entry ?
0 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 first occurrence pos
00000001 number of occurrences
0000001 If New Entry:doc-id else difference to previous:0
1 New Entry ?
0 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 first occurrence pos
00000001 number of occurrences
0000001 If New Entry:doc-id else difference to previous:0
1 New Entry ?
1 contexts map present
0 external context count present
1 first occurrence present
1 number occurrences present
00000001 context map
00000001 first occurrence pos
00000001 number of occurrences
0000010 If New Entry:doc-id else difference to previous:1
```

3.8 Position Occurrences Compressed Data File

The following is an annotated example of the position occurrences data file. The information consumed is on the left, followed by the explanation on the right, and sometimes a field that is processed using the information on the left. For the header entries the left is written in hexadecimal, for the compressed information, the information is written as bit sequences, in which the bit sequences are ordered. The file format is described in section 2.1.14.3.3.

There are two items in the collection. Each token has a set of entries in this file, one for each item where it is present. The header and positional information for the first token only are provided.

```
0x1 0x0 0x0 0x0  Header version 1
0x4 0x0 0x0 0x0  Header version length 4
0x0 0x0 0x0 0x0  MCC 0
00000000000000000000001 Posocc FirstDocId 0
```

```
000000011 First position 2
0 No context
01 Next doc id follows
00000001 Delta DocId 1
000000001 First position 0
0 No context
00  No more docs for this entry
00000000000000000000001 Posocc FirstDocId 0
000000011 First position 2
0 No context
01 Next doc id follows
00000001 Delta DocId 1
000000001 First position 0
0 No context
00  No more docs for this entry
00000000000000000000001 Posocc FirstDocId 0
000000100 First position 3
0 No context
00  No more docs for this entry
```

3.9 Dictionary Paged Data File

The example described in the following subsections is generated using two items with a total of 33 tokens. The example is annotated, and also split into different regions. For this example there exists only one file page. The information consumed is on the left, followed by a textual comment on the type of information, and sometimes followed by the converted value. The file format is described in section 2.1.14.4.3.

3.9.1 Page Header

The following is an annotated example of a page header region.

```
0x0 0x0 0x0 0x0  First Num = 0
0x14 0x0 0x0 0x0  First WordOffset =20
0x21 0x0  Word Count = 33
0x7 0x0  Sparse size = 7
0x3d 0x0  Between size = 61
0x0 0x0  Not used
```

3.9.2 Sparse Region

The following is an annotated example of the sparse region of the file. There are 33 tokens in the dictionary. The number of sparse entries is calculated (`1+floor(33/16)) = 3`, so there are three sparse entries, for each of which there is possible data for four property indexes.

```
0 Sparse coded acc num docs=0
0 Sparse coded boolocc offset=0
1000101011111 Sparse coded posocc offset=96
0 Sparse coded acc num docs=0
0 Sparse coded boolocc offset=0
1000101011111 Sparse coded posocc offset=96
0 Sparse coded acc num docs=0
0 Sparse coded boolocc offset=0
1000101011111 Sparse coded posocc offset=96
0 Sparse coded acc num docs=0
```

```
0 Sparse coded boolocc offset=0
1000101011111 Sparse coded posocc offset=96
1 Sparse Entry present
0 Sparse rice compressed information follows
101101 Sparse compressed number of docs 20
100010000001 Sparse compressed boolocc offset=640
100010011111 Sparse posocc offset=670
1 Sparse Entry present
0 Sparse rice compressed information follows
0111 Sparse compressed number of docs 6
0010101001 Sparse compressed boolocc offset=168
0011010011 Sparse posocc offset=210
0 No sparse entry exists
0 No sparse entry exists
01110110111 Between offset information=950
1 Sparse Entry present
0 Sparse rice compressed information follows
101100 Sparse compressed number of docs 19
100000111101 Sparse compressed boolocc offset=572
100010001011 Sparse posocc offset=650
1 Sparse Entry present
0 Sparse rice compressed information follows
0111 Sparse compressed number of docs 6
0010010001 Sparse compressed boolocc offset=144
0011000100 Sparse posocc offset=195
0 No sparse entry exists
0 No sparse entry exists
```

3.9.3 BETWEEN Region

The following is an excerpt from the **BETWEEN** region, specifically, the entries for the three first tokens in the dictionary.

```
01110101011 Between offset information=938
1 Between entry present
1 Between coded entry follows
11001 Between coded acc num docs=2
00111001 Between coded boolocc offset=56
0111000 Between coded posocc offset=55
0 No Between entry present
0 No Between entry present
0 No Between entry present
11000010110011000011101101000111 Normalized document count=10000000
1 Between entry present
1 Between coded entry follows
11001 Between coded acc num docs=2
00111001 Between coded boolocc offset=56
0111000 Between coded posocc offset=55
0 No Between entry present
0 No Between entry present
0 No Between entry present
11000010110011000011101101000111 Normalized document count=10000000
1 Between entry present
0 Between rice-compressed entry follows
00100101 Between compressed posocc offset=36
0100100 Between compressed posocc offset=35
```

```
0 No Between entry present
0 No Between entry present
0 No Between entry present
11000010101001100001010110100011 Normalized document count=5000000
```

3.9.4 Word Offsets

There are two fewer token offsets for one page than there are tokens. The token offsets are the number of bytes from the beginning of the token region to the LCP entry of the token. The first token for the page has no LCP entry, while the second token has offset 0. So the first offset described is for token number 2 on the current file page.

```
0x4 0x0  Token offset 4
0x10 0x0  Token offset 16
0x13 0x0  Token offset 19
0x1a 0x0  Token offset 26
0x1f 0x0  Token offset 31
0x22 0x0  Token offset 34
0x26 0x0  Token offset 38
0x2d 0x0  Token offset 45
0x34 0x0  Token offset 52
0x3b 0x0  Token offset 59
0x40 0x0  Token offset 64
0x45 0x0  Token offset 69
0x48 0x0  Token offset 72
0x4c 0x0  Token offset 76
0x58 0x0  Token offset 88
0x5f 0x0  Token offset 95
0x64 0x0  Token offset 100
0x6b 0x0  Token offset 107
0x6f 0x0  Token offset 111
0x76 0x0  Token offset 118
0x7d 0x0  Token offset 125
0x83 0x0  Token offset 131
0x86 0x0  Token offset 134
0x8c 0x0  Token offset 140
0x94 0x0  Token offset 148
0x9b 0x0  Token offset 155
0x9e 0x0  Token offset 158
0xa4 0x0  Token offset 164
0xa8 0x0  Token offset 168
0xaf 0x0  Token offset 175
0xb6 0x0  Token offset 182
```

3.9.5 LCP Entries

This part of the file follows the token entries and the token offsets in the file.

```
0x0 0x61 0x54 0x0  LCP entry aT prefix=0
0x0 0x62 0x65 0x61 0x74 0x75 0x69 0x66 0x75 0x6c 0x4c 0x0  LCP entry beatuifulL prefix=0
0x9 0x54 0x0  LCP entry T prefix=9
0x0 0x63 0x69 0x74 0x79 0x4c 0x0  LCP entry cityL prefix=0
0x4 0x4c 0xc7 0x82 0x0  LCP entry LÃé prefix=4
0x4 0x54 0x0  LCP entry T prefix=4
```

```
0x5 0xc7 0x82 0x0  LCP entry Ãé prefix=5
0x0 0x64 0x6f 0x63 0x31 0x54 0x0  LCP entry doc1T prefix=0
0x0 0x64 0x6f 0x63 0x32 0x54 0x0  LCP entry doc2T prefix=0
0x0 0x68 0x74 0x74 0x70 0x54 0x0  LCP entry httpT prefix=0
0x0 0x69 0x6e 0x4c 0x0  LCP entry inL prefix=0
0x0 0x69 0x6e 0x54 0x0  LCP entry inT prefix=0
0x2 0x4c 0x0  LCP entry L prefix=2
0x1 0x73 0x54 0x0  LCP entry sT prefix=1
0x0 0x6c 0x6f 0x63 0x61 0x6c 0x68 0x6f 0x73 0x74 0x54 0x0  LCP entry localhostT prefix=0
0x0 0x70 0x61 0x72 0x6b 0x4c 0x0  LCP entry parkL prefix=0
0x4 0x4c 0xc7 0x82 0x0  LCP entry LÃé prefix=4
0x0 0x70 0x61 0x72 0x6b 0x54 0x0  LCP entry parkT prefix=0
0x5 0xc7 0x82 0x0  LCP entry Ãé prefix=5
0x0 0x72 0x6f 0x6d 0x61 0x4c 0x0  LCP entry romaL prefix=0
0x0 0x72 0x6f 0x6d 0x61 0x54 0x0  LCP entry romaT prefix=0
0x0 0x74 0x68 0x65 0x4c 0x0  LCP entry theL prefix=0
0x3 0x54 0x0  LCP entry T prefix=3
0x0 0x74 0x78 0x74 0x54 0x0  LCP entry txtT prefix=0
0x0 0x74 0x78 0x74 0x54 0xc7 0x82 0x0  LCP entry txtTÃé prefix=0
0x0 0x77 0x61 0x6c 0x6b 0x4c 0x0  LCP entry walkL prefix=0
0x4 0x54 0x0  LCP entry T prefix=4
0x0 0xc7 0x82 0x61 0x4c 0x0  LCP entry ÃéaL prefix=0
0x2 0x61 0x54 0x0  LCP entry aT prefix=2
0x2 0x68 0x74 0x74 0x70 0x54 0x0  LCP entry httpT prefix=2
0x2 0x72 0x6f 0x6d 0x61 0x4c 0x0  LCP entry romaL prefix=2
0x0 0xc7 0x82 0x72 0x6f 0x6d 0x61 0x54 0x0  LCP entry ÃéromaT prefix=0
```

3.10 Dictionary Paged Index File

This example is generated using two items with a total of 33 tokens. The example is annotated, and split into different regions. For this example there exists only one file page. The information consumed is on the left, followed by a text comment on type of data, and sometimes followed by the converted value. For more information, see section 2.1.14.4.4.

```
00000111001001000000000101000101 Magic number
0x2 0x0 0x0 0x0  Version 2
0x8 0x0 0x0 0x0  Header length 8
0x1 0x0  Tagtype 1
0x4 0x0  Taglength 4
000 Unused bits
1 Has posocc compressed file
1 Has boolocc compressed filen
0 No phrase index
1 Has posocc
1 Has boolocc
00000000 Unused byte
0x4 0x0  Nr of property indexes 4
0x61 0x4c 0x0  String aL
```

3.11 Dictionary Sorted Hash File

This example is generated using two items with a total of 33 tokens. The documents that are indexed are very short and contain the following text:

- "Rome is a beautiful city"

- "A walk in the park"

The format is described in section 2.1.14.4.5.

```
          33
2 2 aL
2 2 aT
1 1 beatuifulL
1 1 beatuifulT
1 1 cityL
1 1 cityL‡
1 1 cityT
1 1 cityT‡
1 1 doc1T
1 1 doc2T
2 2 httpT
1 1 inL
1 1 inT
1 1 isL
1 1 isT
2 2 localhostT
1 1 parkL
1 1 parkL‡
1 1 parkT
1 1 parkT‡
1 1 romaL
1 1 romaT
1 1 theL
1 1 theT
2 2 txtT
2 2 txtT‡
1 1 walkL
1 1 walkT
1 1 ‡aL
1 1 ‡aT
2 2 ‡httpT
1 1 ‡romaL
1 1 ‡romaT
```

3.12 Integer Occurrences Bit-vector Index File

This example is generated using two items with a total of 33 tokens. The example is annotated, and split into different regions. For this example there exists one bit-vector index entry. It represents a range of 20 to 26. It points to the beginning of the integer bitmap data file, so the first (and only) bit-vector is referenced. The information consumed is on the left, followed by a text comment on the type of data, and sometimes followed by the converted value. The file format is described in section 2.1.15.4.

```
0x2 0x0 0x0 0x0  Nr of docs 2
0x1 0x0 0x0 0x0  Nr index entries 1
0x14 0x0 0x0 0x0 0x0 0x0 0x0 0x80  Lower key 20
0x1a 0x0 0x0 0x0 0x0 0x0 0x0 0x80  Higher key 26
0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0  Integer bitmap data file offset 0
0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0  Integer occurrence data file offset 0
0x2 0x0 0x0 0x0  Integer occurrences  2
```

```
0x0 0x0 0x0 0x0  4 bytes that are ignored  0
```

3.13 Integer Occurrences Bit-vector Unique Index File

This example is generated using two items with a total of 33 tokens. The example is annotated, and split into different regions. For this example there is only one bit-vector, and therefore one entry in this file. The bit-vector represents a value range from 20 to 26, so the low integer key is 20. The information consumed is on the left, followed by a text comment on the type of data, and sometimes followed by the converted value. The format is described in section 2.1.15.6.

```
0x14 0x0 0x0 0x0 0x0 0x0 0x0 0x80  Low integer key 20
0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0  Integer bit-vector index offset 0
```

3.14 Integer Occurrences Data File

This example is generated using two items with a total of 33 tokens. The example is annotated, and split into different regions. There are two items; each item has one instance of the current numeric field. For more information, see integer occurrence index file. The information consumed is on the left, followed by a text comment on the type of data, and sometimes followed by the converted value. The file format is described in section 2.1.15.7.

```
0x1 0x0 0x0 0x0  Item Identifier 1
0x0 0x0 0x0 0x0  Item Identifier 0
```

3.15 Integer Occurrences Index File

This example is generated using two items with a total of 33 tokens. There are two entries, one for the value 20, and one for the value 26. The offset into the integer occurrence data file for the first entry is 0, for the second entry it is 1. In the integer occurrences data file, the first entry has **item** identifier 1, therefore the item with item identifier 1 has an instance of this numeric field that contains the value 20. Equally, item identifier 0 has an occurrence of this numeric field with the value 26. The information consumed is on the left, followed by a text comment on the type of data, and sometimes followed by the converted value. The file format is described in section 2.1.15.8.

```
0x14 0x0 0x0 0x0 0x0 0x0 0x0 0x80  Integer value 20
0x0 0x0 0x0 0x0  4 bytes to be ignored
0x1 0x0 0x0 0x0  Nr of occurrences 1
0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0  Offset into intocc.dat file 0
0x1a 0x0 0x0 0x0 0x0 0x0 0x0 0x80  Integer value 26
0x0 0x0 0x0 0x0  4 bytes to be ignored
0x1 0x0 0x0 0x0  Nr of occurrences 1
0x1 0x0 0x0 0x0 0x0 0x0 0x0 0x0  Offset into intocc.dat file 1
```

3.16 Document Summary Data File

This example is generated using two items with a total of 33 tokens. The information consumed is on the left, followed by a text comment on type of data, and sometimes followed by the converted value. The file format is described in section 2.1.16.2.

For the following example, it is assumed that the document summary class has the following fields:

```
field internalid type string
field contentid type string
field contentids type string
```

This means that the document summary following the initial document summary class integer consists of three strings.

```
0x0 0x0 0x0 0x0  Document summary class 0
0x23 0x0  String length 35
0x64 0x34 0x66 0x33 0x34 0x35 0x62 0x66 0x66 0x32 0x38 0x38 0x61 0x39 0x35 0x63 0x30 0x63
0x38 0x63 0x63 0x32 0x64 0x63 0x34 0x35 0x36 0x63 0x62 0x34 0x64 0x63 0x5f 0x73 0x70  String
d4f345bff288a95c0c8cc2dc456cb4dc_sp
0x19 0x0  String length 25
0x68 0x74 0x74 0x70 0x3a 0x2f 0x2f 0x6c 0x6f 0x63 0x61 0x6c 0x68 0x6f 0x73 0x74 0x2f 0x64
0x6f 0x63 0x31 0x2e 0x74 0x78 0x74  String http://localhost/doc1.txt
0x19 0x0  String length 25
0x68 0x74 0x74 0x70 0x3a 0x2f 0x2f 0x6c 0x6f 0x63 0x61 0x6c 0x68 0x6f 0x73 0x74 0x2f 0x64
0x6f 0x63 0x31 0x2e 0x74 0x78 0x74  String http://localhost/doc1.txt
```

3.17 Document Summary Index File

This example is generated using two items with a total of 33 tokens. The information consumed is on the left, followed by a text comment on type of data, and sometimes followed by the converted value. It shows that the document summary for the document with item identifier 0 begins at byte 0 in the docsum.dat file, while the item for the document summary associated with item identifier 1 begins at byte 532, and so on throughout the file. The file format is described in section 2.1.16.3.

```
0x0 0x0 0x0 0x0  offset into docsum.dat file 0
0x14 0x2 0x0 0x0  offset into docsum.dat file 532
0x1c 0x4 0x0 0x0  offset into docsum.dat file 1052
```

3.18 Unique Identity Data File

This example is generated using two items with a total of 33 tokens. The information consumed is on the left, followed by a text comment on type of data, and sometimes followed by the converted value. The file format is described in section 2.1.17.

```
0x56 0x65 0x72 0x73 0x69 0x6f 0x6e  The string "Version"
0x0 0x0 0x0 0x0  Version is 0
0x36 0x0 0x0 0x0  Header size 54
0x2 0x0 0x0 0x0  Item count 2
0x1 0x0 0x0 0x0  Pages count 1
0xd4 0xf3 0x45 0xbf 0xf2 0x88 0xa9 0x5c 0xc 0x8c 0xc2 0xdc 0x45 0x6c 0xb4 0xdc  MD5 string
for this item
0x0 0x0 0x0 0x0  CLASS identifier=0
0x1 0x0 0x0 0x0  Collection count 1
0x3 0x0 0x0 0x0  Collection String length 3
0x5f 0x73 0x70  Collection string name _sp
0x13 0xba 0x8e 0x5c 0xd9 0x3d 0x36 0xf2 0xdf 0x9 0xeb 0xaf 0xd2 0xb7 0x7e 0x88  Mapping MD5
string for this item
0x0 0x0 0x0 0x0  Mapping collection identifier 0
0x1 0x0 0x0 0x0  Mapping item identifier 1
0xd4 0xf3 0x45 0xbf 0xf2 0x88 0xa9 0x5c 0xc 0x8c 0xc2 0xdc 0x45 0x6c 0xb4 0xdc  Mapping MD5
0x0 0x0 0x0 0x0  Mapping collection identifier 0
```

```
0x0 0x0 0x0 0x0  Mapping item identifier 0
```

3.19 Dictionary Paged Count File

This example is generated using two items with a total of 33 tokens. The information consumed is on the left, followed by a text comment on type of data, and sometimes followed by the converted value. For the token-differences numbers, not all entries are included. For each bulk of relative numbers, the respective token is described first for the sake of clarity. The file format is described in section 2.1.14.4.1.

```
0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0  before first token-acc-number 0
0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0  before first doc-acc-number 0
0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0  before first token-acc-number 0
0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0  before first doc-acc-number 0
0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0  before first token-acc-number 0
0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0  before first doc-acc-number 0
0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0  before first token-acc-number 0
0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0  before first doc-acc-number 0
0x28 0x0 0x0 0x0 0x0 0x0 0x0 0x0  after last token-acc-number 40
0x28 0x0 0x0 0x0 0x0 0x0 0x0 0x0  after last doc-acc-number 40
0xc 0x0 0x0 0x0 0x0 0x0 0x0 0x0  after last token-acc-number 12
0xc 0x0 0x0 0x0 0x0 0x0 0x0 0x0  after last doc-acc-number 12
0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0  after last token-acc-number 0
0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0  after last doc-acc-number 0
0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0  after last token-acc-number 0
0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0  after last doc-acc-number 0
0x21 0x0 0x0 0x0  token count 33
0x0 0x0 0x0 0x0  token identifier - token count 0

"aT"
0x2 0x0 0x0 0x0  diff-token-acc-number 2
0x2 0x0 0x0 0x0  diff-doc-acc-number 2
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0

"aL"
0x4 0x0 0x0 0x0  diff-token-acc-number 4
0x4 0x0 0x0 0x0  diff-doc-acc-number 4
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0

"beautifulL"
0x5 0x0 0x0 0x0  diff-token-acc-number 5
0x5 0x0 0x0 0x0  diff-doc-acc-number 5
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0
0x0 0x0 0x0 0x0  diff-token-acc-number 0
```

```
0x0 0x0 0x0 0x0  diff-doc-acc-number 0

"beautifulT"
0x6 0x0 0x0 0x0  diff-token-acc-number 6
0x6 0x0 0x0 0x0  diff-doc-acc-number 6
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0

"cityL"
0x7 0x0 0x0 0x0  diff-token-acc-number 7
0x7 0x0 0x0 0x0  diff-doc-acc-number 7
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0

"cityLXX
0x8 0x0 0x0 0x0  diff-token-acc-number 8
0x8 0x0 0x0 0x0  diff-doc-acc-number 8
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0

cytyT
0x9 0x0 0x0 0x0  diff-token-acc-number 9
0x9 0x0 0x0 0x0  diff-doc-acc-number 9
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0

cityTxx
0xa 0x0 0x0 0x0  diff-token-acc-number 10
0xa 0x0 0x0 0x0  diff-doc-acc-number 10
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0

doc1T
0xb 0x0 0x0 0x0  diff-token-acc-number 11
0xb 0x0 0x0 0x0  diff-doc-acc-number 11
0x1 0x0 0x0 0x0  diff-token-acc-number 1
0x1 0x0 0x0 0x0  diff-doc-acc-number 1
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0
```

```
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0

doc2T
0xc 0x0 0x0 0x0  diff-token-acc-number 12
0xc 0x0 0x0 0x0  diff-doc-acc-number 12
0x2 0x0 0x0 0x0  diff-token-acc-number 2
0x2 0x0 0x0 0x0  diff-doc-acc-number 2
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0

0x27 0x0 0x0 0x0  diff-token-acc-number 39
0x27 0x0 0x0 0x0  diff-doc-acc-number 39
0xc 0x0 0x0 0x0  diff-token-acc-number 12
0xc 0x0 0x0 0x0  diff-doc-acc-number 12
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0
0x0 0x0 0x0 0x0  diff-token-acc-number 0
0x0 0x0 0x0 0x0  diff-doc-acc-number 0

0x3 0x0  token length 3
0x6 0x0  token length 6
0x11 0x0  token length 17
0x1c 0x0  token length 28
0x22 0x0  token length 34
0x2a 0x0  token length 42
0x30 0x0  token length 48
0x38 0x0  token length 56
0x3e 0x0  token length 62
0x44 0x0  token length 68
0x4a 0x0  token length 74
0x4e 0x0  token length 78
0x52 0x0  token length 82
0x56 0x0  token length 86
0x5a 0x0  token length 90
0x65 0x0  token length 101
0x6b 0x0  token length 107
0x73 0x0  token length 115
0x79 0x0  token length 121
0x81 0x0  token length 129
0x87 0x0  token length 135
0x8d 0x0  token length 141
0x92 0x0  token length 146
0x97 0x0  token length 151
0x9c 0x0  token length 156
0xa3 0x0  token length 163
0xa9 0x0  token length 169
0xaf 0x0  token length 175
0xb4 0x0  token length 180
0xb9 0x0  token length 185
0xc1 0x0  token length 193
0xc9 0x0  token length 201
0x61 0x4c 0x0  Token string aL
0x61 0x54 0x0  Token string aT
0x62 0x65 0x61 0x74 0x75 0x69 0x66 0x75 0x6c 0x4c 0x0  Token string beatuifulL
0x62 0x65 0x61 0x74 0x75 0x69 0x66 0x75 0x6c 0x54 0x0  Token string beatuifulT
0x63 0x69 0x74 0x79 0x4c 0x0  Token string cityL
```

```
0x63 0x69 0x74 0x79 0x4c 0xc7 0x82 0x0  Token string cityLÃé
0x63 0x69 0x74 0x79 0x54 0x0  Token string cityT
0x63 0x69 0x74 0x79 0x54 0xc7 0x82 0x0  Token string cityTÃé
0x64 0x6f 0x63 0x31 0x54 0x0  Token string doc1T
0x64 0x6f 0x63 0x32 0x54 0x0  Token string doc2T
0x68 0x74 0x74 0x70 0x54 0x0  Token string httpT
0x69 0x6e 0x4c 0x0  Token string inL
0x69 0x6e 0x54 0x0  Token string inT
0x69 0x73 0x4c 0x0  Token string isL
0x69 0x73 0x54 0x0  Token string isT
0x6c 0x6f 0x63 0x61 0x6c 0x68 0x6f 0x73 0x74 0x54 0x0  Token string localhostT
0x70 0x61 0x72 0x6b 0x4c 0x0  Token string parkL
0x70 0x61 0x72 0x6b 0x4c 0xc7 0x82 0x0  Token string parkLÃé
0x70 0x61 0x72 0x6b 0x54 0x0  Token string parkT
0x70 0x61 0x72 0x6b 0x54 0xc7 0x82 0x0  Token string parkTÃé
0x72 0x6f 0x6d 0x61 0x4c 0x0  Token string romaL
0x72 0x6f 0x6d 0x61 0x54 0x0  Token string romaT
0x74 0x68 0x65 0x4c 0x0  Token string theL
0x74 0x68 0x65 0x54 0x0  Token string theT
0x74 0x78 0x74 0x54 0x0  Token string txtT
0x74 0x78 0x74 0x54 0xc7 0x82 0x0  Token string txtTÃé
0x77 0x61 0x6c 0x6b 0x4c 0x0  Token string walkL
0x77 0x61 0x6c 0x6b 0x54 0x0  Token string walkT
0xc7 0x82 0x61 0x4c 0x0  Token string ÃéaL
0xc7 0x82 0x61 0x54 0x0  Token string ÃéaT
0xc7 0x82 0x68 0x74 0x74 0x70 0x54 0x0  Token string ÃéhttpT
0xc7 0x82 0x72 0x6f 0x6d 0x61 0x4c 0x0  Token string ÃéromaL
0xc7 0x82 0x72 0x6f 0x6d 0x61 0x54 0x0  Token string ÃéromaT
```

4 Security Considerations

The index files containing potentially confidential information, secure the index files on a file system where only the administrator and the service account running the query matching component processes have been granted read and write permissions.

5 Appendix A: Product Behavior

The information in this specification is applicable to the following Microsoft products or supplemental software. References to product versions include released service packs:

- Microsoft® FAST™ Search Server 2010

Exceptions, if any, are noted below. If a service pack or Quick Fix Engineering (QFE) number appears with the product version, behavior changed in that service pack or QFE. The new behavior also applies to subsequent service packs of the product unless otherwise specified. If a product edition appears with the product version, behavior is different in that product edition.

Unless otherwise specified, any statement of optional behavior in this specification that is prescribed using the terms SHOULD or SHOULD NOT implies product behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies that the product does not follow the prescription.

6 Change Tracking

No table of changes is available. The document is either new or has had no changes since its last release.

7 Index

E

F

G

I

L

M