

Apache Lucene - Index File Formats

by Doug Cutting

Table of contents

1 Index File Formats.....	3
2 Definitions.....	3
2.1 Inverted Indexing.....	3
2.2 Types of Fields.....	4
2.3 Segments.....	4
2.4 Document Numbers.....	4
3 Overview.....	5
4 File Naming.....	5
5 Primitive Types.....	6
5.1 Byte.....	6
5.2 UInt32.....	6
5.3 UInt64.....	6
5.4 VInt.....	6
5.5 Chars.....	7
5.6 String.....	7
6 Per-Index Files.....	7
6.1 Segments File.....	8
6.2 Lock File.....	9
6.3 Deletable File.....	9
6.4 Compound Files.....	9
7 Per-Segment Files.....	10
7.1 Fields.....	10

7.2 Term Dictionary.....	11
7.3 Frequencies.....	13
7.4 Positions.....	14
7.5 Normalization Factors.....	15
7.6 Term Vectors.....	15
7.7 Deleted Documents.....	17
8 Limitations.....	18

1. Index File Formats

This document defines the index file formats used in Lucene version 2.1. If you are using a different version of Lucene, please consult the copy of `docs/fileformats.html` that was distributed with the version you are using.

Apache Lucene is written in Java, but several efforts are underway to write versions of Lucene in other programming languages. If these versions are to remain compatible with Apache Lucene, then a language-independent definition of the Lucene index format is required. This document thus attempts to provide a complete and independent definition of the Apache Lucene 2.1 file formats.

As Lucene evolves, this document should evolve. Versions of Lucene in different programming languages should endeavor to agree on file formats, and generate new versions of this document.

Compatibility notes are provided in this document, describing how file formats have changed from prior versions.

In version 2.1, the file format was changed to allow lock-less commits (ie, no more commit lock). The change is fully backwards compatible: you can open a pre-2.1 index for searching or adding/deleting of docs. When the new segments file is saved (committed), it will be written in the new file format (meaning no specific "upgrade" process is needed). But note that once a commit has occurred, pre-2.1 Lucene will not be able to read the index.

2. Definitions

The fundamental concepts in Lucene are index, document, field and term.

An index contains a sequence of documents.

- A document is a sequence of fields.
- A field is a named sequence of terms.
- A term is a string.

The same string in two different fields is considered a different term. Thus terms are represented as a pair of strings, the first naming the field, and the second naming text within the field.

2.1. Inverted Indexing

The index stores statistics about terms in order to make term-based search more efficient.

Lucene's index falls into the family of indexes known as an inverted index. This is because it can list, for a term, the documents that contain it. This is the inverse of the natural relationship, in which documents list terms.

2.2. Types of Fields

In Lucene, fields may be stored, in which case their text is stored in the index literally, in a non-inverted manner. Fields that are inverted are called indexed. A field may be both stored and indexed.

The text of a field may be tokenized into terms to be indexed, or the text of a field may be used literally as a term to be indexed. Most fields are tokenized, but sometimes it is useful for certain identifier fields to be indexed literally.

See the Field java docs for more information on Fields.

2.3. Segments

Lucene indexes may be composed of multiple sub-indexes, or segments. Each segment is a fully independent index, which could be searched separately. Indexes evolve by:

1. Creating new segments for newly added documents.
2. Merging existing segments.

Searches may involve multiple segments and/or multiple indexes, each index potentially composed of a set of segments.

2.4. Document Numbers

Internally, Lucene refers to documents by an integer document number. The first document added to an index is numbered zero, and each subsequent document added gets a number one greater than the previous.

Note that a document's number may change, so caution should be taken when storing these numbers outside of Lucene. In particular, numbers may change in the following situations:

- The numbers stored in each segment are unique only within the segment, and must be converted before they can be used in a larger context. The standard technique is to allocate each segment a range of values, based on the range of numbers used in that segment. To convert a document number from a segment to an external value, the segment's base document number is added. To convert an external value back to a segment-specific value, the segment is identified by the range that the external value is in,

and the segment's base value is subtracted. For example two five document segments might be combined, so that the first segment has a base value of zero, and the second of five. Document three from the second segment would have an external value of eight.

- When documents are deleted, gaps are created in the numbering. These are eventually removed as the index evolves through merging. Deleted documents are dropped when segments are merged. A freshly-merged segment thus has no gaps in its numbering.

3. Overview

Each segment index maintains the following:

- Field names. This contains the set of field names used in the index.
- Stored Field values. This contains, for each document, a list of attribute-value pairs, where the attributes are field names. These are used to store auxiliary information about the document, such as its title, url, or an identifier to access a database. The set of stored fields are what is returned for each hit when searching. This is keyed by document number.
- Term dictionary. A dictionary containing all of the terms used in all of the indexed fields of all of the documents. The dictionary also contains the number of documents which contain the term, and pointers to the term's frequency and proximity data.
- Term Frequency data. For each term in the dictionary, the numbers of all the documents that contain that term, and the frequency of the term in that document.
- Term Proximity data. For each term in the dictionary, the positions that the term occurs in each document.
- Normalization factors. For each field in each document, a value is stored that is multiplied into the score for hits on that field.
- Term Vectors. For each field in each document, the term vector (sometimes called document vector) may be stored. A term vector consists of term text and term frequency. To add Term Vectors to your index see the Field constructors
- Deleted documents. An optional file indicating which documents are deleted.

Details on each of these are provided in subsequent sections.

4. File Naming

All files belonging to a segment have the same name with varying extensions. The extensions correspond to the different file formats described below. When using the Compound File format (default in 1.4 and greater) these files are collapsed into a single .cfs file (see below

for details)

Typically, all segments in an index are stored in a single directory, although this is not required.

As of version 2.1 (lock-less commits), file names are never re-used (there is one exception, "segments.gen", see below). That is, when any file is saved to the Directory it is given a never before used filename. This is achieved using a simple generations approach. For example, the first segments file is segments_1, then segments_2, etc. The generation is a sequential long integer represented in alpha-numeric (base 36) form.

5. Primitive Types

5.1. Byte

The most primitive type is an eight-bit byte. Files are accessed as sequences of bytes. All other data types are defined as sequences of bytes, so file formats are byte-order independent.

5.2. UInt32

32-bit unsigned integers are written as four bytes, high-order bytes first.

UInt32 --> <Byte>4

5.3. UInt64

64-bit unsigned integers are written as eight bytes, high-order bytes first.

UInt64 --> <Byte>8

5.4. VInt

A variable-length format for positive integers is defined where the high-order bit of each byte indicates whether more bytes remain to be read. The low-order seven bits are appended as increasingly more significant bits in the resulting integer value. Thus values from zero to 127 may be stored in a single byte, values from 128 to 16,383 may be stored in two bytes, and so on.

VInt Encoding Example

Value	First byte	Second byte	Third byte
0	00000000		

1	00000001		
2	00000010		
...			
127	01111111		
128	10000000	00000001	
129	10000001	00000001	
130	10000010	00000001	
...			
16,383	11111111	01111111	
16,384	10000000	10000000	00000001
16,385	10000001	10000000	00000001
...			

This provides compression while still being efficient to decode.

5.5. Chars

Lucene writes unicode character sequences using Java's "modified UTF-8 encoding" .

5.6. String

Lucene writes strings as a VInt representing the length, followed by the character data.

String --> VInt, Chars

6. Per-Index Files

The files in this section exist one-per-index.

6.1. Segments File

The active segments in the index are stored in the segment info file, `segments_N`. There may be one or more `segments_N` files in the index; however, the one with the largest generation is the active one (when older `segments_N` files are present it's because they temporarily cannot be deleted, or, a writer is in the process of committing). This file lists each segment by name, has details about the separate norms and deletion files, and also contains the size of each segment.

As of 2.1, there is also a file `segments.gen`. This file contains the current generation (the `_N` in `segments_N`) of the index. This is used only as a fallback in case the current generation cannot be accurately determined by directory listing alone (as is the case for some NFS clients with time-based directory cache expiration). This file simply contains an Int32 version header (`SegmentInfos.FORMAT_LOCKLESS = -2`), followed by the generation recorded as Int64, written twice.

Pre-2.1: Segments --> Format, Version, NameCounter, SegCount, <SegName, SegSize> SegCount

2.1 and above: Segments --> Format, Version, NameCounter, SegCount, <SegName, SegSize, DelGen, HasSingleNormFile, NumField, NormGenNumField, IsCompoundFile>SegCount

Format, NameCounter, SegCount, SegSize, NumField --> Int32

Version, DelGen, NormGen --> Int64

SegName --> String

IsCompoundFile, HasSingleNormFile --> Int8

Format is -1 as of Lucene 1.4 and -3 (`SegmentInfos.FORMAT_SINGLE_NORM_FILE`) as of Lucene 2.1.

Version counts how often the index has been changed by adding or deleting documents.

NameCounter is used to generate names for new segment files.

SegName is the name of the segment, and is used as the file name prefix for all of the files that compose the segment's index.

SegSize is the number of documents contained in the segment index.

DelGen is the generation count of the separate deletes file. If this is -1, there are no separate deletes. If it is 0, this is a pre-2.1 segment and you must check filesystem for the existence of

`_X.del`. Anything above zero means there are separate deletes (`_X_N.del`).

`NumField` is the size of the array for `NormGen`, or -1 if there are no `NormGens` stored.

`NormGen` records the generation of the separate norms files. If `NumField` is -1, there are no `normGens` stored and they are all assumed to be 0 when the segment file was written pre-2.1 and all assumed to be -1 when the segments file is 2.1 or above. The generation then has the same meaning as `delGen` (above).

`IsCompoundFile` records whether the segment is written as a compound file or not. If this is -1, the segment is not a compound file. If it is 1, the segment is a compound file. Else it is 0, which means we check filesystem to see if `_X.cfs` exists.

If `HasSingleNormFile` is 1, then the field norms are written as a single joined file (with extension `.nrm`); if it is 0 then each field's norms are stored as separate `.fN` files. See "Normalization Factors" below for details.

6.2. Lock File

A write lock is used to indicate that another process is writing to the index. Note that this file is not stored in the index directory itself, but rather in the system's temporary directory, as indicated in the Java system property "java.io.tmpdir".

The write lock is named "XXXX-write.lock" where XXXX is typically a unique prefix computed by the directory path to the index. When this file is present, a process is currently adding documents to an index, or removing files from that index. This lock file prevents several processes from attempting to modify an index at the same time.

Note that prior to version 2.1, Lucene also used a commit lock. This was removed in 2.1.

6.3. Deletable File

Prior to Lucene 2.1 there was a file "deletable" that contained details about files that need to be deleted. As of 2.1, a writer dynamically computes the files that are deletable, instead, so no file is written.

6.4. Compound Files

Starting with Lucene 1.4 the compound file format became default. This is simply a container for all files described in the next section (except for the `.del` file).

Compound (`.cfs`) --> FileCount, <DataOffset, FileName> FileCount , FileData FileCount
FileCount --> VInt

DataOffset --> Long

FileName --> String

FileData --> raw file data

The raw file data is the data from the individual files named above.

7. Per-Segment Files

The remaining files are all per-segment, and are thus defined by suffix.

7.1. Fields

Field Info

Field names are stored in the field info file, with suffix .fnm.

FieldInfos (.fnm) --> FieldsCount, <FieldName, FieldBits> FieldsCount

FieldsCount --> VInt

FieldName --> String

FieldBits --> Byte

Fields are numbered by their order in this file. Thus field zero is the first field in the file, field one the next, and so on. Note that, like document numbers, field numbers are segment relative.

Stored Fields

Stored fields are represented by two files:

1. The field index, or .fdx file.

This contains, for each document, a pointer to its field data, as follows:

FieldIndex (.fdx) --> <FieldValuesPosition> SegSize

FieldValuesPosition --> UInt64

This is used to find the location within the field data file of the fields of a particular document. Because it contains fixed-length data, this file may be easily randomly accessed. The position of document n 's field data is the UInt64 at n*8 in this file.

2. The field data, or .fdt file.

This contains the stored fields of each document, as follows:

FieldData (.fdt) --> <DocFieldData> SegSize

DocFieldData --> FieldCount, <FieldNum, Bits, Value> FieldCount

FieldCount --> VInt

FieldNum --> VInt

Lucene <= 1.4:

Bits --> Byte

Value --> String

Only the low-order bit of Bits is used. It is one for tokenized fields, and zero for non-tokenized fields.

Lucene >= 1.9:

Bits --> Byte

- low order bit is one for tokenized fields
- second bit is one for fields containing binary data
- third bit is one for fields with compression option enabled (if compression is enabled, the algorithm used is ZLIB)

Value --> String | BinaryValue (depending on Bits)

BinaryValue --> ValueSize, <Byte>^ValueSize

ValueSize --> VInt

7.2. Term Dictionary

The term dictionary is represented as two files:

1. The term infos, or tis file.

TermInfoFile (.tis)--> TIVersion, TermCount, IndexInterval, SkipInterval, TermInfos

TIVersion --> UInt32

TermCount --> UInt64

IndexInterval --> UInt32

SkipInterval --> UInt32

TermInfos --> <TermInfo> TermCount

TermInfo --> <Term, DocFreq, FreqDelta, ProxDelta, SkipDelta>

Term --> <PrefixLength, Suffix, FieldNum>

Suffix --> String

PrefixLength, DocFreq, FreqDelta, ProxDelta, SkipDelta
--> VInt

This file is sorted by Term. Terms are ordered first lexicographically by the term's field name, and within that lexicographically by the term's text.

TIVersion names the version of the format of this file and is -2 in Lucene 1.4.

Term text prefixes are shared. The PrefixLength is the number of initial characters from the previous term which must be pre-pended to a term's suffix in order to form the term's text. Thus, if the previous term's text was "bone" and the term is "boy", the PrefixLength is two and the suffix is "y".

FieldNumber determines the term's field, whose name is stored in the .fdt file.

DocFreq is the count of documents which contain the term.

FreqDelta determines the position of this term's TermFreqs within the .frq file. In particular, it is the difference between the position of this term's data in that file and the position of the previous term's data (or zero, for the first term in the file).

ProxDelta determines the position of this term's TermPositions within the .prx file. In particular, it is the difference between the position of this term's data in that file and the position of the previous term's data (or zero, for the first term in the file).

SkipDelta determines the position of this term's SkipData within the .frq file. In particular, it is the number of bytes after TermFreqs that the SkipData starts. In other words, it is the length of the TermFreq data.

2. The term info index, or .tii file.

This contains every IndexInterval entry from the .tis file, along with its location in the "tis" file. This is designed to be read entirely into memory and used to provide random access to the "tis" file.

The structure of this file is very similar to the .tis file, with the addition of one item per record, the IndexDelta.

TermInfoIndex (.tii)--> TIVersion, IndexTermCount, IndexInterval, SkipInterval, TermIndices

TIVersion --> UInt32

IndexTermCount --> UInt64

IndexInterval --> UInt32

SkipInterval --> UInt32

TermIndices --> <TermInfo, IndexDelta> IndexTermCount

IndexDelta --> VLong

IndexDelta determines the position of this term's TermInfo within the .tis file. In particular, it is the difference between the position of this term's entry in that file and the position of the previous term's entry.

SkipInterval is the fraction of TermDocs stored in skip tables. It is used to accelerate TermDocs.skipTo(int). Larger values result in smaller indexes, greater acceleration, but fewer accelerable cases, while smaller values result in bigger indexes, less acceleration and more accelerable cases.

7.3. Frequencies

The .frq file contains the lists of documents which contain each term, along with the frequency of the term in that document.

FreqFile (.frq) --> <TermFreqs, SkipData> TermCount

TermFreqs --> <TermFreq> DocFreq

TermFreq --> DocDelta, Freq?

SkipData --> <SkipDatum> DocFreq/SkipInterval

SkipDatum --> DocSkip, FreqSkip, ProxSkip

DocDelta, Freq, DocSkip, FreqSkip, ProxSkip --> VInt

TermFreqs are ordered by term (the term is implicit, from the .tis file).

TermFreq entries are ordered by increasing document number.

DocDelta determines both the document number and the frequency. In particular, DocDelta/2 is the difference between this document number and the previous document number (or zero

when this is the first document in a TermFreqs). When DocDelta is odd, the frequency is one. When DocDelta is even, the frequency is read as another VInt.

For example, the TermFreqs for a term which occurs once in document seven and three times in document eleven would be the following sequence of VInts:

15, 8, 3

DocSkip records the document number before every SkipIntervalth document in TermFreqs. Document numbers are represented as differences from the previous value in the sequence. FreqSkip and ProxSkip record the position of every SkipIntervalth entry in FreqFile and ProxFile, respectively. File positions are relative to the start of TermFreqs and Positions, to the previous SkipDatum in the sequence.

For example, if DocFreq=35 and SkipInterval=16, then there are two SkipData entries, containing the 15th and 31st document numbers in TermFreqs. The first FreqSkip names the number of bytes after the beginning of TermFreqs that the 16th SkipDatum starts, and the second the number of bytes after that that the 32nd starts. The first ProxSkip names the number of bytes after the beginning of Positions that the 16th SkipDatum starts, and the second the number of bytes after that that the 32nd starts.

7.4. Positions

The .prx file contains the lists of positions that each term occurs at within documents.

ProxFile (.prx) --> <TermPositions> TermCount

TermPositions --> <Positions> DocFreq

Positions --> <PositionDelta> Freq

PositionDelta --> VInt

TermPositions are ordered by term (the term is implicit, from the .tis file).

Positions entries are ordered by increasing document number (the document number is implicit from the .frq file).

PositionDelta is the difference between the position of the current occurrence in the document and the previous occurrence (or zero, if this is the first occurrence in this document).

For example, the TermPositions for a term which occurs as the fourth term in one document, and as the fifth and ninth term in a subsequent document, would be the following sequence of VInts:

4, 5, 4

7.5. Normalization Factors

Pre-2.1: There's a norm file for each indexed field with a byte for each document. The `.f[0-9]*` file contains, for each document, a byte that encodes a value that is multiplied into the score for hits on that field:

Norms (`.f[0-9]*`) --> `<Byte> SegSize`

2.1 and above: There's a single `.nrm` file containing all norms:

AllNorms (`.nrm`) --> NormsHeader,`<Norms> NumFieldsWithNorms`

Norms --> `<Byte> SegSize`

NormsHeader --> 'N','R','M',Version

Version --> Byte

NormsHeader has 4 bytes, last of which is the format version for this file, currently -1.

Each byte encodes a floating point value. Bits 0-2 contain the 3-bit mantissa, and bits 3-8 contain the 5-bit exponent.

These are converted to an IEEE single float value as follows:

1. If the byte is zero, use a zero float.
2. Otherwise, set the sign bit of the float to zero;
3. add 48 to the exponent and use this as the float's exponent;
4. map the mantissa to the high-order 3 bits of the float's mantissa; and
5. set the low-order 21 bits of the float's mantissa to zero.

A separate norm file is created when the norm values of an existing segment are modified. When field *N* is modified, a separate norm file `.sN` is created, to maintain the norm values for that field.

Pre-2.1: Separate norm files are created only for compound segments.

2.1 and above: Separate norm files are created (when adequate) for both compound and non compound segments.

7.6. Term Vectors

Term Vector support is an optional on a field by field basis. It consists of 4 files.

1. The Document Index or .tvx file.

This contains, for each document, a pointer to the document data in the Document (.tvd) file.

DocumentIndex (.tvx) --> TVXVersion<DocumentPosition> NumDocs

TVXVersion --> Int

DocumentPosition --> UInt64

This is used to find the position of the Document in the .tvd file.

2. The Document or .tvd file.

This contains, for each document, the number of fields, a list of the fields with term vector info and finally a list of pointers to the field information in the .tvf (Term Vector Fields) file.

Document (.tvd) --> TVDVersion<NumFields, FieldNums, FieldPositions,> NumDocs

TVDVersion --> Int

NumFields --> VInt

FieldNums --> <FieldNumDelta> NumFields

FieldNumDelta --> VInt

FieldPositions --> <FieldPosition> NumFields

FieldPosition --> VLong

The .tvd file is used to map out the fields that have term vectors stored and where the field information is in the .tvf file.

3. The Field or .tvf file.

This file contains, for each field that has a term vector stored, a list of the terms, their frequencies and, optionally, position and offset information.

Field (.tvf) --> TVFVersion<NumTerms, Position/Offset, TermFreqs> NumFields

TVFVersion --> Int

NumTerms --> VInt

Position/Offset --> Byte

TermFreqs --> <TermText, TermFreq, Positions?, Offsets?> NumTerms

TermText --> <PrefixLength, Suffix>

PrefixLength --> VInt

Suffix --> String

TermFreq --> VInt

Positions --> <VInt>TermFreq

Offsets --> <VInt, VInt>TermFreq

Notes:

- Position/Offset byte stores whether this term vector has position or offset information stored.
- Term text prefixes are shared. The PrefixLength is the number of initial characters from the previous term which must be pre-pended to a term's suffix in order to form the term's text. Thus, if the previous term's text was "bone" and the term is "boy", the PrefixLength is two and the suffix is "y".
- Positions are stored as delta encoded VInts. This means we only store the difference of the current position from the last position
- Offsets are stored as delta encoded VInts. The first VInt is the startOffset, the second is the endOffset.

7.7. Deleted Documents

The .del file is optional, and only exists when a segment contains deletions.

Although per-segment, this file is maintained exterior to compound segment files.

Pre-2.1: Deletions (.del) --> ByteCount, BitCount, Bits

2.1 and above: Deletions (.del) --> [Format], ByteCount, BitCount, Bits | DGaps (depending on Format)

Format, ByteSize, BitCount --> UInt32

Bits --> <Byte> ByteCount

DGaps --> <DGap, NonzeroByte> NonzeroBytesCount

DGap --> VInt

NonzeroByte --> Byte

Format is Optional. -1 indicates DGaps. Non-negative value indicates Bits, and that Format is

excluded.

ByteCount indicates the number of bytes in Bits. It is typically $(\text{SegSize}/8)+1$.

BitCount indicates the number of bits that are currently set in Bits.

Bits contains one bit for each document indexed. When the bit corresponding to a document number is set, that document is marked as deleted. Bit ordering is from least to most significant. Thus, if Bits contains two bytes, 0x00 and 0x02, then document 9 is marked as deleted.

DGaps represents sparse bit-vectors more efficiently than Bits. It is made of DGaps on indexes of nonzero bytes in Bits, and the nonzero bytes themselves. The number of nonzero bytes in Bits (NonzeroBytesCount) is not stored.

For example, if there are 8000 bits and only bits 10,12,32 are set, DGaps would be used:

(VInt) 1 , (byte) 20 , (VInt) 3 , (Byte) 1

8. Limitations

There are a few places where these file formats limit the maximum number of terms and documents to a 32-bit quantity, or to approximately 4 billion. This is not today a problem, but, in the long term, probably will be. These should therefore be replaced with either UInt64 values, or better yet, with VInt values which have no limit.