Portable format for Apache Lucene indexes

Andrzej Białecki <ab@apache.org>

Draft 2011.10.04

1 Introduction

Apache Lucene (http://lucene.apache.org) indexes use binary formats to represent various index data in a way that is compact, efficient to write and read, and able to support all aspects of the indexing and search functionality.

There are however differences in the layout of this format between various versions of the library. These differences came about as a natural result of the continuing development and the need to support new areas of functionality (e.g. per-document values, term vectors, shared document stores, compound formats, various field flags, etc) or the need to resolve certain deficiencies (e.g. char vs. byte[] terms).

As a result, nearly every released version of Lucene produces slightly different file formats. There is a limited support for backwards-compatibility so that certain newer versions of Lucene can still open indexes created with older versions.

This document describes the rationale for and a design of a portable index format that is expressive and generic enough to support all currently used versions of Lucene indexes with easy to add extensions to support future releases.

1.1 Backwards-compatibility

The maintenance of the backwards-compatibility code becomes increasingly difficult and time-consuming as the number of supported versions increases. This cost of maintenance is a function of the number of source formats and the number of target formats that need to be supported.

One solution to this problem is to limit the number of supported versions to an absolute minimum.

An alternative solution, to be presented here, is to come up with an intermediate format that is expressive and flexible enough to represent all aspects and details of various versions in such a way that it's easy for different Lucene versions to pick and choose only those aspects that are supported by a particular version, and without breaking due to format differences. The working hypothesis is that the cost of back-compat maintenance should be lower in this case, because each Lucene version would have to implement only a compatibility layer with the intermediate format.

In recent versions of Lucene the CodecProvider and Codec API-s can be used to implement various on-disk formats for index data. Still, the Codec API doesn't cover all parts of the index yet (e.g. term vectors, stored fields, norms, global index data, …). There is a SimpleText Codec that supports human-readable serialization of the parts that Codec-s support, but still it is inflexible in that it supports only a particular version of Lucene and has no provisions for accommodating extensions or for skipping unrecognized sections. However, both the Codec API and the SimpleTextCodec can benefit from the ideas presented here, and hopefully they can be either extended or partially reused to implement the tools described below.

Luke (http://code.google.com/p/luke) has a module for exporting parts of an index to an XML format. It supports exporting stored fields, term vectors and overall index statistics, but it doesn't support exporting other parts of the index (postings, term dictionary), and it doesn't have any support for a reverse operation, i.e. importing of the XML data to create an index.

Please note that the expected workflow for upgrading / downgrading the index data will always involve at least one export operation from the source index to the portable format. While this incurs significant IO cost it still seems worthwhile instead of re-indexing.

1.2 Backup and archival

In some cases it may be useful to periodically back up the content of the index. Currently this means making a copy of the binary files in a format specific to a concrete version of Lucene. However, as time passes by and applications use newer versions of Lucene, it may no longer be possible to use such a backup copy e.g. for diagnosis, testing, or examination of data.

If an index is first exported to the portable format then newer versions of Lucene will still be able to open and use it.

1.3 External text processing

Lucene provides an extensive support for text analysis in the form of Analyzer-s, Tokenizer-s, TokenFilter-s, and an extensible format for TokenStream-s that produce Token-s with arbitrary Attribute-s.

However, throughout this API the accent is put on performing text analysis within the context of Lucene, and only limited support exists for feeding documents pre-processed by external text processing pipelines. Lucene API supports this via Field.setTokenStream() and Solr support is provided by a PreAnalyzedField available in SOLR-1535 Jira issue.

It would be relatively easy to enhance Lucene/Solr API by adding support for the following operations:

- (trivial) add a Field constructor that takes the stored value and TokenStream.
- Use a more expressive serialization format for submitting pre-analyzed documents – the format implemented in SOLR-1535 is too limited. E.g. it doesn't support arbitrary attributes, docvalues, term vectors, or submitting several documents in one data stream.
- add an API for adding fully-inverted documents – this data wouldn't have to form a standalone index, nor does it have to be in an optimized format – e.g. instead of a single term dictionary and combined postings list each document could carry its own inverted data, so that the external process wouldn't have to aggregate cross-document information (as is the case with standalone indexes).
- Add an API for adding full indexes created with incompatible versions of Lucene, or created with a different inverted-index search libraries – as long as that external process can structure the data according to the intermediate format described here.

2 Functional requirements

This section discusses functional requirements for the portable index data format.

2.1 Data model requirements

2.1.1 Functionality over implementation details

Data representation should focus on data semantics and not on a particular data representation (e.g.

Java class internals, binary encoding, or compression) used by the implementation. This means that in some cases the portable representation will be less efficient and will consume more space, at the benefit of portability.

In some justified cases a selected lowest common denominator encoding can be used in order to reduce the size of data. E.g. for term dictionary this could be a front coding, for postings this could be a simple delta coding.

Data model should favor the logical function of the data and its most basic representation rather than its actual representation in a specific implementation. Thus e.g. terms should be viewed logically as an array of bytes and not as BytesRef-s or String-s. Similarly, common term attributes such as positions or start/end offsets should be represented as primitive integers, and identified by symbolic (functional) names instead of using fully-qualified class names.

2.1.2 Expressive enough for all existing and many future index formats

As of the time of writing this document, Lucene versions 3.x are in wide-spread use, and there are numerous users of the yet-unreleased version 4.x. There is also a large community of users that still use versions 2.4 up to version 2.9. The data model should cover all aspects of index formats for these versions, plus it should allow easy extending to support new functionality being added or planned for 4.x versions.

2.1.3 Versioned

Even though the aim of this specification is to support all existing and many future formats, it's conceivable that future extensions will be needed. Data model should support future extensions in a way that allows graceful fallback by clients that use earlier versions of the model.

2.1.4 Faithfully represent all index data

It should be possible to serialize and deserialize a complete index without any loss of data for the same versions of producer and consumer. This means that the model should be able to represent all currently known types of data in the index in a way that allows for unambiguous and complete serialization/deserialization by the same versions of Lucene.

2.1.5 Graceful handling of unsupported features

Clients using different versions of Lucene should be able to easily ignore data that they don't support, in a way that doesn't interfere with processing of other data that they do support.

2.2 Serialization format requirements

2.2.1 Easy to produce, easy to consume

It should be possible to produce and consume data in this format with minimal or no dependencies on Java libraries except the core platform library.

2.2.2 Fast random access to sections

It should be possible to access sections of the data randomly in an efficient way. This is potentially at odds with the requirement for streaming – the serialization format may need separate modes or separate representations to support either type of access.

2.2.3 Support for arbitrary number and type of sections

This is a consequence of the extensibility requirement. There are well-known index sections that should be pre-defined, but the format should allow for adding other sections.

Clients should be able to easily ignore any sections unknown to them.

2.2.4 Support hierarchy of sections

Some data may be better represented in a nested hierarchy of sections (e.g. segments in an index, parts of segments in a segment). An open issue is the overhead per section – it's likely that only major sections could be represented this way, and individual data sections would use other more compact formats.

2.2.5 Support per-section metadata

In addition to the name of the section the format should support other arbitrary metadata, in the form of a collection of key-value strings. Support for non-string values would be useful but not required.

2.2.6 Reasonably efficient, low overhead, optional compression

2.2.7 (Optional?) Single file

It should be possible to form a single file that includes all index data.

2.2.8 (Optional?) Streamable

It should be possible to produce and consume the data using streams, without first caching the complete data stream and having to use random access.

2.2.9 (Optional?) Allow for IndexWriter / IndexReader API

This optional requirement means that the serialization format could be suitable for creating via the IndexWriter API, and it could be possible to read data from this format using the IndexReader API.

2.2.10 (Optional) A human-readable variant

Sections of the file may use binary representation for efficiency and compactness. However, this prevents casual reading by humans, which is useful for diagnostics or for manual creation / editing of data. Optionally it should be possible to serialize the data in a human-readable format, and deserialize it without any data loss or functional loss.

2.2.11 Non-requirements

The index format doesn't have to offer the same level of performance for all operations as the regular binary formats do – its purpose is only to transfer the data, and not to be used as an on-line backing store for regular Lucene operations.

3 Design

3.1 Data model of Lucene indexes

Note that the model presented below is not concerned about the details of implementations, rather it tries to outline an information model that the portable format needs to support.

Lucene indexes can be viewed as containers that consist of a hierarchical structure of parts (sections), with each section storing different type of data. Hierarchy reflects the multi-segment nature of Lucene indexes, and the fact that a single index carries multiple types of data about multiple documents.

3.1.1 Metadata

At different levels in the model there is a need to express metadata. This model proposes to use a straightforward Map<String,String>. Numeric values are represented by String.valueOf(value). Binary values are represented using a single-line (no wrapping) Byte64 encoding.

3.1.2 Map-like structure

Conceptually various parts of the index and various parts of data in each section of the index can be represented using map-like structures, with a set of keys that point to values, values being either nested sections or primitive data.

This model is very flexible but also very opaque. There are well-known sections of indexes that remain unchanged across many versions of Lucene, and these should use well-known predefined key names. Initially the sections present in Lucene 4.x will be defined, because they are a strict superset of section types present in earlier versions of Lucene.

3.1.3 Index-level data

Lucene indexes consist of global index metadata and segments.

Index metadata logically consists of:

- the list of commit points, each commit point listing segments (sub-readers) that are included in a commit point

Note that it might not be required to represent this structure as a hierarchy – the list of commit points may simply refer to a flat listing of segments that follows.

3.1.4 Commit point data

Each commit point consists of:

- commit point metadata
- “user info” metadata (key/value pairs of strings)
- a list of segments

3.1.5 Segment (sub-reader) data

Each segment logically consists of:

- sub-index metadata: this includes the number of documents in the sub-index, list of field names and their metadata (flags), and other per-segment metadata
- deleted documents
- stored document fields
- term dictionary
- frequencies

- field norms
- positions (postings)
- term vectors
- docvalues (4.x only)

Future formats may add other parts to this list.

3.1.5.1 Deleted documents

Deleted documents are represented as a bitset where set bits mark deleted documents. This data could be serialized using the existing DGap serialization, with no additional headers/trailers.

3.1.5.2 Stored document fields

Stored fields usually contain strings, but in general may contain arbitrary byte arrays. The interpretation of field values depends on per-field flags, which can be considered as per-field metadata (available in SegmentInfo). Logically fields for one document are grouped together, and then they are followed by fields for the next document, and so on, in ascending document id order. Since there could be gaps in document ids each group of stored fields should explicitly state the document id. Stored fields could be represented in a way similar to the SimpleTextCodec (escaped byte strings).

3.1.5.3 Term dictionary

Term dictionary consists of terms in alphabetical order (again, in general they could be arbitrary byte arrays) and associated per-term metadata, such as term frequency and term postings. Due to the size of the postings data this information is modeled as another section, but per-term global frequency information may be modeled as a per-term metadata instead of a separate section. There is also a dictionary-level metadata that provides parameters to implementations (e.g. skip interval, skip levels, term count, etc).

Lucene term dictionary contains also pointers to appropriate sections in the postings files – in the portable representation this could be modeled by an abstract term number that ties together sections in the term dictionary and entries in the postings' section.

For efficiency term dictionary may use the lowest common denominator encoding, such as delta front coding, which should be marked in the term dictionary metadata.

3.1.6 Field norms

Field normalization factors currently consist of a single small float value per document per field (that uses a normalization factor). This data could be logically presented as a separate section that is explicitly numbered using the same document id-s as other per-document sections, with normalization factors for all fields in a document grouped together. Alternatively, this could be added to the stored fields section as a per-document metadata.

3.1.7 Positions (postings)

TBD. This should use an encoding that can be easily extended to support other posting attributes/sections. Currently we serialize positions and payloads.

3.1.8 Term vectors

TBD. This could use a combination of encodings used for term dictionary and postings.

3.1.9 Per-document values (docvalues)

TBD.

3.2 Serialization format

There are many options for serialization formats, and many of them are supported by the standard Java library:

- plain line-oriented text: this is e.g. the schema that SimpleTextCodec uses
- XML: this is the format that Luke's XMLExporter uses
- JSON: one of the options for Solr input documents (and output results)
- Avro: this is a toolkit for creating and consuming self-describing strongly typed formats, either binary or JSON. It would add a small dependency (300kB JAR), but benefits could outweigh this cost.
- many other options ...

However, none of these formats (except for Avro) are particularly suited for serialization of large amounts of binary data. None of them have provisions for quickly locating specific sections of data. It's awkward also to apply compression – either the whole file needs to be compressed (and then individual sections cannot be quickly found) or if individual sections are compressed then their binary data needs to be converted into a text-like format.

For quick random access within large multi-record sections (e.g. term dictionary or postings) we could use the same design as the StandardCodec uses for term dictionary (the same pattern is present e.g. in Hadoop MapFile-s): create an adjacent lookup index file that lists keys and pointers to every N-th record in the data file. N should be sufficiently large to be able to load the complete lookup index in memory.

When it comes to the format for a single container file, surprisingly majority of the requirements are satisfied with the standard ZIP file format, which is supported by classes in java.util.zip package. This format supports stream writing and stream reading, as well as quick random access to the entries, optional compression of entries, and it also supports per-entry metadata. It's also supported by many non-Java tools, which makes it easy to create and examine. However, in current version of Java (1.6) this format is limited to files smaller than 4GB, and Lucene indexes often exceed this size. Also the metadata ("extra") is an unstructured byte array.

It's also fine to consider the existing Directory abstraction as a suitable container format – and if a single file representation is needed then the content of a Directory can be packed into a single archive using any one of the supported archive formats (ZIP, or via commons-compress TAR).

For now we assume that the output format will be the Directory abstraction. This format is not hierarchical, but this limitation can be worked around with suitable structured naming of files, which is already in use by all other codecs. Lack of structured metadata can be worked around in a way described below.

3.2.1 Serialization of metadata

Lucene's Directory doesn't explicitly support structured per-file metadata. This specification defines that optional per-file metadata will be stored in files with ".meta" extensions, and the interpretation of their content is to be an output compatible with the output of java.util.Properties.storeToXml(..), and to use the UTF-8 encoding.

Currently known metadata should use predefined key names.

3.2.2 Serialization of known index parts

Currently known index parts should use predefined names for specific entries. Again, we can reuse and extend the SimpleTextCodec approach.

Path names in Directory can express hierarchy of parts by using hierarchical, path-like naming patterns. Even though the Directory listing produces a flat list of all entries regardless of their nesting level, the reader can still re-construct a hierarchy of entries based on their names (e.g. with a "." character reserved as a path element separator).

It's not feasible to use separate files for items that occur in large numbers (e.g. terms or postings). These parts should be serialized as multi-record streams. Initially we could use the format that SimpleTextCodec uses for postings, or similar line-oriented formats for other parts of the index that are not supported by the Codec API. Readers and writers of these formats should be lenient and according to the requirements they should be able to easily skip unrecognized sections.

TBD.

3.2.3 Adding extensions and handling unknown metadata, sections or attributes.

Extensions to the currently known data model can be expressed as:

- additional metadata at any level. New metadata that is semantically different from the existing metadata should use different unique metadata keys. At the same time, if a metadata is a specialization of some other common metadata that occurs in different versions of Lucene, then an appropriate generalized metadata should also be checked (or added). In case of conflict a more specific metadata wins.
- additional sections at any level. New sections should follow the naming as described above.

Implementations are required to ignore sections that they don't support, and should use best effort approach to re-create the index from information available from sections that they support and recognize. Similarly, at a finer-grained level, any attributes or metadata keys that occurring within sections, which are not recognized, should be ignored and sensible defaults specific to a particular version of Lucene should be applied. Fatal errors or exceptions should be thrown only in case of I/O errors or in cases where a recognized section cannot be processed without significant information loss.

4 Implementation

4.1 PortableCodecProvider / PortableCodec

TBD. This is a codec that serializes parts of the index that are handled by the Codec API. For the initial implementation a text-oriented format like the one used by SimpleTextCodec is used. Later on probably a more compact representation needs to be designed.

4.2 PortableIndexExporter

This is a utility class that takes an IndexReader and writes its data in the portable format.

TBD.

4.3 PortableIndexImporter

This is a utility class that opens a file in portable format and writes out an index in a format specific to the current version of Lucene.

TBD.

4.4 PortableIndexReader

This is a utility class that opens a file in portable format and presents its content as an IndexReader.

TBD. This class wouldn't be needed if all parts of the index were under control of CodecProvider/Codec. It's likely that this class will have to create some throwaway lookup structures in order to support some random seek operations efficiently.

4.5 PortableIndexWriter

This is a utility class that presents the API of IndexWriter and writes data in the portable format.

TBD. Again, this wouldn't be needed if Codecs supported all parts of the index.