jamk.fi

Enhancing organization and maintenance of big data with Apache Solr in IBM WebSphere Commerce deployments

Rudolf Grigeľ

Bachelor’s thesis
May 2015

Degree Programme in Software Engineering
School of Technology, Communication and Transport

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Description

Author(s) Grigeľ, Rudolf	Type of publication Bachelor's thesis	Date 01.05.2015
		Language of publication: English
	Number of pages 56	Permission for web publication: x

Title of publication
Enhancing organization and maintenance of big data with Apache Solr in IBM WebSphere Commerce deployments

Degree programme
Software Engineering

Tutor(s)
Lappalainen-Kajan, Tarja

Assigned by
Descom Oy

Abstract

The main objective of this thesis was to enhance the organization and maintenance of big data with Apache Solr in IBM WebSphere Commerce deployments. This objective can be split into several subtasks: reorganization of data, fast and optimised exporting and importing, efficient update and cleanup operations.

E-Commerce is a fast growing and frequently changing environment. There is a constant flow of data that is rapidly growing larger and larger every day which is becoming an serious problem in the current process of data handling. Apache Solr is an enterprise search platform used by IBM WebSphereCommerce. It is a fast indexing engine that can handle large amounts of data with proper configuration, data organization and custom extensions.

This thesis results in extensions for Apache Solr programmed in Solr's Java API. Other expected results of this thesis are data organization rules and recommendations that can help with the current big data situation with Apache Solr in IBM WebSphere Commerce.

Testing and evaluation of the results are an important part of this thesis and tests are run with currently deployed IBM WebSphere Commerce systems and data that comes from these eCommerce systems.

Keywords/tags
Apache Solr, Java, Big Data, eCommerce

Miscellaneous

Contents

Figures and tables

Acronyms

eCommerce, **E-commerce**	electronic commerce
B2C	Business to Customer
B2B	Business to Business
SQL	Structured Query Language
NoSQL	Not only SQL
REST	Representational State Transfer
API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformation
XPath	XML Path language
JSON	JavaScript Object Notation
CSV	Comma Separated Values
RDBMS	Relational Database Management System
DIH	Data Import Handler
JDBC	Java Database Connectivity Technology
URL	Uniform Resource Locator
HTML	HyperText Markup Language
regex	regular expression
JVM	Java Virtual Machine
HDFS	Hadoop Distributed File System

1 Organization and maintenance of data

1.1 Introduction

“Enterprise data is defined as data shared by the users of an organization, generally across departments and/or geographic regions. Because enterprise data loss can result in significant financial losses for all the parties involved, enterprises spend time, money and other limited resources on careful and effective data modeling, security and storage.” (Technology dictionary, 2015)

“Another main component of enterprise data organization is the analysis of relatively structured and unstructured data. Structured data comprises data in tables that can be easily integrated into a database and, from there, fed into analytics software or other particular applications. Unstructured data is data that is raw and unformatted, the kind of data that you find in a simple text document, where names, dates and other pieces of information are scattered throughout random paragraphs. Experts have developed tech tools and resources to handle relatively unstructured data and integrate it into a holistic data environment.” (Technology dictionary, 2015)

Enterprise businesses accept data organization strategies for the sake of better usage of the data assets that they own. E-commerce business is a world where data sets represent some of the most valuable assets. Executives and other professionals may focus on data organization as a component of a comprehensive strategy to streamline business processes, get better business intelligence and generally improve a business model. (Technology dictionary, 2015)

Data management encapsulates different techniques that provide step by step processes beginning with creation, processing, deletion and analysis. Operative framework, robust yet easily scalable infrastructure and technological resources are defining factors of data management solutions.

1.2 Project goals

The objectives of this thesis can be divided into several interrelated and independent subtasks. The subtasks can be divided as following:

- enhancing current organization of data

- Apache Solr exporting and importing extensions that would be able to handle Big Data
- optimized delta and full imports of data into Apache Solr
- efficient update, remove and cleanup operations

Meeting each of the given project goals depends highly on the following set of tasks:

- deep understanding of data
- efficient implementation
- testing in real eCommerce environment,
- retrieval of relevant results
- evaluation of results

These tasks can be understood as means of accomplishing the given objectives and they are significant in different stages of this process. Results and knowledge from each of these steps are used as input to the following step. With deep understanding of data it is possible to create implementation tailored for the specific needs and also for a general purpose. If the implementation is ready to use it should be tested in a real environment. Testing is an important part of the life cycle of every software because the results provided from testing can be analyzed and later evaluated. Comparison of solutions is a key to improve the current process of handling Big Data and it would not be possible without testing and evaluating the results.

It is expected that by meeting the set goals it could be possible to improve the ability to efficiently maintain large amounts of data in existing and future projects based on IBM WebSphere Commerce and Apache Solr. Making the process of maintenance more efficient means spending less amount of the limited resources, which is leads to cutting down the costs important for both the customer and the eCommerce solution provider.

2 Big Data

2.1 Overview

“Big Data matters to business because it is directly attributable to analytics and not only that. The need for analytics has evolved from being a business initiative to business imperative.” (Zikopoulos, et al., 2013)

Big Data is often defined by special V words and these four are the most commonly used:

- volume
- variety
- velocity
- veracity

The issue about the volumes of data is that data changes rapidly over the time. Phrase data grow larger and larger can be backed up by some interesting numbers. In 2009, it was estimated that the world had about 0.8 zeta bytes of data, in 2010, this number crossed 1 zeta bytes border and at the end of 2011 that number was estimated to be around 1.8 zeta bytes. (Zikopoulos, et al., 2013) To have a complete picture one zeta byte is equal to 10^{21} bytes.

Variety is characteristic of Big Data that comes from the different nature of data itself. Data can be formally divided into structured or unstructured; however there is no such matter as completely unstructured data. There is still some level of hierarchy or order in seemingly unstructured content. Variety is about capturing all forms of data. (Zikopoulos, et al., 2013)

Another important trait of Big Data is called velocity. It can be defined as the speed of data arrival and processing within the system. Faster reactions to changes provide advantages and insights that are valuable for businesses. (Zikopoulos, et al., 2013)

Veracity can commonly be referred to as quality or trustworthiness of data. (Zikopoulos, et al., 2013)

Organizations find it valuable to capture more data and process it more quickly. They find it expensive, if even possible, to do so with relational databases. The primary reason is that a relational database is designed to run on a single machine, however it is usually more economical to run large data and compute loads on clusters of many smaller and cheaper machines. Many NoSQL databases are designed explicitly to run on clusters, thus they make a better fit for Big Data scenarios. (Sadalage & Fowler, 2013)

The benefits of traditional relational database solutions cannot be ignored. Traditional solutions and Big Data solutions should be seen as complements. Only when the most of both of these worlds is obtained success can be spoken of, especially in eCommerce environment.

Data governance is an important term in connection with data and not only the Big Data. Governance can be understood as a set of policies or rules that define how the data should be managed. The policy may or may not involve actively governing the data (cleansing it, securing, etc.) but it should be clear for all the data if it is be governed or not. If not reactive governance is discussed, which means the problems are fixed as they arise and that is more costly and complex for enterprises. (Zikopoulos, et al., 2013)

2.2 Scenarios

Product pricing is one of the several Big Data scenarios. Efficient and on demand price alteration is not a trivial task if large businesses with hundreds of thousands or millions of products are considered which have to be available to customers with an exact price in exact moment. This problem can be extended even further considering there are multiple groups of customers. There are B2C customers and B2B customers and they certainly should be differentiated. There are special campaigns going on depending on seasons, for example. Prices are also dependent on location. Here the Big Data can be introduced because as can sees from one product price there can be twenty and the applications have to be able to process this kind of information as fast as possible.

Web store assets data searching can also be seen as one of the scenarios including Big Data. This particular scenario emerges from the volume of searchable data. Fast and relevant results are the necessary requirements that must be met.

There are many more scenarios that may or may not be relevant to eCommerce industry; however, the above-mentioned product pricing and web store assets data searching are the most relevant to this project.

3 WebSphere Commerce

3.1 E-commerce and WebSphere Commerce

“E-commerce can be defined as the use of the Internet and the Web to transact business. More formally, digitally enabled commercial transactions between and among the organizations and the individuals.” (Traver & Laudon, 2007)

IBM WebSphere Commerce is an eCommerce omni-channel software platform that provides extensible support for marketing, sales, orders and others. It enables B2C and B2B sales to customers across multiple channels including Web, mobile, stores, etc. It is built on top of the Java and Java EE platform with usage of open standards. (WebSphere Commerce product overview, 2015)

3.2 Architecture

WebSphere Commerce is a complex ecosystem divided into multiple layers according to their functionality. General knowledge about WebSphere Commerce architecture is a necessary requirement since without this knowledge it would not be possible to understand how the data is consumed and processed within the system. (See Figure 1.)

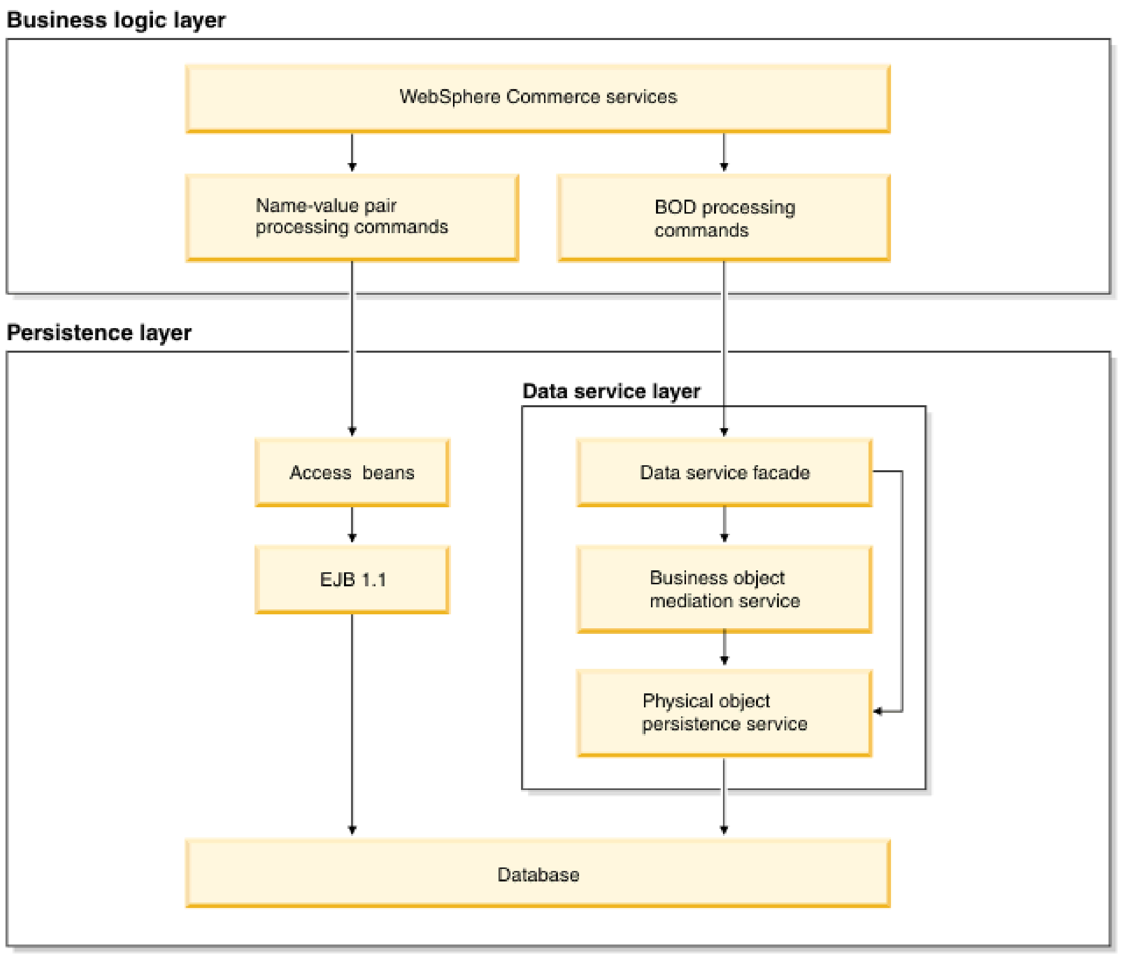
Business logic layer
WebSphere Commerce services
Name-value pair processing commands
BOD processing commands
Persistence layer
Data service layer
Access beans
EJB 1.1
Data service facade
Business object mediation service
Physical object persistence service
Database

Figure 1. WebSphere Commerce common architecture (WebSphere Commerce product overview, 2015)

Performance can be maintained for hundreds of thousands of documents stored in a database behind the persistence layer; however, when the data start to increase larger and larger from several hundreds of thousands of documents to tens of millions and higher the performance starts to fall down drastically. This affects negatively the ability to maintain such data, not to mention the negative impact on whole system that is highly dependent on quick data access.

3.3 Solr search integration

WebSphere Commerce integrates into its suite a complex, enterprise search solution built on top of the open platforms called Apache Solr and Apache Lucene. It provides separated components extending the scope of searchable content for business users for both structured and unstructured data. The programming model illustrates various search components such as expression and indexing preprocessors and postprocessors, expression providers and filters. (See Figure 2.)

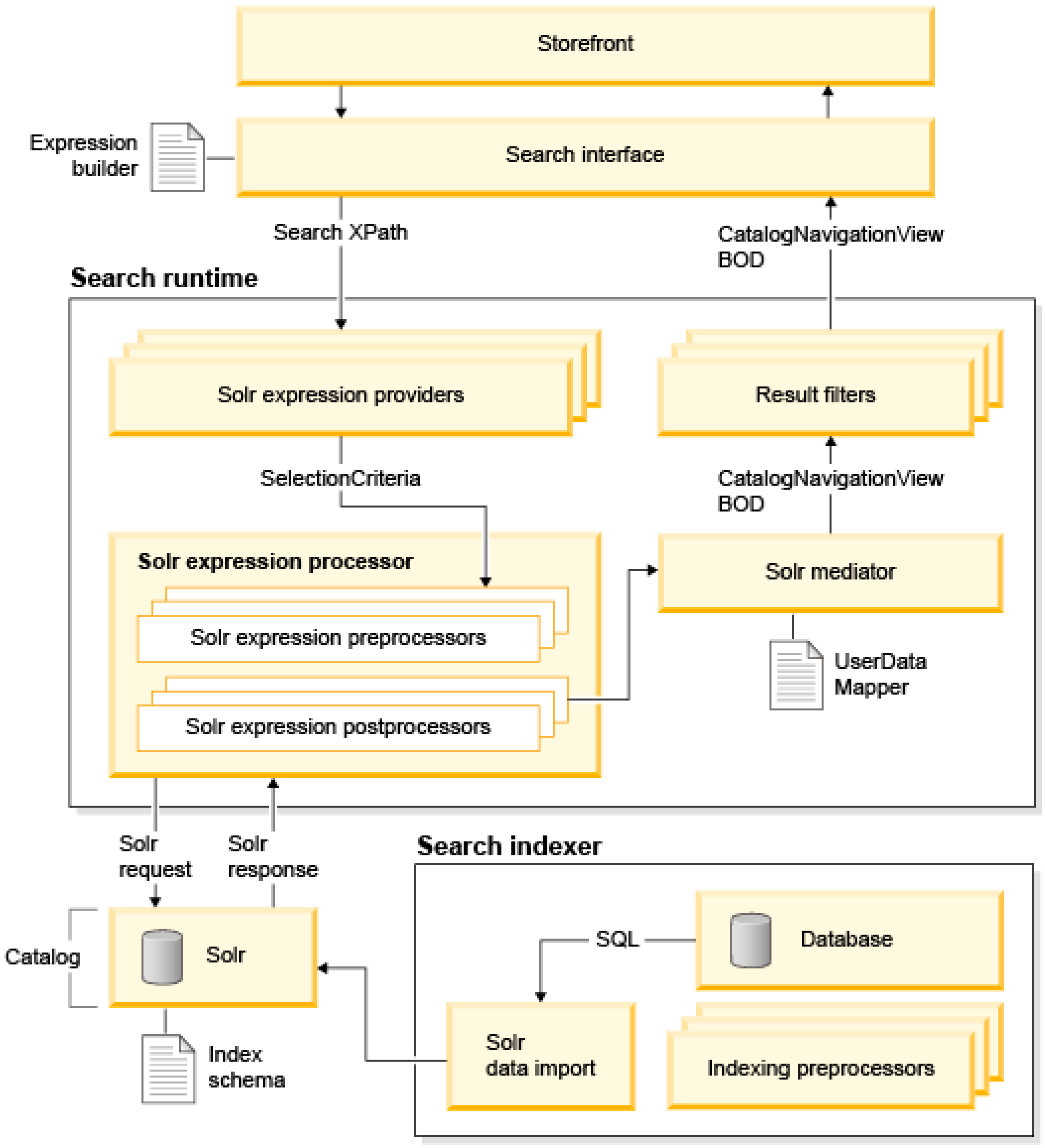

Figure 2. WebSphere Commerce Search Programming Model (WebSphere Commerce product overview, 2015)

Understanding **Search** is a key concept in the process of enhancement of the Big Data maintenance. Advantage can be taken of this architecture in order to handle Big Data scenarios in WebSphere Commerce environment.

3.4 Dataload

WebSphere Commerce platform provides common pattern for loading data into the commerce environment. The Dataload framework supports indexing data directly into the search server based on Apache Solr. A SolrJ Java client is used to index a flat data structure from an external data sources in XML or CSV format. The Dataload processing is done by multiple components: the reader, the mediator and the writer. (See figure 3.)

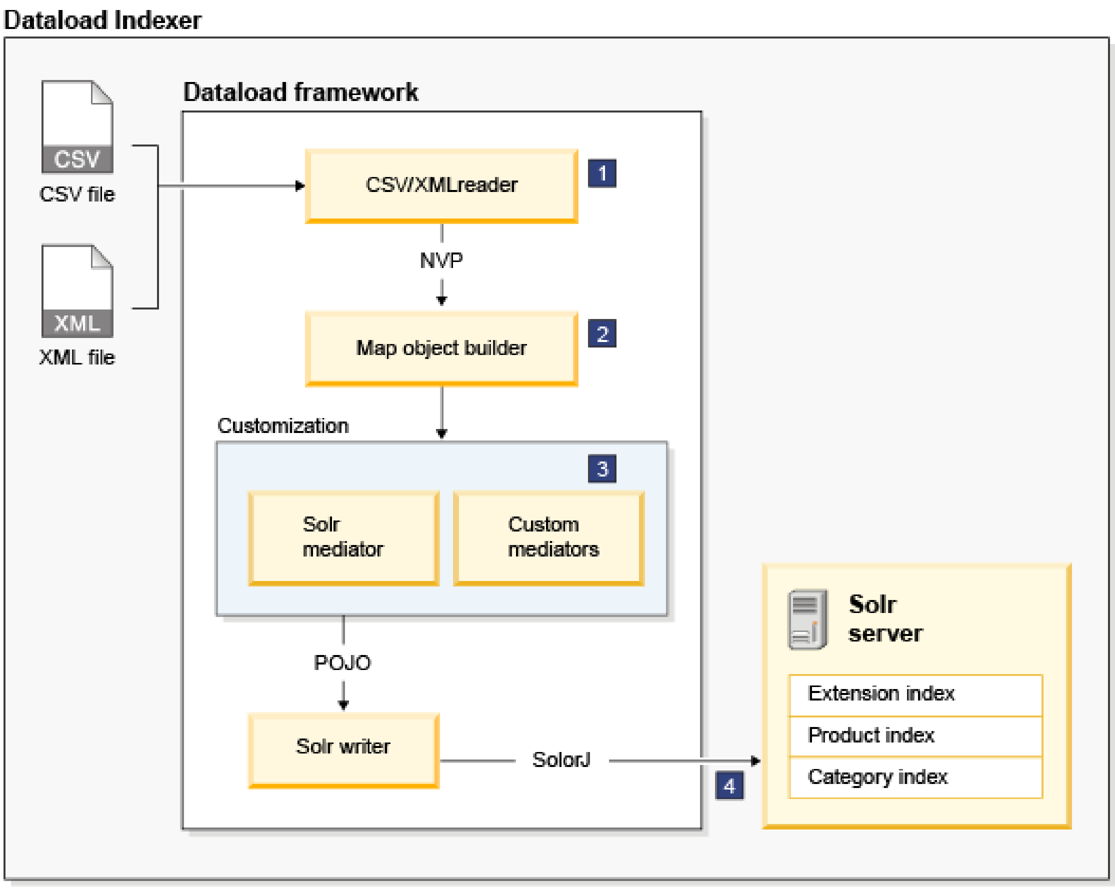

Figure 3. WebSphere Commerce Dataload Indexer (WebSphere Commerce product overview, 2015)

Dataload framework is recommended method for loading data into the WebSphere Commerce; however, in certain situations it can block resources that are needed by other processes. Especially when the huge amount of documents needs to be processed. The Dataload cannot be used by other processes until it finishes current processing.

4 Apache Solr

4.1 Introduction

Solr is standalone, open source, enterprise and search server with REST like API. Solr is based on Apache Lucene that is an open source, high performance text search engine library written in Java programming language. It uses for communication standard technologies such as HTTP, XML, JSON and others. This means that a wide variety of clients are able to use Solr, not just the web applications but also the mobile applications and any other platform capable of communicating through HTTP. Solr is written in Java as well as Lucene, and the same programming language is used to further extend and modify Solr. Multiple plugin interfaces are used as extension points. Apache Solr conceptual architecture can be seen below. (See Figure 4.)

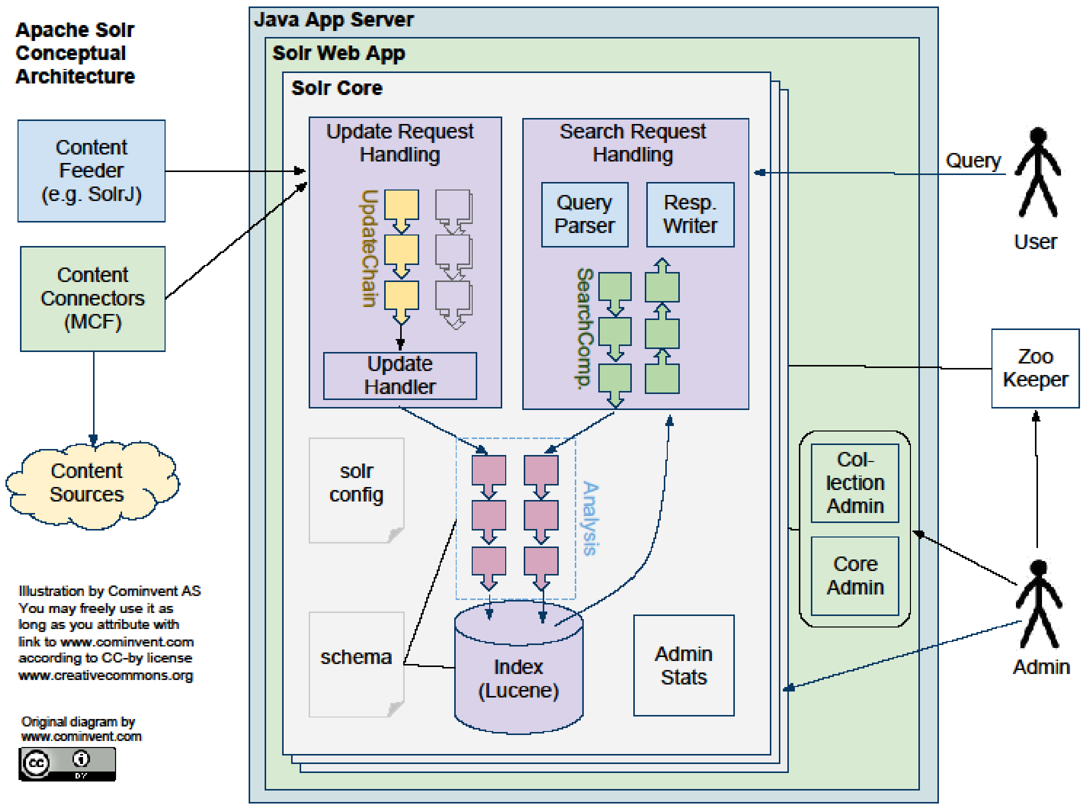

Figure 4. Apache Solr conceptual architecture (Lucene Apache Jira, 2015)

Few of the features supported by Solr are listed as follows:

- advanced full-text search capabilities

- results faceting, highlighting and spellchecking
- simple configuration files for cores, schemas
- several caches for faster response times
- support for cloud computing and distributed search
- high scalability and fault tolerance

4.2 Apache Lucene

“Lucene is a high performance, scalable Information Retrieval (IR) library. It lets users add indexing and searching capabilities to the applications. Lucene is a mature, free, open source project implemented in Java and provides simple and powerful core API that requires minimal understanding of full text searching and indexing.” (Gospodnetic & Hatcher, 2005)

Features supported by Apache Lucene:

- near real time indexing
- inverted index for efficient document retrieval based on indexed terms
- rich set of chainable text analysis components, such as tokenizers, language specific stemmers and transformers
- custom query syntax with appropriate query parser
- scoring algorithm based on Information Retrieval principles

General architecture of Apache Lucene is shown in figure below. (See Figure 5.)

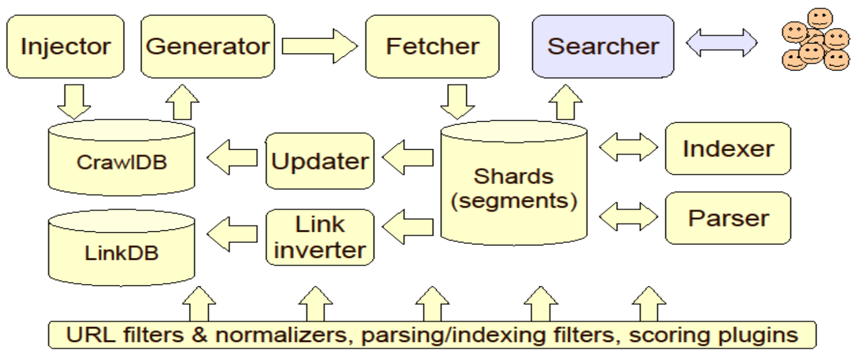

Figure 5. Apache Lucene Architecture (Lucene Apache Jira, 2015)

4.3 Schema design

A key idea to learn is that the queries needed to be supported by Solr completely drive the schema design. On the other hand, relational databases typically use decomposition to third normal form, mainly because rows containing data are commonly in strong joint relationships. The fact that the queries drive the Solr schema design means that all the data needed to match a document must be present in the document to be matched, not in the related document. To satisfy that requirement, data that would otherwise exist in one place is copied into related documents that need it to support a search. (Smiley & Pugh, 2011) Solr is fully capable of supporting not just the design based on some well-known schema but it is also capable of supporting schema-less design. This can be used as advantage compared to traditional RDBMS systems because they do not support schema-less design very well in order to keep their tables in normalized forms.

4.4 Solr schema

Field types and fields of those types are defined in the file called `schema.xml`. This file is important since it stores index metadata. This file belongs to the configuration directory for a particular Solr instance. By default `schema.xml` contains basic field type definitions, field definitions and unique key element definition.

4.4.1 Field types

Solr schema is divided into sections and the first section contains definitions of field types. In other words, the field type defines how Solr should interpret data in a field and how the field should be indexed and queried.

Solr provides number of field types included in its distribution by default; however, it is also possible to define its own field types. Most of them are text related fields; however there are also numeric, date and other specific field types. (See Table 1.) It is important to mention that the numeric field types are sorted by their natural numeric order instead of lexicographical order.

An interesting group of field types provided by Solr are so called trie types. Trie types are based on special tree data structure that enables effective searches with the usage of common prefixes of indexed values. An important property of trie field types is precisionStep property. This property is used to tune indexing and searching behaviors. If the value of precisionStep is higher than zero, Solr splits the value to be indexed into multiple values and index them accordingly. The number of values that are created from the original field with particular precisionStep depends on the field type implementation class. Lower precision step value but higher than zero leads to a bigger index and slightly slower indexing process compared to higher precision step value; however, the major advantage are faster range queries. This can be useful especially with TrieDateField because range queries on this field are usually very slow if some rounding strategy is not used.

Table 1. Solr basic field types (Apache Solr Wiki, 2015)

Class	**Description**
BinaryField	Field that stores binary data.
BoolField	Boolean field type (true/false).
CollationField	Field that supports Unicode collation. It is used for sorting and range queries.
CurrencyField	Field that supports different exchange rates and currencies.
StrField	Unicode or UTF-8 encoded String value.
TextField	Text field that usually contains multiple words.
TrieDateField	Precise date field supporting fast range queries and sorting.
TrieDoubleField	Double number field type with 64-bit floating point precision.
TrieFloatField	Float number field type with 32-bit floating point precision.
TrieIntField	Signed integer with 32-bit precision.
TrieLongField	Long field type with 64-bit precision.
UUIDField	Universally Unique Identifier.
PointType	Filed type that represents n-dimensional point.
LatLonType	Represents coordinate in format of latitude and longitude.
ExternalFileField	This field type uses values from file on disk instead of index.

Field type definition contains the following:

- mandatory name,
- mandatory implementation class,
- optional description for field type analysis,
- mandatory or optional properties depending on the implementation class, these properties can be also overridden at the field declaration, (See Table 2.)
- text analysis configuration.

Table 2. General properties of field type (Apache Solr Wiki, 2015)

Name	Description
name	Defines unique name of field type.
class	Implementation class of the field type.
positionIncrementGap	Defines gap between values of multivalued field types.
autoGeneratePhraseQueries	If enabled Solr will automatically generate special phrase queries.

Part of the `schema.xml` is shown below. It contains sample field type definition with name, class and possitionIncrementGap properties along with text analysis components such as filters and tokenizers.

```
<fieldType name="text_general" class="solr.TextField"
           positionIncrementGap="100">
     <analyzer type="index">
          <tokenizer class="solr.StandardTokenizerFactory"/>
          <filter class="solr.StopFilterFactory" ignoreCase="true"
                 words="stopwords.txt" />
     </analyzer>
     <analyzer type="query">
          <tokenizer class="solr.StandardTokenizerFactory"/>
          <filter class="solr.StopFilterFactory"
                 ignoreCase="true" words="stopwords.txt" />
          <filter class="solr.SynonymFilterFactory"
                 synonyms="synonyms.txt"
                 ignoreCase="true" expand="true"/>
          <filter class="solr.LowerCaseFilterFactory"/>
     </analyzer>
</fieldType>
```

4.4.2 Defining fields

Fields are carriers of information for a particular part of the document. Every document in Solr consists of fields. There are various types of fields in Solr, each designated for a specific purpose. There are normal fields that are very similar to RDBMS columns. There are special dynamic fields that are very important for schema-less designs. There are also copy fields used in use cases when modified values of some field need to be saved without actually modifying the original field.

Fields are defined similarly as field types in a document called `schema.xml` and they are placed between the `<fields> </fields>` tags. Field definitions contain a set of properties that can be divided into basic properties and optional properties. (See Table 3.)

Table 3. Basic field properties (Apache Solr Wiki, 2015)

Property	Description
name	Field name. Java naming conventions is recommended for field names.
type	Field type name used for specific field.
default	Default value for the field when the value is missing in indexed document.

Optional properties can be defined in both field definition and field type definition. In the case of conflicting property definition, field definition of property takes precedence over the field type definition of property. It is more convenient and safer to have optional field properties defined in field instead of field type. Multiple fields with same field type but with different properties can be specified. (See Table. 4)

Table 4. Optional field properties (Apache Solr Wiki, 2015)

Property	Description	Values
indexed	If true, indexes the value of the field.	true or false
stored	If true, stores the value of the field.	true or false
docValues	If true, field values is stored in special columns oriented structure.	true or false
sortMissingFirst sortMissingLast	Defines the sorting behavior when the field value is missing.	true or false

multiValued	If true, field can contain multiple values.	true or false
omitNorms	If true, omits the norms associated with this field.	true or false
omitTermFreqAndPositions	If true, omits term frequency, positions and payloads from postings for this field and reduces the actual index size.	true or false
required	If true, Solr will not index the documents that are missing this field.	true or false

A part of the `schema.xml` is shown below. It contains sample fields definitions with basic and optional properties.

```
<fields>
     <field name="firstName" type="string" indexed="true"
            stored="true" required="true"/>
     <field name="lastName" type="string" indexed="true"
            stored="true" required="true"/>
     <field name="dateOfBirth" type="tdate" indexed="true"
            stored="true" required="true"/>
</fields>
```

4.4.3 Dynamic fields

Dynamic fields allow to dynamically name fields of indexed document before the very process of indexing starts. If at index time a document contains a field that is not matched by an explicit field definition; however, it has a name matching defined pattern, then it gets processed according to that definition. Dynamic field is declared just like a regular field in the same section. However, the element is named dynamicField instead of field and it has a name attribute that must either start or end with an asterisk. (Smiley & Pugh, 2011)

Dynamic field definition that creates new fields for existing fields which names end with "_d" characters is presented as follows:-

```
<dynamicField name="*_d" type="double" indexed="true" stored="true"/>
```

4.4.4 Copy fields

Copy fields are another type of fields that can be defined in schema. They are useful when a value needs to be copied to additional fields and to be indexed differently.

Copy fields have special properties that specify a source field that has to be used for copying from, destination which is the name of the field that is to be copied to and finally, maxChars which defines the maximum number of characters that should be copied from the source.

At index time, each field of indexed document has its name compared against the source attribute of all copy fields defined in schema. The source attribute can include an asterisk wildcard. It is possible that the input might match more than one copy field. If a wildcard is used in the destination, then it must refer to a dynamic field and furthermore the source must include a wildcard, too, otherwise a wildcard in the destination is an error. If there is a match found for a copy field, value of matched field is duplicated to the field which name is defined in destination attribute. (Smiley & Pugh, 2011)

Copy field definition that creates a new field named fullName from existing fields named, for example, firstName and lastName is presented as follows:-

```
<copyField source="*Name" dest="fullName" maxChars="25000" />
```

4.5 Text analysis

An important role of search platforms such as Apache Solr is to provide fast and good search results. The core of the text analysis functionality of Apache Solr is part of the Apache Lucene library. Text analysis is a tool that affects indexing of documents and not only that. Text analysis is also applied during the document querying phase. Text analysis can be defined as a sequence of text-processing steps, for example:

- synonym matching
- case normalization
- tokenization
- text stemming

Text analysis is in charge of converting text from a particular field into sequence of separate terms. The term can be understood as the fundamental unit that is actually stored and searched upon in the Lucene index. During the indexing phase the original value for the indexed field is converted into a sequence of terms and these terms are indexed into a storage medium in special Lucene index structure. During the querying phase the terms provided in the query string are also converted into sequence of terms and analyzed before the actual searching in the index structure happens. (Smiley & Pugh, 2011)

Apache Solr provides three basic text analysis components:

- field analyzers
- tokenizers
- filters

It is important to mention that text analysis components are attached to particular field types but not all the field types support text analysis. The field types are defined in the `schema.xml` configuration file and so are the text analysis components. The most important field type that supports text analysis is `solr.TextField`. It is also possible to define custom text analysis components that can extend the functionality of existing components or they can add completely new text analysis functionality on top of the existing components. Correct configuration of text analysis can dramatically affect the indexing time and time needed for query to return relevant response.

4.5.1 Analyzers

An analyzer examines the text of fields and generates a token stream. Analyzers are specified as a child of the `<fieldType>` element. The easiest way to configure analyzer is with a single `<analyzer>` element whose class attribute is a fully qualified Java class name. Class that is used in the `<analyzer>` element must be derived from the `org.apache.lucene.analysys.Analyzer` class. (Apache Solr Wiki, 2015) There is a possibility to configure more complex analyzers by chaining the other components inside of the `<analyzer>` element. The analyzer can also specify the place of the configured text analysis. There is an optional type attribute that can hold value of either index or query. If the type attribute is not specified an analyzer is applied in both stages.

Simple analyzer configuration that uses standard WhiteSpaceAnalyzer is described as follows:-

```
<fieldType name="textField" class="solr.TextField">
    <analyzer class="org.apache.lucene.analysis.WhitespaceAnalyzer"/>
</fieldType>
```

A more complex analyzer that chains multiple analysis components is shown below:-

```
<fieldType name="textField" class="solr.TextField">
     <analyzer>
          <tokenizer class="solr.StandardTokenizerFactory"/>
          <filter class="solr.StandardFilterFactory"/>
          <filter class="solr.LowerCaseFilterFactory"/>
          <filter class="solr.EnglishPorterFilterFactory"/>
     </analyzer>
</fieldType>
```

4.5.2 Tokenizers

A tokenizer is an analysis component declared with the `<tokenizer>` element that takes text in the form of a character stream and splits it into so called tokens, most of the time skipping less significant parts of the text such as whitespaces. This means that tokenizer is not directly aware of the analyzed field. An analyzer should have exactly one tokenizer configured. (Smiley & Pugh, 2011)

Standard tokenizer is to be configured inside analyzer component.

```
<analyzer>
     <tokenizer class="solr.StandardTokenizerFactory"/>
</analyzer>
```

4.5.3 Filters

Filters are very similar to tokenizers in a way that they process input and produces stream of tokens. Difference between the tokenizers and filers is that the filters take as input stream of tokens produced by either the tokenizer or another filter. Filters can replace, pass or discard the currently processed token from stream. There is also possibility of analyzing multiple tokens at once. (Apache Solr Wiki, 2015)

The chained process of text analysis using one tokenizer and multiple filters is illustrated as follows:-

```
<analyzer>
     <tokenizer class="solr.StandardTokenizerFactory"/>
     <filter class="solr.StandardFilterFactory"/>
     <filter class="solr.LowerCaseFilterFactory"/>
     <filter class="solr.EnglishPorterFilterFactory"/>
</analyzer>
```

4.6 Indexing and other operations

4.6.1 Communication

Applications interact with Solr over HTTP protocol. This can either be done directly using any HTTP client API or indirectly via Solr integration API such as SolrJ or Sunspot. This HTTP interaction can evoke that the data that needs to be indexed to

Solr must be transferred to Solr over HTTP; however, this is not necessarily true. (Smiley & Pugh, 2011) There are multiple ways how to index data from various sources such as file system, network drives, RDBMS systems and others.

4.6.2 Supported formats

Solr provides indexing support for various file formats such as:

- Solr’s Update-XML
- Solr’s Update-JSON
- Java-Bin
- CSV
- rich documents

Solr’s Update-XML along with Solr’s Update-JSON are two specific formats derived from simple XML and JSON. They provide extended support for specific needs of files to be indexed to Solr. Java-Bin is an efficient binary variation of Solr’s Update-XML format that is officially supported only by SolrJ client. Rich documents represent commonly used formats such as PDF, XLS, DOC, PPT. (Smiley & Pugh, 2011)

4.6.3 Indexing commands

Documents that have been sent to Solr are not immediately visible in the index. This behavior depends on the commit strategy. It is very similar to traditional RDBMS systems where there is also a similar command called commit. Commit provides functionality to persist data to the actual storage. There is an easy way to issue this commit command in Solr. If sending HTTP request commit = true parameter can be added to the URL parameter or commit command can also be included in the indexed document itself. After the successful commit it is possible to search upon the committed documents.

There are several important issues to mention regarding the Solr’s commit command. Commit operation is a slow and resource hungry operation in Apache Solr. Commit speed depends on index size, auto warming configuration and also on actual cache state. Probably the major difference between Solr’s commit and the commit known from traditional RDBMS systems is lack of transactional isolation in Solr. This means that if there are more clients making changes to index simultaneously they can easily commit each other's changes without knowing that the commit occurred. This behavior also applies to rollback operation that is the opposite of commit operation. If there are multiple clients modifying the index at the same time and one of them issues the rollback command it is possible that the changes of another client that were not committed will be discarded. One solution how to prevent such a behavior is to set up one process or one client that will be in charge of low level index changes. Another problem that can occur while there are simultaneous commits occurring is closely connected to query warming which is part of the commit process. Too many simultaneous query warming processes can overhaul the system and it can raise errors; however, the index changes will take effect despite the errors. (Smiley & Pugh, 2011) Solr provides users with two basic commit strategies: autoCommit and commitWithin. These options can be configured in the `solrconfig.xml` file.

Optimize command is something unique for Solr and in particular to Lucene. When a buffer of indexed documents gets flushed to disk it creates a new segment. Lucene merges those segments according to their sizes and other parameters. When there is just one segment Lucene is in optimized state. The more segments there are the more the query performance degrades. (Smiley & Pugh, 2011) Optimize command can be understood similarly as a disk defragmentation. Optimize command is resource hungry and also considerably slow. It should be run every once in a while, especially after huge loads.

Rollback command has the same functionality as the RDBMS rollback command. It discards all the changes made prior to previous commit. Rollback command can be sent to Solr in the same way as commit command.

4.7 Querying

Querying is one of the basic functionalities provided by Apache Solr. It is necessary to understand querying mechanism in Solr because the queries provide major source of information during the process of schema definition. When the data is successfully indexed it is available to searchers. Multiple components are involved in the querying process. (See Figure 6.)

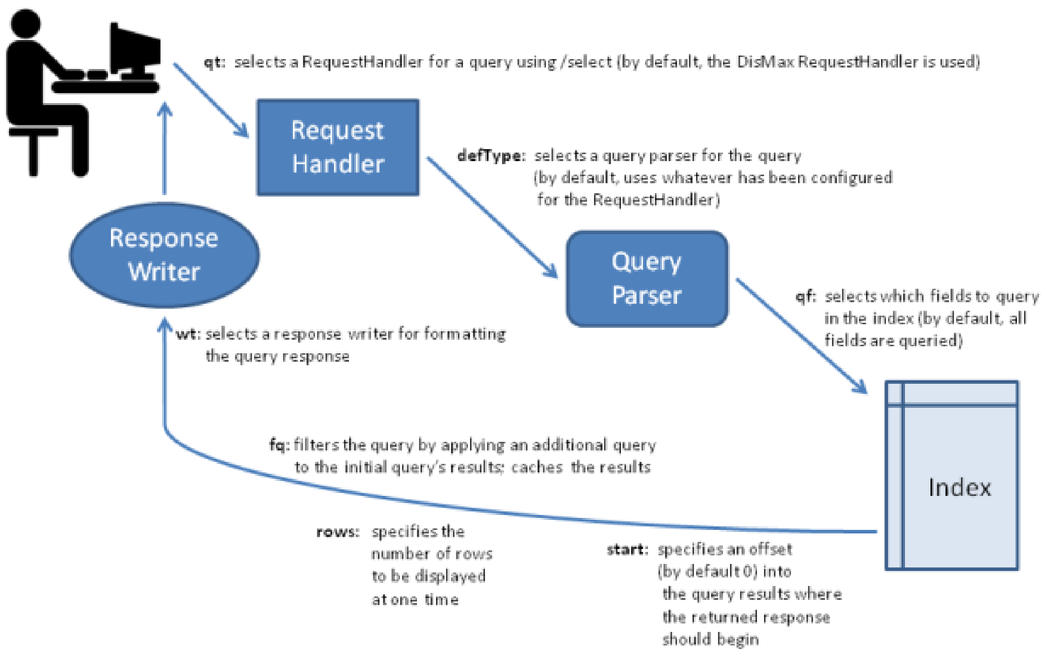
qt: selects a RequestHandler for a query using /select (by default, the DisMax RequestHandler is used)
Request Handler
defType: selects a query parser for the query (by default, uses whatever has been configured for the RequestHandler)
Query Parser
qf: selects which fields to query in the index (by default, all fields are queried)
Index
Response Writer
wt: selects a response writer for formatting the query response
fq: filters the query by applying an additional query to the initial query's results; caches the results
rows: specifies the number of rows to be displayed at one time
start: specifies an offset (by default 0) into the query results where the returned response should begin

Figure 6. Query processing (Apache Solr Wiki, 2015)

The process of querying starts when the specific request handler is invoked. In the next step the request handler calls a query parser. Query parser is component, which is responsible for interpretation of query parameters and terms. There are multiple query parser, which support different syntax. Solr includes an earlier standard Lucene parser, DisMax query parser and Extended DisMax query parser. (Apache Solr Wiki, 2015)

Query parser input consists of following parts:

- search string
- parameters affecting the index search (See Table 5.)
- parameters controlling the presentation of response (See Table 5.)

Table 5. Basic query parameters (Lucene Apache Jira, 2015)

Parameter	**Description**
defType	Defines a specific query parser.
sort	Defines the ascending or descending order of documents returned in response.
start	Defines starting offset of documents matched by the query.
rows	Defines number of documents returned by the query.

fq	Defines a specific query filter.
fl	Defines list a specific list of fields returned in response.
wt	Defines a specific format of the response (CSV, JSON, XML).

4.8 Request handlers

Request that is coming to Solr is processed by component called request handler. It is possible to have multiple request handlers configured. There are multiple types of requests that Solr can process, for example: query for data, index updates, status query, etc. It is convenient to have separate request handler defined for each request type. Request handler is configured with specific name and implementation class. Solr provides basic implementation classes; however users can define custom classes, which extend the functionality. Configured handler name is used by request for determining, which specific request handler has to process the request. Request handler contains definition of parameters, which are used with queries that are processed. These parameters can be overridden by client, which is sending the request. Each request handler can be configured with multiple types of parameters such as: defaults, invariants, appends. (Apache Solr Wiki, 2015)

Parameters defined in defaults section provide default parameters that are used by handler when there are no parameters provided in client request. Parameters defined as appends are added to each of the client's query which is processed by specific handler. Appends parameters cannot be overridden by client's query. Parameters defined as invariants cannot be overridden by a client and they are used with every request. (Apache Solr Wiki, 2015)

Search handler configuration containing default parameters definition is shown below:-

```
<requestHandler name="/select" class="solr.SearchHandler">
     <lst name="defaults">
          <str name="fl">name</str>
     </lst>
     <lst name="invariants">
          <str name="rows">10</str>
     </lst>
</requestHandler>
```

It is also possible to define request handlers with the common parameters. An `<initParams>` section provides named parameters configuration that can be used by multiple request handlers.

Common parameters definition named sampleParams, which is used by defaulthandler and myhandler.

```
<initParams name="sampleParams" path="/defaulthandler,/myhandler">
     <lst name="defaults">
          <str name="fl">name</str>
     </lst>
     <lst name="invariants">
          <str name="rows">10</str>
     </lst>
     <lst name="appends">
          <str name="fq">loyalCustomer:true</str>
     </lst>
</initParams>
```

4.9 Data import handler

Apache Solr includes various contribution modules and data import handler (DIH) is one of them. DIH module is used for importing data from various sources. It is data processing pipeline built specifically for Solr. Following list contains data import handler capabilities (Smiley & Pugh, 2011):

- supports data import from database systems through JDBC
- supports data import from URL
- supports data import from files
- supports emails import from an IMAP server, including attachments
- supports combining data from different sources
- extracts text and metadata from rich document formats
- supports XSLT transformations and XPath extraction on XML data
- supports scheduled full imports and delta imports
- includes a diagnostic and development tool

Sample data import handler configuration is show below:-

```
<requestHandler name="/dataimport"
class="org.apache.solr.handler.dataimport.DataImportHandler">
     <lst name="defaults">
          <str name="config">dataimport-config.xml</str>
     </lst>
</requestHandler>
```

Data import handler is configured as request handler and it has to contain path to the configuration file under the defaults parameters.

4.9.1 DIH configuration file

Data import handler configuration file contains specific details bounded to importing process. DIH configuration file contains various components such as:

- data sources
- entity processors
- transformers
- fields

Data sources are used to define source of data that can be referenced in entities. Data source is configured in `<datasource />` tag. There are several data sources provided by Solr. (See Table 6.) It is also possible to create custom data source. Type attribute of data source contains provided data source class or fully qualified java path to the custom data source class.

Table 6. Data import handler data sources (Smiley & Pugh, 2011)

Name	**Description**
JdbcDataSource	Database source that uses JDBC.
FieldStreamDataSource, FieldReaderDataSource	Used for extracting binary and character data from columns from JdbcDataSource.
BinFileDataSource, FileDataSource	Binary or text file sources.
URLDatasource	Specify path to the text source either in disk or remote location.
BinContentStreamDataSource, ContentStreamDataSource	Binary or text data that is directly posted to DIH.

After the data source definition in configuration file the `<document>` element is specified. This element represents document to be indexed and it contains entity processors. Entity processor is defined in the `<entity />` element with type attribute where the specific processor is set. Entity processor is responsible for producing the documents depending on the processor type, parent entity processor and data source used. There must be one entity processor defined as root entity. There can be multiple entities defined in one document element but only the documents produced by root entity are indexed to Solr. (Smiley & Pugh, 2011) Chaining of entity processors is useful because it provides easy and convenient way how to import complex documents, especially documents with multivalued fields that contain separate documents. Entity processors provided by Solr can be seen in the table below. (See Table 7.) It is also possible to define custom entity processors.

Table 7. Data import handler basic entity processors (Smiley & Pugh, 2011)

Name	**Description**
SqlEntityProcessor	References JdbcDataSource and executes specific SQL query defined in the query attribute.
CachedSqlEntityProcessor	Same as SqlEntityProcessor but this processor caches the results of the SQL query defined in the query attribute.
XPathEntityProcessor	Processes XML from data source. It separates the XML into separate documents according to the XPath expression. XPath expression is also used to reference the fields.
PlainTextEntityProcessor	Processes the text from data source as a single field value.
LineEntityProcessor	Processes text from data source line by line.
FileListEntityProcessor	Processes all files matching the specific criteria. Each file represents one document.
TikaEntityProcessor	Processes text from a binary data source, using the Apache Tika. Tika provides support for file types such as: HTML, PDF and Microsoft Office documents.
MailEntityProcessor	Processes emails with attachments from IMAP mail server.

An `<field />` element is defined within the `<entity />` element. This element represents how the specific fields are mapped from entity processor to Solr. The field element contains column attribute and name attribute. Column attribute is used to retrieve value for specified field from entity processor and name attribute is used as mapping to specific field defined in schema.

Entity can contain additional transformer attribute. This attribute is used to specify one or more transformers that are used in process of importing. Transformers are used to modify fields of processed documents. Following list of transformers is provided by default: (See Table 8.)

Table 8. Data import handler transformers (Smiley & Pugh, 2011)

Name	Description
TemplateTransformer	Modifies field value based on string template. The template can contain references to other fields and data import handler variables.
RegexTransformer	Splits field value into separately named fields or values. It is also capable of string substitution based on provided regex expression.
DateFormatTransformer	Transforms the date field to a Solr’s date format.
NumberFormatTransformer	Transforms the field according to defined locale and number format.
HTMLStripTransformer	Removes all the HTML tags from the text.
ClobTransformer	Transforms CLOB value into plain string value.
ScriptTransformer	Supports user defined transformer code in any of the JVM compatible languages.

Used components are entirely dependent on importing process. There is no common universal configuration for the importing process. In many cases the importing process can be configured more than once with different configurations, which lead to the same result. It is important to know how to configure DIH properly because it can have a serious performance impact on Solr.

4.9.2 Commands

Data import handler provides support for multiple commands. (See Table 9.)

Table 9. Data import handler commands

Command name	Description
full-import	New thread is started and the importer status is set to busy. This thread executes the importing process. This process is not blocking client queries to Solr.
delta-import	Incremental imports with support to document changes detection.
status	Status of data import handler with importing statistics.
reload-config	Reloads the data import handler configuration.
abort	Aborts the currently running operation.

Specific command can be executed by sending the HTTP GET request with path to the configured data import request handler and the command parameter with the name of the command to execute. Sample GET request that triggers full import process is illustrated below:-

```
http://<host>:<port>/solr/corename/dataimport?command=full-import
```

4.10 Basic configuration and management

Solr indices data to cores, which are defined mainly in two configuration files. Structure of the documents to be stored in the specific core is defined in the `schema.xml`. Various properties and components including the request handlers are defined in the `solrconfig.xml` file for each core. There is also configuration file for the whole Solr instance called solr.xml. Solr instance can contain single core or multiple cores and solr.xml is the configuration file that contains definitions of the cores present in the specified Solr instance. Sample solr.xml configuration that contains multiple cores is shown below:-

```
<solr persistent="true" sharedLib="lib">
     <cores adminPath="/admin/cores">
          <core name="core0" instanceDir="core0">
               <property name="dataDir" value="/data/core0" />
          </core>
          <core name="core1" instanceDir="core1" />
          <core name="core2" instanceDir="core2" />
     </cores>
</solr>
```

Solr provides several core administration commands, which are listed in the table below. (See Table. 10)

Table 10. Core administration commands (Smiley & Pugh, 2011)

Name	Description
STATUS	Gets the current status of the cores. Provides basic statistics and other useful information about the cores.
CREATE	This option is used to create new based on some existing core.
RENAME	Renames the existing core.
SWAP	Swaps the defined cores. Swap command is executed as atomic operation. This ensures that the clients cannot receive mixed documents from both cores.
RELOAD	Reloads specified core without the need to restart whole Solr server. This command is useful when some changes were made to core configuration files.
UNLOAD	Unloads specified core from Solr. Currently running queries are finished; however, no new queries can be run on unloaded core.
MERGEINDEXES	Merges two or more cores into another core.

Specific admin command can be executed by sending the HTTP GET request with path to the configured admin request handler. Sample GET request that returns status of the cores is show below:-

```
http://<host>:<port>/solr/admin/cores?action=STATUS
```

4.11 Caching options

One of the key components of Apache Solr is caching, which can be used for scalability and performance improvements. Caches must be properly configured because there is a risk of wasting resources like RAM and others when the cache is not properly configured. There are several types of caches used in Solr; however, the Least Recently Used in-memory cache is implementation that is used in most cases. For cache types see table below. (See Table 11.) Index searchers are components that are directly responsible for retrieving the data from the index. Caching is specifically used to store subset of data in memory and thus reducing the need of pulling the data from the disk when the query is send to Solr. Every commit operation results in new index searchers to be opened and auto warmed. Auto warming is process of populating the new searcher with the data from queries, which ran in the previously opened searcher. Auto warming queries can be configured in `solrconfig.xml` file. Only when the the auto warming process finishes the new index searcher is used to process requests for data. (Smiley & Pugh, 2011)

Table 11. Solr caches (Smiley & Pugh, 2011)

Name	**Description**
filterCache	This cache stores unordered lists of documents that match certain queries.
queryResultCache	This cache stores ordered lists of document ids. Order of ids in lists is dependent on the passed sort parameter.
documentCache	This cache stores field values for the fields that were defined in the `schema.xml` as stored.

Every cache is configured with a set of the following parameters:

- class
- size
- initialSize
- autowarmCount
- minSize
- acceptableSize
- cleanupThread
- timeDecay
- showItems

Class parameter defines the cache implementation class. Supported implementation classes are LRUCache, FastLRUCache and LFUCache. Size parameter defines number of items to be stored in cache. The higher the value the more RAM is allocated for the cache. Initial size parameter defines initial capacity of the cache. Autowarm count parameter defines number of items that should be copied from old searcher to newly created searcher during the auto warming process. Min size is optional parameter defined only for FastLRUCache and it defines the size that should be maintained by the cache when it hits the size limit. Acceptable size is similar to min size parameter. It is only defined for the FastLRUCache and it defines the size of the cache when the cache cannot regulate its size to min size after it hits the size limit. Cleanup thread parameter can be set only for a FastLRUCache and setting it to true means that the dedicated thread will be created for the removal of the old entries from the cache. Time decay parameter is used for lowering the entries hits when the cache searches size limit. Show parameter is usable only for debugging purposes with the LRUCache and the FastLRUCache. (Apache Solr Wiki, 2015)

4.12 Replication

Replication is the process of copying the data from the master index to one or more slave indices. Common replication pattern in Solr can be described as follows:

“Master index is updated with new data and all queries for data are handled by the slave indices. This division enables Solr to scale and provide adequate responsiveness to queries against large search volumes.” (Apache Solr Wiki, 2015)

Following figure illustrates a simple replication process for one master index and three slave indices. (See Figure 7.)

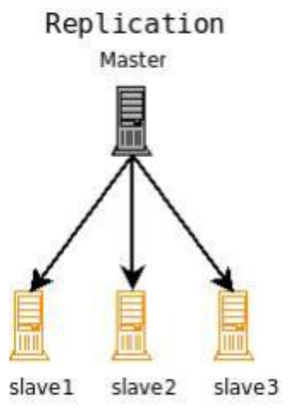

Figure 7. Solr replication

Solr index replication properties:

- entire replication is configured in `solrconfig.xml`
- replication of index data and configuration files

- OS file system independent
- replication is implemented as a special request handler

Sample replication request handler configuration for master index is show below:-

```
<requestHandler name="/replication" class="solr.ReplicationHandler" >
     <lst name="master">
          <str name="replicateAfter">optimize</str>
          <str name="backupAfter">optimize</str>
          <str name="confFiles">
               schema.xml,stopwords.txt,elevate.xml</str>
          <str name="commitReserveDuration">00:00:10</str>
     </lst>
     <int name="maxNumberOfBackups">2</int>
     <lst name="invariants">
          <str name="maxWriteMBPerSec">16</str>
     </lst>
</requestHandler>
```

Sample replication request handler configuration for slave index is shown below:-

```
<requestHandler name="/replication" class="solr.ReplicationHandler" >
     <lst name="slave">
          <str
name="masterUrl">http://host:port/solr/core/replication</str>
          <str name="pollInterval">00:00:20</str>
          <str name="compression">internal</str>
          <str name="httpConnTimeout">5000</str>
          <str name="httpReadTimeout">10000</str>
          <str name="httpBasicAuthUser">username</str>
          <str name="httpBasicAuthPassword">password</str>
     </lst>
</requestHandler>
```

Following figure illustrates a replication process in the WebSphereCommerce platform. (See Figure 8.) WebSphereCommerce uses the common pattern when the master index is updated with new data and slave indices are used for handling the queries from clients.

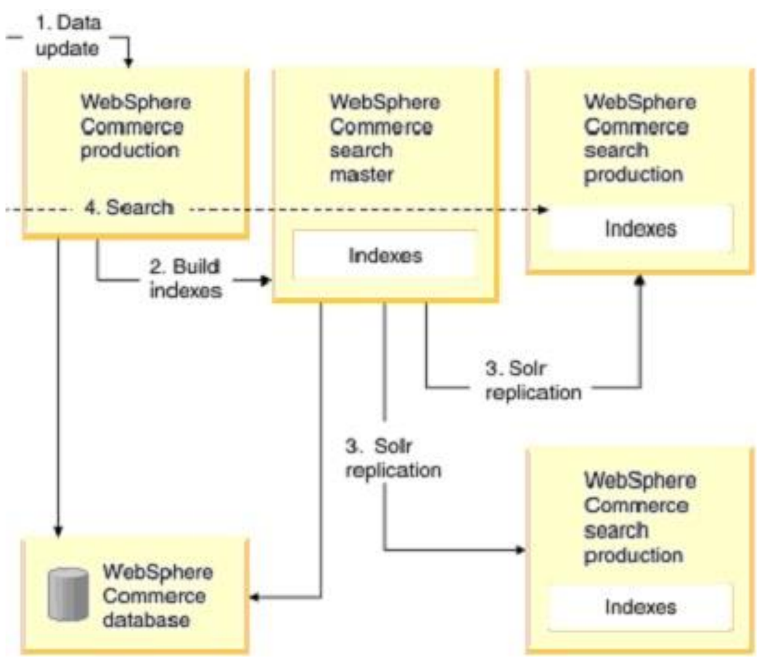
1. Data update
WebSphere Commerce production
4. Search
2. Build indexes
WebSphere Commerce search master
Indexes
WebSphere Commerce search production
Indexes
3. Solr replication
3. Solr replication
WebSphere Commerce search production
Indexes
WebSphere Commerce database

Figure 8. Solr replication in WebSphereCommerce

4.13 SolrCloud

SolrCloud is a feature of Apache Solr, which provides distributed indexing and searching. With the introduction of SolrCloud many of the manual steps, which were required are not automated. SolrCloud provides leader to replica based approach instead of traditional maste to slave approach. Apache Zookeeper is a key technology behind the SolrCloud. (Scaling Apache Solr, 2015) SolrClound is designed to overcome the problem when the amount of data to be stored and processed is too much for a single Solr server. (Kuc, 2013)

4.13.1 Sharding

Sharding is one of the key concepts in SolrCloud. Sharding process is basically process of splitting the index into multiple indices. It can be understood as horizontal partitioning of data. This process is a common database scaling strategy when the single database instance is not capable of handling the load. Solr sharding can be specifically defined as breaking up a single Solr core into multiple cores on multiple servers, each with identical schema. Apache Solr provides aggregation capabilities, which mean when the query is running across multiple shards multiple sets of results are returned and these results must be aggregated into single result set and this result set must be returned to the client that issued the particular query. Sharding is recommended when the queries start to take too long to process on a single machine. Sharding is typically needed when the index contains millions of documents and there are complex queries upon large number of documents that take too much resources as CPU and memory. (Smiley & Pugh, 2011)

5 Specific scenario

5.1 Product pricing

E-commerce is a high-paced, competitive and constantly changing environment. Merchants compete with each other for customers. Multiple factors have influence on this kind of competition; however, one of the key factors is dynamic product pricing.

Specific product pricing scenario that was used as a foundation for this project can be described as follows. There is a company that sells various goods in the multiple states. This company owns nearly one hundred local stores. All the stores have their own assortment inventory and specific product pricing depending on the location (country, town), tax regulations, special campaigns, dates and many more. According to these specifications there was introduced the concept of price rule. Price rule describes the pricing conditions for the specific product. Currently there are approximately forty millions of these price rules managed in the system. The current way of managing these price rules was sufficient enough until the company decided to add another one hundred stores into the system. Expected raise in the price rules was to be from forty millions to the approximately one hundred and fifty millions of price rules. The deed for the changes emerged from this expected raise.

5.2 Schema definition

Price rule can be described by structured document with fields containing specific pricing information. (See Table 12.) Most of the fields are of textual format but there are date fields that can take advantage of trie date type with certain precision set. Trie date type is especially useful when there is a need to run the range queries on date fields.

Table 12. Price rule fields

Name	**Description**
ext_documentid	Key field created from other fields.
KOTABNR	SAP table number.
KSCHL	SAP condition.

ext_lookupseq	SAP condition lookup sequence number.
VKORG	Seller organization (country).
VTWEG	Delivery channel.
WERKS	Local store code.
KNUMH	Condition identifier.
DATAB	Price rule start date.
DATABI	Price rule end date.
MATNR	Stock keeping unit part number.
ext_hier_ht2	Subcategory hierarchy identifier.
ext_hier_ht1	Subcategory hierarchy identifier.
MATKL	Master category identifier.
KONDA	Customer group identifier.
ext_customer	Ordering party identifier.
KBETR	Price or discount type.
KONWA	Currency or % symbol.
KPEIN	Pricing amount.
KMEIN	Pricing unit.
KNUMA_AG	Discount table number.
AKTNR	Campaign number.

The schema contains two field types. The `string` field type uses `solr.StrField` implementation class, which provides basic functionality for indexing simple strings. There are no other components such as tokenizers or analyzers defined for this field type because they are not necessary. Queries that are made for price rules documents are so called exact match queries and the relevancy search is not used in this case so the text analysis components are not used during the indexing process. This field type contains set of additional properties such as: `omitNorms`, `termVectors`, `termPossitions`, `termOffsets`. These properties are additional properties useful for faster and better relevancy searches but they are not useful in the exact match searches. Second field type defined in the schema is special `solr.TrieDateType`. This field type supports faster range queries when the precistionStep value is well configured; however, lower the precisionStep value more index space is taken by fields defined with this type. In this case the faster queries are more important than size of an index. Part of the price rule document schema that contains field type definitions is shown below:-

```
<types>
   <fieldType name="string" class="solr.StrField"
              sortMissingLast="true" omitNorms="true"
              termVectors="false" termPositions="false"
              termOffsets="false"/>
   <fieldType name="date" class="solr.TrieDateField"
              omitNorms="true" precisionStep="6"
              positionIncrementGap="0"/>
</types>
```

All the fields defined in the schema are indexed, stored and they are not multivalued. The ext_documentid is required because it is document's key field. Part of the price rule document schema that contains field definitions is shown below:-

```
<fields>
<field name="ext_documentid" type="string" indexed="true" stored="true"
       required="true" multiValued="false"/>
<field name="KOTABNR" type="string" indexed="true" stored="true" multiValued="false"/>
<field name="KSCHL" type="string" indexed="true" stored="true" multiValued="false"/>
<field name="ext_lookupseq" type="string" indexed="true" stored="true"
       multiValued="false" />
<field name="VKORG" type="string" indexed="true" stored="true" multiValued="false"/>
<field name="VTWEG" type="string" indexed="true" stored="true" multiValued="false"/>
<field name="WERKS" type="string" indexed="true" stored="true" multiValued="false"/>
<field name="KNUMH" type="string" indexed="true" stored="true" multiValued="false"/>
<field name="DATAB" type="date" indexed="true" stored="true" multiValued="false"/>
<field name="DATBI" type="date" indexed="true" stored="true" multiValued="false"/>
<field name="MATNR" type="string" indexed="true" stored="true" multiValued="false"/>
<field name="ext_hier_ht2" type="string" indexed="true" stored="true"
       multiValued="false" />
<field name="ext_hier_ht1" type="string" indexed="true" stored="true"
       multiValued="false" />
<field name="MATKL" type="string" indexed="true" stored="true" multiValued="false" />
<field name="KONDA" type="string" indexed="true" stored="true" multiValued="false"/>
<field name="ext_customer" type="string" indexed="true" stored="true"
       multiValued="false" />
<field name="KBETR" type="string" indexed="true" stored="true" multiValued="false"/>
<field name="KONWA" type="string" indexed="true" stored="true" multiValued="false"/>
<field name="KPEIN" type="string" indexed="true" stored="true" multiValued="false"/>
<field name="KMEIN" type="string" indexed="true" stored="true" multiValued="false"/>
<field name="KNUMA_AG" type="string" indexed="true" stored="true"
       multiValued="false"/>
<field name="AKTNR" type="string" indexed="true" stored="true" multiValued="false"/>
</fields>
```

6 Solr extensions

Apache Solr provides a large set of components and great functionality; however, sometimes it is not enough to cover all the needs of the modern eCommerce applications. On the other hand Apache Solr provides excellent API that enables developers to create various extensions and customizations, which are fully compatible with existing components.

6.1 Export handler

Export request handler is custom Solr extension, which adds export to file functionality to Solr. Solr is capable of exporting the data in various formats; however, response containing formatted data cannot be directly exported to the files in the file system. Clients communicate and request data from Solr through the HTTP requests, which are processed by Solr. When the request is processed by Solr, it sends the response with appropriate data to client. This is the basic concept how clients communicate with Solr; however, when the client wants to export large number of documents response can get too big to be handled by the client. This extensions is useful in such cases when the client wants to export whole index or substantial subset of the index.

Backup process is good example when the client needs to export whole index in some specified format, which can be later used as an input to import process. Backup process is very important process in the data maintenance perspective because it is one of the methods how to prevent the data loss.

Another case when the export handler extension would be very useful is index partitioning. When there is a special need to split the index according to defined criteria the export handler can create files, which contain partition specific data and then these files can be used to build separate indices, each for the defined partition.

There are methods how to export large numbers of documents from Solr; however they can rely on third party software, which is not under control of the Solr developers and these methods can be significantly slower when there is a need to handle large numbers of documents.

6.1.1 Configuration

Every request handler that is used by Solr needs to be defined and configured in the configuration file called `solrconfig.xml`. Following part of the configuration file contains definition of custom export to file handler:-

```
<requestHandler name="/dataexport"
class="com.commerce.solr.handler.dataexport.DataExportToFileHandler">
     <lst name="defaults">
          <str name="exportFolderPath">
               B:\PriceRules\export\
          </str>
          <str name="exportFileNamePrefix">export</str>
          <str name="exportFileFormat">csv</str>
          <bool name="exportFileOverwrite">true</bool>
          <str name="batchSize">100000</str>
     </lst>
</requestHandler>
```

Custom data export to file handler definition contains name and class mandatory attributes together with defaults configuration properties. Every request handler defined in the configuration file is accessible via path that is defined in the name attribute located in the `<requestHandler>` tag. Specific path to the handler is as follows:-

`http://<host>:<port>/solr/corename/dataexport`

Export to file handler has defined number of default properties, which can be overwritten by the users. (See Table 13.) These properties are parsed and sent to the implementation class, which provides the exporting logic.

Table 13. Export to file handler attributes and properties

Name	Description
exportToFolderPath	Location of the exported files in the file system.
exportFileNamePrefix	Exported files have form of the <exportFileNamePrefix><batchNumber>.<exportFileFormat>. This naming is used to distinguish between exported files.
exportFileFormat	Format of the exported data. Supported formats are CSV, JSON and XML.
exportFileOverwrite	This attribute defines if the previously created files in the defined location should be overwritten.

batchSize	This attribute defines number of documents in one batch. Every exported file is created from one batch.

6.1.2 Implementation

Custom export to file handler is provided as one Java package, which consists of three main components: (See Figure 9.)

- DataExportToFileHandler
- DataExporter
- DataExportStats

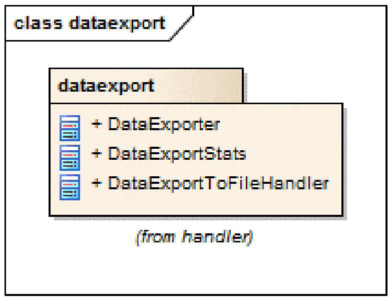

Figure 9. Export handler package structure

Class diagram shown below provides detailed view of the classes and relationships between the classes. (See Figure 10.)

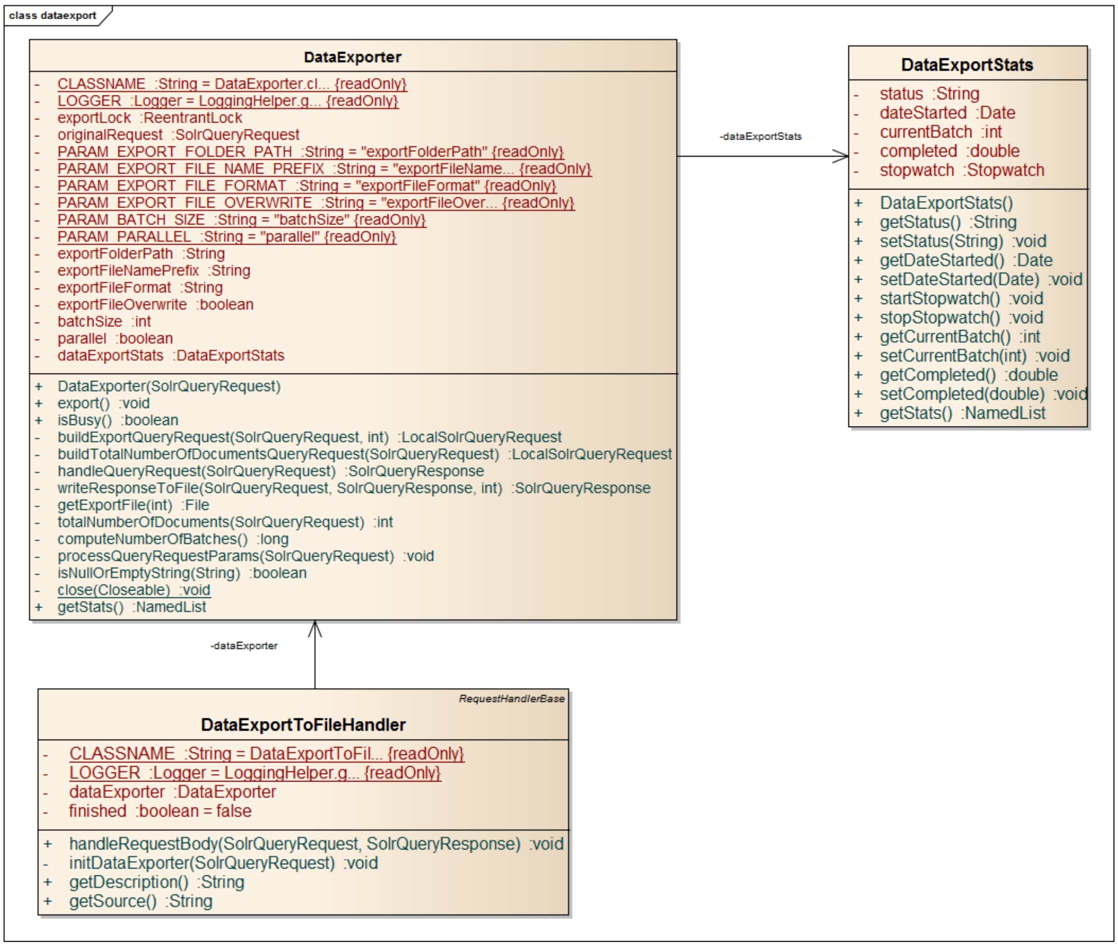
class dataexport
DataExporter
- CLASSNAME :String = DataExporter.cl... {readOnly}
- LOGGER :Logger = LoggingHelper.g... {readOnly}
- exportLock :ReentrantLock
- originalRequest :SolrQueryRequest
- PARAM_EXPORT_FOLDER_PATH :String = "exportFolderPath" {readOnly}
- PARAM_EXPORT_FILE_NAME_PREFIX :String = "exportFileName... {readOnly}
- PARAM_EXPORT_FILE_FORMAT :String = "exportFileFormat" {readOnly}
- PARAM_EXPORT_FILE_OVERWRITE :String = "exportFileOver... {readOnly}
- PARAM_BATCH_SIZE :String = "batchSize" {readOnly}
- PARAM_PARALLEL :String = "parallel" {readOnly}
- exportFolderPath :String
- exportFileNamePrefix :String
- exportFileFormat :String
- exportFileOverwrite :boolean
- batchSize :int
- parallel :boolean
- dataExportStats :DataExportStats
+ DataExporter(SolrQueryRequest)
+ export() :void
+ isBusy() :boolean
- buildExportQueryRequest(SolrQueryRequest, int) :LocalSolrQueryRequest
- buildTotalNumberOfDocumentsQueryRequest(SolrQueryRequest) :LocalSolrQueryRequest
- handleQueryRequest(SolrQueryRequest) :SolrQueryResponse
- writeResponseToFile(SolrQueryRequest, SolrQueryResponse, int) :SolrQueryResponse
- getExportFile(int) :File
- totalNumberOfDocuments(SolrQueryRequest) :int
- computeNumberOfBatches() :long
- processQueryRequestParams(SolrQueryRequest) :void
- isNullOrEmptyString(String) :boolean
- close(Closeable) :void
+ getStats() :NamedList
-dataExportStats
DataExportStats
- status :String
- dateStarted :Date
- currentBatch :int
- completed :double
- stopwatch :Stopwatch
+ DataExportStats()
+ getStatus() :String
+ setStatus(String) :void
+ getDateStarted() :Date
+ setDateStarted(Date) :void
+ startStopwatch() :void
+ stopStopwatch() :void
+ getCurrentBatch() :int
+ setCurrentBatch(int) :void
+ getCompleted() :double
+ setCompleted(double) :void
+ getStats() :NamedList
-dataExporter
RequestHandlerBase
DataExportToFileHandler
- CLASSNAME :String = DataExportToFil... {readOnly}
- LOGGER :Logger = LoggingHelper.g... {readOnly}
- dataExporter :DataExporter
- finished :boolean = false
+ handleRequestBody(SolrQueryRequest, SolrQueryResponse) :void
- initDataExporter(SolrQueryRequest) :void
+ getDescription() :String
+ getSource() :String

Figure 10. Export handler class diagram

DataExportToFileHandler extends RequestHandleBase class, which is provided as a part of the Solr's API. Every request for exporting is going through object of this class. DataExportToFileHandler overrides method called `handleRequestBody`. This method is invoked every time user makes a request to handler. This object's responsibilities are to initialize and start the DataExporter and to handle the user's request with proper data including the exporting statistics.

DataExporter class provides the export to file capabilities for the DataExportToFileHandler. Object of this class is using simple logic, which can be described in following steps:

- parse the request's attributes
- precompute the number of batches
- query for the specific batch
- write the batch to the file

It is very important to divide the documents into the batches because of the limited RAM memory. It is possible to query for all the documents from the index at once; however if the index contains significant amount of documents all the documents will be fetched into the memory. Size of all the documents combined can easily exceeds the memory limit and when it does there is a risk of JVM crash and system crash. In order to enable batch processing separate batches of documents are retrieved by separate queries to Solr.

DataExportStats class represents object, which contains exporting statistics such as: current exporting status, precise time when the export process began, percentage of currently exported batches and currently exported batch. Object of this class is used by DataExporter for gathering the statistic, which can be displayed to user on request.

6.1.3 Usage

Data export handler is easy to use because the user just have to provide the correct query parameter to the handler. Example request, which triggers export of all the documents located in the PriceRules core is shown below:-

```
http://<host>:<port>/solr/PriceRules/dataexport?q=*:*
```

Export handler's response in JSON format for the above export request is following:-

```
<response>
 <lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">0</int>
 </lst>
 <lst name="params">
  <str name="exportFileFormat">csv</str>
  <str name="exportFolderPath">B:\SapPriceRulesTest\export\</str>
  <str name="batchSize">100000</str>
  <str name="exportFileOverwrite">true</str>
  <str name="exportFileNamePrefix">export</str>
  <str name="q">*:*</str>
 </lst>
 <lst name="exportStats">
  <str name="status">running</str>
  <date name="started">2015-05-11T17:05:42.465Z</date>
  <str name="timeElapsed">65911 ms</str>
  <int name="currentBatch">8</int>
  <str name="completed">80.0 %</str>
 </lst>
</response>
```

6.2 Import handler

Solr can process and handle large amounts of documents without serious problems; however the first step is to index these documents into the Solr. Importing process is important part of the data organization and maintenance because without the data being indexed there is nothing to organize and maintain. Importing process is even more important when the eCommerce systems such as WebSphereCommerce rely on huge amounts of data that has to be ready to retrieve and process as fast as possible.

Data import handler is Solr's contribution module, which provides configurable and extended import functionality for Solr. It provides various components that supports import from RDBMS systems, file systems, email servers, etc. DIH also provides extensive API for developers to build custom extensions. Custom CSVImportHandler extension for price load scenario is built on top of the default DIH and it provides configuration driven way to import large numbers of price rules documents stored in CSV format.

6.2.1 Configuration

Data import handler is very similar to basic request handler and it needs to be defined and configured in the `solrconfig.xml`; however, the import specific components and import process configuration needs to be configured in `dataimport-config.xml`. Following part of the `solrconfig.xml` configuration file contains definition of data import handler:-

```
<requestHandler name="/dataimport"
class="org.apache.solr.handler.dataimport.DataImportHandler">
   <lst name="defaults">
      <str name="config">dataimport-config.xml</str>
      <str name="importFolderPath">B:\SapPriceRulesNew\import</str>
      <bool name="clean">false</bool>
   </lst>
</requestHandler>
```

Custom csv data import handler definition contains name and class mandatory attributes together with defaults configuration properties. Every request handler defined in the configuration file is accessible via path that is defined in the name attribute located in the `<requestHandler>` tag. Specific path to the handler is as follows:-

```
http://<host>:<port>/solr/corename/dataimport
```

CSV import handler has defined number of default properties, which can be overwritten by the users. (See Table 14.)

Table 14. Data import handler default properties

Name	Description
config	Location to the import specific configuration file.
importFolderPath	Location of the files to be imported in the file system.
clean	Defines if the index should be cleaned before import process.

Price load data import configuration file is shown below:-

```
<dataConfig>
   <dataSource type="FileDataSource" name = "fds"/>
   <document onImportStart="com.rautakesko.commerce.price
                           .rule.solr.handler.dataimport
                           .DataImportStartEventListener"
             onImportEnd="com.rautakesko.commerce.price
                         .rule.solr.handler.dataimport
                         .DataImportEndEventListener">
      <entity name="sourceFile"
              processor="FileListEntityProcessor"
              baseDir="${dataimporter.request.importFolderPath}"
              fileName=".*csv"
              recursive="true"
              rootEntity="false">
         <entity name="line"
                processor="com.rautakesko.commerce.price
                          .rule.solr.handler.dataimport
                          .CSVLineEntityProcessor"
                url="${sourceFile.fileAbsolutePath}"
                separator=","
                transformer="com.rautakesko.commerce.price
                            .rule.solr.handler.dataimport
                            .SapPriceRuleDocumentIdTransormer">
           <field column="MATKL" name="MATKL" />
           <field column="MATNR" name="MATNR" />
           <field column="KPEIN" name="KPEIN" />
           <field column="KONDA" name="KONDA" />
           <field column="ext_lookupseq" name="ext_lookupseq" />
           <field column="KMEIN" name="KMEIN" />
           <field column="KBETR" name="KBETR" />
           <field column="WERKS" name="WERKS" />
           <field column="KNUMH" name="KNUMH" />
           <field column="AKTNR" name="AKTNR" />
           <field column="VKORG" name="VKORG" />
           <field column="ext_hier_ht1" name="ext_hier_ht1" />
           <field column="ext_hier_ht2" name="ext_hier_ht2" />
           <field column="ext_documentid" name="ext_documentid" />
           <field column="KONWA" name="KONWA" />
```

```
                    <field column="KOTABNR" name="KOTABNR" />
                    <field column="DATAB" name="DATAB" />
                    <field column="VTWEG" name="VTWEG" />
                    <field column="ext_customer" name="ext_customer" />
                    <field column="DATBI" name="DATBI" />
                    <field column="KSCHL" name="KSCHL" />
                    <field column="KNUMA_AG" name="KNUMA_AG" />
                </entity>
            </entity>
        </document>
</dataConfig>
```

This particular configuration contains one data source component, two event listener components and two entity components. Data source component is set to be `FileDataSource` because the files to be imported are located in the file system. The `<document>` tag contains special event listeners, which are used for logging and special postprocessing purposes. There are two event listeners defined: onImportStart event listener is triggered when the import process begins and the onImportEnd event listener is triggered after the successful import. Data import configuration file also contains definition for entities, which can be understood as higher level documents that are used for building the specific documents to be indexed to Solr. Price load data import handler uses two entities with processors defined as follows:

- FileListEntityProcessor
- CSVLineEntityProcessor

The `FileListEntityProcessor` is used for fetching the files containing the documents. It produces the files, which are consumed by the next processor in the chain and that is `CSVLineEntityProcessor`. This processor reads the lines of the files and it produces the documents that are indexed to Solr. The `CSVLineEntityProcessor` is defined with multiple properties. (See Table 15.)

Table 15. CSVLineEntityProcessor properties

Name	**Description**
url	Path to the file retrieved by `FileListEntityProcessor`.
separator	Character that is used as separator in the file to be imported.

transformer	Comma separated list of transformers that are used during the document processing.

The `CSVLineEntityProcessor` is capable of marking certain documents for delete from the index during the importing process. This can be especially useful if the index contains invalid data. Document can be marked for delete by providing unique id or there is also possibility to delete whole subset of documents matching specific query.

6.2.2 Implementation

Custom price load import handler is provided as one Java package, which consist of multiple components:

- CSVLineEntityProcessor
- CsvReader
- DataImportStartEventListener
- DataImportEndEventListener
- PriceLoadAggregator
- SapPriceRuleDocumentIdTransformer

Class diagram shown below provides detailed view of the Java classes, which provide implementation to Solr components. (See Figure 11.)

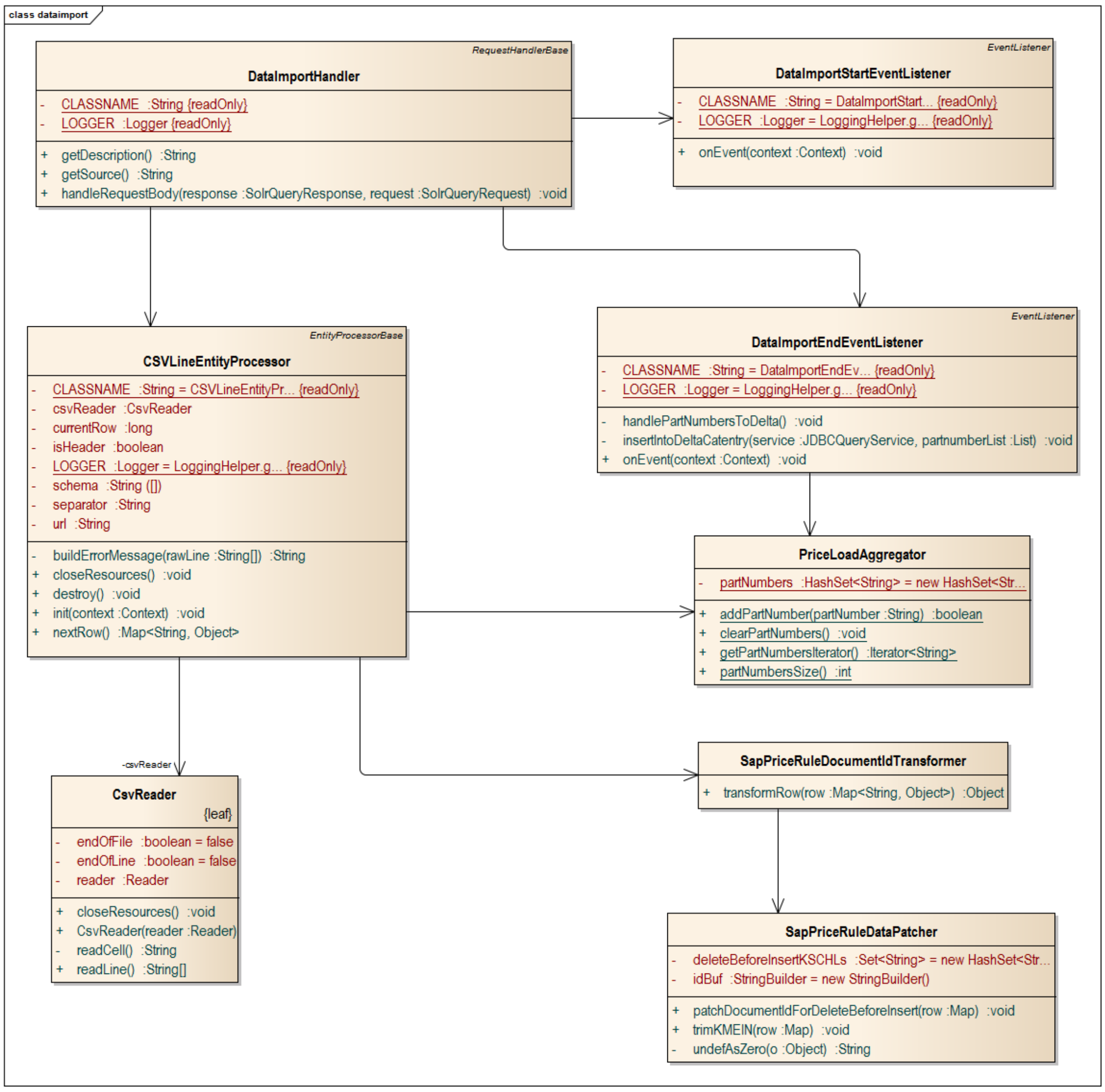
class dataimport
RequestHandlerBase
DataImportHandler
- CLASSNAME :String {readOnly}
- LOGGER :Logger {readOnly}
+ getDescription() :String
+ getSource() :String
+ handleRequestBody(response :SolrQueryResponse, request :SolrQueryRequest) :void
EventListener
DataImportStartEventListener
- CLASSNAME :String = DataImportStart... {readOnly}
- LOGGER :Logger = LoggingHelper.g... {readOnly}
+ onEvent(context :Context) :void
EntityProcessorBase
CSVLineEntityProcessor
- CLASSNAME :String = CSVLineEntityPr... {readOnly}
- csvReader :CsvReader
- currentRow :long
- isHeader :boolean
- LOGGER :Logger = LoggingHelper.g... {readOnly}
- schema :String ([])
- separator :String
- url :String
- buildErrorMessage(rawLine :String[]) :String
+ closeResources() :void
+ destroy() :void
+ init(context :Context) :void
+ nextRow() :Map<String, Object>
EventListener
DataImportEndEventListener
- CLASSNAME :String = DataImportEndEv... {readOnly}
- LOGGER :Logger = LoggingHelper.g... {readOnly}
- handlePartNumbersToDelta() :void
- insertIntoDeltaCatentry(service :JDBCQueryService, partnumberList :List) :void
+ onEvent(context :Context) :void
PriceLoadAggregator
- partNumbers :HashSet<String> = new HashSet<Str...
+ addPartNumber(partNumber :String) :boolean
+ clearPartNumbers() :void
+ getPartNumbersIterator() :Iterator<String>
+ partNumbersSize() :int
-csvReader
CsvReader
{leaf}
- endOfFile :boolean = false
- endOfLine :boolean = false
- reader :Reader
+ closeResources() :void
+ CsvReader(reader :Reader)
- readCell() :String
+ readLine() :String[]
SapPriceRuleDocumentIdTransformer
+ transformRow(row :Map<String, Object>) :Object
SapPriceRuleDataPatcher
- deleteBeforeInsertKSCHLs :Set<String> = new HashSet<Str...
- idBuf :StringBuilder = new StringBuilder()
+ patchDocumentIdForDeleteBeforeInsert(row :Map) :void
+ trimKMEIN(row :Map) :void
- undefAsZero(o :Object) :String

Figure 11. Price load import handler package diagram

The `CSVLineEntityProcessor` class is most important among all of these because it is responsible for creating the correct Solr documents that are indexed by document writer. This class extends the `EntityProcessorBase` and overrides three methods of the parent class. These methods are `init`, `nextRow` and `destroy`. This class uses modified `apache.wink.CsvReader` for efficient parsing of the CSV documents.

DataImportStartEventListener and DataImportEndEventListener provides way of how to log information before the very process of import starts and after the process successfully ends. DataImportEndEventListener is also used for additional postprocessing of the successfully indexed documents.

6.2.3 Usage

Data import handler is used with commands it provides. Example request, which triggers import of all the CSV files located in the folder configured in the `dataimport-config.xml` file is as follows:-

```
http://<host>:<port>/solr/PriceRules/dataimport?command=f
ull-import
```

Example response with current status of the data import handler is shown below:-

```
<response>
 <lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">0</int>
 </lst>
 <lst name="initArgs">
  <lst name="defaults">
   <str name="config">dataimport-config.xml</str>
   <str name="importFolderPath">B:\SapPriceRulesTest\import</str>
   <bool name="clean">false</bool>
  </lst>
 </lst>
 <str name="command">status</str>
 <str name="status">busy</str>
 <str name="importResponse">A command is still running...</str>
 <lst name="statusMessages">
  <str name="Time Elapsed">0:38:59.756</str>
  <str name="Total Requests made to DataSource">0</str>
  <str name="Total Rows Fetched">6789221</str>
  <str name="Total Documents Processed">6789152</str>
  <str name="Total Documents Skipped">0</str>
  <str name="Full Dump Started">2015-05-14 16:22:16</str>
 </lst>
 <str name="WARNING">This response format is experimental.</str>
</response>
```

7 Testing

Testing is an important part of this project because it provides valuable information necessary for evaluation of results. Tests can be run with various configurations for the tested components and thus they provide comparable data. This data can be used to determine, which specific configuration should be used for the specific tested component. Testing is highly dependent on many factors such as: testing environment, testing parameters and others. A specific development environment was used as testing environment for the custom Solr’s extensions. (See Table 16.)

Table 16. Testing environment

WebSphere Commerce version	WebSphere Commerce V7.0 Feature Pack 7
Apache Solr version	4.3.0
Solr buffer size	256 MB
Java version	1.7
Java Heap size	1536 MB
Operating system	Microsoft Windows 7 (Virtual environment in VMWare)
RAM	8 GB
Number of processor cores	3

The exporting test was run multiple times with different parameters and with different number of documents in the index. The most important parameter is the batch size parameter. This parameter defines the number of documents in one exported file. The table with results contains the basic test parameters such as number of documents that were exported, number of test runs, exporting batch size parameter and average, minimum and maximum export times in milliseconds. (See Table 17.)

Table 17. Export test results table

Number of documents	Number of runs	Batch size	Average time (ms)	Min time (ms)	Max time (ms)
100000	10	100000	8278.1	8133	8522
1000000	10	100000	80770.9	80201	83409
10000000	10	100000	1402279	1399205	1409940
10000000	10	1000000	841014.2	838891	844980
28000000	10	100000	8157983.2	8125335	8195564
28000000	10	1000000	2859509.4	2799984	2900142

In order to obtain useful results it is necessary to run multiple tests with a different number of documents to import. The table with results from importing tests contains similar information as table with exporting tests. Importing table contains the total number of imported documents, number of files containing the documents and number of CSV files used for import. The table also contains the average, minimum and maximum importing times in milliseconds. (See Table 18.)

Table 18. Import test results table

Total number of documents	Number of files	Number of documents in one file	Number of runs	Average time (ms)	Min time (ms)	Max time (ms)
100000	1	100000	10	37412.4	29375	45340
1000000	10	100000	10	373075.2	321345	412230
10000000	100	100000	10	3758635	3721564	3798507
28000000	280	100000	10	10939755.6	10741215	11103459

8 Results and discussion

The main objective of this thesis was to enhance the existing way of Big Data organization and Big Data maintenance in the WebSphere Commerce environment. This project was based on specific eCommerce pricing scenario from an existing project, and the current way of handling a large number of documents was not sufficient enough. Multiple steps were taken to meet the goals of this project:

- Existing document schema for specific pricing was changed to suit the needs of the application better. This change resulted in the reduction of the index size and slightly faster query response times.
- Export to file data handler extension was created to enable developers to backup existing data from Solr in an easy and configuration driven way.
- Import from CSV file data handler extension was created to support the import of large amounts of data into the Solr. The extended version of this component is used to import tens of millions of product price rules into the existing eCommerce environment.

The new way of importing the price rules documents into the WebSphere Commerce provides a great alternative to the Dataload utility, which is the recommended and default way how to import data into the WebSphere Commerce; however Dataload utility is used by other components and price loading can block the utility for multiple hours depending on the number of imported documents. The most notable difference is the speed of the importing process. The created data import handler extension is able to import tens of millions of documents into the system much faster than Dataload utility because it utilizes Solr’s internal components.

There are multiple issues that come with the created solution and these can be improved in the future. A few of the issues are directly connected to the Apache Solr version 4.3.0, which is used by WebSphere Commerce V7.0 Feature Pack 7. For example: the deep paging problem is the most serious issue that has impact on the export to file handler; however, this issue is resolved in the Apache Solr version 4.7.0 that is used in WebSphere Commerce V7.0 Feature Pack 8.

Another step, which can improve the performance and scalability is to consider the usage of Solr Cloud; however, before that single machine optimization should take place.

Apache Solr and Apache Lucene fully support the usage of the various file systems. Different file systems can be used with the Apache Solr and most interesting of them is the HDFS file system, which was specifically created to work with the large data and Big Data.

The created solution was tested for various configurations and it is used in the existing project; however more tests are needed in order to find the best possible configuration for the specific scenario. More tests are also needed to determine the performance impact of the created components on Solr’s server and WebSphere Commerce server.

References

Apache Solr Wiki. Accessed on 23 April 2015. Retrieved from Apache Solr Wiki: https://cwiki.apache.org/confluence/display/solr

Gospodnetic, O., & Hatcher, E. (2005). *Lucene in action.* Greenwitch: Manning Publications.

Kuc, R. (2013). *Apache Solr 4 Cookbook.* Birmingham: Packt Publishing Ltd.

Lucene Apache Jira. Accessed on 22 April 2015. Retrieved from Apache Jira: https://issues.apache.org/jira/browse/LUCENE-2412

Sadalage, P. J., & Fowler, M. (2013). *NoSQL Distilled, A Brief Guide to the Emerging World of Polygot Persistence.* Crawfordsville: Pearson Education, Inc.

Scaling Apache Solr. Accessed on 22 April 2015. Retrieved from Safari books online: https://www.safaribooksonline.com/library/view/scaling-apache-solr/9781783981748/ch01s04.html

Smiley, D., & Pugh, E. (2011). *Apache Solr 3 Enterprise Search Server.* Birmingham: Packt Publishing Ltd.

Technology dictionary. Accessed on 15 April 2015. Retrieved from Technopedia webpage: http://www.techopedia.com/dictionary

Traver, C. G., & Laudon, K. C. (2007). *E-commerce: business, technology, society.* New Jersey 07458: Prentice Hall.

WebSphere Commerce product overview. Accessed on 15 April 2015. Retrieved from IBM Knowledge Center: http://www-01.ibm.com/support/knowledgecenter/

Zikopoulos, P., Deroos, D., Parasuraman, K., Deutsch, T., Corrigan, D., & Giles, J. (2013). *Harness the Power of Big Data.* McGraw-Hill.