WebSphere Commerce Search Solution Performance Best Practices

Optimizing e-commerce performance on a Solr-based search engine

Yang Gao (ygaobj@cn.ibm.com)  
Software Engineer  
IBM China

Kevin Yu (kevinyu@ca.ibm.com)  
Performance Architect  
IBM

Search solution is an important feature in the WebSphere Commerce product, and is widely used by Commerce customers. Therefore, it is important for customers to understand the best practices to optimize Search solution performance.

Optimizing search when planning and deploying a production environment

Planning the topology of a WebSphere Commerce site

WebSphere Commerce integrates the open source Solr search engine, which is deployed as a separate WebSphere application to maximize scalability options. When you plan the capacity for the entire Commerce site, ensure that you add the appropriate hardware resources to the Solr application server. Separating servers for the Solr application reduces the capacity requirement of the Commerce application server. With this topology design, you must also consider the network bandwidth that is needed to transmit request and response data between the Commerce and Solr applications. The speed or the bandwidth of the internal network does not need to be a major concern, but both variables need to be considered and tested. You might also consider deploying the Solr application on the same Commerce server with a separate Java Virtual Machine (JVM), which conserves server and network bandwidth, but consumes the same amount of server capacity.

The Solr application also requires a web server. You can deploy a separate web server specifically for the Solr cluster, which avoids potential security risks and reduces potential performance problems, because requests now pass through an extra firewall. However, a separate web server...
requires more hardware, load balancing, and other considerations. All of these requirements must be considered at an early stage of the solution planning and design.

**Planning the topology of a Solr application**

On a large site, there are typically thousands or tens of thousands of concurrent users. It is difficult for a single Solr server's runtime JVM to handle that many requests while it maintains satisfactory performance requirements. Similar to Commerce and other WebSphere Application Server applications, a horizontal and vertical cluster can be used for Solr servers to handle scalable and high-availability requirements. A single main service node (master server) is configured to control the synchronization of search index data on the cluster members (subordinate servers), as shown in Figure 2. The master node is used for indexing update purposes only and is not a member of the Solr cluster. The Commerce production environment does not access the master node; instead, it accesses only the subordinate nodes. Any updates on the master node are detected and “pulled” by subordinate nodes to modify the copying of data on the subordinate nodes.

**Figure 1. Solr master-subordinate configuration diagram**
Figure 2. Solr cluster structure diagram

Figure 3. System deployment diagram

Deployment Manager

The following configuration is a sample configuration for a Solr master node:

Listing 1. Solr master node configuration

```xml
<requestHandler name="/replication" class="solr.ReplicationHandler">
  <lst name="master">
    <!-- Replicate on 'optimize'. Other values can be 'commit', 'startup'. It is possible to have multiple
```
The following configuration is a sample configuration for a Solr subordinate node:

Listing 2. Solr subordinate node configuration
<requestHandler name="/replication" class="solr.ReplicationHandler">
  <lst name="subordinate">
    <!-- The fully qualified URL for the master replication handler. It is possible to pass this as a request parameter for the fetchindex command. -->
    <str name="masterUrl">http://localhost:port/solr/corename/replication</str>

    <!-- Interval in which the subordinate should poll the master. Format is hh:mm:ss. If this is absent the subordinate does not poll automatically. However, a fetchindex can be triggered from the admin or the HTTP API. -->
    <str name="pollInterval">00:00:20</str>

    <!-- THE FOLLOWING PARAMETERS ARE OPTIONAL -->
    <!-- To use compression while transferring the index file. Possible values are 'internal' or 'external'. If the compression value is 'external' make sure that your master Solr has the settings to honor the accept-encoding header. See http://wiki.apache.org/solr/SolrHttpCompression for details. If the compression value is 'internal' everything will be taken care of automatically. Important: Use this only if your bandwidth is low. This can actually slow down replication in a LAN. -->
    <str name="compression">internal</str>

    <!-- The following values are used when the subordinate connects to the master to download the index files. Default values implicitly set as 5000ms and 10000ms respectively. The user DOES NOT need to specify these unless the bandwidth is extremely low or if there is an extremely high latency. -->
    <str name="httpConnTimeout">5000</str>
    <str name="httpReadTimeout">5000</str>

    <!-- If HTTP Basic authentication is enabled on the master, then the subordinate can be configured with the -->
Optimize Solr index delta update from staging server

Regularly scheduling the replication in Listing 2 is not recommended. The search index that is used for staging might contain data that is not production-ready. Therefore, IT admin must carefully time the search index replication to avoid data contamination in production.

The Solr index delta update from the staging server is a new feature that was introduced in WebSphere Commerce, Feature Pack 6. The objective is to both optimize and synchronize the process flows of the database update, Solr index update, and DynaCache invalidation. The flow begins with an update on the staging server and ends by invalidating the DynaCache on production after staging propagates the updates into production. The following diagram shows the flow:

**Figure 4. Timeline of indexing with staging population**

![Timeline Diagram](image)

The Solr search engine server that resides in the staging environment is the master node. The search index repeater is used both as a source and a target for search replication. The search index repeater is used as a target when replication is scheduled with the staging search index.
The staging search index is the source, and the repeater is the target and acts as a backup of the search index for production. When the first replication is completed from staging, the repeater communicates the changes to its target nodes in production. The repeater then becomes the source, where all nodes from the search index cluster are configured to pull changes from the repeater on a regular pre-configured fixed-time interval. This time interval is defined in the solrconfig.xml file under replication.

Replication between the repeater and all the search index clusters in production can be automated because the indexed data in the repeater always matches the indexed data in production. The search index repeater must be a target to the staging search server and a source to the production search server. The automated tool is the indexprop utility. The indexprop utility is used by IT administrators to propagate the WebSphere Commerce search index and to invalidate the cached content in production. The indexprop utility performs the following tasks in sequence:

1. Pre-processing the WebSphere Commerce search index data
2. Building the WebSphere Commerce search index
3. Replicating the WebSphere Commerce search index
4. Performing cache invalidation for the WebSphere Commerce search in production

You are recommended to use the indexprop utility for staging propagation because the utility ensures that data is updated and that cache content is refreshed for the updates. For more information about detailed tool usage, see Propagating the WebSphere Commerce search index.

Optimizing the HTTP server network parameters

All search service requests are sent to the HTTP server and are then forwarded to WebSphere Application Server. This request process requires the optimization on the number of concurrent connections as well as other aspects of the HTTP server. In addition to an HTTP connection to handle page responses, extra connections to handle the Search requests are also required. Therefore, an appropriate tuning on the HTTP server is needed. For best performance, it is recommended that Solr applications use a different HTTP server than the Commerce instance.

In the following equation, $N$ denotes the number of connections for the administration and maintenance of Solr server.

\[
\text{The number of actual concurrent connections to the Solr HTTP server} = \text{The number of allowed concurrent connections} + N
\]

The connections, $N$, are normally used for creating or updating the search index, and synchronizing the search index within the Solr nodes, and so on. $N$ must be adjusted on a case-by-case basis, according to the specific site's requirements. Typically, for a small site it is reasonable for the value of $N$ to range 10 - 20. You are suggested to use separate servers for the Solr application and WebSphere Commerce to avoid mutual impairment and resource competition, as well as to simplify problem determination.

You are recommended to validate the tuning of the HTTP server connection during a performance load test that simulates expected peak traffic volumes and workload patterns. HTTP server
connections can be monitored by enabling the mod_status Apache module in the IBM HTTP Server server configuration.

**Optimizing the Commerce network parameters**

As a client of the Solr application, WebSphere Commerce controls its own network tuning parameters. These parameters are defined in the Commerce runtime environment configuration file, `wc-search.xml`. In the following example, the current configuration can be found by using the name attribute:

**Listing 3. Sample configuration for runtime network parameters**

```xml
<config:server name="AdvancedConfiguration">
    allowCompression="true" connectionTimeout="5000"
    defaultMaxConnectionsPerHost="600" followRedirects="false"
    maxRetries="1" maxTotalConnections="600"
    retryTimeInterval="6" soTimeout="5000"/>
</config:server>
```

All time-related runtime network parameter values are in seconds:

- **connectionTimeout**
  Determines the threshold for a network connection timeout, and *soTimeout* is the threshold for the network socket connection timeout. When the production environment has a smaller workload, *connectionTimeout* and *soTimeout* are set to small values to avoid long page response times. The ideal value range is 3 - 5 seconds. Increase the values of *soTimeout* and *connectionTimeout* if frequent timeout errors that are caused by a high level of concurrent accesses occur.

- **defaultMaxConnectionPerHost**
  Determines the maximum number of allowable connections for each host. *maxTotalConnections* is the total maximum number of allowable connections from all hosts. When WebSphere Commerce and Solr are on separate servers and have a one-to-one relationship, you can set *defaultMaxConnectionPerHost* and *maxTotalConnections* to the same value. However, in a clustered environment, *maxTotalConnections* must be greater than *defaultMaxConnectionPerHost*. Similar to the HTTP server, you might need to adjust *defaultMaxConnectionPerHost* and *maxTotalConnections* according to the number of possible concurrent visits. Smaller values can cause access denied errors.

- **maxRetries**
  Determines the maximum number of retries that are allowed when a Solr request fails. *retryTimeInterval* determines the time interval before the next retry. Do not set the *retryTimeInterval* value too high to avoid access denied errors for subsequent requests that are caused by waiting for the retry interval.

**Optimizing the JVM parameters**

Based on Solr's runtime characteristics, optimization of the Java virtual machine (JVM) parameters rely mainly on the JVM heap size. Ensure that you assign a sufficient heap size to ensure that the
Solr application can operate efficiently for various logic executions. Depending on your particular production environment, the heap size for the Solr application can be affected by the following attributes:

1. The object size of batch processing on database records during index building.
2. The size of the search index data. Whenever possible, ensure that you place index data in the file system cache. To determine the size of the index data:
   a. Open `<SOLR_DIRECTORY>/solr.xml` to check the properties of `instanceDir` for each Solr core.
   b. Check the file size of the `instanceDir/data` directory.
3. The cache size of Solr runtime data.
4. The complexity and object size of Solr data.

When you create a new instance of the Solr runtime engine, the default value for the Solr runtime JVM heap size is 512 - 1024 MB. Before the site goes live, enable the JVM Verbose Garbage Collection (verbose GC) to monitor and analyze the behavior of JVM garbage collection. Performance tuning is considered optimized if the JVM garbage collection overhead is less than 5%. Depending on the allotment for garbage collection, further tuning might be required. If you are using the out-of-the-box Solr caching setting for a 64-bit application server, set the maximum heap size to 3 GB, minimum heap size to 2 GB, and maintain a 1 GB nursery with a `gencon` garbage collection policy.

**CSV or XML direct index update customization**

The Solr engine supports direct index updates from the input of CSV/XML text-formatted files. Solr uses an out-of-the-box handler component to handle such direct update requests. The out-of-the-box handler component is convenient and easy to integrate with existing ETL tools. The handler component also bypasses heavy access routes to other complex data sources, such as the database.

**Introduction**

To import or transform data that might be saved in multiple sources with multiple formats to the Solr index, you can either use the `DataImportHandler` object, or the Solr client-side library, Solrj. `DataImportHandler` is a Solr component that is registered in the `solrconfig.xml` file. You can do a full import or a delta update, and the `DataImportHandler` component performs the following configurations or customizations:

1. Defines where to fetch the data from supported data sources. You can also define your preferred data sources.
2. Defines how to process or transform source data. `Processor` is a Java class that handles each entity that is retrieved from the data source. `Transformer`, also a Java class, modifies source field or creates new fields. `Transformer` classes can be "chained". For example, you can define three `Transformer` classes that each process the source data consecutively. The output of the first `Transformer` is consumed by the second `Transformer`, and then the output of the second `Transformer` is consumed by the third. You can register your own `Processor`
and Transformer classes in the `wc-data-config.xml` file. In WebSphere Commerce, new Processor classes are introduced to handle Character Large Objects (CLOB), Apache Tika, and other web content entities. For more examples of Processor class definitions, see the files in the home directory of the out-of-the-box Solr deployment with WebSphere Commerce: `<WEBSPHERE_COMMERCE_INSTANCE>/search/solr/home/
3. Extends and registers the new DataImportHandler object.

It might be convenient to use the DataImportHandler component when data is stored in a relational database management system. However, if data is stored in CSV/XML format, it is more efficient to use the Solrj API for direct index updates. Solrj is a Java client that is used to operate Solr. Solrj offers a Java interface to add to, update, delete from, or query the Solr index. In Figure 5, the existing WebSphere Commerce `dataload` framework accepts the Solr server as a type of target data source:

**Figure 5. The logical diagram of a dataload-based index update**

- The **Business Object Layer** contains the SDO Builder, BOD Mediator, and Solr Mediator. The Business Object Layer handles the conversion of business objects into physical objects. Physical objects can be persisted into the database. The Business Object Mediator transforms business objects into the data source that is supported by the physical business object. When the `dataload` utility is used to create or update the Solr index, the SolrInputDocumentMediator converts the business object that was built from either the CSV or the XML file to SolrInputDocument.
- The **Persistence Layer**, which contains the JDBC Writer and the Solr Writer, is where the physical object is persisted into the data source. The SolrDataWriter component is
used to add or create a SolrInputDocument in the Solr server. SolrDataWriter initializes
HttpSolrServer and commits SolrInputDocument objects to the server.

- You can use the Business Context Service in any of the layers when the business context
data is needed.

Direct Index Update Customization

WebSphere Commerce provides a flexible programming framework for its Search solution
based on the Solr runtime engine. The following tutorial is an example to customize the direct
index update by adding features from CSV or XML data sources. Customization also includes
adding customized Java classes and updating existing Solr configuration files. You can make all
customization changes in the solrconfig.xml file:

1. Initialize either a HttpSolrServer or EmbeddedSolrServer, and set the server connection
   settings as in the following example code. You can upload content in XML format, which is the
default, or in binary format. Because the Solrj client fetches results in binary format, uploading
   the content in binary format improves performance by reducing stress on the system with
   XML marshalling:

   ```java
   HttpSolrServer server = new HttpSolrServer("http://localhost:8080/solr2/");
   server.setRequestWriter(new BinaryRequestWriter());
   server.setConnectionTimeout(15000);
   server.setSoTimeout(15000);  // socket read timeout
   server.setMaxTotalConnections(600);
   server.setFollowRedirects(false);  // defaults to false
   server.setAllowCompression(true);
   server.setMaxRetries(1); // defaults to 0. > 1 not recommended.
   server.setDefaultMaxConnectionsPerHost(600);
   ```

2. Ensure that you include the following code to enable the binary update request handler:

   ```xml
   <!-- Binary Update Request Handler http://wiki.apache.org/solr/javabin -->
   <requestHandler name="/update/javabin" class="solr.BinaryUpdateRequestHandler" />
   ```

3. Include the following code to initialize and add a SolrInputDocument object:

   ```java
   SolrInputDocument doc1 = new SolrInputDocument();
   doc1.addField( "id", "id1", 1.0f );
   doc1.addField( "name", "doc1", 1.0f );
   doc1.addField( "price", 10 );
   if (updateReq == null) {
       updateReq = new UpdateRequest();
   }
   if (solrDocDO.getOpMode() == DataLoadConstants.DL_OP_DELETE) {
       updateReq.deleteById((String)solrDocDO.
       getIdFieldName()));
   } else {
       updateReq.add(solrDocDO.getSolrInputDocument());
   }
   ```

4. Include the following code to flush and commit:

   ```java
   updateReq.process(iSolrServer);
   updateReq.clear();
   iSolrServer.commit();
   ```

5. The following configuration shows you the logic flow of the various components that are used
   in the configuration. It includes the dataload utility, the CSVReader, which is used as a loader
   layer, the MapObjectBuilder and SolrInputDocumentMediator objects, with the super classes
   SDOBuilder and SolrMediator. These components are used to convert name-value pair data,
   which is read in from CSV format to the SolrInputDocument object:
Performance tuning considerations

The Solr application's index update handler work flow is depicted in Figure 6:
Figure 6. High-level index update logic diagram

Figure 6 shows that the Solr index update logic is synchronous, meaning that a single component functions at a time. For example, the Solr update handler awaits data input, while the data load utility composes the next data object to send. This process is inefficient, but it can be improved by splitting the data files and running several data load utility processes in parallel on the Solr server. The most efficient starting ratio for data load utility instances versus the Solr server instances is a 4:1 ratio, considering the locking and mutex access parts that are embedded in the Solr update logic codes.

When the data load utility sends the composed objects to the Solr server, it uses a variant format in the data stream. If the data stream is text-based, it is read in a line-buffer manner, meaning that Solr reads and parses each line at a time until the entire data stream is read. A line-buffer reading is inefficient. You can also switch to a binary format data stream if you change the data load configuration file.

When the memory buffer for updates is filled, the Solr server persists extraneous data to disk. The Solr server then sets a global exclusive lock for the entire JVM. The global exclusive lock prevents other threads from completing an update operation, even if those threads handle unrelated files.
or records. The effects of the global exclusive lock can be mitigated if you tune the Solr server parameters for commit timing and buffer size in the `solrconfig.xml` file for the updated Solr core:

a. Enlarge the value of `ramBufferSizeMB`. The maximum value is 2048 MB:

```xml
<ramBufferSizeMB>2048</ramBufferSizeMB>
```
b. Disable the document-based count buffered setting:

```xml
<!----<maxBufferedDocs>1000</maxBufferedDocs>---->
```
c. Disable the server side auto-commit trigger:

```xml
<autoCommit>
  <maxDocs>10000</maxDocs>
  <maxTime>1000</maxTime>
</autoCommit>
```

## Pre-processing the Commerce database

In WebSphere Commerce, business objects are related to database objects that are dispersed across multiple tables. To improve the data import performance in the search engine, the data needs to be "flattened" across fewer tables. This "flattening" process is called data pre-processing and is completed by the `di-preprocess` utility. The `di-preprocess` utility creates temporary tables, queries all satisfied data records, and then bulk inserts the data into the temporary tables. Data pre-processing performance depends on two aspects: query records, and bulk inserts. Querying performance depends on the complexity and optimization of written SQL queries. Data insertion performance depends on database tuning, such as tuning the buffer pools or system input/output.

The following optimization suggestions are to improve data pre-processing on an IBM DB2 database system. However, the same suggestions can be applied to other database systems, including Oracle. Apply optimizations based on your particular database system:

- Create a separate table space for the temporary tables in pre-processing, and pre-assign sufficient physical storage. This ensures that the pre-processing has enough space to complete the process. Ensure that the allocated space is continuous. If the records in the temporary table are too large, increase the table space page size.
- Set a large buffer pool for this separate table space to improve disk input/output efficiency. The following example code creates a separate table space `MYTAB32K` with 32K page size and the corresponding buffer pool:

```sql
CREATE BUFFERPOOL MYBUFF32K IMMEDIATE ALL DBPARTITIONNUMS SIZE 4000 NUMBLOCKPAGES 0
  PAGESIZE 32K
CREATE REGULAR TABLESPACE MYTAB32K PAGESIZE 32K MANAGED BY AUTOMATIC STORAGE
  EXTENTSIZE 32 BUFFERPOOL MYBUFF32K INITIALSIZE 600M INCREASESIZE 500M
```
- If the insert data contains large amounts of text data but can be accommodated by the `VARCHAR` column, change the type from `LOB` to `VARCHAR`.
- If the text data is too large and the `LOB` type must be used, create a table space and open the `FILESYSTEM CACHING` switch, as shown in the following code:

```sql
CREATE REGULAR TABLESPACE MYTAB32K PAGESIZE 32K MANAGED BY AUTOMATIC STORAGE
  EXTENTSIZE 32 BUFFERPOOL MYBUFF32K FILE SYSTEM CACHING INITIALSIZE 600M INCREASESIZE 500M
```
- After you create the specified table space, modify the utility configuration files so that the temporary tables are created in the table space that was created in Recommendation 4:
You can also fine-tune the performance of the temporary tables by removing the logging and improving the pre-processing speed. The *di-preprocess* utility contains several XML configuration files, each with a similar format to define SQL statements for entity object table creation and processing.

- Adjust the concurrent input/output parameters in your database management system. Refer to the disks used in your actual environment and set the number of concurrent input/output processes to the allowed number of parallel input/output processes in your disk system. In an environment with only one disk, a single process - responsible for reading and writing to the disk - is set in the following example:

  ```sql
  UPDATE DB CFG FOR MALL USING NUM_IOCLEANERS 1
  ```

  If you are using a disk array in your environment, you might consider setting the stripe depth to the same units as database input/output parameter units. For example, you could set the stripe depth to the block size of DB2 Extent to avoid disk contention between concurrent input/outputs.

- Adjust the threshold of synchronization between the buffered data and the disk files to avoid disk system overload for large synchronous data:

  ```sql
  UPDATE DB CFG FOR MALL USING CHNGPGS_THRESH 50
  ```

- In WebSphere Commerce, Version 7, Feature Pack 3, the *di-preprocess* utility supports multi-threaded data processing mode, which greatly enhances the pre-processing performance. To enable the *di-preprocess* utility, see [di-preprocess utility](https://www.ibm.com/support/knowledgecenter/SSY37A_7.0.0/com.ibm.wcs.doc/operations/optimizing_search_performance.html) in the WebSphere Commerce IBM Knowledge Center.

### Building the search index

You can build the search index data with the *di-buildindex* utility, or the Solr application user interface. There are several parameters that can affect performance. You must decide how to tune the parameters to yield the best performance on your particular system:

**useCompoundFiles**

Determines whether Solr uses multiple files to compose the entirety of the index data. The value of *useCompoundFiles* is set to *FALSE* to improve the performance of index construction, considering the limitations of the file descriptor of a system process and the file system cache.

**mergeFactor**

Determines the number of segments that are allowed in a file. The index data file is separated into "segments" or "sections". When the index is built, new data is added to the segment that is being processed. When the number of segments in a file exceeds the threshold value that is determined by *mergeFactor*, the segments are merged into one segment. A small *mergeFactor* value improves search performance. However, if the value of *mergeFactor* is too small, the segment merge operation occurs too frequently, which can negatively affect performance. The default value of *mergeFactor* is 10, which yields good performance for most environments. You can adjust *mergeFactor* according to your particular environment.
maxBufferedDocs
Determines the number of document objects that are cached in memory. In Solr, each data record is called a "document object". When the maxBufferedDocs value is reached, the data is synchronized to the disk. Parameters must be adjusted to complement the Java virtual machine heap size settings. Adjust the value of maxBufferedDocs according to the data structure definition for a Solr document object. When possible, provide enough memory cache to improve index building performance.

ramBufferSizeMB
Determines the amount of memory allotted for caching document objects. If cached content reaches the threshold that is defined by ramBufferSizeMB, the data is synchronized to the disk. When possible, provide enough memory cache to improve index building performance.

maxMergeDocs
Determines the number of document objects that are allowed in a segment of the index data file. A large maxMergeDocs value reduces the merge operation frequency and improves performance.

autoCommit
Includes the parameters maxDocs and maxTime. maxDocs determines the threshold value of the maximum allowed number of documents. maxTime determines the threshold value of the allowed time for automatic submission of updated documents. Frequent submissions might guarantee the accuracy of the index data, but negatively impacts performance. For a complete rebuild of the index data, ensure that you set maxDocs and maxTime to large values.

If the index data is large and cannot be completed in routine maintenance, you can adjust the parameters to disable index optimization during index building. This minimizes the impact of problems that might occur when the production environment is accessed. Index optimization is a single-threaded operation, and it is the internal core function of Solr and Lucene. It is difficult to effectively optimize its performance. The non-optimization does not affect the Solr work normally and might only slightly extend the search response time. In the case where there is no effect on the use of the production environment, the index optimization can be completed on the Solr master node, and, at a more convenient time, synchronize the index to the Solr subordinate nodes for the production environment. The following sample URL triggers an index rebuild with the parameter that disables index optimization:

http://<HOSTNAME><PORT>/solr/<SolrCore_name>/dataimport?command=full-import&optimize=false

After the index is built, you can use the following URL to perform the index optimization separately:

http://<HOSTNAME><PORT>/solr/<SolrCore_name>/update?optimize=true

Many file operations occur during the index creation process. As a result, ensure that you increase the limit for processing open files at the operating system level, such as the ulimit command for Linux/UNIX systems, or the registry operation for Windows systems. For more information about adjusting your system's ulimit setting or equivalent, see the documentation of your particular operating system.
**Note:** You can find all configuration parameters in the Solr core configuration file, solrconfig.xml in the following directories:

- For versions of WebSphere Commerce, Feature Pack 3+:
  solrhome/MC_<masterCatalogId>/<locale>/CatalogGroup/conf/
- For all other versions and feature packs of WebSphere Commerce:
  solrhome/MC_<masterCatalogId>/<locale>/CatalogEntry/conf/

**Multiple-Index component customization**

**Overview**

Following the official Solr request architecture, the RequestHandler object accepts search query conditions from inbound HTTP requests and dispatches tasks to different components to construct a QueryResponse object. This action provides an opportunity to customize Solr to read in data from multiple indexes and join the resulting set at the given correlation key, similar to a SQL JOIN statement. With the customization, the Solr engine can now isolate data into separate indexes according to the data category and volatile characteristics. Figure 7 displays a diagram of the enhanced MultipleReader customization process. A new Multi-Index Reader is added to the Solr engine to intercept incoming queries from the e-commerce system. The Multi-Index Reader parses the queries and separates them into several subqueries by index. The subqueries then read the data from each index, and then merge into a joined resulting set before they are returned to the e-commerce system.

**Figure 7. MultipleReader customization**
MultipleReader Customization

To customize the MultipleReader:

1. You might want to customize the RequestHandler object. For example, you can pre-process request data before it is handled by the RequestHandler. You can also customize the RequestHandler to perform other actions before it is sent from the Solr engine. However you decide to customize the requestHandler, ensure that you enter the updated code into the solrconfig.xml file.

2. Because the SearchComponent object contains the main logic for performing search actions on inbound search queries, it is the most important component when requests are fulfilled within the Solr search engine. The SearchComponent is chained with other components, such as a FacetComponent or QueryComponent to fulfill the incoming query request. These components have the same reference to the SolrQueryRequest and SolrQueryResponse. Any change in the logic of the components is reflected in the response. You can customize the components to improve the search logic and the search result.

3. With an extended index in WebSphere Commerce, after the SolrQueryResponse is on the master core, the extended information from extended index must be appended. You must customize QueryComponent and FacetComponent to append the extended information. In the customized QueryComponent:

   ```java
   public void prepare(ResponseBuilder responseBuilder) throws IOException {
       final String METHODNAME = "prepare(ResponseBuilder)";
       SearchLogger.entering(SEARCHLOGGER, CLASSNAME, METHODNAME, new Object[]{responseBuilder});
       tryPrepareSolrCores();
       super.prepare(responseBuilder);
       SearchLogger.exiting(SEARCHLOGGER, CLASSNAME, METHODNAME);
   }
   
   4. As you can see in the prepare method in Step 3, you must prepare extended Solr cores before you prepare the master Solr cores. In the process method, you can parse the query if extended core information is needed, and append the extended core information to the main response. WebSphere Commerce also customizes FacetComponent to support facets on extended fields on extended indexes:

   ```java
   // Request for a searcher instance and perform a sub-index search
   QueryResult queryResult = subSearcher.search(
       new QueryResult(), subQueryCommand);

   // Prefetch all returned document into the searcher and
   // merge the 2nd search result (after reading from the index
   // reader) into the main searcher's document cache.
   // Note: Do not modify the DocSet or DocList inside of the
   // QueryResult because the original DocSet or DocList may
   // have already been cached by Solr
   DocIterator docIterator = queryResult.getDocList().iterator();
   while (docIterator.hasNext()) {
       // If more than one entry returned, turn into a
       // multi-value field. Otherwise, just add the field with
       // the current value into its corresponding document
       // cache in the base index.
       Document document = subSearcher
   ```
5. After you define your own `QueryComponent` and `FacetComponent`, you must include it in the `solrconfig.xml` file:

```xml
<searchComponent name="wc_query" class="com.ibm.commerce.foundation.internal.server.services.search.component.solr.SolrSearchMultipleQueryComponent">
  <int name="cacheSize">1320000</int>
  <str name="referenceField">catentry_id</str>
</searchComponent>
<searchComponent name="wc_facet" class="com.ibm.commerce.foundation.internal.server.services.search.component.solr.SolrSearchMultipleFacetComponent">
</searchComponent>
```

**Runtime customization**

WebSphere Commerce provides a complete set of functional interfaces for the Solr application. In addition to search, the Solr application can also be used by other components, such as catalogs and orders, and serves as a data provider. These service interfaces provide a wealth of built-in features and scalable customization features.

Figure 8 is a flow diagram of the WebSphere Commerce search infrastructure service. As you can see in the diagram, characteristic features are widely used by other components.
Proper customization and optimization of the search interface is important for improving runtime performance.

### Optimizing the search service call

Figure 9 shows the typical flow of a SOA search service call. Typically, the JavaServer Pages (JSP) receive required content from the SOA search interface. The process uses the Expression Builder and Mediator Logic, which are both obtained through the corresponding configuration of search profile.
As demonstrated in the following example configuration, the search profile contains a `<config:query>` configuration element to build the search query, and a `<config:result>` to filter the initial set of search results:

### Listing 4. Example search profile configuration

```xml
<config:profile indexName="CatalogEntry" name="IBM_findCatalogGroupDetails">
  <config:query>
    <config:param name="maxRows" value="50"/>
    <config:param name="maxTimeAllowed" value="15000"/>
    <config:param name="preview" value="1"/>
    <config:param name="price" value="1"/>
    <config:param name="statistics" value="false"/>
    <config:provider classname="com.ibm.commerce.catalog.facade.server.services.search.expression.solr.SolrSearchIndexNameValidator"/>
    <config:provider classname="com.ibm.commerce.catalog.facade.server.services.search.expression.solr.SolrSearchIndexSynchronizer"/>
    ...
    <config:provider classname="com.ibm.commerce.catalog.facade.server.services.search.expression.solr.SolrSearchProductEntitlementExpressionProvider"/>
  </config:query>
  <config:sort/>
  <config:result>
    <config:filter classname="com.ibm.commerce.catalog.facade.server.services.search.metadata.solr.SolrSearchCatalogEntryViewPriceResultFilter"/>
    <config:filter classname="com.ibm.commerce.catalog.facade.server.services.search.metadata.solr.SolrSearchCatalogEntryViewSingleSKUResultFilter"/>
  </config:result>
  <config:highlight simplePost="</span></strong>" simplePre="<strong><span class=font2>"/>
  <config:facets>
    <config:param name="sort" value="count"/>
    <config:param name="minCount" value="1"/>
    <config:param name="limit" value="10"/>
    <config:category scope="all">
      <config:facet converter="com.ibm.commerce.catalog.facade.server.services.search.metadata.solr.SolrSearchCategoryFacetMetaDataConverter" name="parentCatgroup_id_search"/>
    </config:category>
    ...
  </config:facets>
  <config:spellcheck>
</config:profile>
```

**Figure 9. SOA search service call**

![Image of SOA search service call diagram]

---

WebSphere Commerce Search Solution Performance Best Practices

Page 20 of 33
The runtime logic dynamically invokes defined method classes with the Reflection method. Selecting and optimizing the correct methods can improve the performance of the search interface services. From the process logic of the SOA search interface, you can optimize the JSP code in any of the following ways:

- Ensure that you call the SOA search correctly. The Solr application is a high performance search engine that provides stable search response times with data sets of varying sizes. The SOA search is suitable in situations where quick search responses are required and complex semantic conditions are presented. SOA search capabilities are not usually provided by a normal database or other data sources.

The following JSP code fragment is an example of an incorrectly called SOA search that obtains a description of the subdirectory only through the call:

```jsp
<wcf:getData type="com.ibm.commerce.catalog.facade.datatypes.CatalogNavigationViewType" var="catGroupDetailsView" expressionBuilder="getCatalogNavigationCatalogGroupView">
  <wcf:param name="UniqueID" value="${catUniqueId}"/>
  <wcf:contextData name="storeId" data="${WCParam.storeId}" />
  <wcf:contextData name="catalogId" data="${WCParam.catalogId}" />
  <wcf:param name="searchProfile" value="IBM_findCatalogGroupDetails"/>
</wcf:getData>
```

In this case, you can improve performance by storing the subdirectory description in the database and retrieving it using a SQL query. However, this query does not involve complex searching. Instead, a database index and memory buffer can accelerate query performance.

- Ensure that you do not excessively use the search service. To complete a search service call, more than one query is typically needed. The Solr application relies heavily on CPU resources. If there are too many query requests, the Solr application can exhaust CPU resources and in turn, diminish overall performance. When possible, merge multiple repeated Solr query calls into a single query to improve query performance.

- Ensure that you properly customize the search service profile. For each call, the ExpressionProvider that is listed in the search profile determines the search criteria and the complexity in the resulting query. The Mediator method determines how to process the data. Carefully select the list of ExpressionProvider objects and the list of Mediator methods based on the actual requirements and the format of the returned data for each call to ensure that all calls are demand queries. Too many method calls, extra return data, and unnecessary processing logic can negatively affect performance.

- Evaluate the performance of your updated code. Since the Solr application relies heavily upon CPU resources, use performance testing or other methods to examine the overall system throughput and response time. The differences in the consumption of CPU resources and the number of requests are good indicators of the relative performance differences between builds.

- Based on the Solr query from the code, monitor the processing performance of the Solr server. From the WebSphere Application Server Administration Console on the Solr server, set the log level to * = info. Doing so writes each query as a log entry in the systemout.log
file. In each entry is the field, QTime, which is the time it takes Solr to handle an inbound search query request. Using QTime, you can identify slower queries that require improvement, and determine the Solr server's performance metrics.

- Use the Solr extensible plug-in. The Solr application is built with an open framework. To adapt to various functional requirements, Solr is provided with an extensible plug-in mechanism. Supported plug-in features include pre-processing of queries, format handling of return data, and the treatment of query results. However, unsuitable usage and unreasonable logic in plug-in applications can significantly diminish Solr server performance. An example solrconfig.xml file has the configuration:

```
<valueSourceParser name="getSequenceByCatalogAndCategory"
    class="com.ibm.commerce.foundation.internal.server.services.search.function.
    solr.SolrSearchGetSequenceByCatalogAndCategoryFunctionParser" />
```

For each query result in Solr, the configuration dynamically calculates its _val_ value to determine the degree of matching with the query to meet the functional requirements of the query sort. Because it is a dynamic calculation, the methods in the class must be processed iteratively for each record in each returned result set. As a result, performance is directly proportional to the size of the returned result set. When a large catalog is browsed in the production environment, the query might return all of the subcategories and products in the top category for each top-level category page. The number of returned results is large and the Solr query on the page is affected by the processing logic of the plug-in. In fact, when a catalog is browsed, this type of query is rarely required. You can remove the ExpressionProvider from the corresponding configuration in search profile for page browsing:

```
<config:provider
classname="com.ibm.commerce.catalog.facade.server.services.search.expression.
solr.SolrSearchSequencingExpressionProvider"/>
```

- WebSphere Commerce provides a catalog filter feature, called "contract entitlement" that filters catalogs according to different shopper segments. As the search engine foundation is introduced, the "contract entitlement" feature is enhanced, allowing it to use product attributes and properties as a filtering condition in search queries. Although the catalog filter function is much more powerful, it generates more performance overhead due to code logic. Therefore, unless a dynamic catalog filter is required, for example in a business-to-consumer store model, you can remove the newer logic and restore the older, but simpler, logic:

  a. Before you begin, ensure that you back up the configuration file:

  `<WC_DEMO.EAR>/xml/config/com.ibm.commerce.catalog-fep/wc-search.xml`

  b. Search the file for the following entries. Comment out or delete them all:

  ```
  <config:provider
  classname="com.ibm.commerce.catalog.facade.server.services.search.expression.
  solr.SolrSearchProductEntitlementExpressionProvider"/>
  <config:provider
  classname="com.ibm.commerce.catalog.facade.server.services.search.expression.
  solr.SolrSearchCategoryEntitlementExpressionProvider"/>
  ```

  c. Back up the database and delete all records that are returned by following SQL statements:
d. Restart the WebSphere Commerce application server to verify the store function.

- WebSphere Commerce, Feature Pack 5 introduced facet management, and display functions, which allowed administrators to add facet fields for catalogs to make catalog navigation more convenient. However, the addition of too many facet fields can negatively affect performance:
  
  a. **WebSphere Commerce runtime:** If catalog pages have facets fields to be displayed, the code must fetch the facet fields information, and receives the search result from the Solr engine. If the facet definition is based on attributes in the Attribute dictionary, the server must iterate through the entire dictionary to retrieve displayable information for all facets. The performance of this attribute retrieval can be improved with the data cache. For more information about tuning the performance with the Data Cache, see [Optimizing Solr native caching performance](#). You can find the used cache instances WCSearchNavigationDistributedMapCache and WCSearchAttributeDistributedMapCache in the services/cache/ directory.

  b. **Solr** When Solr processes a one-facet field, an object in which to store the mapping information of the facet field and all indexed documents is created. The more documents there are in an index, the larger an object is created, and the more slowly the object is processed. Solr caches the object to speed up the process the next time, but caching take up memory. For more information about Solr caching, see [Optimizing Solr native caching performance](#).

  c. In the case that there are too many defined facets for memory to hold, you can use the WebSphere Commerce APAR fix, JR43818. This fix might be helpful in the case that each facet field has few unique values. You can also further improve performance if you tune the Solr filterCache. After you apply the APAR fix, tune the Solr filterCache according to the best practice equation:

  \[
  \text{New filterCache size} = \text{Original filterCache size} + \text{Number of Facets} \times \text{Number of values for each facet}
  \]

  Ensure that you also tune the spell check index. There might be a decrease in performance if you enable spell checking for WebSphere Commerce search terms. You might see a performance improvement in transaction throughput if spell checking is skipped where necessary, or when a customer searches for products with catalog overrides. For example, a search term that is submitted in a different language than the storefront requires extra resources for spell checking. However, product names with catalog overrides are already known and do not require any resources for spell checking. The spell check index ensures that automatically suggested search terms accurately reflect the terms in the search index. It is built automatically during commits (build index and replication), including target search nodes in a clustered environment. The automatic build is defined in the solrconfig.xml file:
<lst name="spellchecker">
  <str name="name">default</str>
  <str name="field">spellCheck</str>
  <str name="spellcheckIndexDir">spellchecker</str>
  <str name="classname">solr.IndexBasedSpellChecker</str>
  <str name="field">spellCheck</str>
  <str name="buildOnCommit">true</str>
  <str name="buildOnOptimize">true</str>
  <str name="spellcheckIndexDir">./spellchecker</str>
</lst>

If you run frequent delta updates during the day, you might notice high CPU usage on your search target servers. If the CPU usage is excessive, you can set the `buildOnCommit` parameter to `false`, and then manually trigger a build of the spellcheck index for a specific index by using the following command:

```
http://host_name:search_port/solr/MC_masterCatalogId_CatalogEntry_locale/select?
q=query&spellcheck=true&spellcheck.collate=true&spellcheck.build=true
```

**Optimizing Commerce JSP caching performance**

The data cache usage of the WebSphere Commerce search framework, is similar to the data usage of other components. Generally speaking, the complexity of the code logic, data reusability, and the correct functionality with the cache invalidation mechanism determines what can be placed in the cache. To improve performance, use the following order of priority when you consider the contents in the cache:

**Whole page > Page fragment > Java Command cache > Data cache**

---

**Whole page or page fragment caching in new search features**

The `cachespec.xml` file contains a default sample configuration for dynamic caching with store pages. Depending the store type, you can find the `cachespec.xml` file in the `<WC_INSTALL>/samples/dynacache/<STORE_TYPE>/` directory. The focus is on store, catalog, category, and product page caching because the cache on such pages has good reusability. If you need to cache the results of a keyword search, ensure that the content of the keyword cache has a small chance of being reused. Instead, consider customizing the dynamic cache for hot keywords that are being cached only.

Keyword search and facet filtering introduces a large variance in the combinations of attributes and URL parameters. Because of the variance of keywords and there being few facet selections, the reuse might be low for pages, even if sufficient space is available for caching all possible combinations. Caching at a lower level, as opposed to a full page, is where you can attempt to improve the cache hit ratio. For example, you can cache the product thumbnails individually so that the thumbnails can be reused across different facet selections. Caching product thumbnails individually is reusable and is highly suggested for optimizing caching for facets. In Figure 10, the following products are displayed in the initial load of the Women's Dress category. Ensure that you cache each product thumbnail individually:
Figure 10. Products shown in Category View

Figure 11. Filtered by a particular brand

In Figure 12, you can see that the filtered view displays many of the same product thumbnails in the category page, proving that individually cached products can be reused:
Optimizations that can improve the cache reuse of product thumbnails and reduce service demand include:

- Listing products at the thumbnail level
- Limiting cache keys that are required for the thumbnails
- Optimizing the number of product attributes that are shown in the thumbnails to minimize system costs

For example, the JSP implementation of an out-of-the-box store uses `CatalogEntryDisplay.jsp` to display product thumbnail fragments. The cache policy can be defined as following the code in Listing 5 to cache and reuse the fragment in other pages. The entirety of the `cache-id` configuration is not shown here:
Listing 5. Customized cache policy

```xml
<cache-entry>
  <class>servlet</class>
  <name>/Aurora/Widgets/CatalogEntry/CatalogEntryDisplay.jsp</name>
  <property name="do-not-consume">true</property>
  <property name="save-attributes">false</property>
  <property name="ignore-get-post">true</property>
  <property name="consume-subfragments">true</property>
  <cache-id>
    <component id="storeId" type="parameter">
      <required>true</required>
    </component>
    <component id="catalogId" type="parameter">
      <required>true</required>
    </component>
    <component id="langId" type="parameter">
      <required>true</required>
    </component>
  </cache-id>
...  
</cache-entry>
```

**Note:** You can further customize this cachespec.xml entry if you set `do-not-consume` to `false` to consume `CatalogEntryDisplay.jsp` within its parent cache entries. This reduces the number of fetches of the cache entry at the cost of increasing the occurrences of invalidation ID build-ups. Ensure that you test and validate the optimal configuration for your specific workload, volume, and topology.

### Optimizing Solr native caching performance

**Search data cache**

The search data cache differs from other WebSphere Commerce data caches because the data comes from the Solr search engine. The search data cache is defined in the Solr core configuration file, `solrconfig.xml`. In general, if the memory on the Solr server is large enough, greater cache settings yield better performance. For more information about tuning for an increase in cache sizing, see Optimizing the JVM parameters.

**Filter cache**

Solr uses query filters to narrow the search range to improve search performance. For each filter, the results can be placed in a dedicated cache. This dedicated cache uses the filter query as a cache keyword filter query so that the same query gets results quickly by cache hit. For example, see the following fragment in the `solrconfig.xml` configuration file:

```xml
<filterCache class="solr.FastLRUCache" size="8192" initialSize="4096" autowarmCount="0"/>
```

**Query cache**

The Query cache caches the resulting sets of a cache. Because cache contents are the value of the object ID of the Solr document, it has a small memory footprint. You can set the cache size to a large number for better performance. For example, see the following fragment in the configuration file:

```xml
<queryResultCache class="solr.LRUCache" size="8192" initialSize="4096" autowarmCount="0"/>
```
Field Value cache

The Field Value cache is important for facet calculation performance. The calculated result is cached for each facet field. The size of each entry in this cache depends on the index size, as in following equation:

\[
\text{Field value cache entry size} = 4 \times B \times \text{Number of Indexed documents}
\]

**Note:** Although this cache instance is not defined explicitly in the `solrconfig.xml` file, the Solr server creates a cache instance with a hard-coded size. When many facets are defined in site, the cache with a hard-coded size leads to out-of-memory errors easily. Ensure that you consider the quantity of possible facets and tune the cache instance in the configuration file accordingly:

```xml
<fieldValueCache class="solr.FastLRUCache" size="512" showItems="32" autowarmCount="129"/>
```

Document Object Cache

The Document Object cache caches the contents of a document object. If the document object is large and contains much data, you can use this cache to reduce the time to fetch complete data from the search index. For example, see the following fragment in the configuration file:

```xml
<documentCache class="solr.LRUCache" size="8192" initialSize="4096" autowarmCount="0"/>
```

Custom Cache

If you are able to customize Solr, you can add a custom user-defined cache to the configuration file. For example:

```xml
<cache name="myUserCache" class="solr.LRUCache" size="8192" initialSize="4096" autowarmCount="0" />
```

Other performance considerations

In WebSphere Commerce, Feature Pack 7 there are other new features that were introduced that rely on the search framework and might also need to be optimized for improved performance:

Catalog filter

The catalog filter is a new feature in the Contract components. The catalog filter selects catalog data by managing the Solr query. It adds an "Include" or "Exclude" option to the category or product nodes with extra properties or descriptions to refine filter criteria.

Be careful when you add conditions with the catalog filter because too many conditions can affect query response times. Also, avoid adding lengthy additional conditions because they might exceed the length limitation on the HTTP server. Usually a reasonable number of nodes is approximately 50. At the same time, to reduce the computational load on filters, you can increase the size of the corresponding Java Command cache. Refer the following `cachespec.xml` files in the following directories:

- `<WC_INSTALL>/samples/dynacache/Contract/`
Search rules

Search rules is a new type of marketing function that provides some flexible recommendations based on customers' search keywords or the search result set. Like the catalog filter, too many rules or rules that are too complex can negatively affect page response performance.

Pay particular attention to search rules that contain a "Search Criteria And Result" element. In the returned result set, the "Search Criteria And Result" element triggers the Marketing component to call the search SOA to refine results. The logic uses complex data and might generate multiple queries, depending on its configuration. When there are multiple similar rules, the number of Solr queries is increased and often leads to a scalability issue based on the number of rules. This has a great impact on performance. Where possible, ensure that you avoid or reduce these types of search rules.

**Figure 13. Example of the search rule**

![Diagram of search rule](image)

Optimize Search Server in Feature Pack 7

Search Server is a new component that is introduced in WebSphere Commerce, Feature Pack 7. Search Server is a new WebSphere Application Server runtime instance with an embedded Solr runtime, which includes a published RESTful based API to interact with the WebSphere Commerce runtime instance. It de-couples the store pages into the presentation layer and business logic layer of Search solution. Since the Feature Pack 7 release, the B2C store component pages uses the RESTful API to retrieve necessary data from the Search Server instance and present it to an online store visitor. For more information about the details of Search Server, see the Feature Pack 7 related content of Search Server administration in the WebSphere Commerce IBM Knowledge Center.

Important tuning parameters

An important performance tuning point in Search Server is to control the response timeout value when a WebSphere Commerce runtime instance interacts with a Search Server instance. Because it is a bidirectional interaction, there are two timeout values that can be tuned:

1. **A request from Search Server to the WebSphere Commerce instance:** In the foundation component configuration file on the Search Server, the REST-related properties are within the `<config:configgrouping name="CommerceServerSetting">` element. The following table summarizes the types of configurable properties in the file; refer to the comments within the file for more details as below table.
Table 1. Foundation Component configuration file containing REST properties

<table>
<thead>
<tr>
<th>Section</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timeout settings</td>
<td><strong>RemoteRestRequestConnectTimeout</strong> The timeout value that is used to create an HTTP connection from the search server to the WebSphere Commerce server for REST requests.</td>
</tr>
<tr>
<td></td>
<td><strong>RemoteRestRequestReadTimeout</strong> The timeout value that is used to terminate a REST request from the search server to the WebSphere Commerce server.</td>
</tr>
</tbody>
</table>

To change any properties in the component configuration file for the REST API, you must create a customized version of the `wc-component.xml` file in a new folder. The customized version of the file must contain only the changed properties. For more information about detailed steps to tune this parameter, see Changing REST configuration properties in the component configuration file (wc-component.xml) in the WebSphere Commerce IBM Knowledge Center.

2. **The request is from Commerce instance to Search Server:** Such requests are mainly used in the store pages, which initial the RESTful request against Search Server in the JSP page code. In the main release code, the timeout value is hard-coded in the JSP files. APAR JR50713 was issued to make the timeout value configurable. APAR JR50713 adds the capability to tune the timeout value of RESTful requests initiated against the Solr server to a WebSphere Commerce instance. To tune, open the `wc-component.xml` file in the foundation component and change the values of the parameters in Table 1.

**Disable remote Business Context Service (BCS) call**

When catalog filter and REST services entitlement checks are enabled, the remote BCS call can be used to validate contracts based on the value of `runAsId`. The BCS call then triggers a REST call to WebSphere Commerce. Therefore, a page visit to WebSphere Commerce might trigger a BCS callback to WebSphere Commerce from the Search Server. Because all web interactions are handled by web container threads, more than one thread is required to handle one page visit, which leads to the resource constraint and competition at a high concurrent workload. You can also use local contract validation on the search server to disable remote BCS callbacks. For more information about how to disable a remote BCS call, see Disabling the remote Business Context Service (BCS) call in the WebSphere Commerce IBM Knowledge Center.

**Recommended APARs**

There are additional APARs for WebSphere Commerce, Feature Pack 7 to further improve performance. View the APAR details and adopt the appropriate fixes for your particular environment:

- **WebSphere Commerce Version 7 Feature Pack 7 Cumulative1 Interim Fix (JR50553)** is a cumulative interim fix for Feature Pack 7 with a prerequisite APAR of Fix Pack 8, **WebSphere Commerce Version 7 Fix Pack 8 Cumulative1 Interim Fix (JR50553)**. For more information
about APAR content details, see Cumulative interim fix to improve store performance in the WebSphere Commerce IBM Knowledge Center.
Resources

- Participate in the discussion forum for this content.
- For more information about WebSphere Commerce, see WebSphere Commerce Version 7 on the IBM Knowledge Center.
- For more information about the Apache Solr search engine internals, see the Apache Solr Wiki.
About the authors

Yang Gao
Yang Gao is a Software Engineer in the IBM China Development Lab. Yang has five years of experience on e-commerce product system testing and performance optimization.

Kevin Yu
Kevin Yu is the Industry Solutions Performance Architect for Smarter Commerce. Kevin works on application servers, enterprise commerce, and database products. Kevin led multiple multi-disciplinary teams to solve critical customer situations and to achieve new records in transaction volumes and sales on their busiest online shopping days of the year.

© Copyright IBM Corporation 2015
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)