**SEARCH – QUERIES & INDEXING**

Along with the transition to Oak in AEM 6, some major changes were made to the way that queries and indexes are managed.  Under Jackrabbit 2, all content was indexed by default and could be queried freely.  In Oak, indexes must be created manually under the **oak:index** node.  A query can be executed without an index, but for large datasets, it will execute very slowly, or even abort.

This article will outline when to create indexes as well as when they are not needed, tricks to avoid using queries when they are not necessary, and tips for optimizing your indexes and queries to perform as optimally as possible.

Additionally, make sure to read the [Oak documentation on writing queries and indexes](https://helpx.adobe.com/experience-manager/6-3/sites/deploying/using/queries-and-indexing.html).  In addition to indexes being a new concept in AEM 6, there are syntactical differences in Oak queries that need to be taken into account when migrating code from a previous AEM installation.

**When to Use Queries**

**Repository and Taxonomy Design**

When designing the taxonomy of a repository, several factors need to be taken into account. These include access controls, localization, component and page property inheritance among others.

While designing a taxonomy that addresses these concerns, it is also important to consider the "traversability" of the indexing design. In this context, the traversability is the ability of a taxonomy that allows content to be predictably accessed based on its path. This will make for a more performant system that is easier to maintain than one that will require a lot of queries to be executed.

Additionally, when designing a taxonomy, it is important to consider whether ordering is important. In cases where explicit ordering is not required and a large number of sibling nodes are expected, it is preferred to use an unordered node type such as **sling:Folder** or **oak:Unstructured**. In cases where ordering is required, **nt:unstructured** and **sling:OrderedFolder** would be more appropriate.

**Queries in Components**

Since queries can be one of the more taxing operations done on an AEM system, it is a good idea to avoid them in your components. Having several queries execute each time a page is rendered can often degrade the performance of the system. There are two strategies that can be used to avoid executing queries when rendering components: **traversing nodes** and **prefetching results**.

**Traversing Nodes**

If the repository is designed in such a way that allows prior knowledge of the location of the required data, code that retrieves this data from the necessary paths can be deployed without having to run queries in order to find it.

An example of this would be rendering content that fits within a certain category. One approach would be to organize the content with a category property that can be queried to populate a component that shows items in a category.

A better approach would be to structure this content in a taxonomy by category so that it can be manually retrieved.

For example, if the content is stored in a taxonomy similar to:

|  |  |
| --- | --- |
|  | **/content/myUnstructuredContent/parentCategory/childCategory/contentPiece** |

the **/content/myUnstructuredContent/parentCategory/childCategory** node can simply be retrieved, its children can be parsed and used to render the component.

Additionally, when you are dealing with a small or homogenous result set, it can be faster to traverse the repository and gather the required nodes, rather than crafting a query to return the same result set. As a general consideration, queries should be avoided where it is possible to do so.

**Prefetching Results**

Sometimes the content or the requirements around the component will not allow the use of node traversal as a method of retrieving the required data. In these cases, the required queries need to be executed before the component is rendered so that optimal performance is ensured for the end user.

If the results that are required for the component can be calculated at the time that it is authored and there is no expectancy that the content will change, the query can be executed when the author applies settings in the dialog.

If the data or content will change regularly, the query can be executed on a schedule or via a listener for updates to the underlying data. Then, the results can be written to a shared location in the repository. Any components that need this data can then pull the values from this single node without needing to execute a query at runtime.

**Query Optimization**

When running a query that is not using an index, warnings will be logged regarding node traversal. If this is a query that is going to be run often, an index should be created. To determine which index a given query is using, the [Explain Query tool](https://helpx.adobe.com/experience-manager/6-3/sites/administering/using/operations-dashboard.html#main-pars_title_1097830066) is recommended.  For additional information, DEBUG logging can be enabled for the relevant search APIs.

**Note:**

After modifying an index definition, the index needs to be rebuilt (reindexed). Depending on the size of the index, this may take some time to complete.

When running complex queries, there may be cases in which breaking down the query into multiple smaller queries and joining the data through code after the fact is more performant. The recommendation for these cases is to compare performance of the two approaches to determine which option would be better for the use case in question.

AEM allows writing queries in one of three ways:

* Via the [QueryBuilder APIs](https://helpx.adobe.com/experience-manager/6-3/sites/developing/using/querybuilder-api.html) (recommended)
* Using XPath (recommended)
* Using SQL2

While all queries are converted to SQL2 before being run, the overhead of query conversion is minimal and thus, the greatest concern when choosing a query language will be readability and comfort level from the development team.

**Note:**

When using QueryBuilder, it will determine the result count by default, which is slower in Oak as compared to previous versions of Jackrabbit. To compensate for this, you can use the [guessTotal parameter](https://helpx.adobe.com/experience-manager/6-3/sites/developing/using/querybuilder-api.html" \l "main-pars_title_2032974994).

**The Explain Query Tool**

As with any query language, the first step to optimizing a query is to understand how it will be executed. To enable this activity, you can use the [Explain Query tool](https://helpx.adobe.com/experience-manager/6-3/sites/administering/using/operations-dashboard.html#main-pars_title_1097830066) that is part of the Operations Dashboard.  With this tool, a query can be plugged in and explained. A warning will be shown if the query will cause issues with a large repository as well as execution time and the indexes that will be used. The tool can also load a list of slow and popular queries that can then be explained and optimized.

**DEBUG Logging for Queries**

To get some additional information about how Oak is choosing which index to use and how the query engine is actually executing a query, a **DEBUG** logging configuration can be added for the following packages:

* org.apache.jackrabbit.oak.plugins.index
* org.apache.jackrabbit.oak.query
* com.day.cq.search

Make sure to remove this logger when you have finished debugging your query as it will output a lot of activity and can eventually fill up your disk with log files.

For more information on how to do this, see the [Logging documentation](https://helpx.adobe.com/experience-manager/6-3/sites/deploying/using/configure-logging.html).

**Index Statistics**

Lucene registers a JMX bean that will provide details about indexed content including the size and number of documents present in each of the indexes.

You can reach it by accessing the JMX Console at **http://server:port/system/console/jmx**

Once you are logged in to the JMX console, perform a search for **Lucene Index Statistics** in order to find it. Other index statistics can be found in the **IndexStats** MBean.

For query statistics, take a look at the MBean named **Oak Query Statistics**.

If you would like to dig into your indexes using a tool like [Luke](https://code.google.com/p/luke/), you will need to use the Oak console to dump the index from the **NodeStore** to a filesystem directory. For instructions on how to do this, please read the [Lucene documentation](http://jackrabbit.apache.org/oak/docs/query/lucene.html).

You can also extract the indexes in your system in JSON format. In order to do this, you need to access **http://server:port/oak:index.tidy.-1.json**

**Query Limits**

**During Development**

Set low threshholds for **oak.queryLimitInMemory** (eg. 10000) and oak.**queryLimitReads** (eg. 5000) and optimize the expensive query when hitting an UnsupportedOperationException saying “The query read more than x nodes..."

This helps avoiding resource intensive queries (ie. not backed by any index or backed by less covering index). For example, a query that reads 1 million nodes would lead to increased I/O, and negatively impact the overall application performance. Any query that fails due to above limits should be analyzed and optimized.

**Post-Deployment**

* Monitor the logs for queries triggering large node traversal or large heap memory consumption :
  + **\*WARN\* ... java.lang.UnsupportedOperationException: The query read or traversed more than 100000 nodes. To avoid affecting other tasks, processing was stopped.**
  + Optimize the query to reduce the number of traversed nodes
* Monitor the logs for queries triggering large heap memory consumption :
  + **\*WARN\* ... java.lang.UnsupportedOperationException: The query read more than 500000 nodes in memory. To avoid running out of memory, processing was stopped**
  + Optimize the query to reduce the heap memory consumption

For AEM 6.0 - 6.2 versions, you can tune the threshold for node traversal via JVM parameters in the AEM start script to prevent large queries from overloading the environment.

The recommended values are :

* **-Doak.queryLimitInMemory=500000**
* **-Doak.queryLimitReads=100000**

In AEM 6.3, the above 2 parameters are preconfigured OOTB, and can be persisted via the OSGi QueryEngineSettings.

More information available under : http://jackrabbit.apache.org/oak/docs/query/query-engine.html#Slow\_Queries\_and\_Read\_Limits

**Tips for Creating Efficient Indexes**

**Should I Create an Index?**

The first question to ask when creating or optimizing indexes is whether they are really required for a given situation. If you are only going to run the query in question once or only occasionally and at an off-peak time for the system through a batch process, it may be better to not create an index at all.

After creating an index, every time the indexed data is updated, the index must be updated as well.  Since this carries performance implications for the system, indexes should only be created when they are actually required.

Additionally, indexes are only useful if the data contained within the index is unique enough to warrant it. Consider an index in a book and the topics that it covers. When indexing a set of topics in a text, usually there will be hundreds or thousands of entries, which allows you to quickly jump to a subset of pages to quickly find the information that you are looking for. If that index only had two or three entries, each pointing you to several hundred pages, the index would not be very useful. This same concept applies to database indexes. If there are only a couple unique values, the index will not be very useful.  That being said, an index can also become too large to be useful as well. To look at index statistics, see [Index Statistics](https://helpx.adobe.com/experience-manager/6-3/sites/deploying/using/best-practices-for-queries-and-indexing.html#main-pars_title_1472859131) above.

**Lucene or Property Indexes?**

Lucene indexes were introduced in Oak 1.0.9 and offer some powerful optimizations over the property indexes that were introduced in the initial launch of AEM 6. When deciding whether to use Lucene indexes or property indexes, take the following into consideration:

* Lucene indexes offer many more features than property indexes. For example, a property index can only index a single property while a Lucene index can include many. For more information on all of the features available in Lucene indexes, consult the [documentation](http://jackrabbit.apache.org/oak/docs/query/lucene.html).
* Lucene indexes are asynchronous. While this offers a considerable performance boost, it can also induce a delay between when data is written to the repository and when the index is updated. If it is vital to have queries return 100% accurate results, a property index would be required.
* By virtue of being asynchronous, Lucene indexes cannot enforce uniqueness constraints.  If this is required, then a property index will need to be put in place.

In general, it is recommended you use Lucene indexes unless there is a compelling need to use property indexes so that you can gain the benefits of higher performance and flexibility.

**Solr Indexing**

AEM also provides support for Solr indexing by default. This is mainly leveraged to support full text search, but it can also be used to support any type of JCR query. Solr should be considered when the AEM instances do not have the CPU capacity to handle the number of queries required in search intensive deployments like search driven websites with a high number of concurrent users. Alternately, Solr can be implemented in a crawler based approach to leverage some of the more advanced features of the platform.

Solr indexes can be configured to run embedded on the AEM server for development environments or can be offloaded to a remote instance to improve search scalability on the production and staging environments. While offloading search will improve scalability, it will introduce latency and because of this, is not recommended unless required. For more info on how to configure Solr integration and how to create Solr indexes see the [Oak Queries and Indexing documentation](https://helpx.adobe.com/experience-manager/6-3/sites/deploying/using/queries-and-indexing.html#main-pars_title_5).

**Note:**

While taking the integrated Solr search approach would allow for offloading of indexing to a Solr server. If the more advanced features of the Solr server are used through a crawler based approach, additional configuration work will be required. Headwire has created an [open source connector](http://www.aemsolrsearch.com/#/) to accelerate these types of implementations.

The downside to taking this approach is that while by default, AEM queries will respect ACLs and thus hide results that a user does not have access to, externalizing search to a Solr server will not support this feature. If search is to be externalized in this way, extra care must be taken to ensure that users are not presented with results that they should not see.

Potential use cases where this approach may be appropriate are cases where search data from multiple sources may need to be aggregated. For instance, you may have a site being hosted on AEM as well as a second site being hosted on a third party platform. Solr could be configured to crawl the content of both sites and store them in an aggregated index. This would allow for cross-site searches.

**Design Considerations**

The Oak documentation for Lucene indexes lists several considerations to make when designing indexes:

* If the query uses different path restrictions, utilize **evaluatePathRestrictions**. This will allow the query to return the subset of results under the path specified and then filter them based on the query. Otherwise, the query will search for all results that match the query parameters in the repository and then filter them based on the path.
* If the query uses sorting, have an explicit property definition for the sorted property and set **ordered** to **true** for it. This will allow the results to be ordered as such in the index and save on costly sorting operations at query execution time.
* Only put what is needed into the index. Adding unneeded features or properties will cause the index to grow and slow performance.
* In a property index, having a unique property name would help to reduce the size on an index, but for Lucene indexes, use of **nodeTypes** and **mixins** should be made to achieve cohesive indexes. Querying a specific **nodeType** or **mixin** will be more performant than querying **nt:base**. When using this approach, define **indexRules** for the **nodeTypes** in question.
* If your queries are only being run under certain paths, then create those indexes under those paths. Indexes are not required to live at the root of the repository.
* It is recommended to use a single index when all of the properties being indexed are related to allow Lucene to evaluate as many property restrictions as possible natively. Additionally, a query will only use one index, even when performing a join.

**CopyOnRead**

In cases where the **NodeStore** is stored remotely, an option called **CopyOnRead** can be enabled. The option will cause the remote index to be written to the local filesystem when it is read. This can help to improve performance for queries that are often run against these remote indexes.

This can be configured in the OSGi console under the **LuceneIndexProvider** service and is enabled by default as of Oak 1.0.13.

**Removing Indexes**

When removing an index, it is always recommended to temporarily disable the index by setting the **type** property to **disabled**and do testing to ensure that your application functions correctly before actually deleting it. Note that an index is not updated while disabled, so it may not have the correct content if it is reenabled and may need to be reindexed.

After removing a property index on a TarMK instance, compaction will need to be run to reclaim any disk space that was in use. For Lucene indexes, the actual index content lives in the BlobStore, so a data store garbage collection would be required.

When removing an index on a MongoDB instance, the cost of deletion is proportional to the number of nodes in the index. Since deleting a large index can cause problems, the recommended approach is to disable the index and delete it only during a maintenance window, using a tool such as **oak-mongo.js**. Please note that this approach should not be employed for regular node content as it can introduce data inconsistencies.

**Note:**

For more information about oak-mongo.js, see the [Command Line Tools section](http://jackrabbit.apache.org/oak/docs/command_line.html) of the Oak documentation.

**Re-indexing**

This section outlines the **only** acceptable reasons to re-index Oak indexes.

Outside the reasons outlined below, initiating re-indexes of Oak indexes will **not** change behavior or resolve issues, and unncessarily increase load on AEM.

Re-indexing of Oak indexes is to be avoided unless covered by a reasons in the tables below.

**Note:**

Prior to consulting the tables below to determine is re-indexing is useful,**always**verify:

* the query is correct
* the query resolves to the expected index (using [Explain Query](https://helpx.adobe.com/experience-manager/6-3/sites/administering/using/operations-dashboard.html#Diagnosistools))
* the indexing process has completed

**Oak index configuration changes**

The only acceptable non-erring conditions for re-indexing Oak indexes, is if the configuration of an Oak index has changed.

*Re-indexing should always be approached with proper consideration on its impact to AEM's overall performance, and performed during periods of low activity or maintenance windows.*

The following detail possible issues together with resolutions:

* [Property Index Definition Change](https://helpx.adobe.com/experience-manager/6-3/sites/deploying/using/best-practices-for-queries-and-indexing.html#PropertyIndexDefinitionChange)
* [Lucene Index Definition Change](https://helpx.adobe.com/experience-manager/6-3/sites/deploying/using/best-practices-for-queries-and-indexing.html#LuceneIndexDefinitionChange)

**Property Index Definition Change**

* Applies for/if:
  + All Oak versions
  + Only [property indexes](http://jackrabbit.apache.org/oak/docs/query/property-index.html)
* Symptoms:
  + Nodes existing prior to property index's definition update missing from results
* How to Verify:
  + Determine if missing nodes were created/modified prior to the deployment of the updated index definition.
  + Verify the **jcr:created** or **jcr:lastModified** properties of any missing nodes against the index's modified time
* How to Resolve:
  + [Re-index](https://helpx.adobe.com/experience-manager/6-3/sites/deploying/using/best-practices-for-queries-and-indexing.html?cq_ck=1496173166571#HowtoReindex) the lucene index
  + Alternatively, touch (perform a benign write operation) to the missing nodes
    - Requires manual touches or custom code
    - Requires the set of missing nodes to be known
    - Requires changing any property on the node

**Lucene Index Definition Change**

* Applies for/if:
  + All Oak versions
  + Only [lucene indexes](http://jackrabbit.apache.org/oak/docs/query/lucene.html)
* Symptoms:
  + Lucene index does not contain expected results
  + Query results do not reflect expected behavior of index definition
  + Query plan does not report expected output based on index definition
* How to Verify:
  + Verify the index definition was changed using the Lucene Index statistics JMX Mbean (LuceneIndex), method **diffStoredIndexDefinition**.
* How to Resolve:
  + Oak versions prior to 1.6:
    - [Re-index](https://helpx.adobe.com/experience-manager/6-3/sites/deploying/using/best-practices-for-queries-and-indexing.html#HowtoReindex) the lucene index
  + Oak versions 1.6+
    - If existing content is not effected by the changes then only a refresh is needed
      * [Refresh](http://jackrabbit.apache.org/oak/docs/query/lucene.html#stored-index-definition) the lucene index by setting [oak:queryIndexDefinition]@refresh=true
    - Else, [re-index](https://helpx.adobe.com/experience-manager/6-3/sites/deploying/using/best-practices-for-queries-and-indexing.html#HowtoReindex) the lucene index
      * Note: The index state from the last good re-indexing (or initial indexing) will be used until a new re-indexing is triggered

**Erring and exceptional situations**

The following table describes the only acceptable erring and exceptional situations where re-indexing Oak indexes will resolve the issue.

If an issue is experienced on AEM that does not match the criteria outlined below, do **not** re-index any indexes, as it will not resolve the issue.

The following detail possible issues together with resolutions:

* [Lucene Index Binary is Missing](https://helpx.adobe.com/experience-manager/6-3/sites/deploying/using/best-practices-for-queries-and-indexing.html#LuceneIndexBinaryisMissing)
* [Lucene Index Binary is Corrupt](https://helpx.adobe.com/experience-manager/6-3/sites/deploying/using/best-practices-for-queries-and-indexing.html#LuceneIndexBinaryisCorrupt)

**Lucene Index Binary is Missing**

* Applies for/if:
  + All Oak versions
  + Only [lucene indexes](http://jackrabbit.apache.org/oak/docs/query/lucene.html)
* Symptoms:
  + Lucene index does not contain expected results
* How to Verify:
  + The error log file contains an exception saying a binary of the Lucene index is missing
* How to Resolve:
  + Perform a traversing repository check; for example:  
    <http://localhost:4502/system/console/repositorycheck>  
    traversing the repository determines if other binaries (besides lucene files) are missing
  + If binaries other than lucene indexes are missing, restore from backup
  + Otherwise, [re-index](https://helpx.adobe.com/experience-manager/6-3/sites/deploying/using/best-practices-for-queries-and-indexing.html#HowtoReindex) *all* lucene indexes
  + Note:  
    This condition is indicative of a misconfigured datastore that may result in ANY binary (eg. assets binaries) to go missing.  
    In this case, restore to the last known good version of the repository to recover all missing binaries.

**Lucene Index Binary is Corrupt**

* Applies for/if:
  + All Oak versions
  + Only [lucene indexes](http://jackrabbit.apache.org/oak/docs/query/lucene.html)
* Symptoms:
  + Lucene index does not contain expected results
* How to Verify:
  + The **AsyncIndexUpdate** (every 5s) will fail with an exception in the error.log:  
     **...a Lucene index file is corrupt...**
* How to Resolve:
  + Remove the local copy of the lucene index
    1. Stop AEM
    2. Delete the local copy of the lucene index at **crx-quickstart/repository/index**
    3. Restart AEM
  + If this does not resolve the issue, and the **AsyncIndexUpdate** exceptions persist then:
    1. [Re-index](https://helpx.adobe.com/experience-manager/6-3/sites/deploying/using/best-practices-for-queries-and-indexing.html#HowtoReindex) the erring index
    2. Also file an [Adobe Support](https://helpx.adobe.com/support.html) ticket

**How to Re-index**

**Re-indexing Property indexes**

* Set the async-reindex property to true on the property index  
  + [oak:queryIndexDefinition]@reindex-async=true
* Re-index the property index asyncronously using the Web Console via the **PropertyIndexAsyncReindex**MBean;  
  for example,  
  <http://localhost:4502/system/console/jmx/org.apache.jackrabbit.oak%3Aname%3Dasync%2Ctype%3DPropertyIndexAsyncReindex>

**Re-indexing Lucene Property indexes**

* AEM's [Index Manager](https://helpx.adobe.com/experience-manager/6-3/sites/administering/using/operations-dashboard.html#TheIndexManager)
* Setting **[oak:queryIndexDefinition]@reindex=true** (typically via CRXDE Lite or Sling POST Servlet)

**Note:**

The preceding section summarizes and frames the Oak re-indexing guidance from the [Apache Oak documentation](http://jackrabbit.apache.org/oak/docs/query/indexing.html#reindexing) in the context of AEM.

**Text Pre-extraction of Binaries**

Text pre-extraction is the process of extracting and processing text from binaries, directly from the Data Store via an isolated process, and directly exposing the extracted text to subsequent re/indexings of Oak indexes.

* Oak text pre-extraction is recommended for re/indexing Lucene indexes on repositories with large volumes of files (binaries) that contain extractable text (eg. PDFs, Word Docs, PPTs, TXT, etc.) that qualify for full-text search via deployed Oak indexes; for example **/oak:index/damAssetLucene**.
* Text pre-extraction will only benefit the re/indexing of Lucene indexes, and NOT Oak property indexes, since property indexes do not extract text from binaries.
* Text pre-extraction has a high positive impact when the full-text re-indexing of text-heavy binaries (PDF, Doc, TXT, etc.), where as a repository of images will not enjoy the same efficiencies since images do not contain extractable text.
* Text pre-extraction performs the extraction of full-text search related text in a extra-efficient manner, and exposes it to the Oak re/indexing process in a way that is extra-efficient to consume.

**When CAN text pre-extraction be used?**

Re-indexing an **existing** lucene index with binary extraction enabled

* Re-indexing processing **all** candidate content in the repository; when the binaries to extract full-text from are numerous or complex, an increased computational burden to perform the the full-text extraction is placed on AEM. Text pre-extraction moves the "computationally costly work" of text-extraction into an isolated process that directly accesses AEM's Data Store, avoiding overhead and resource contention in AEM.

Supporting the deployment of a **new** lucene index to AEM with binary extraction enabled

* When a new index (with binary extraction enabled) is deployed into AEM, Oak automatically indexes all candidate content on the next async full-text index run. For the same reasons described in re-indexing above, this may result in undue load on AEM.

**When can text pre-extraction NOT be used?**

Text pre-extraction cannot be used for new content added to the repository, nor is it necessary.

New content is added to the repository will naturally and incrementally be indexed by the async full-text indexing process (by default, every 5 seconds).

Under normal operation of AEM, for example uploading Assets via the Web UI or programmatic ingest of Assets, AEM will automatically and incrementally full-text index the new binary content. Since the amount of data is incremental and relatively small (approximately the amount of data that can be persisted to the repository in 5 seconds), AEM can perform the full-text extraction from the binaries during indexing without effecting overall system performance.

**Prerequisites to using text pre-extraction**

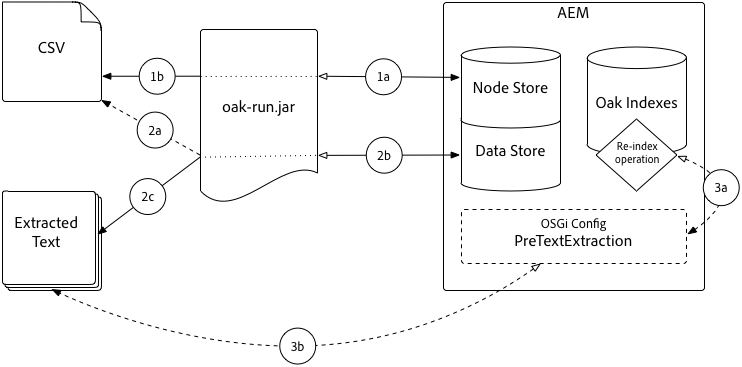
* You will be re-indexing a lucene index that performs full-text binary extraction or deploying a new index that will full-text index binaries of existing content
* The content (binaries) from which to pre-extract text, must be in the repository
* A maintenance window to generate the CSV file AND to perform the final re-indexing
* Oak version: 1.0.18+, 1.2.3+
* [oak-run.jar](http://mvnrepository.com/artifact/org.apache.jackrabbit/oak-run/)version 1.7.4+
* A file system folder/share to store extracted text accessible from the indexing AEM instance(s)
  + The Text pre-extraction OSGi config requires a file system path to the extracted text files, so they must be accessible directly from the AEM instance (local drive or file share mount)

**How to perform text pre-extraction**

**Note:**

***The oak-run.jar commands outlined below are fully enumerated at***[***https://jackrabbit.apache.org/oak/docs/query/pre-extract-text.html***](https://jackrabbit.apache.org/oak/docs/query/pre-extract-text.html)

The above diagram and steps below serve to explain and compliment the technical text pre-extraction steps outlined in the Apache Oak documentation.



**Generate list of content to pre-extract**

*Execute Step 1(a-b) during a maintenance window/low-use period as the Node Store is traversed during this operation, which may incur significant load on the system.*

1a. Execute **oak-run.jar --generate** to create a list of nodes that will have their text pre-extracted.

1b. List of nodes (1a) is stored to the file system as a CSV file

Note the entire Node Store is traversed (as specified by the paths in the oak-run command) every time **--generate** is executed, and a **new** CSV file is created. The CSV file is **not** re-used between discrete executions of the text pre-extraction process (Steps 1 - 2)

**Pre-extract text to file system**

*Step 2(a-c) can be executed during normal operation of AEM is it only interacts w the Data Store.*

2a. Execute **oak-run.jar --tika** to pre-extract text for the binary nodes enumerated in the CSV file generated in (1b)

2b. The process initiated in (2a) accesses binary nodes defined in the CSV in Data Store directly, and extracts text.

2c. Extracted text is stored on file system in a format ingestible by the Oak re-indexing process (3a)

Pre-extracted text is identified in the CSV by the binary fingerprint. If the binary file is the same, the same pre-extracted text can be used across AEM instances. Since AEM Publish is usually a sub-set of AEM Author, the pre-extracted text from AEM Author can often be used to re-index AEM Publish as well (assuming the AEM Publish have file-system access to the extracted text files).

Pre-extracted text can be incrementally added to over time. Text pre-extraction will skip extraction for previously extracted binaries, so it is best practice to keep pre-extracted text in case re-indexing must happen again in the future (assuming the extracted contents is not prohibitively large. If it is, evaluate zipping the contents in the interim, since text compresses well).

**Re-index Oak indexes, sourcing full-text from Extracted Text files**

*Execute re-indexing (Steps 3a-b) during a maintenance/low-use period as the Node Store is traversed during this operation, which may incur significant load on the system.*

3a. [Re-index](https://helpx.adobe.com/experience-manager/6-3/sites/deploying/using/best-practices-for-queries-and-indexing.html#HowtoReindex) of Lucene indexes is invoked in AEM

3b. The Apache Jackrabbit Oak DataStore PreExtractedTextProvider OSGi config (configured to point at the Extracted text via a file system path) instructs Oak to sourced full-text text from the Extracted Files, and avoids directly hitting and processing the data stored in the repository.