

COMPOSITE TEXT ANALYSIS

OPERATIONAL CONCEPT DOCUMENT

Version 1.0

Zac Bergquist - zbergqu@syr.edu

CSE 681: Software Modeling and Analysis

Project 1, September 2013

TABLE OF CONTENTS

Figures	4
Executive Summary	5
Introduction	5
Objective	5
Metadata	5
Queries	6
Users	7
Content Providers	7
Adding Content	7
Retrieving Content	7
Customers	8
Q/A	8
Activities	9
Getting Input	9
Processing Files	10
Displaying Results	10
Partitions	11
Driver	11
Metadata Processor	12
XML Reader	12
Index	12
Query Processor	12
Critical Issues	12
working set size (total number of documents)	12
Text Search Mechanism	13
Concurrent Users	13

Formatting of Results 14

Human Factors.....15

FIGURES

Figure 1: Functional Overview	5
Figure 2: XML Metadata Sample	6
Figure 3: Interface-independent Queries	9
Figure 4: Query Processing Activity Diagram.....	10
Figure 5: Analyzer Package Diagram.....	11
Figure 6: Query Processing	13
Figure 7: Handling Queries from Concurrent Users	14

EXECUTIVE SUMMARY

This document describes the requirements, use cases, design considerations, and overall architecture for a composite text analyzer (referred to as “the analyzer” from here on).

The analyzer operates on a set of text files and associated metadata. It accepts queries as input, and returns a list of all matching files as output.

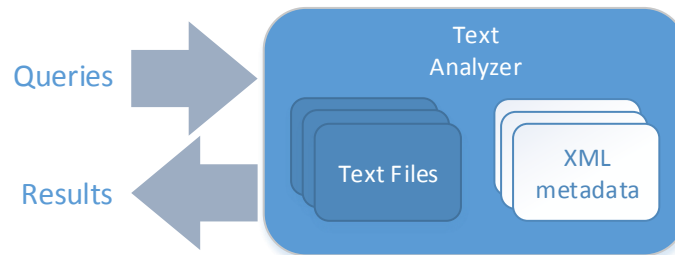


Figure 1: Functional Overview

INTRODUCTION

OBJECTIVE

The purpose of the analyzer is to aid in the management of large collections of documents. It is designed from the ground up to be easily reused and adapted for different projects and use cases. For the initial phase of this effort, the user interface to the text analyzer will be through the command line. However, the design of the analyzer allows for any front-end to be hooked up. It could easily accommodate a graphical user interface (GUI) or a scripting/automation interface.

The text analyzer is being designed as part of a larger document vault project. While the text analyzer can stand on its own, the use cases and design considerations presented in this document are accounting for the analyzer’s future use as the document vault’s backend.

METADATA

The metadata for these text files is stored in extensible markup language (XML) format, and includes the following fields:

- File name
- Date and time of last modification
- Version number
- Description
- Size of file (bytes)
- Dependencies (a set of 0 or more dependent files)
- Keywords (optional)

The following figure shows an example of an XML metadata file that would be accepted by the analyzer:

```

<?xml version="1.0"?>

<file
  name="SampleTextFile.txt"
  last_updated=""
  version_number="1"
  description="this is a sample text file"
  size="1024"
  keywords="sample,testing,demo">

  <dependencies>
    <d name="dependency1.txt"/>
    <d name="dependency2.txt"/>
  </dependencies>

</file>

```

Figure 2: XML Metadata Sample

The analyzer will ship with a separate tool dedicated to creating these metadata files. The metadata builder is a simple tool that provides a command line interface for building metadata files. It is designed to be small and fast so that it can be embedded in future tools.

QUERIES

The analyzer supports two different types of queries: text and metadata queries. Text queries search for a string of text in the contents of all the files. Metadata queries allow the user to specify a set of criteria based on the metadata of a particular file. Metadata queries, by nature, can be much faster than text queries since the analyzer doesn't need to scan the contents of entire text files to service the query. Metadata queries can be combined with text queries and as such are a powerful tool for efficiently searching through very large data sets.

TEXT QUERIES

Text queries are useful when searching for a particular string of text. For the initial phase of this project (command line interface), text queries will be specified via **/T** command line flags. For example, to search for the string "foo bar", the command line argument **/T"foo bar"** would be used.

A single query can contain multiple text strings to match. Each string should be provided as a separate **/T** argument. When multiple text strings are provided, the user can specify whether the query should be conjunctive (**/A**) or disjunctive (**/O**). If neither of these options are specified, queries are conjunctive by default.

Table 1: Example Text Queries

#	TEXT QUERY FORMAT	DESCRIPTION
1	/T"string 1" /T"string2" /A	Results contain both "string1" AND "string2"
2	/T"string1" /T"string2"	Identical to query #1 (/A is the default if not specified)
3	/T"string1" /T"string2" /O	Results contain either "string1" OR "string2" (or both)

METADATA QUERIES

Metadata queries can be used to request that various metadata is included for the files in the result set. Metadata queries are specified on the command line. The user must add a command line flag for each metadata field that they want included in the results. Metadata queries are specified using /M tags. For example, the following query arguments request both the file size and version number: */Msize /Mversion_number*.

If text and metadata queries are both present on the command line, the text query runs first, and then metadata is provided. If metadata queries are provided alone, then the metadata is retrieved for every file in the input directory that matches the input pattern. For more details on the query processing, see the Processing Files section.

USERS

Since the text analyzer will eventually serve as the backend for a document vault system, the following use cases include the larger context encompassing both the analyzer and the document vault.

CONTENT PROVIDERS

The term content provider describes any user that wishes to submit their work to the analyzer's working set. Depending on the type of documents the analyzer is managing, this could be several different types of users. This includes software developers, bloggers, authors, reporters, and any other types of users that create text-based content.

Content providers will use the system in several different ways:

ADDING CONTENT

The first step to interacting with the system is being able to add new content. Since many different types of users fall into this category, each with different skill sets and experiences, the main consideration here is to provide a generic interface to accommodate for future requests.

For example, software developers are highly technical and may wish for the tool to automatically detect when new source files are added to a project. They may want to write their own scripts and tools to use the text analyzer/document vault. On the other hand, a reporter will probably want a simple GUI that allows them to select a file to upload.

The project will be designed so that these various "front ends" can be added in a short period of time. If we are able to accommodate the requests of different customer bases, the project may even take off in multiple forms, each serving the same purpose but tailored to different contexts.

RETRIEVING CONTENT

We expect that in the majority of cases, the user that adds their content to the system is also the user who wishes to use the system to help manage their data. This user will likely be the user that makes queries to search through and retrieve their content. We do however recognize that there may be cases where one user is adding the data so that another user can query it.

Retrieving content could come in several forms. The content provider may be performing simple queries to find an old document that needs revisions, or they may be at a meeting or customer site wishing to pull up a document for review. The purpose of the text analyzer is simply to perform the query and report the results. It should not care where the query comes from or how it was formed, or even where it needs to send the results.

The same considerations apply to retrieving content as adding content – the design needs to be generic to start and then customized as needed. This will maximize our potential customer base and provide an opportunity for follow-on work.

CUSTOMERS

Customers may wish to use system to monitor progress and keep tabs on projects that they contract out. Potential uses include periodically running reports to find out which documents have been added or changed since the last report, or searching through release notes and other materials to find information about previous versions of a product.

This brings up a critical issue – in some cases, the document vault users might not want their customers to have access to everything in the vault. See the Critical Issues section for more information.

Note: project managers can be considered internal customers as they have the same type of use cases – monitoring progress, running reports, etc. Perhaps the only difference is that project managers don't have any of the limitations or viewing restrictions that external customers might have.

Q/A

A Q/A team would be responsible for validating and maintaining the tool. They will wish to exercise the software more than traditional users. In particular, some sort of scripting interface and batch query mechanism would be useful for testing.

The Python programming language is a popular scripting language that is widely used for Q/A and testing purposes due to its dynamic nature. Since this project will be implemented in C# .NET, Iron Python (www.ironpython.net) is an ideal Python implementation for this task.

However, batch queries and scripting capabilities might be useful to other users and not just Q/A. For example, if a user needs to make several queries into a very large collection, they may wish to set them up in advance, run the queries overnight or throughout the weekend, and come back and view the results at a later date. Since not all users will have the technical skills to develop Python code, a special query object will be developed to be used as input to the text analyzer.

Any executive that can create these query objects can be used with the text analyzer. Since IronPython runs on the CLR, this will meet the needs of users who wish to use Python to exercise the text analyzer. It also allows for any number of tools to be created to generate queries. The command line application that will be developed for the initial phase of this effort is just one example of such a query generator.

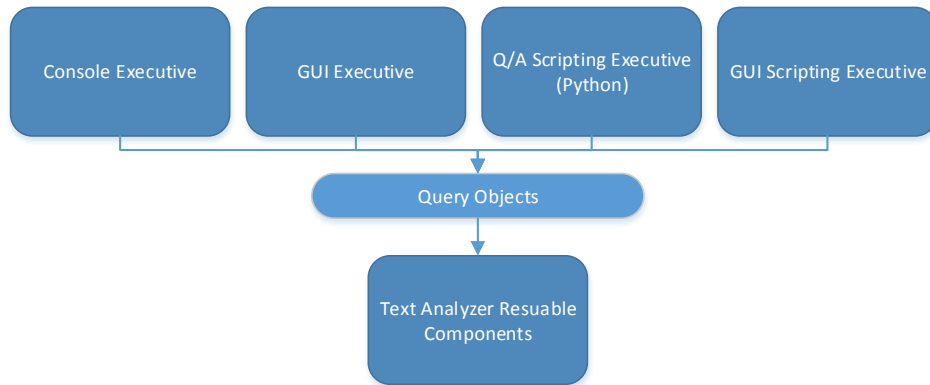


Figure 3: Interface-independent Queries

With this query language in place, a number of different executives can be built to generate queries, and the rest of the text analyzer can be ignorant as to where the query came from. The user may learn this query language and build their own queries, they could provide scripts that call into the command line interface for generating queries, or they could use Python to generate the queries. In addition, a GUI scripting executive could be built which serves the same purpose as the Python scripting, but visually walks the user through the steps of building a query rather than requiring them to learn Python.

For more detail on the box labeled “Text Analyzer Reusable Components” see Figure 5: Analyzer Package Diagram.

ACTIVITIES

GETTING INPUT

For the initial phase of this development, input will be provided in the form of command line arguments. The command line inputs include:

- Directory to search
- File pattern
- /T text query
- /A conjunctive text query
- /O disjunctive text query
- /M metadata query
- /R recursive search

In addition, the metadata generator tool that will ship with the text analyzer supports the following command line inputs:

- Fully qualified file name
- /T description text
- /D list of dependencies
- /K list of keywords

For specific details on command line input for queries, see the Queries section.

PROCESSING FILES

The following diagram shows the activities related to processing a query.

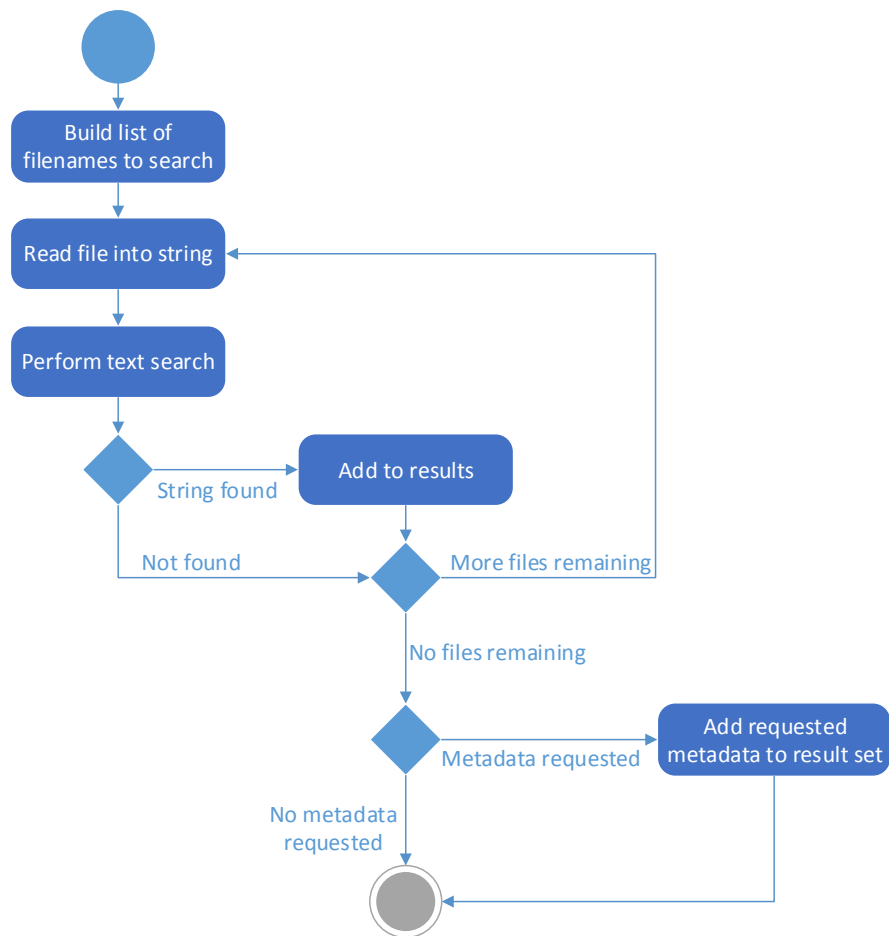


Figure 4: Query Processing Activity Diagram

As you can see, the processing is very sequential in nature. The first step is to gather a list of all files that must be searched. This includes all text (and possibly Word doc) files in the specified directory, as well as subdirectories if the recursive option was specified. After the list is generated, each file is read into a string, scanned for the text, and if a match was found, added to the result set. When all files have been scanned and the result set is complete, the software checks if the user requested any metadata fields. If so, the metadata is added to the result set, and the processing phase is complete.

DISPLAYING RESULTS

After processing completes, the result set will be passed on to the display stage, where results are formatted for display. In this case, the results are simply printed to the console.

PARTITIONS

The following package diagram shows the partition of the analyzer into modules grouped by functionality.

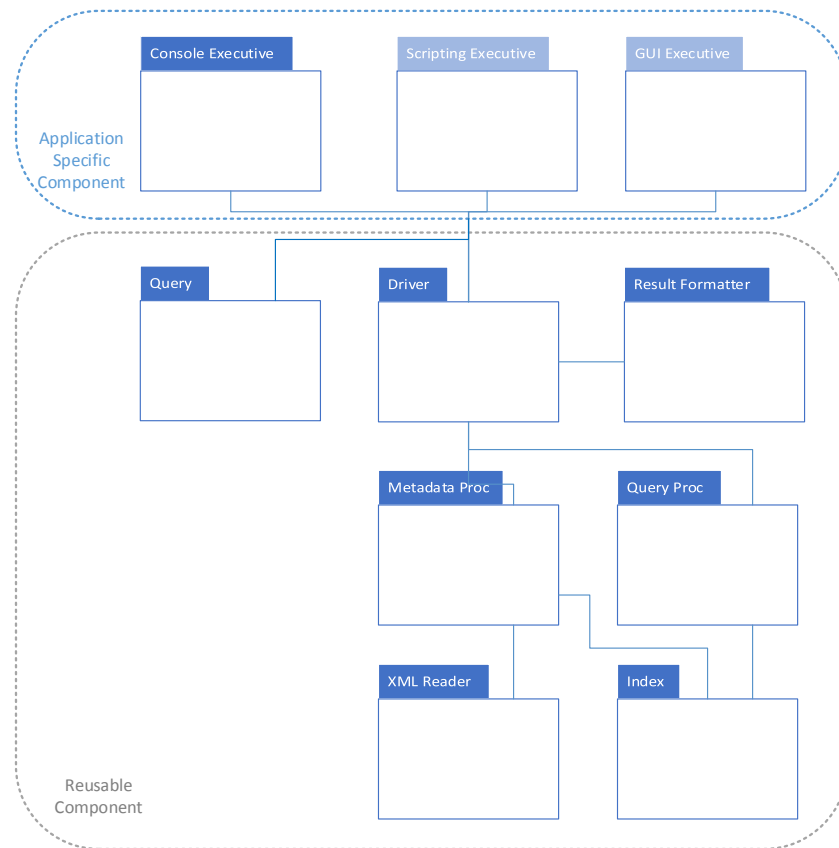


Figure 5: Analyzer Package Diagram

DRIVER

In this package diagram, the driver is the interface to any external components that wish to interact with the text analyzer. Its public interface exposes a generic representation of a query and analyzer results in order to maximize reuse. The driver will typically be controlled by an executive. In this case, a console executive will provide a command line wrapper to the driver's interface. Examples of other executives include GUI applications and scripting executives.

This boundary allows for several forms of reuse. The packages in the gray "reusable" boundary can be compiled into a class library, or included as source code for use by other applications. Additionally custom executives can be developed to provide for different forms of reuse. For example, a network executive could be developed that would accept queries over a network connection, allowing different GUI clients to submit queries to a remote text analyzer.

METADATA PROCESSOR

The metadata processor is responsible for enumerating the metadata files and building an index. Since the metadata will be in the form of XML files, most of the work will be accomplished by calling into the XML reader. The metadata processor serves as an interface between the driver and the XML reader. Since the driver isn't reading the metadata files directly, this design allows for flexibility in the event that other (non XML) metadata formats are used in the future.

XML READER

The XML reader provides basic XML parsing functions and is used by the metadata processor for pulling metadata elements from the XML files.

INDEX

The index is meant to be an in-memory representation of the data set that can be used to process queries. Since it isn't feasible to keep entire documents in memory, the index will simply serve as a data structure to hold any metadata that has been parsed from XML.

While this may seem unnecessary at first, the index package is provided as a potential extension point to allow for more advanced queries and features in the future.

QUERY PROCESSOR

The query processor takes in query strings and parses them in order to determine how to handle the query. After parsing, the query processor calls into the index in order to acquire the query results.

CRITICAL ISSUES

WORKING SET SIZE (TOTAL NUMBER OF DOCUMENTS)

The text analyzer and document vault may be used with very large working sets. The total number of documents managed by the tool could number into the thousands or even higher. If the analyzer were to search the entire collection for each query, there could be performance problems for very large collections.

The performance bottleneck for this application will most likely be the disk I/O. Each file has to be first loaded into memory and then searched for the query string. A naïve design might try to parallelize the application to either run multiple queries in parallel, or to split a single query into separate searches, each run in parallel. This should be avoided, especially for systems which use traditional platter-based disk drives. With these drives, the disk head can only be reading from a single sector at any given time, so attempts to run parallel queries would result in the disk bouncing back and forth to seek to different sectors. This would cause page faults and hinder performance.

Since we can't do much about improving the disk I/O (aside from purchasing new hardware) we need to address the performance issue in another way. The solution will be to allow files to belong to categories. A category is a tag that can be applied to a file. Each category consists of one or more files, and a single file can belong to multiple categories. The category information for the file will be embedded in the XML metadata. The software will then

allow the user to specify one or more categories in which to run their query. The software will only open files and scan for the query string in files that belong to the specified categories.

For maximum flexibility, it will be up to the user to decide how best to tag the data with categories. In some cases, it might make sense to group all files belonging to a particular project into a single category. Other times, the user may wish to group categories based on the subject matter or type of document.

Categories will be totally optional, so users with small data sets don't need to bother with setting them up or forming category-specific queries. Users with extremely large collections who decide that the time required to tag the data with categories is well worth the performance improvement are free to use categories as they see fit.

TEXT SEARCH MECHANISM

Since the text search mechanism is looking for exact substrings, the user might run into cases where they aren't getting the results they expect from a query. For example, a text query looking to match the string "the quick brown fox jumped over the lazy dog" might fail when the actual document reads "the quick brown fox *jumps* over the lazy dog."

In true information retrieval engines, these issues are resolved using different types of stemming algorithms and query expansions. An implementation of these techniques could be added to the query processor package, but would cost a significant amount of development time. In the meantime, these issues can be worked around by constructing smarter queries when necessary. For example, if the user didn't receive any results after their first query, they could create a query looking to match both "quick brown fox" and "lazy dog."

CONCURRENT USERS

When the text analyzer is eventually integrated into the document vault system, multiple networked users may attempt to query the same data simultaneously. This introduces several concerns in both correctness and performance. First, this introduces the potential for race conditions and other problems. The software has to be able to recognize and successfully process multiple queries, even if they are received milliseconds from one another.

In addition, performance problems related with concurrent operations must also be considered. We have already discussed how parallelizing a single query would likely be slower than running the query sequentially, but it is unknown how the software will fair when two unrelated queries are executed simultaneously.

At a high level, query processing can be described in three steps:

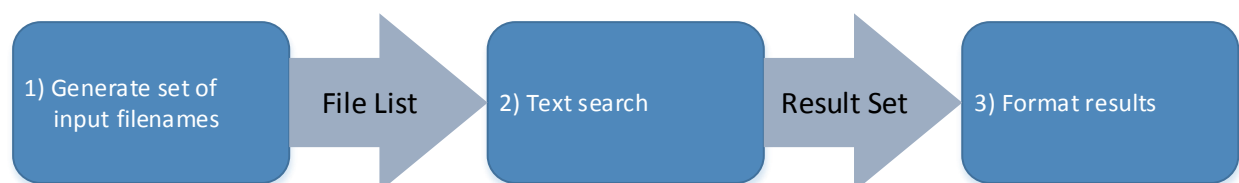


Figure 6: Query Processing

It is clear that operations 1 and 3 can be safely performed in parallel, as they are primarily computation. Some testing and investigation must be done to determine whether it is faster to serialize the text searches (option A),

which would be cache friendly but might introduce a bottleneck, or to parallelize entire queries (option B), which would likely increase the time required for each query but might perform better in the long run. Both option A and option B are depicted graphically in the following diagram:

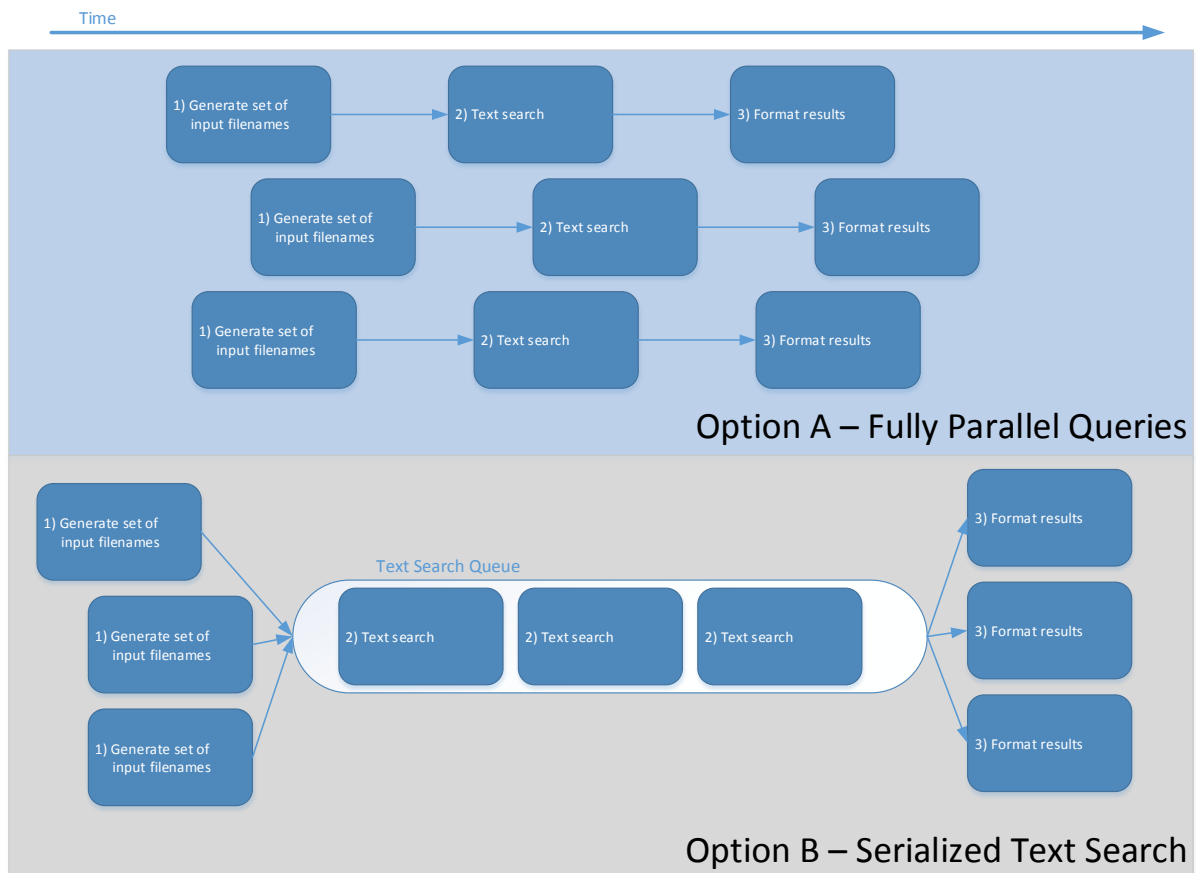


Figure 7: Handling Queries from Concurrent Users

FORMATTING OF RESULTS

In some cases, the users making queries shouldn't have full access to the dataset and results. For example, if a company is using the document vault to help track a project they are building for a customer, they may wish to provide the customer access to the vault. The customer could use the vault to periodically check on progress and search for updates. While this can be useful, the developers may not want the customer to have access to the entire collection of documents. The vault may contain data related to other projects or other customers that shouldn't be accessible outside of the organization.

Fortunately, since the query processing and results generation are decoupled, the software can support these use cases. Separate result formatters can be written to filter or reformat the resulting data before it is presented to the user. The query logic would remain the same, regardless of the result formatter that is used.

HUMAN FACTORS

This issue focuses on usability and making the tool useful to its users. This issue includes concerns for the following items:

- Making output that is useful and easy to understand
 - Reports could be expanded with visualizations, charts, etc.
- Dealing with long running queries
 - Estimating how long a query will take
 - Showing progress of a query (processed X out of Y files, Z files remaining)
 - Providing options to cancel a long running query
 - Allowing the user to construct the next query while the current query is running
- Making the tools easy to learn
 - Command line help/usage flag or manual page
 - Exploratory GUI tools (tooltips, help menus, etc.)
 - Accompany with README and user's guides