Object Matrix

Content Search

Version 3.2, Jan 2016

Contents

1 Overview

MatrixStore Content Search has been introduced to MatrixStore v3.1 to allow for API users to access a high-performance, full-featured text search engine. It is a technology suitable for nearly any application that requires full-text search and features:

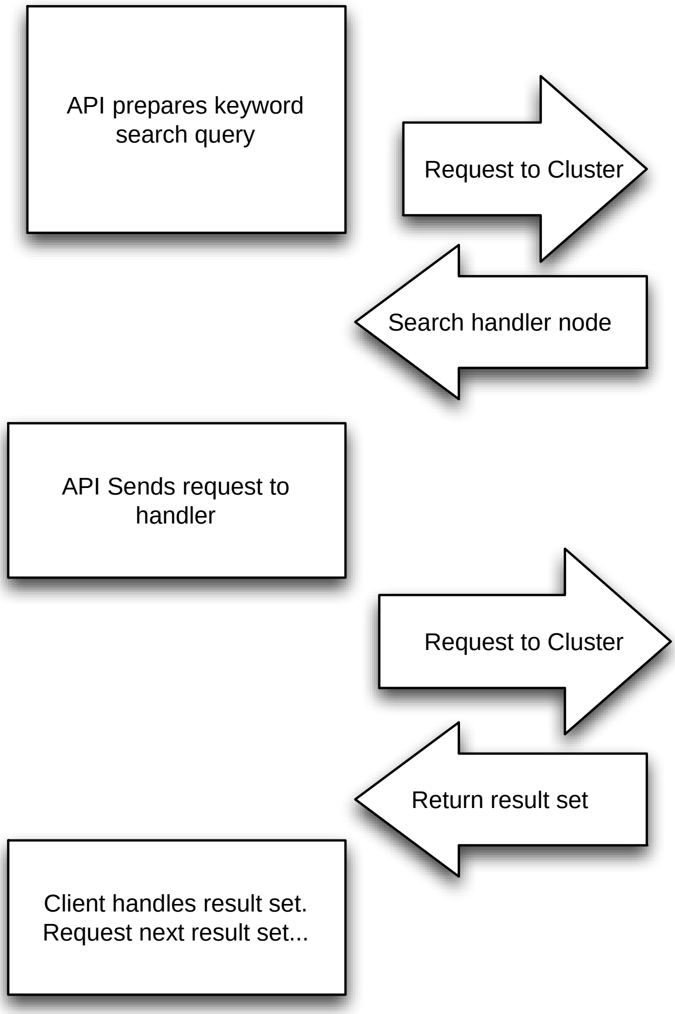
- many powerful query types: phrase queries, wildcard queries, range queries and more
- fielded searching (e.g. title, author, contents)
- sorting by any field
- multiple-index searching with merged results

2 MatrixStore Search Architecture

MatrixStore supports two search engines:

- Standard keyword search
- Content search

Keyword search has been present within MatrixStore since day 1 and guarantees highest levels of performance. When a client performs a search it starts by making a single call to the API. In the background the API performs the following actions:


API prepares keyword search query
Request to Cluster
Cluster assigns search handler node
Search handler node
API Sends request to handler
Request to Cluster
Node contacts cluster for result set
Return result set
Client handles result set. Request next result set...

This is true for both standard and content search. The main difference is:

- Standard Keyword search results can always be returned without the cluster accessing disk
- Content search requires each online node to access disk to build a result set and thus will typically be slower than standard keyword search

However, whilst keyword search can only search for exact matches, content search offers a wealth of search features, including:

- many powerful query types: phrase queries, wildcard queries, range queries and more, e.g.,
    - "Jon*": can return "Jon", "Jonathan"…
    - "?old": can return "cold", "hold" etc.
- fielded searching (e.g. title, author, contents)
- sorting by any field
- multiple-index searching with merged results

- fast-search can return requested field values in the search results

2.1 Fields within Content Search

Content search can be made on:

- All key value pairs added as metadata to an object
- All critical data on an object (length, date stored etc)
- Content extracted from an object (see contract extraction)

2.2 Content Search Fast-Search

Fast search can be made in both standard search and content search modes. Fast search allows the user to define which fields should be returned in the search results, and should be considered carefully since it can result in searching that is 100x faster.

Imagine a standard search to fill a list of files in a directory, the search might be:

1. Send query (“Get all objects in a directory”)
    a. Server returns 100 results
2. For each object:
    a. Request Object filename
    b. Request Object length
    c. Request Object last accessed time
3. Sort and display sorted results

The above query would result in at least 300 round trips to the server.

Instead, a fast query does the following:

1. Send query (“Get all objects and return to me object ID, name, length, last accessed time”)
    a. Server returns 100 results
2. Sort and display sorted results

This can result in just 1 round trip (or 2 or 3 depending on packet sizes etc) to the server.

Even better, content search can sort results and return just the top hits, removing the need to return all results before displaying the appropriate data.

2.3 Content Extraction

MatrixStore can examine objects added to the store for searchable metadata and keywords. To have MatrixStore extract content for search, add the following metadata attribute to the object at the creation of the object.

```
new Attribute(Constants.CONTENT_TYPE, Constants.TEXT);
```

In V1, the following field types are supported for content extraction:

TEXT = "text/plain";

RTF = "text/rtf";

HTML = "text/html";

PDF = "application/pdf";

MSWORD = "application/msword";

3 Prefixes (New in version 3.2)

In MatrixStore V3.2 **two fields** are indexed for each attribute: a **text** field and a field of the **attribute type**. The field name of the latter will contain one of the next **prefixes** indicating the type info:

*_mxs_string_, _mxs_int_, _mxs_long_, _mxs_double_, _mxs_float_ or _mxs_boolean_*

Example:

```
Mxs attribute:
metadata.writeString(“title”, “A Clash of Kings”);

Fields in the index:
       -   title (text field)
       -   _mxs_string_title (string field)


Mxs attribute:
metadata.writeInt(“season”, 2);

Fields in the index:
       -   season (text field)
       -   _mxs_int_season (int field)
```

These prefixes give the user more flexibility during searching and are especially useful when searching **string** fields or creating range queries using the Lucene query parser syntax.

3.1 String queries

Given the next attribute:

```
metadata.writeString(“title”, “A Clash of Kings”);
```

We have now multiple options to find the object:

// case-sensitive and exact match

```
search(“_mxs_string_title: \“A Clash of Kings\”);
```

// case-sensitive and wildcards

```
search(“_mxs_string_title: A*”);
```

// case-insensitive and no exact match (text field)

```
search(“title: \“a clash of Kings\”);
search(“title: \“a clash of\”);
```

// etc. (regexp, fuzzy …)

3.2 Range queries

Prefixes allow us to create range queries using the Lucene query parser syntax, which let us to create more complex queries. Previous syntax is still compatible (see section 3.3).

Example:

// normal range

```
search(“_mxs_int_season: [1 TO 5]”);
```

// open range

```
search(“_mxs_int_season: [1 TO *]”);
```

// multiple ranges

```
search(“title: kings AND (_mxs_int_season: [1 TO 2] OR _mxs_int_year: [2012 TO
2014”));
```

4 Content Search Language

Content search queries are made up of the following parts:

Query

Keywords

Ranges

Sorts

4.1 Query

Queries are based upon Lucene 5.3.1 syntax.

4.1.1 Terms

A query is broken up into terms and operators. There are two types of terms: Single Terms and Phrases.

A Single Term is a single word such as "test" or "hello".

A Phrase is a group of words surrounded by double quotes such as "hello dolly".

Multiple terms can be combined together with Boolean operators to form a more complex query (see below).

Note: The analyser used to create the index will be used on the terms and phrases in the query string. So it is important to choose an analyser that will not interfere with the terms used in the query string.

4.1.2 Fields

Query supports fielded data. When performing a search you can either specify a field, or use the default field. The field names and default field is implementation specific.

You can search any field by typing the field name followed by a colon "\u241D " and then the term you are looking for.

As an example, let's assume an index contains two fields, title and text and text is the default field. If you want to find the document entitled "The Right Way" which contains the text "don't go this way", you can enter:

title: "The Right Way" AND text:go

or

title:"The Right Way" AND go

Since text is the default field, the field indicator is not required.

Note: The field is only valid for the term that it directly precedes, so the query

title:The Right Way

Will only find "The" in the title field. It will find "Right" and "Way" in the default field (in this case the text field).

4.1.3 Term Modifiers

Query supports modifying query terms to provide a wide range of searching options.

Wildcard Searches

Query supports single and multiple character wildcard searches within single terms (not within phrase queries).

To perform a single character wildcard search use the "?" symbol.

To perform a multiple character wildcard search use the "*" symbol.

The single character wildcard search looks for terms that match that with the single character replaced. For example, to search for "text" or "test" you can use the search:

te?t

Multiple character wildcard searches looks for 0 or more characters. For example, to search for test, tests or tester, you can use the search:

test*

You can also use the wildcard searches in the middle of a term.

te*t

Note: You cannot use a * or ? symbol as the first character of a search.

Regular Expression Searches

Query supports regular expression searches matching a pattern between forward slashes "/". The syntax may change across releases, but the current supported syntax is documented in the RegExp class. For example to find documents containing "moat" or "boat":

/[mb]oat/

Fuzzy Searches

Query supports fuzzy searches based on Damerau-Levenshtein Distance. To do a fuzzy search use the tilde, "~", symbol at the end of a Single word Term. For example to search for a term similar in spelling to "roam" use the fuzzy search:

roam~

This search will find terms like foam and roams.

An additional (optional) parameter can specify the maximum number of edits allowed. The value is between 0 and 2, For example:

roam~1

The default that is used if the parameter is not given is 2 edit distances.

Boolean Operators

Boolean operators allow terms to be combined through logic operators. Query supports AND, "+", OR, NOT and "-" as Boolean operators(Note: Boolean operators must be ALL CAPS).

OR

The OR operator is the default conjunction operator. This means that if there is no Boolean operator between two terms, the OR operator is used. The OR operator links two terms and finds a matching document if either of the terms exist in a document. This is equivalent to a union using sets. The symbol || can be used in place of the word OR.

To search for documents that contain either "Object Matrix" or just "Matrix" use the query:

"Object Matrix" Matrix

or

"Object Matrix" OR Matrix

AND

The AND operator matches documents where both terms exist anywhere in the text of a single document. This is equivalent to an intersection using sets. The symbol && can be used in place of the word AND.

To search for documents that contain "Object Matrix" and "Object MatrixStore" use the query:

"Object Matrix" AND "Object MatrixStore"

+

The "+" or required operator requires that the term after the "+" symbol exist somewhere in a the field of a single document.

To search for documents that must contain "matrix" and may contain "object" use the query:

+matrix object

NOT

The NOT operator excludes documents that contain the term after NOT. This is equivalent to a difference using sets. The symbol ! can be used in place of the word NOT.

To search for documents that contain "object matrix" but not "matrix store" use the query:

"object apache" NOT "matrix store"

Note: The NOT operator cannot be used with just one term. For example, the following search will return no results:

NOT "object apache"

-

The "-" or prohibit operator excludes documents that contain the term after the "-" symbol.

To search for documents that contain "object matrix" but not "matrix store" use the query:

"object apache" -"matrix store"

Grouping

Query supports using parentheses to group clauses to form sub queries. This can be very useful if you want to control the boolean logic for a query.

To search for either "object" or "matrix" and "website" use the query:

(object OR matrix) AND website

This eliminates any confusion and makes sure you that website must exist and either term object or apache may exist.

Field Grouping

Query supports using parentheses to group multiple clauses to a single field.

To search for a title that contains both the word "return" and the phrase "pink panther" use the query:

title:(+return +"pink panther")

Escaping Special Characters

Query supports escaping special characters that are part of the query syntax. The current list special characters are

+ - && || ! ( ) { } [ ] ^ " ~ * ? : \ /

To escape these character use the \ before the character. For example to search for (1+1):2 use the query:

\(1\+1\)\:2

4.2 Keywords

Fast search is supported by adding the following line to a query:

```
keywords: key1, key2, key3, …
```

Results then contain:

```
Object_Id key1=value1 key2=value2 key3=value3, etc
```

Where key1 is the key. When a key is a system keyword the prefix

```
__mxs__name
```

contains two underscores, mxs, two underscores, name of field.

The following system keywords are valid:

Keyword	Description
__mxs__id	Object id
__mxs__locked	Object Locked?
__mxs__inCompliantStore	Object in a compliant store?
__mxs__ storesCurrentRetentionPeriod	Retention period of the vault (seconds)
__mxs__creationTime	Time object was created
__mxs__modifiedTime	Time object was last user modified
__mxs__accessedTime	Time object was last user accessed
__mxs__length	Length of object data in bytes
__mxs__calc_adler32	Adler32 checksum of the object
__mxs__calc_md5	MD5 of the object
__mxs__location	Nodes the object is physically located upon
__mxs__online	Is the object online or has it been archived?

Results are always text.

4.3 Ranges

Range Queries allow one to match documents whose field(s) values are between the lower and upper bound specified by the Range Query. Range queries can consist of one or more lines of

```
    range:field\u241D[string|int|long|float|double]\u241D[>=|>]\u241Dfrom\u241D
[<=|<]\u241Dto
```

4.4 Sorts

Sorts allow the result set to be sorted according to field values. Multiple sorts can be added, with the primary sort coming first. Sort syntax is:

```
    sort:[>|<]\u241Dfield\u241D[string|int|long|float|double]
```

4.5 Text attribute Vs String attribute

`Note: See changes in version 3.2`

To create a String attribute using the API you can use either, a **String** type or a **Text** type:

```
…
ObjectTypedAttributeView metadata = object.getAttributeView();
…
metadata.writeString(“title”, “A Clash of Kings”);

or

metadata.writeText(“title”, “A Clash of Kings”);
```

When performing a **field query**, content search will return different results depending on the type of the attribute as explained in the next sections.

4.5.1 Text attribute

A **Text** attribute is indexed and tokenized so that you can search it using particular terms (case insensitive) contained in the attribute value. The query

```
title:clash
```

will find an object with the **Text** title attribute value “A Clash of Kings”.

4.5.2 String attribute

A **String** field is indexed but not tokenized: the entire String value is indexed as a single token and with the original case. For example this might be used for a “country” field or an “id” field. In this case, the query

```
title:clash
```

will not find an object with the **String** title attribute value “A Clash of Kings”. Even the query

```
title:”A Clash of Kings”
```

will not find the object as the parser lowercases the words of the query so they do not match with the original title value.

The way to search by a String attribute is using a range query:

```
String GS = "\u241D";
String mytitle= “A Clash of Kings”

*\nrange:title" + GS + "string" + GS + ">=" + GS + mytitle + GS + "<=" + GS +
mytitle
```

4 Examples

Each example below calls upon a method called contentSearch.

Please note that the escape character is “\u241D”

The code for this is:

```
          private static Set<String> contentSearch(Vault vault, String query) throws
IOException {
              Attribute contentQuery = new Attribute(Constants.CONTENT, query);
              return vault.searchObjects(contentQuery, 10);
          }
```

Sample searches are:

```
          // find all objects with “three” in the content
          ids = contentSearch(vault, "three");
          System.out.println("Exact Matching found " + ids.size() + " objects");

          // search all objects with exact matching of attribute incrementingInt
= 10005
          ids = contentSearch(vault,
"*\nrange:incrementingInt\u241Dint\u241D>=\u241D10000\u241D<=\u241D10000");
          System.out.println("Exact Matching found " + ids.size() + " objects");

          /*
           * Range Matching
           */
          // search all objects with attribute incrementingInt between 100 and
20000
                  ids = contentSearch(vault,
"*\nrange:incrementingInt\u241Dint\u241D>=\u241D100\u241D<=\u241D20000");
          System.out.println("Range Matching found " + ids.size() + " objects");

          /*
           * Field Sorting
           */
          // find all objects with “two” in the content, sort results by
__mxs__id
          ids = contentSearch(vault, "two\nsort:>\u241D__mxs__id\u241Dstring");
          System.out.println("Field Sorting found " + ids.size() + " objects");

          // find all objects with “one” in the content, sort the results on an
integer field called “incrementingInt”
          ids = contentSearch(vault,
"one\nsort:>\u241DincrementingInt\u241Dint");

          /*
           * Keywords
           */
          ids = contentSearch(vault, "five\nkeywords:__mxs__id,__mxs__location");
          for (String id: ids) {
              System.out.println("Result with keywords embedded: " + id);
          }

          /*
           * Wildcard Search
           */
          ids = contentSearch(vault, "five*");
          System.out.println("Wildcard Search found " + ids.size() + " objects");

          /*
```

```
           * Fuzzy Search
           */
          ids = contentSearch(vault, "fiva~");
          System.out.println("Fuzzy Search found " + ids.size() + " objects");

          /*
           * AND Search
           */
          createSimpleObject(vault, "ANDSearches0 ANDSearches1");
          while (!object1.isIndexed()){
              try{
                  Thread.sleep(5000);
              }catch (InterruptedException e){}
          }

          ids = contentSearch(vault, "*\nANDSearches0 AND ANDSearches1");
          System.out.println("AND Search found " + ids.size() + " objects");

          /*
           * OR Search
           */
          createSimpleObject(vault, "ORSearches0");
          createSimpleObject(vault, "ORSearches1");
          ids = contentSearch(vault, "*\nORSearches0 OR ORSearches1");
          System.out.println("OR Search found " + ids.size() + " objects");

          /*
           * NOT Search
           */
          createSimpleObject(vault, "noSearches");
          createSimpleObject(vault, "noSearches MatrixStore");
          ids = contentSearch(vault, "noSearches NOT MatrixStore");
          System.out.println("NOT Search found " + ids.size() + " objects");

          /*
           * Grouping
           */
          createSimpleObject(vault, "Grouping0");
          createSimpleObject(vault, "Grouping1 MatrixStore");
          ids = contentSearch(vault, "((Grouping0 OR Grouping1) AND
MatrixStore)");
          System.out.println("Grouping found " + ids.size() + " objects");
```

The query consists of a string containing:

```
query: lucene syntax query
keywords:[keyword],[keyword],...
range:field\u241D[string|int|long|float|double]\u241D[>=|>]\u241Dfrom\u241D[<=|<
]\u241D to    (can be many)
sort:[>|<]\u241Dfield\u241D[string|int|long|float|double]    (can be many)
```