UCRL-TR-235781

LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

Native Language Processing using Exegy Text Miner

J. Compton

October 24, 2007

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Native Language Processing
Using Exegy Text Miner

John Compton
Computing Applications and Research
Lawrence Livermore National Laboratory

10/24/07

Executive Summary

Lawrence Livermore National Laboratory's New Architectures Testbed recently evaluated Exegy's Text Miner appliance to assess its applicability to high-performance, automated native language analysis. The evaluation was performed with support from the Computing Applications and Research Department in close collaboration with Global Security programs, and institutional activities in native language analysis.

The Exegy Text Miner is a special-purpose device for detecting and flagging user-supplied patterns of characters, whether in streaming text or in collections of documents at very high rates. Patterns may consist of simple lists of words or complex expressions with sub-patterns linked by logical operators. These searches are accomplished through a combination of specialized hardware (i.e., one or more field-programmable gates arrays in addition to general-purpose processors) and proprietary software that exploits these individual components in an optimal manner (through parallelism and pipelining). For this application the Text Miner has performed accurately and reproducibly at high speeds approaching those documented by Exegy in its technical specifications. The Exegy Text Miner is primarily intended for the single-byte ASCII characters used in English, but at a technical level its capabilities are language-neutral and can be applied to multi-byte character sets such as those found in Arabic and Chinese.

The system is used for searching databases or tracking streaming text with respect to one or more lexicons. In a real operational environment it is likely that data would need to be processed separately for each lexicon or search technique. However, the searches would be so fast that multiple passes should not be considered as a limitation *a priori*. Indeed, it is conceivable that large databases could be searched as often as necessary if new queries were deemed worthwhile.

Project Goals

This project is concerned with evaluating the Exegy Text Miner installed in the New Architectures Testbed running under software version 2.0. The concrete goals of the evaluation were to test the speed and accuracy of the Exegy and explore ways that it could be employed in current or future text-processing projects at Lawrence Livermore National Laboratory (LLNL). This study extended beyond this to evaluate its suitability for processing foreign language sources. The scope of this study was limited to the capabilities of the Exegy Text Miner in the *file search mode* and does not attempt simulating the streaming mode. Since the capabilities of the machine are invariant to the choice of input mode and since timing should not depend on this choice, it was felt that the added effort was not necessary for this restricted study.

Hardware/Software

The system used in this study is a single Exegy Text Miner with the basic configuration. This means that only one type of search can be performed at a time (exact, approximate, or regular expression), but within each of these types Boolean operators and proximity operators can be applied. The encryption/decryption module was not included as extra hardware over and above this basic capability. For the focused scope of this study, this additional hardware was not deemed necessary.

The local hardware configuration for this study includes two Raid 0 partitions and two Raid 5 partitions of dedicated disk space (called "high-speed persistent cache" in the Exegy technical specifications). Each partition is 1.2 terabytes for a total of 4.8 TB, of which only 4.4TB is effectively usable. Our network connections consist of two 1 gigabit/sec, and two 10 Gb/sec interfaces, all of the GigE variety. Exegy quotes its sustained throughput performance as 5.2 Gb/sec and its peak as 7.1 Gb/sec.

Internally the Text Miner has two AMD dual-core Opteron processors, and a Xilinx Virtex-4 FPGA board (with three FPGA's). (For additional capability the number of FPGA's can be increased at added expense.)

The Exegy on-board software is accessible to the user through its API. (Interfaces are provided for C, C++, Java, and Perl.) Exegy provided several test programs containing calls to this API that provide the user with a UNIX command-line interface to the internal software of the Exegy. These programs were used with little modification since they allowed for testing of all the available search features. Some modifications of their software (C version) were made to allow for more user-friendly display of search results in the case of the tests on the Arabic and Chinese corpora.

Approach

The combination of high-performance disk drives, large input buffers, and internal parallelization and pipelining make the Exegy ideal for high-speed scanning of an input stream (for example, news articles) for the presence of user-defined strings (byte patterns) associated with characters in word lists or other patterns that can be defined via regular expressions. The user can also specify that matches of sub-patterns are contingent on proximity constraints with other sub-patterns. For example, one can require that the words "stock" and "bond" will be reported only if they occur within a fixed number of bytes from each other in the text. Allowances for spelling errors and other imprecision are permitted through approximate matching to a level (number of deviations) likewise specified by the user.

The software provided by Exegy was used for applying queries to one or more files at a time (typically hundreds to thousands at a time). The source used for English queries consists of news articles provided by Reuters for 1995-1996. Arabic and Chinese sources, also news articles from several agencies, are from the ACE corpora provided by the Linguistic Data Consortium for the year 2000. The Text Miner 2.0 software was evaluated for speed and accuracy in its various text search modes.

Queries in Arabic

In order to formulate a query for foreign text it is necessary to know how that text is encoded. The ACE documents used here for testing in Arabic and Chinese were encoded in Unicode, and so the following examples conform to that model. Each character in Unicode (which spans a large array of languages), regardless of the character set to which it belongs, has a unique representation that is one to three bytes long. The most significant bits of the most significant byte encode the character set, and the lower order bits encode the particular character. For example, Arabic characters in their basic disconnected forms span the range 0600-06FF in hexadecimal. These are sufficient for internal representation of a document in any of the many languages that use some version of the Arabic alphabet, and they appear in the ACE corpus used here. Two separate sets of presentation forms (the connected forms in which Arabic characters actually appear on screen or a printed page) occupy the ranges FB50-FDFF and FE70-FEFF, but are not needed for the queries made here.

An individual Arabic character code is embedded in Unicode as follows. We will use the letter *laam* (ل) as an example. Its code is 0644 hex. We strip off the most significant four bits to get 644. These twelve bits (‘011001000100’) are then divided in half. The upper six bits are appended to ‘11’ to give ‘11011001’ or D9 in hex. The lower six bits are appended to ‘10’ to give ‘10000100’ or 84 in hex. So in the text *laam* appears in two sequential bytes as D984. (This is the UTF-8 encoding form used in the ACE corpus. Other forms are possible but not used here.) In order to construct the sequence for ‘Lebanon’ (‘*lubnaan*’ in Arabic, or لبنان) we will need the additional characters *baa’* (ب), *nuun* (ن), and *’alif* (ا). Their UTF-8 encodings for these characters are D8A8, D986, and D8A7 hex, respectively. The sequence of characters for ‘*lubnaan*’ is *laam, baa’, nuun, ’alif, nuun*. Placing all these characters in a string for an exact-match query results in, for example:

```
file-search -q 'EXACT:\xd9\x84\xd8\xa8\xd9\x86\xd8\xa7\xd9\x86' -f ./arab_data
```

In the following example we have the results of searching the corpus of Arabic news articles of the year 2000 from the Linguistic Data Consortium’s (LDC) Automatic Content Extraction (ACE) Project (http://projects.ldc.upenn.edu/ace/). The query was for any article containing the Arabic equivalent of ‘America’ (أمريكا) or ‘Lebanon’ (لبنان). The following is an excerpt of the results of that search:

```
./arab_data/bn/fp1/NTV20001016.1530.1140.sgm [2537 - 2548]: أمريكا (1)
>>>  ين ا أن نعرف وأخيرا أن أمريكا هي مستعمرة إ
./arab_data/bn/fp1/NTV20001016.1530.1140.sgm [2622 - 2633]: أمريكا (1)
>>>
./arab_data/bn/fp2/NTV20001222.1530.0613.sgm [382 - 391]: لبنان (0)
>>>  الحدود الإسرائيلية اللبنانية بأنه حساس ج
./arab_data/bn/fp2/NTV20001222.1530.0613.sgm [604 - 613]: لبنان (0)
>>>  شأ بسبب فشل الحكومة اللبنانية في فرض احتر
./arab_data/bn/fp2/NTV20001222.1530.0613.sgm [910 - 919]: لبنان (0)
>>>  ائيليين ونشر الجيش اللبناني لإعادة السل
```

```
./arab_data/bn/fp2/NTV20001222.1530.0613.sgm [999 - 1008]: نانبل (0)
>>>
./arab_data/bn/fp2/NTV20001222.1530.0613.sgm [1040 - 1049]: نانبل (0)
>>>  بنان قد رفضت نشر قوات لبنانية في الجنوب ا
./arab_data/bn/fp2/NTV20001222.1530.0613.sgm [1077 - 1086]: نانبل (0)
>>>  ت لبنانية في الجنوب اللبنان ي إلى أن يتم ا
```

The first four lines of this output detail how two occurrences of 'America' occur in one article (NTV20001016.1530.1140.sgm) at bytes 2537 – 2548 and 2622 – 2633. Following that are the letters that spell out 'America' in Arabic, but since Arabic is written from right to left that are backwards from the Arabic perspective. The '(1)' simply indicates that 'America' follows 'Lebanon' (numbered '0' below this) in the query not shown here. In lines 2 and 4 we can see the context in which the word occurs. This time the order of the letters is correct, but they are not connected to each other in the normal fashion, so we see 'أمريكا' or ''-m-r-ii-k-aa' instead of 'أمريكا' or ''mriikaa'. (In addition, some context lines are blanks. These are only minor irregularities of software that was not written to accommodate foreign languages at all initially. The fix is rather simple.) Note that short vowels are not written, and so the initial 'a' is pronounced but not written. (There is no written or phonetic equivalent for the 'e' in "America" either.) The initial (') represents a glottal stop, a consonant in Arabic, required because a word in Arabic must begin with a consonant. The remaining lines of the excerpt above represent six occurrences of 'Lebanon' or some variant of it such as the equivalent of 'Lebanese' in another article. The interpretation of these lines is exactly for those described above.

Queries in Chinese

Searches in Chinese are constructed in much the same way as for Arabic. The main difference is in the number of possible characters to be used in a search (in the tens of thousands), and the fact that the characters require two bytes for their individual codes (and three bytes for some rarely used characters) and three bytes in the UTF-8 encoding. In order to look up the encoding one needs to use a special on-line dictionary and enter either the phonetic spelling or an English equivalent. One must also decide whether to use the simplified characters of the Chinese mainland or the traditional character set in use elsewhere (depending upon the encoding of the source documents.) So for example, if we want to search for 'China' (中国, zhong1 guo4), then we look up the two encodings as 4E2D and 56FD, respectively. The first hex digit is appended to '1110' and the last three digits are treated as with Arabic above to yield E4B8AD and E59BBD, respectively. So now a query for 'China' looks like:

/file-search -q 'EXACT:\xe4\xb8\xad\xe5\x9b\xbd' -f ./chinese_data

In our next example we again use a corpus from LDC's ACE project. We try a more sophisticated query for the words 'America' (美国, mei3 guo4, literally "beautiful country," where the second character is the simplified version used in mainland China) and 'China' (中国, zhong1 guo4, literally "middle country"). The digits in the Pinyin phonetic transcriptions (e.g., in 'mei3') are tone markings. In actual pronunciation the unstressed syllables 'guo4' in both words drop their falling tones for a neutral tone. (The traditional version of 'guo4' used outside of mainland China is '國', which should be treated as equivalent to the simplified '国' in any search. Google does this. The alternative is to detect whether the article uses traditional or simplified characters and apply the appropriate queries, which must then be maintained as two separate sets.) Our query here (not shown) says that we should flag any occurrence of 'America' if it occurs within 50 bytes of 'China'. This is known as a proximity match. The following is an excerpt of the results:

```
./chinese_data/bn/fp1/CBS20001126.1000.0700.sgm [210 - 238]: 美国...中国 (-1)
>>> 国国务院发言人鲍瑞什表示，美国21号搁置因为中国大陆
./chinese_data/bn/fp1/CBS20001217.1000.0600.sgm [335 - 376]: 中国...美国 (-1)
>>> 联席会议主席希尔顿将军正在中国大陆进行首次的访问
```

Now we see in line 3 that 'America' ('美国') occurs some 9 characters before 'China' ('中国') in the article CBS20001126.1000.0700.sgm . The first two intervening characters '2' and '1' take only one byte each, but the five Chinese characters '号搁置因为' take three bytes each in this particular encoding (Unicode UTF-8), for a total of 2+3×5 or 17 bytes in all, thus well within the limit of 50 bytes of separation (which was chosen arbitrarily here). In the next article at line 4 'China' occurs before 'America'. We can see 'China' in line 6 at bytes 335-376 of the article. The location of 'America' is not indicated as the location of 'China' was in line 1 (simply a peculiarity of the software that was being re-written at the time of the query). The '(-1)' in lines 2 and 5 indicates a proximity match.

Timing Results

Several factors influence the speed with which the Text Miner can process text in the form individual files. By far the biggest effect is the time required to read the files into the system. The fastest rates we observed (about 850 megabytes/sec for exact matches and about 750 MB/sec for regular expression matches) were achieved with text files already on local disk, that were reasonably large (on the order of megabytes), and with the number of matches on the order of tens of thousand per second or less. At the opposite extreme, if files are small or non-resident on the local disk, or if the number of hits exceeds roughly 100,000/second then the rates drop. The problem is that the operating system cannot satisfy the appetite of the hardware under these latter circumstances (on input), or cannot store the output quickly enough if there are large numbers of hits. Once the input/output barriers are overcome we observe a surge in performance levels that is stable over a large range of queries, hundreds of which were tried in this study. (The barrier effect will be illustrated below.)

We will quantify the observations above with some actual timings. First we emphasize the extremes. A simple example is the query:

```
'(REGEX:"bond" andthen[50] REGEX:"stock")'
```

searches for the word 'bond' preceding 'stock' with a separation of 50 bytes or less. When an entire year's worth of Reuters news articles from 2000-2001 is scanned as individual files from a non-local disk, the time is 6471 seconds, or nearly two hours, at a rate of 0.362 MB/sec. (The number of hits is 1992, a relatively small number for such a large corpus of data.) When we instead gather these articles into a few ".tar" files and place them on the local disk, we get a time of 4 seconds and an ingest rate of 725 MB/sec. This is a speed-up of about 2000 times. This rate was a bit lower than the maximum possible because we used the regular expression hardware, needlessly as it turns out, since an exact match is simpler and gives the identical set of hits. So for the query:

```
'(EXACT:"bond" andthen[50] EXACT:"stock")'
```

we get the fastest rates of 0.393 MB/sec and 837 MB/sec at the two extremes. Rates as high as 870 MB/sec were observed on other queries.

Now we look at the phenomenon of reduction of processing rates as a function of the number of hits. We consider the query where we check for a sequence of five of the first *n* letters of the alphabet (with *n* from 3 to 10) within 50 bytes of the same type of sequence. (The corpus is the same Reuters news articles mentioned in the paragraph above.) For example, ‘ceded’ within 50 bytes of ‘ebbed’ constitutes a match for a value of *n* of 5 or greater, because the letters fall in the range ‘a’ to ‘e’. The following table shows the scan rates:

n	Total sequences	Proximity hits	Ingest rate
3	95,219	5456	752 MB/sec
4	125,371	5728	751 MB/sec
5	537,486	15,243	752 MB/sec
6	857,900	37,384	751 MB/sec
7	1,106,530	99,800	752 MB/sec
8	1,987,988	262,025	96.5 MB/sec
9	4,872,467	823,101	25.6 MB/sec
10	5,232,899	1,302,700	14.1 MB/sec

The numbers in the second column represent the total number of sequences of 5 characters with letters up to the *n*th in the corpus (that is, where n=3 represents ‘a’ through ‘c’ and so on), and the third column represents the number of pairs of such sequences that occur within 50 bytes of each other. Notice that the ingest rates rise dramatically for decreasing *n* and become quite stable for n<8, that is, below a threshold of about 10^6 sequences and 10^5 hits. Clearly there is no input bottleneck involved here since the same corpus is scanned each time. Some combination of internal computation and/or output buffering must be responsible. (Below a purely computational example will be given.)

Another effect is important, namely the case where files are already resident in a large internal buffer (as when one performs a query twice in a row on data that does not exceed buffer size). In that case the second and subsequent queries on the same data will be faster, but this is a somewhat artificial situation and not of great interest.

A completely different influence on speed is computational in nature and has to do with processing of data internally in the Text Miner. Let us assume that two queries A and B each generate millions of matches in a data set. However, if A and B rarely occur near each other inside a document, then the proximity query for A near B will be somewhere below peak speed, even though the number of overall matches is small. Internally the Text Miner is forced to consider the proximity of millions of A hits to millions of B hits, which is computationally intensive. The example query which shows this situation is somewhat artificial, but illuminating:

```
'("q" near[10] "v") andthen[12] ("x" near[4] "y")'
```

Here there are three proximity constraints. First, ‘q’ and ‘v’ (in either order) must be within 10 bytes of each other, ‘x’ and ‘y’ (again in either order) must be within 4 bytes of each other, and in addition, the ‘q’ and ‘v’ must precede the ‘x’ and ‘y’ by 12 bytes or less (but not be overlapping). These letters occur in the corpus with the following frequencies: ‘q’ 4,018,257 times, ‘v’ 15,415,318 times, ‘x’ 6,640,603 times, and ‘y’ 17,522,729 times. However, the set of proximity conditions in the query restricts the total number of hits to only 479. The resulting ingest rate is only 81 MB/sec, or about a tenth of the peak speed. Clearly the number of proximity constraints that need to be checked is large enough to slow processing down. This is necessarily due to purely computational overhead rather than input/output bottlenecks. This is a very contrived example and such behavior was not observed in other queries.

Conclusions

As a result of this work it was possible to determine conditions that enable peak performance of the Text Miner consistent with rates quoted by the vendor. Most important is to guarantee that the hardware can be kept busy by minimizing latencies in inputting data. The most important factors in this are to use large files (by concatenating smaller files if necessary) on the order of at least a megabyte, and to place these files on one of the hardware’s local disks. Dividing up the work among multiple disks was not found to be helpful, though it is possible that this approach might be beneficial in some situations. To achieve good performance it is important to address input/output bottlenecks.

Many hundreds of queries were performed to check the accuracy and performance of the hardware and software. Some changes in the output format were made to make it clearer and more consistent. Searches in non-Western foreign languages was tried for the first time on this system and found to be very little different from searches in English.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.