Vol.2015-AL-154 No.3
2015/9/28

A Method for Dynamic Packing of Data Blocks for Over-the-Network Indexing

MARAT ZHANIKEEV[1,a)]

Abstract: The problem of over-the-network indexing has been raised in recent literature. Indexing is traditionally done on a local filesystem. When processing/access and storage are separated by network, traditional methods perform poorly, even if rewritten for over-the-network logic. The new engine called Stringex was newly proposed with over-the-network efficiency in mind. However, although blocksize is optimized by the method, it is fixed for the entire index. This paper looks into a way to allow for dynamic blocksize. The problem is formulated as dynamic packing of unit blocks for optimal over-the-network access. The new method also takes into account the issue of atomicity of operations in multiuser environments, where each of the multiple users can experience drastically different performance on end-to-end network paths.

Keywords: over-the-network applications, resource-constricted indexing, resource-efficient indexing, high-volume indexing, resource efficiency, resource optimization, network performance

1. Introduction and Problem Statement

Imagine a local application in form of a Javascript web application *(webapp)* running in a web browser. The Javascript code could be downloaded from a remote server, but it runs locally. It is also possible to run standalone webapps from an HTML file in the local filesystem. The webapp retrieves information from web pages as you are browsing them. Let us assume that this webapp retrieves metadata from pages on a scientific portal, where the unit page shows information for one scientific paper. Such an application needs to index the retrieved information, which is accumulated in multiple sessions over a prolonged period of time. The indexing is necessary in order to be able to browse or search the information later. Note that **indexing** for now is a generic term which includes tables, databases, etc., until it will be defined later in this paper.

You are also part of social collaboration which means that your index is shared with other people. In the age of clouds, it has become convenient to share files stored in the cloud. All you need to do is share your files (or a folder), and get access info from your cloud provider (Google, Dropbox, etc.). This information is readily available as part of your account. Your webapp will use it for authentication which is normally done using the OAuth protocol, details about which can be omitted for clarity.

This hypothetical situation creates interesting challenges. Since our webapp runs in a browser, computing and storage resources are limited. The indexing engine has to be designed accordingly. Clouds might impose limitations on the number of files as well as the total size of the storage. Finally, end-to-end throughput between the webapp and cloud storage has a physical limit. This optimization problem is referred to as **the Stringex Problem** in this paper, where the term is the concatenation of the words *stringent* and *indexing*. Indexing now can be fully defined as a type of database without the relational part – the technology has become popular recently as an alternative to structured data storage, where indexing provides the unstructured alternative. Lucene [5] is the de-facto industry standard of indexing tools. Lucene, of course, runs only locally and is not optimized for over-the-network operation. Hence the need for a Stringex-like alternative to Lucene.

There is some evidence of recent development in this general direction. For example, Fullproof [4] is a Javascript-based indexing engine which was designed from scratch to be able to run under HTML5. Fullproof, however, does not use cloud storage and does not consider the above optimization problem being simply a browser-based indexer.

Fig.1 is the visual representation of the above problem. The figure is split into left (conventional) and right (proposed) parts.

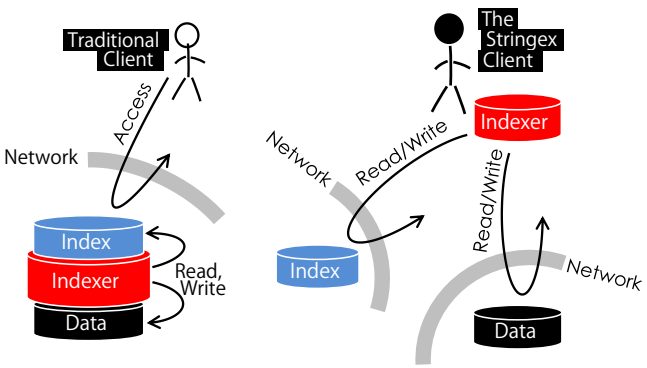

Fig. 1 The fundamental problem of over-the-network indexing.

[1] Computer Science and Systems Engineering
Kyushu Institute of Technology
Kawazu 680-4, Iizuka-shi, Fukuoka-ken, 820–8502 Japan
[a)] maratishe@gmail.com

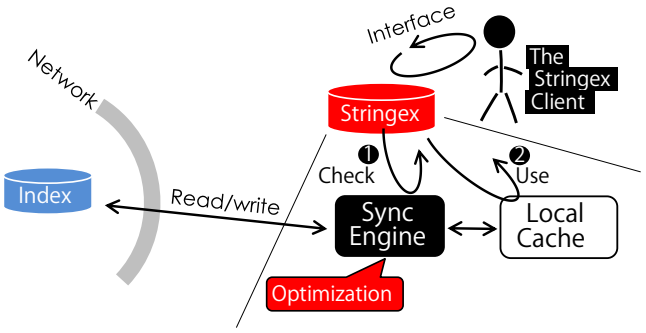

Fig. 2 The core design of a Stringex client implemented as a sync engine between local caching and remote index.

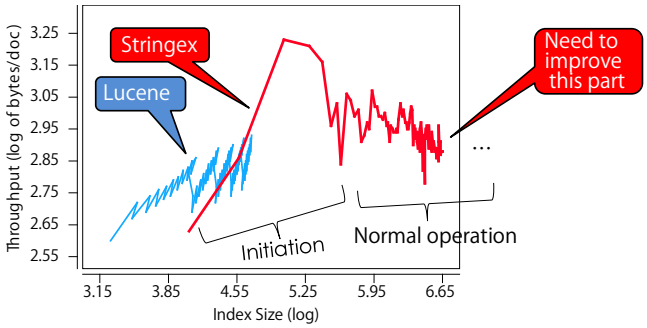

Fig. 3 Snapshot of performance under Stringex v.1, compared to the industry standard Lucene.

In the *conventional* indexing engines, while it is possible to allow users to have an over-the-network access (web API, remote shell, etc.), the index and indexer are tightly coupled, which means that they are commonly located either on the same machine or very closely located separate machines. In fact, as the figure shows, it is quite common to store the entire documents in Lucene, in which case even the data is stored at the same location.

So, what needs to change on the right side of Fig.1? For starters, metadata and data could be separated not only logically but also physically – meaning that the two could be stored on different machines. This is not a new way of thinking. The bigger is the size of data the more often the servicing technology strives to separate data from metadata. For example, this is a *must-do* for Big Data-level bulk – read the Hadoop overview at [8].

Also, even more importantly, the core of the Stringex formulation is the separation of **Indexer** from both the index and data. It is, in fact, the main premise of the proposal – handling of both the metadata and the data parts is to be done in **over-the-network** manner.

Unfortunately, both Lucene and Hadoop (bigdata) do not consider over-the-network access at present time. Instead, most research on the two tools is dedicated to the problems of concurrent access [9] and heterogeneous load [10].

2. Stringex Basics

When performing data-intensive activities over the network, one has to invest heavily into traffic efficiency. The two fundamental problems are:

- one cannot use any form of *locking* at remote location because deadlocks from leftover or abandoned locks are inevitable and cannot be counteracted effectively;
- concurrency can only be supported in form of *shared reads* but *isolated writes* – if one attempts to share writes, the problem of the first item have (but cannot) to be resolved.

Once the above fundamental problems are treated, the Stringex client can start working on efficiency.

Fig.2 shows the efficiency logic adopted by early Stringex and retained by the next virsion discussed in this paper. The logic follows the general *caching* approach by implementing a *Sync Engine* which serves as the border between local and remote content. A limited size of *Local Cache* is maintained by the Sync Engine and serves as the buffer for both metadata and data updates.

The efficiency offered by the Sync Engine should be obvious. For example, if human user updates the same document two or more times within a short span of time (common in practice), then the efficiency can be achieved simply by delaying the first update for some time. If the second update happens before the first update is commited – where **commit** involves sending the metadata/data to the remote index and removing it from local cache – then the syncing is done only once for any number of updates happening within the *timeout* time span within each other.

The hidden part of the efficiency is in the *optimization* potential of such a method. The basic optimization formulation can be found in the first paper on the Stringex problem in [1]. In a manner of speaking, this paper continues looking into optimization while presenting the upgraded version of the Stringex client.

3. Early Stringex: Performance Bottlenecks

Software for the old version, which also contains the benchmark code for comparison with Lucene can be found at [3]. It is based on the implementation of the logic presented in [1]. Note that while that paper also formulated the optimziation problem, the software implementation was simplified, avoiding the complexity of runtime optimization. Once of the simplifications called for a *fixed block size* for both metadata and documents – the very problem resolved by this paper.

Fig.3 shows a performance snapshot of Stringex versus Lucene with annotations. First, it is obvious that Stringex client has two main stages of operation. At the *initiation stage*, the client has to create many new files for both metadata and data, which is why the data exchange with the index – both in terms of file number and byte count – is high. However, this process saturates naturally, which is when the Stringex client reaches a more moderate level of data exchange at the *operation* stage. Note that the main reason for the spike at the Initiation stage is that the client uses *random hashing* to define prefixes of both metadata and document files, which is why the saturation is steep but short.

Comparing the performance with Lucene is difficult simply because Lucene writes to local files and does not care about throughput while the Stringex client writes to remote files over the network and cares *only* about traffic efficiency. However, Fig.3 still attempts to compare the two tools in the volume of the byte stream they generate. The figure shows that Lucene peaks at almost 1 order of magnitude higher than Lucene, but settles only slightly above Lucene (0.05 of an order of magnitude).

Still, having a similar level of performance to Lucene is not a

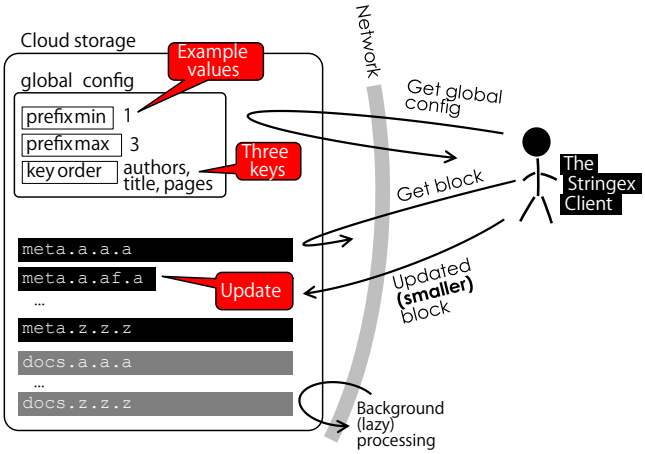

Fig. 4 The completely new design that simplifies the metadata layer but also allowed for variable size blocks.

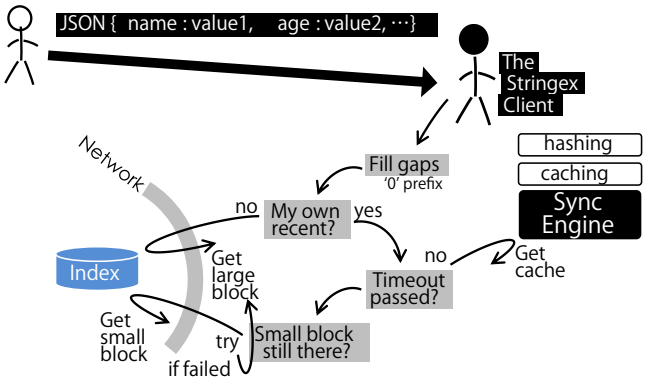

Fig. 5 The new algorithm implemented by Stringex v2 for any operation (lookup, update, create, etc.) triggered by user.

good thing. Again, since Lucene does not care about the throughput it generates, supporting the same level of traffic in the Stringex client will definitely result in poor overall performance. Specifically, users might have to wait for multiple seconds or even minutes while their documents sync to the index. So, as the annotation in Fig.3 says, this part in Stringex operation has to be improved.

4. Advanced Stringex: Performance Tweaks

Let us first set up the objectives for the new Stringex engine as an attempt to fix the problems described in the previous section:

- having *many metadata files* is messy – the old version of Stringex would keep separate file lists for each *key* in metadata, resulting in the Stringex client having to update multiple files when performing a major update;
- having *large files* contributes to sluggishness of the tool – however, when the block size is fixed, this issue becomes the unpleasant tradeoff between the number of files and download times.

The following improvements are made.

Combined metadata resolves the problem of multiple metadata files. In the current version, metadata is viewed as a flattened multiparameter space where each parameter is a *key* in metadata. When values are *hashed*, one can easily predict where in the long sequence a given set of metadata values is located. This design even allowes for *empty* keys by replacing the file prefix with a string which is not encountered in a conventional hashkey – the client to use such strings to fill in the empty keys (search, lookup, etc.) when bulding the name of a metadata file sync.

Asymmetric/Variable Blocks resolves the problem with heavy traffic exchange. It applies to both metadata and document files. The idea is to allow each independent Stringex Client to *upload* smaller blocks – ideally the minimal chunks which would hold the updated part of docs/metadata. This is a tricky part of the new engine, but is achievable in practice.

Fig.4 shows the design of the new Stringex engine. The left side shows the structure and filesystem design at the (remote) index location while the right side shows the simple way to deal with variable side blocks.

The *config file* has some global information. The two important parameters are *prefixmin* and *keyorder*, while the rest can be fine-tuned by each client without breaking the index itself. *Prefixmin* is important because this file is downloaded by each Stringex Client when metadata/docs are used for the first time. All the following handling can be done on a greatly reduced file – this is accomplished by increasing prefix length for one or more parts in the filename. Fig.4 shows an example, where the second *a* is grown to *af* – the length increases from 1 to 2. The *prefixmax* parameter in global configuration can be used to place a cap on the maximum length of prefixes, but index-side automation can be built to deal with any length, which is why it is advised that this decision is made at client side (possible user setting).

Note that this version of the Stringex client stores all metadata in *meta** files. Filename for any particular combination of values of keys can be calculated easily by taking hashes of values and building the filename from prefixes of hash values. Note that this is why *key order* is a global parameter – it is necessary not to confuse the order of prefixes in filename.

Also note that *doc.** files follow the same pattern – these filenames are also constructed from multiple prefixes obtained from shortened hash functions of metadata values. This is the connection between metadata and data. The earlier version of the Stringex client required metadata to point to filenames which contains the documents for any given configuration of metadata. In this version, both metadata and documents can be looked up by the client separately.

This logic also requires a *lazy background* processing job as is marked in Fig.4. This is because small files have to be merged with the large files on a given (fairly long) timeout, after which the contents should be merged into the main bulk at the maximum globally defined size. For example, it is wise to have timeouts at the range of *60s* or more at client side (in browser). In case of the lazy background processing, small blocks can be allowed to live for *hours* or even *days*. If a given metadata or document are *popular* among multiple users, this can drastically improve the responsiveness of over-the-network access to the index.

5. The New Algorithm

Fig.5 shows the new algorithm. As before, the user communicates with the Stringex Client using the JSON format [6]. In fact, the internal format is JSON for the index as well, with the only exception that JSON is stringified, compressed, and stored in a

one-line-per-item format in the files at the index side. To explain the algorithm, it is not necessary to make a distinction between the various handling functions, such as *create*, *lookup*, *update*, and others. The same generic algorithm applies to any handling function.

As the first step, the *filenames* have to be properly constructed. We know the global order of keys, but just in case some of the keys are not set (very common) by the user, the *gaps have to be filled*. The figure shows the *0* prefix, but anything meaningful (anything not to appear in a hash function) can be used instead, including the empty prefix, which can still be detected by counting the dots.

Having the legitimate filenames, one can now interact with the local sync engine by looking up the locally cached content. Note that the lookup is fast because it is local, so, the local engine can iterate through the various prefix length in order to find out whether or not a very specific part of metadata can be found in local cache.

There are situations when the file has been handled within the current session but has already timeouted and has been uploaded/synced to the remote index. This still leaves the possibility that the specific (small) file can still be found at the remote index (difference between timeouts is normally large), which needs confirmation. The confirmation can be implemented as *try*, that is, an attempt to simply download the file, and only when the operation files (no such file), the client can fall back to getting a larger block.

Note that the logic is built is such a way that smaller blocks are used either in local cache or remote index as much as possible, falling back to large blocks only as the last resort. Also note that, while the Stringex Client in this version has retained its main modules – specifically the *hashing* (MD5), *caching*, and the *syncing* logic, the structure of the index itself as well as the algorithm of interacting with the index has undergone a major upgrade.

6. Conclusion

This paper presented an upgraded version of the Stringex Client – the client side of the technology for over-the-network indexing. The remote side (index itself) of the technology has also undergone a major upgrade to support the changes in the client. All in all, the second version of this software has been rendered completely incompatible with the old version. In fact, the possibility of compatibility has completely disappeared after the first goal – variable block size – has been reached in this paper. However, including the block size, the new version offers several major improvements in performance and is recommended over the early version.

The first version of the software has been plagued with two main problem, both having to do with the structure of the index. On one hand, the metadata layer was split into sub-layers for each *key* (field, column, etc.) which resulted in having to sync multiple files for each update that created or changes one or more keys (which is very common). On the other hand, block size was fixed at a tradeoff point between having too many files and having to wait for a long time to sync large files over the network. Both problems are resolved in the new version of the software. The total number of files is reduced by merging all the metadata into flat sequence. However, smaller files are allowed to be created temporarily in the updated areas of metadata or documents, which does increase the total number of files but in a controlled/focused/local manner.

The work on the Stringex client will continue. One of the performance tricks in this version allowed for independent lookups in metadata and document spaces using the same lookup tuples. This is convenient when the purpose of the action is to find and read a document – in this case the metadata can be left untouched. However, this also opens the possibility of having *metadata-less* versions of the Stringex Client. Such versions might be useful in cases where the documents themselves are mostly indistinguishable from metadata. These are, in fact, the usecases covered by the traditional Lucene community [5], where it is common to store documents as part of the metadata.

Although the practical purposes of the Stringex Client might not be immediately obvious, there is a large class of *crowdsourcing* web applications [2] where the proposed functionality can helps support structured data exchange across the large social communities of contributors.

References

[1] M.Zhanikeev, “A New Practical Design for Browsable Over-the-Network Indexing”, International Conference on Information Science, Electronics and Electrical Engineering (ISEEE), pp.1686–1690, April 2014.
[2] M.Zhanikeev, “Maps2Graphs: A Socially Scalable Method for Generating High-Quality GIS Datasets Based on Google Maps API”, IEICE Technical Report on Intelligent Transport Systems Technology (ITS), vol.113, no.337, pp.73–76, December 2013.
[3] Stringex Project Repository. [Online]. Available: https://github.com/maratishe/stringex (January 2015)
[4] fullproof: Browser Side Indexing. [Online]. Available: https://github.com/reyesr/fullproof (September 2015)
[5] Apachet Lucene. [Online]. Available: lucene.apache.org (September 2015)
[6] JSON format. [Online]. Available: www.json.org (September 2015)
[7] Dropbox Core API. [Online]. Available: shttps://www.dropbox.com/developers/core (September 2015)
[8] Shvachko K., “HDFS Scalability: the Limits to Growth”, the Magazine of USENIX, vol.35, no.2, pp.6–16, 2012.
[9] Gimme all resources you have - I can use them!. [Online]. Available: http://blog.trifork.com/2011/04/01/gimme-all-resources-you-have-i-can-use-them/ (September 2015)
[10] Rasooli A., Down D., “COSHH: A Classification and Optimization based Scheduler for Heterogeneous Hadoop Systems”, Technical report of McMaster University, Canada, 2013.