Caching for faster duplicate and irrelevant record detection using a non-relational database & MD5 Checksum – A Case Study

Sri Harsha Anand Pushkala
Product Research &Development
Infosys Ltd
Bangalore,India
SriHarsha_A01@infosys.com

An.P.Dananjay
Senior Consultant
TCS Ltd
Japan
dananjay.dbz@gmail.com

Dr.G.S.Anandan
Chief Executive Officer
BTL Edl. Institutions
Bangalore
drgsanandan@gmail.com

Abstract **- In this paper, we propose a formal analysis approach to detect certain inconsistent and redundant records in day-to-day transactions of a retail domain by making such a process faster and much more reliable. Towards this goal, we introduce a interim storage stack by creating caches out of a No-SQL DB and using a hashing algorithm while inserting the data into the primary indexes of the caches. Also to speed up the processing and also to automate it, we need an efficient batch processing program that performs the storage into the stack and retrieval of the data for comparison at a later date, in a distributed environment. Our experimental evaluation confirms the viability of such a caching approach.**

Key words—Caching, No-SQL DB, Hashing Algorithm, Distributed and Parallel Processing

I. INTRODUCTION

With the advent of the era of Big-Data [5], there has been a great increase in the need to process enormous amount of data in the shortest possible time. Especially in a retail domain, there are daily transactional data that comes in huge volumes [6] of 1 – 10 Terabytes of data.

The first challenge is that such enormous files often have redundant data being processed which if removed can result in much faster processing of the data. To achieve faster processing we require a concurrent processing platform along with an interim storage functionary that enables faster insertion and retrieval of the data. We solved this problem with the usage of a document based non-relational database which facilitates faster insertion and retrieval of data for the challenge at hand.

To achieve faster processing of the data and to solve the first challenge, we have used a cache [8]. Cache, is normally a special high speed storage mechanism. It can be, either a reserved section of main memory or an independent high-speed storage device. A cache is a component that transparently stores data so that future requests for that data can be served faster. Recently many Web2.0 companies such as Twitter, Digg and Reddit that switch from MySQL to non-relational database (NOSQL) to provide scalable data storage solutions, has aroused strong developers interest on NOSQL. Today companies are even considering using the non-relational databases as caching mechanism. In this paper we discuss about the possibilities of using a No-SQL as a viable substitute for the existing conventional DB approaches and the existing caching mechanism.

The second challenge normally faced is that, conventional approach to records storage would involve the entire record being stored into the cache. This not only is wastage of the memory, but would result in a slower processing when the retrieval process begins. It has also been observed in most retail scenarios that the files would be based on the frequency of the transaction occurrence, for instance monthly, weekly or daily storage of the daily business transactions.

To address the second challenge this paper suggests using a hashing algorithm [12] that would further our cause of reducing the latency incurred in the reads and writes. Hashing algorithms such as MD5 checksum [11] is utilized which would generate a value for each record. During comparison we would compare these hashed values instead of comparing individual columns which would result in a faster processing of the data. In addition we have used the primary indexing of MongoDB to enable faster retrieval and insertion which is further discussed in section II (E).

The paper is structured as follows; section two specifies the background for the research conducted where we define the need for the caching mechanism, about the primary indexing as seen in MongoDB and also brief description of MongoDB and the hashing algorithm MD5 checksum. In the third section, we illustrate the core processes that take place and also how the threshold value is calculated. In the final section discussion the results of the empirical analysis are given.

In addition we also use a concurrent processing platform based out of a well known batch processing stack like Spring Batch enables us to process the data in a faster and efficient manner.

II. BACKGROUND

The work reported in this paper is part of an ongoing research project in the development of a Distributed platform. Our research is based on the following understanding: the need for a caching mechanism, the implications of using such a caching mechanism, the

ISSN: 2278-0181

functioning of MongoDB as a storage functionary and also using a Parallel programming stack.

A. Need for caching Mechanism :

In a typical retail domain scenario where there are millions of records there is need to ensure only valid records, need to be processed. To weed out the duplicate records [2, 4] and non-conforming records, where the files are in the order of Terabytes in the shortest possible time and in order to speed up this process, we need an interim caching functionary that would hold data under process before finally persisting valid records into a relational database.

But the need of the day is a caching stack that would be both fault tolerant and scalable. The cache should be able to handle any type of failure elegantly and able to deliver at any point in time. Since the size of the files being processed is huge, any loss of the data that need to be reloaded into the cache will result in the slow-down of the entire process. To solve this problem we use a non-relational database, namely MongoDB that has faster reads and writes [10] and also is fault tolerant [14].

***B.** Relevant Records cache and Unique Records Cache:*

In the non-relational database we create two types of cache, Relevant Records cache and Unique Records Cache.

Relevant Records cache is required for the irrelevant records detection. Irrelevant records are those that have anomalous entries in the pre-defined subset of columns that have been decided by the user to calculate irrelevancy. Any deviation of the data in these columns in the file needs to be weeded out. This constitutes for "Irrelevant Record Detection" and we utilize the relevant records cache to store the relevant records based on which "Irrelevant Record Detection" is done.

The second cache used is the Unique Records Cache which only holds non-duplicate records pertaining to a particular partner. The platform utilizes this cache to weed any duplicate entries that may come in a file sent by the customer. This constitutes for the "Redundant Record detection".

C. Insertion into the cache:

Initially during our research we tried inserting the records one after the other into the cache. Though the process might look simple, it was time consuming and the performance was very low. In order to improve the performance and reduce the latency in insertion of the data, we programmatically induced the bulk inserts [25] into the cache based on a predefined block size. The calculation of the block size (θ) was done as follows:

$$\theta = \beta * (\text{JVM Heap Space / Size of the file}) \qquad \dots (1)$$

In the above formula "β" is a Constant (Taken to be 0.20). Based on the above formula we calculate the block-size (θ) based on two important parameters: the JVM Heap Size and the size of the file. After the data is divided based on the block-size (θ), the blocks of data are inserted into the cache.

The graph in the Figure 1 depicts the time in milliseconds taken against the number of records in millions, to compare normal record-by-record insert compared for the induced bulk inserts. We can see from the Figure 1 significant amount of improvement on the performance as compared to normal record-by-record insert and reduced insert time.

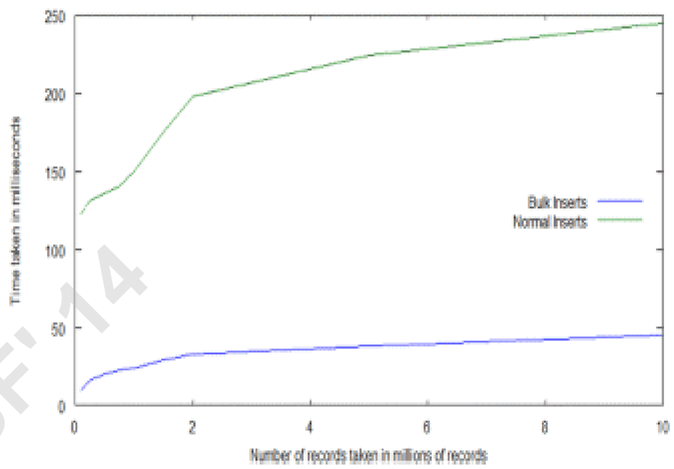

Figure 1: The above graph depicts the time in milliseconds along the x-axis against the number of records in millions along the y-axis and a comparison of normal record-by-record insert compared to induced bulk inserts.

***D.** MD5 Checksum:*

An MD5 checksum [11] is a very reliable way to verify data integrity. The MD5 algorithm takes data of arbitrary length and produces a 128-bit fingerprint of characters and numbers from that data.

It is proposed that it is computationally infeasible to produce two messages having the same output of numbers and characters. In, our scenario we will be using MD5 checksum to calculate a unique value for each and every record in the file and for the whole file too. So, that we can have row level as well as file level duplicate detection of the files.

In our platform we have used MD5 checksum for generating the hashed values for the data in a record, which expedites the process of data comparison, since we would compare only the hashed values of the entire data record instead of individual entries in the record under consideration.

ISSN: 2278-0181

E. *Primary Indexing in MongoDB:*

An index is a data structure that allows you to quickly locate documents based on the values stored in certain specified fields. Indexes in MongoDB [1, 7] are similar to indexes in other databases.

Creating an index comes with a contract of writing or updating data which could become a slow process as it requires an overhead of writing few extra bytes. In, our scenario we have a terminology of write once, use many times. All MongoDB indexes use a B-tree data structure [1, 16].MongoDB can use this representation of the data to optimize query responses.

MongoDB comes with two types of indexing _id indexing and secondary indexing. In our scenario we are using _id indexing which by default ensure Index on the primary key. We can enter any unique value in this field and MongoDB will create an index on the same by default and returns the result using only the index; MongoDB does not need to look at the documents, only the index is enough to fulfill the query.

III. CORE PROCESSES:

This paper suggests two types' processes. As seen in the figure 2, these processes happen sequentially on to find and weed out the invalid records from the file under duplicate and irrelevant record detection.

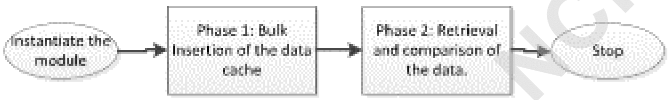

Figure 2: Illustrates the overall core process.

Phase 1:

During the phase 1 we have the following operations occurring:

1. Initially the supposed data corresponding to a partner resides in a conventional relational database. Using native commands we first copy the data into a file as direct insert into a cache was time consuming.
2. Once the data is stored on a file we have the bulk insertion happening into the cache. Using the formula in equation (1) we calculate the block-size based on which the data in the file needs to be divided for the bulk inserts to take place.
3. After the blocks division based on the block-size, firstly we calculate the hash value of each record as discussed in II(C) and then the bulk insertion process begins in the unique records cache. Secondly, we retrieve certain column constraints that the user has pre-defined for the irrelevant record detection. Whilst comparison any deviation from the set norms would deem the compared record from the user file as irrelevant and needs to be weeded out. So based on the column constraints we concatenate the data into a single string and then insert the same in relevant records cache.

Thus we have the data that is now prepared for the duplicate detection and irrelevant detection to happen. This process is repeated for any new client that may be using the platform.

Phase 2:

During the second phase of the core operations, we have actual retrieval and comparison process happening on two separate threads. There are two types of duplicate detection that will happen as discussed below:

a) Redundant Record Detection:

This defines a process that is used to check if the record coming in is a unique record and has not occurred in the past transactional records. In our research we found that approximately 20-40% of the records that may come in the form of transactions had duplicated data. This process ensures only the unique records get passed further on for processing.

In order to speed up the process of duplicate record detection the data that is read from the files is divided into chunks or blocks of data and processed parallel on different threads. Since there would be a write lock in MongoDB [1] and we would be using the primary indexes [1, 7] of the MongoDB, the duplicate records would not be inserted even if they are processed in parallel.

First we retrieve the data from the cache and create blocks of data based on a pre-configured block size (viz.1000, meaning less than or equal to1000 records in each block).After the division into blocks, we calculate checksum for each record in the block simultaneously on separate threads. Then the checksum is looked up in the MongoDB cache. Since duplicate records cannot be inserted into the primary index an error would thrown which would be logged in our logging framework that it is a redundant record. If not found, then the record is unique, the record is stored and can be sent further for processing, as illustrated in the figure 3.

ISSN: 2278-0181

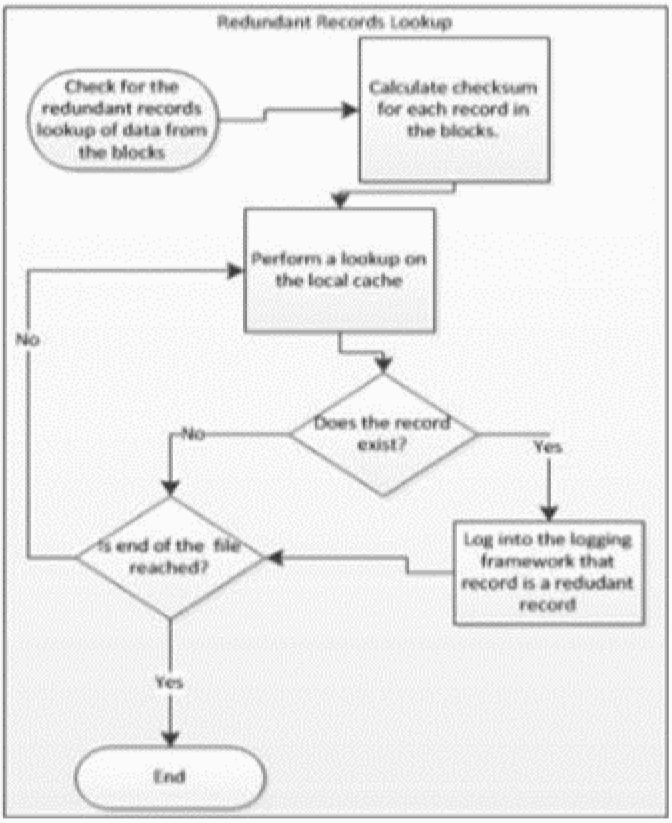

Figure 3: Illustrates the process of Redundant Record Detection.

b) Irrelevant Record Detection:

After the redundant records are weeded out, the records now need to be checked; if they conform to a pre-set standard and used to check if the record coming in is a irrelevant record (record that is not legitimate such as the product code of the transaction record does not exist).

The following steps are involved in the suggested detection, as also seen in the figure 4:

1. First step is detailing out which of the given columns in the files are to be used for the Irrelevant Record Detection. This is pre-set by the owner of the file. This helps us set the conditions based on which irrelevant records are determined.
2. Then the platform concatenates each column needed for irrelevant record lookup into a single string.
3. Then the platform checks if the concatenated columns are within the cache.
4. If not found, log into the logging framework that it is a irrelevant record and weed out the data from the file.
5. If found, then the record is relevant to the customer's file and can sent be further for processing.

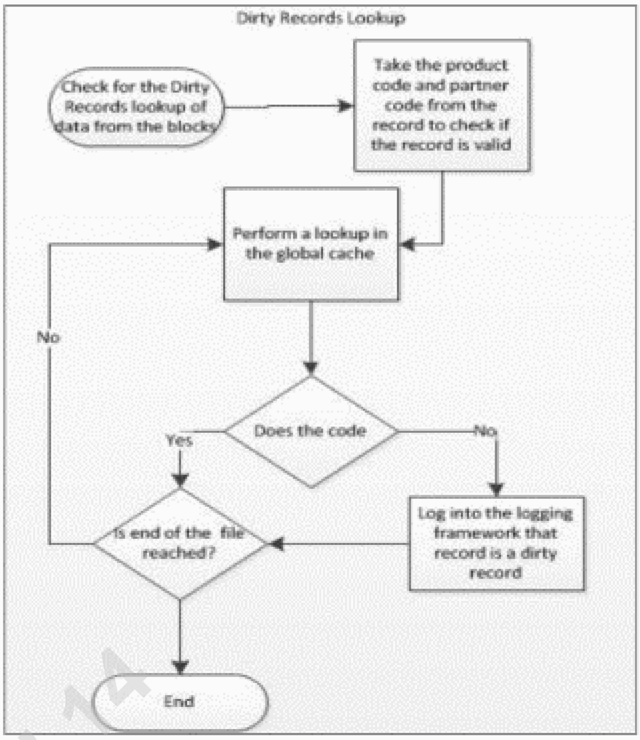

Figure 4: Illustrates the process of Irrelevant Record Detection.

IV. BENEFITS OF THE PROPOSED SOLUTION

Faster caching with reduced retrieval and insertion:

- By doing bulk inserts we are able to do faster inserts, as seen in section II (B).
- We avoid creating our own secondary indexes by using primary indexes [1, 7].
- Also since the primary indexing needs to be unique, if there are duplicate records within the same file the records are also weeded out automatically.
- Also since, MongoDB [1,18] uses resident mapping technique to map the latest data into the virtual memory using the LRU algorithm[1,18], there are chances that after the first retrieval for comparison, the data resides in the memory, thus causing lesser page faults and faster retrieval.

High Availability and Partition tolerant:

- This design provides us Availability and Partition Tolerance under the CAP's Theorem [23, 24] (Brewer's Theorem). But consistent view across all the nodes in the replica sets is something that is not done as the secondary of the replica set is replicated asynchronously. Also during sharding the collections, the first time data comes in, it would reside only on the primary shards. Then the data gets moved to the other shards based on the shard keys.

V. Empirical analysis

A comparative study was conducted as part of the our research at Trade Edge where MongoDB's performance was measured with that of the performance of Memcached [21], a in-memory cache.

The study was conducted with both caches operating on similar test conditions of 400 reads and writes of normal loads with 'Checksum' calculated on each record of a file and then the response time is tabulated. The tests were conducted on a commodity system with the following system Specification: 3GB Ram, O/S- Windows 32-bit, Processor: Pentium® Dual Core E5400 @2.70 GHz.

An observation was made that though Memcached [16] is comparatively faster than MongoDB in the initial stages. But its performance degrades as seen by the interim spikes in the figure 5, which causes delay in the insertion and retrieval of data as seen by our empirical analysis having both the caches function under similar conditions.

The results of our analysis can be seen in the Table 1 and the graphical representation is seen in figure 5:

S.No	Comparative results of MemcacheDB and MongoDB.		
	Time	MemcacheDB in mill-seconds	MongoDB in mill-seconds
1	9:40:01	0.732	1.842
2	9:40:02 AM	3.894	3.998
3	9:40:03	0.345	0.154
4	9:40:04 AM	0.166	0.563
5	9:40:05	0.235	0.416
6	9:40:06 AM	0.093	0.649
7	9:40:07	0.278	0.793
8	9:40:08 AM	0.124	0.543
9	9:40:09	0.453	0.559
10	9:40:10 AM	0.25	0.239
11	9:40:11	0.241	0.357
12	9:40:12 AM	0.125	0.45
13	9:40:13	0.675	0.839
14	9:40:14 AM	0.753	0.645
15	9:40:15	1.986	0.839
16	9:40:16 AM	3.879	0.673
17	9:40:17	0.578	0.563
18	9:40:18 AM	2.673	0.938
19	9:40:19	5.175	0.235
20	9:40:20 AM	0.437	0.213
21	9:40:21	3.968	0.498
22	9:40:22 AM	0.237	0.987

Table 1: Response time taken with 400 reads and Writes of normal loads with 'Checksum' calculated on each record of a file. System Specification: 3GB Ram, O/S- Windows 32-bit, Processor: Pentium® Dual Core E5400 @2.70 GHz

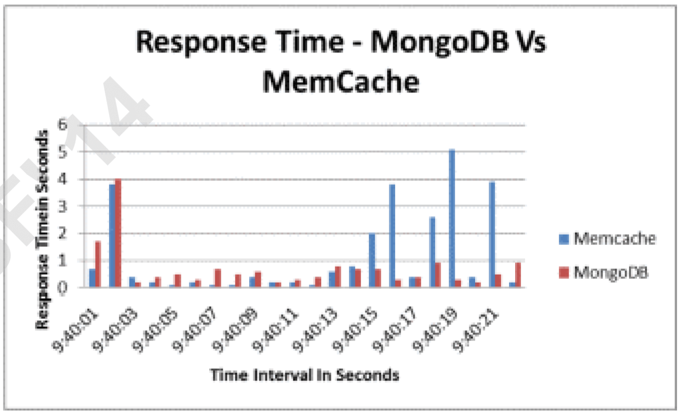

Figure 5: Depicts the graphical representation of the tabulated entries in the table 1.

VI. Existing work

- "Travel IQ" [20] travel company has used a similar type of caching based out of MongoDB but they had not used any hashing algorithm before inserting the data into the cache. If this was implemented then this would have further enhanced their performance.

VII. Future Work

The scope of this methodology is varied and could be leveraged in any domain where there is need to weed out any redundant or irrelevant records in the most effective and fastest way.

- Future Development could leverage open source Big-Data stack like Hadoop to provide much higher levels of concurrent and faster lookups.
- Other caching and No-SQL frameworks can also be considered for the design such as CouchDB, Cassandera , Hazelcast.

ISSN: 2278-0181

- Other checksum methodologies are also being considered like SHA-1, SHA-256 and so on.

VIII. ACKNOWLEDGMENT

I acknowledge with profound thanks and gratitude, the works of various genius from Dr. Rakesh, Principal Research Scientist-Infosys, where many definitions and ideas have been utilized in this paper. I would also like to thank Mr. Jude Fernandez, Research Analyst - Infosys who helped me through the entire process of this paper. Finally, I would like to also thank Dr. G.S Anandan and Dr. Dananjay who gave me their valuable inputs during the drafting of this paper and helped all the way along in co-authoring this paper.

REFERENCES

[1]. "Brewer's CAP Theorem", julianbrowne.com, Retrieved 02-Mar-2010.

[2]. Banker, Kyle. MongoDB in action. Manning Publications Co., 2011.

[3]. Bivas Das, "Decentralized fault tolerant caching with Memcached" (January 1, 2011). ETD Collection for University of Texas, El Paso. Paper AAI1494339.

[4]. Burtica, Ruxandra, et al. "Practical application and evaluation of no-SQL databases in Cloud Computing." Systems Conference (SysCon), 2012 IEEE International. IEEE, 2012.

[5]. Chodorow, Kristina, and Michael Dirolf. MongoDB: the definitive guide. O'Reilly Media, 2010.

[6]. Chodorow, Kristina. Scaling MongoDB. O'Reilly, 2011.

[7]. Christopher M. Bishop (2006) Pattern Recognition and Machine Learning, Springer ISBN 0-387-31073-8.

[8]. Chu, Cheng, et al. "Map-reduce for machine learning on multicore." Advances in neural information processing systems 19 (2007): 281.

[9]. http://cs229.stanford.edu/notes/cs229-notes1.pdf

[10]. http://en.wikipedia.org/wiki/Cache_(computing)

[11]. http://en.wikipedia.org/wiki/MD5

[12]. http://www.cse.iitb.ac.in/~sudarsha/db-book/slide-dir/ch12.pdf

[13]. Journal Article 2004 0885-6125 Machine Learning Lessons and Challenges from Mining Retail E-Commerce Data http://dx.doi.org/10.1023/B%3AMACH.0000035473. Kluwer Academic Publishers 2004-10-01 data mining data analysis business intelligence web analytics web mining OLAP visualization reporting data transformations retail e-commerce Simpson's paradox sessionization bot detection clickstreams application server web logs data cleansing hierarchical attributes business reporting data warehousing Kohavi, Ron Mason, Llew Parekh, Rajesh Zheng, Zijian 83-113 English

[14]. Khan, Iqbal. "Distributed Caching On The Path To Scalability". MSDN (July 2009).

[15]. Lith, Adam; Jakob Mattson (2010). "Investigating storage solutions for large data: A comparison of well performing and scalable data storage solutions for real time extraction and batch insertion of data" (PDF). Göteborg: Department of Computer Science and Engineering, Chalmers University of Technology. p. 70. Retrieved 12 May 2011.

[16]. Location and Detection of near-duplicate files in Distributed Systems VaibhavGautam, University of Southern California

[17]. Lynch, Clifford. "Big data: How do your data grow?." Nature 455.7209 (2008): 28-29.

[18]. Manyika, James, et al. "Big data: The next frontier for innovation, competition, and productivity." (2011).

[19]. MongoDB as a queryable cache-Martin Tepper, monogreen.de2011-03-25 http://www.slideshare.net/mongodb/mongodb-as-a-fast-and-queryable-cache.

[20]. Nancy Lynch and Seth Gilbert, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services", ACM SIGACT News, Volume 33 Issue 2 (2002), pg. 51-59

[21]. Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C&Venkataramani, V. (2013, April). Scaling Memcache at Facebook. InProceedings of the 10th USENIX conference on Networked Systems Design and Implementation (pp. 385-398). USENIX Association.

[22]. NoSQL Relational Database Management System: Home Page". Strozzi.it. 2 October 2007

[23]. Paul, S; Z Fei (1 February 2001). "Distributed caching with centralized control". Computer Communications 24 (2): 256–268.

[24]. Phil Simon (March 18, 2013). Too Big to Ignore: The Business Case for Big Data. Wiley. p. 89. ISBN 978-1118638170.

[25]. Smith, Alan Jay. "Cache memories." ACM Computing Surveys (CSUR) 14.3 (1982): 473-530.

[26]. Vassilios S Verykios, Ahmed K Elmagarmid, Elias N Houstis, Automating the approximate record-matching process, Information Sciences, Volume 126, Issues 1–4, July 2000, Pages 83-98, ISSN 0020-0255,http://dx.doi.org/10.1016/S0020-0255(00)00013-X.

[27]. Vassilios S Verykios, Ahmed K Elmagarmid, Elias N Houstis, Automating the approximate record-matching process, Information Sciences, Volume 126, Issues 1–4, July 2000, Pages 83-98, ISSN 0020-0255, http://dx.doi.org/10.1016/S0020-0255(00)00013-X.