STUDIA INFORMATICA 2012
Volume 33 Number 2B (106)

Paweł KASPROWSKI
Politechnika Śląska, Instytut Informatyki

CHOOSING A PERSISTENT STORAGE FOR DATA MINING TASK

Summary. The amount of data available for mining or machine learning is increasing. Therefore one of the main problems of nowadays mining is decision how to persistently store that data in the way that it is easy and fast to load and save by mining algorithms. When data is too big to fit in the memory, there are two common ways to handle it: text or binary file in own format or ready-to-use universal database engine. Both have advantages and disadvantages. As for database engine, the most popular storage is a relational database server. Recently another promising option became non-relational databases like document-oriented databases. The work presented in this paper analyses how different storages behave for big amounts of data. Experiments compare efficiency of these storages for some classic mining tasks.

Keywords: persistent storage, data mining, document-oriented database

WYBÓR SPOSOBU PRZECHOWYWANIA DANYCH STOSOWANYCH W DATA MININGU

Streszczenie. Ilość danych dostępnych do analizy i algorytmów uczenia się z roku na rok rośnie. W związku z tym coraz większym problemem staje się przechowywanie tych danych w sposób trwały, który umożliwi szybki odczyt i jednocześnie bezpieczny zapis wyników analizy. Są dwa najpopularniejsze rozwiązania problemu trwałego zapisu danych: pliki we własnym formacie binarnym lub tekstowym albo relacyjne bazy danych. Oba te sposoby mają swoje zalety i wady. W ostatnich czasach popularność zaczynają zdobywać nierelacyjne bazy danych. W artykule zaprezentowano eksperyment mający na celu porównanie możliwości tych trzech sposobów przechowywania danych.

Słowa kluczowe: data mining, bazy nierelacyjne, przechowywanie danych

1. Introduction

The problem of storage of data used in machine learning or data mining tasks is often underestimated. Most libraries implementing data mining and ML algorithms assume that all data fits in memory. However, present data sets are becoming bigger and bigger and, despite of increasing amount of memory available for computers, the problem of data storing should get more consideration. Moreover, even when all data fits in memory, every analyzed data set must be eventually somehow persistently stored.

Data set may be stored in a specialized task-specific storage. However creating own storage is expensive – in most cases it is better to use a ready to use, universal solution. Although there are efforts to create universal storage for data mining tasks like inductive databases [23], most researchers pay attention to usage of standard relational database and to use SQL language to manipulate the data [21, 25]. The main advantage of such choice is that SQL language is widely known and that relational DBMS' are stable and reliable. The drawback is that relational data structure doesn't suit well for data mining [8].

Recently, there is also a growing attention to so called NoSQL databases which don't work according to relational paradigm. Such storage may be used instead of classic relational database [18, 17]. It seems to be competitive to relational databases when considering usage for storing data for mining.

2. Previous work

As it was stated in the previous section the problem of saving data used for mining has been researched for years. In one of the most notable examples [24] authors compared different algorithms which could be used for mining associations. Authors considered using pure SQL language and SQL enhanced with object-relational extensions in IBM DB2 database engine. Subsequent works tried to propose new pure SQL solutions [21, 20] or suggested some SQL extensions [10, 14]. Some authors proposed moving mining logic from client to DBMS side using stored procedures and user defined functions (UDF) [19]. Some other works handle with very large databases by aggregating and sampling [27, 5].

3. Contribution

The paper analyses possible choices how to persistently store data to be able to retrieve it fast and to use it without the need to load the whole dataset into memory. Several possibilities how to store data were chosen and compared.

The contribution of the paper is the comparison of efficiency of four different storage engines for the same data and the same set of operations. The main novelty of the paper is usage of document-oriented database engine MongoDB which occurs to be competitive.

4. Assumptions

Standard data for machine learning tasks consists of list of objects which are analyzed (to cluster, classify, find associations or similarities etc.). The object is defined by some number of attributes. For simplicity, in further researches we assumed that every object has a set of attributes which have only simple types (i.e. strings or numbers). It's not always the case because in general object may be build from other objects. However it seems that such 'flat' objects containing only simple attributes are the most common. Every attribute has name (key) and value, values may have different types.

There are mainly two factors which influence data mining algorithms efficiency: number of objects (n) and number of attributes (p) [8]. For some tasks number of attributes may be relatively small and well defined. However there are plenty of examples where p is quite big and, moreover, it is different for every object. One of them may be a classic basket analysis where single transaction may consist of number of items varying from 1 to even 1000 [24]. Another example may be inverted index used in text mining, where for every indexed word an attribute is created. Yet another example may be data taken from biometric measurements [13]. In further research in this paper we assume that sets of attributes for different objects may differ and may be considerably big.

The task was to store the same data probe in different storages and then test how it is possible to maintain standard data mining tasks on this probe.

5. Possible data storages

Every data mining task starts with gathering data. The data may be delivered in different formats but to start mining it must be gathered together in one unified format. The assumption was made that the data set consists of n objects. For simplicity we assumed that every object has one integer attribute *id* which is unique and different number of additional numeric (integer) attributes. Such a probe must be kept in some persistent storage and be available for direct analyses.

Although the available operational memory is increasing – personal computer with more than 4GB of memory is not rare – the same is for data mining databases. When data has several GB it is not possible to fit it into memory. Experiments provided in the paper assume that amount of mining data cannot be fully loaded from a hard disk.

5.1. Plain file

The most obvious storage of persistent mining data is a plain file. Because creating own binary format may be complicated, it was assumed that plain file means plain text file in some structural format. XML [7] has established position as universal format for data storage in text files. However, because XML produces a lot of configuration data, XML files are quite big. That's why a considerably more consistent but equivalent JSON [11] format was used in this experiment. One line in file represents one object. Every object has attribute *id* which is unique among objects and a set of integer attributes denoted $a_0 \ldots a_{p-1}$ which number may be different for each object.

Example of one object definition in JSON format:

```
{"id":6,"a63":4,"a4":56,"a61":99,"a62":16,…}
```

To achieve tests fairness, access to all data storages was implemented in the same Java environment.

Implementation of JSON text file storage in Java is very easy. One doesn't need any special drivers, only some ready-to-use classes from JSON.ORG [12] are required. Of course, the main disadvantage of the plain file is that access to data is always sequential. The other disadvantage is lack of concurrent access to data. When one mining application uses the data, it cannot be used by others.

5.2. Relational database

Another choice may be a classic relational database storage in client-server architecture [4]. Access to data in relational database is not sequential – from application's point of view it just "requests" some data in SQL language and database server is responsible for delivering it. Moreover data may be concurrently used by multiple users – assuming that they only read data, no locks are necessary.

Because of the assumption that number of possible attributes may be big and different for every object, it was impossible to create one table *objects* with all attributes as columns in horizontal manner. Therefore the key-value vertical structure of the table was proposed. A table *objects* was created with columns: *id, key, value*. The inconveniency of such format was that objects were spread among multiple rows. To retrieve all attributes of one object it was necessary to retrieve all rows from the table which had the same value of *id* attribute. The attribute *val* had type `int` because it was used to store only integer values but in more general example it could have `varchar` type for storing both numeric and text attributes.

The script creating objects table

```
create table objects (id int, key varchar(10), val int)
```

Two database servers MySQL [16] and PostgreSQL [22] were used as data storage because both are light, free and easy to handle.

Similarly to the previous case, Java implementation of application that uses relational database storage is very easy. One needs a proper JDBC driver and must create a set of SQL select statements to retrieve data. The obvious advantage over plain text file is that access to data is not necessary sequential and selections of specific data are more efficient, especially when using indexes. However, the main problem is that data belonging to one object is spread among multiple independent rows what makes querying more difficult.

5.3. Document-oriented database

The last choice was a solution which is relatively new but gaining more and more attention – a document-oriented database. There was one of the most popular solutions chosen – MongoDB [15]. Such kind of database is not relational, it means it doesn't have a classic table-column-primary_key-foreign_key structure. Objects in MongoDB database are stored in 'collections' as 'documents' in the same way as rows in tables. What differs a collection from relational table is that documents in the same collection may have different structure – a different set of attributes. MongoDB stores data in Binary JSON (BSON) format. It works as classic server expecting requests on default port 27017. To communicate with MongoDB server one needs to use a driver. There are drivers for different environments available including Java, Perl and others. Application communicates with MongoDB server in a special query language.

Similarly to relational databases, MongoDB implementation in Java involves using a driver and sending queries created in a special language similar to Query by Example. The queries may be quite complex. The query is in fact just a JSON document which is sent to the server.

6. Test probes

To simulate database mining tasks a set of data probes was generated. Every probe consists of *n* objects. Every object has obligatory *id* attribute and may have maximum of *p* other attributes (with keys $a_0..a_{p-1}$ respectively). As we assume equal distribution of number of attributes for each object, the average number of attributes may be computed as $m=p/2$.

Every attribute's value is an integer from range 0-MAX. After some experiments MAX was arbitrary chosen as 1000 for every probe.

There were 3 probes with different *n* and *p* values tested. The choices are presented in Table 1.

Table 1

List of test probes taken into consideration in experiments

Name	*n* – number of object	*p* – max number of attributes
test100k2k	100,000	2000
test200k1k	200,000	1000
test50k4k	50,000	4000

7. Experiment

The aim of this experiment was to compare how different storages act with classic data mining tasks. We emphasized the part of mining that involves reading object or object's attributes from data set and (for one task) saving the results. Because in this case only the code retrieving data was interesting for us, the 'mining' itself was maximally simplified – only some interesting parts of known algorithms were tested. The time of execution was measured for every Task and every storage. These times were then compared.

The first task (Task1) was finding objects having a specific set of attributes. It is a common task for text mining where we are looking for documents fulfilling a query (set of words) [8]. It may be also useful in algorithms searching for associations. Because the result may be quite big and may be used in further algorithms – it was persistently stored in another file or another table or collection (depending on the storage used).

Second task (Task2) was very similar to the previous one: it was finding number of objects with a specific set of attributes. Such a task may be used for example as a part of Apriori algorithm. In this algorithm we are looking for meaningful associations and therefore we are interested in frequency of specific itemset (e.g. in basket analysis tasks) [8, 24].

The third task (Task3) was finding average values of the specified list of attributes. Such problem is a part of k-means algorithm when we are centering the position of the cluster among all objects assigned to it.

The last task (Task4) was finding the objects most similar to example. The example is an object with some attributes. We used a classic Euclidean distance as the measure of similarity between two objects. This task is of course very frequent task in data mining – for instance in k-means algorithm when searching for object-to-cluster assignments.

All tasks computational complexities obviously depend on the analyzed data size – number of objects (*n*) and average number of attributes (*m*). But another important factor in tasks 1, 2 and 3 is the size of attributes' set provided as the parameter (*s*). Similarly, in task 4 the number of attributes of object given as an example is important. Therefore all experiments were provided for datasets with different values of three factors:

- *n* – number of objects taken into consideration,
- *m* – average number of attributes for objects,
- *s* – number of attributes given as parameters.

8. Implementation of tests

To evaluate how different storages work in the same Java environment, it was necessary to create different implementations of one common interface. The class *MyObject* is a very simple class containing a map of attributes and id field.

There were overall six implementations of *ITester* proposed:

- TextFile – implementation that opens file in JSON format and sequentially reads it to extract objects or values that should be returned.
- MongoDB – implementation that reads data from MongoDB database.
- MySQL – implementation reading data from database stored in local MySQL server using JDBC driver.
- MySQL_I – implementation that uses indexes created on fields *id* and *key* and one composite index on both fields *id,key*.
- PostgreSQL – implementation using local PostgreSQL database via JDBC driver.
- PostgreSQL_I – the same as PostgreSQL implementation but it also uses indexes on fields *id* and *key* and one composite index on both fields *id,key*.

It's worth to notice that there was one more implementation tested in the beginning, namely: MongoDBIndex. MongoDBIndex used MongoDB database with indexes created on all attributes taken into consideration. Because results were comparable to MongoDB implementation without indexes it was removed from final presentation for clarity.

9. Test results

All tested storages (i.e. text file as well as MongoDB, MySQL and PostgreSQL databases) were filled with exactly the same data. Therefore the results of methods (averages, numbers of objects etc.) were exactly the same for every storage for every test. The testing application measured the time of execution for every method and checked if it returned proper values. The tests were invoked multiple times for every storage and the results of measurements presented are the averages of that multiple trials. It is worth to notice that database engines try to cache results of aggregations, so, to make tests fair, a query cache had to be cleared before every trial.

All tests were performed on four computers of the same type (Lenovo, Xeon W3550, 4GB RAM) in the same Win7/64-bit environment. For every sample the same set of tasks was executed for different number of attributes given as tasks' parameters (*s*). The experiments were executed for numbers of parameters (*s*): 1, 10, 50 and 200.

The tests used three freely available database servers:

- MySQL 5.5.11 Winx64,
- MongoDB 1.8.1 x86_64,
- PostgreSQL 9.03 64bit.

Because results for samples test50k4k and 200k1k were very similar to results obtained for sample 100k2k, only results of the latter were presented.

Table 2

Results of Task1 for test100k2k sample [s]

Type	*s*=1	*s*=10	*s*=50	*s*=200
TextFile	92	77	76	76
MongoDB	139	26	31	27
MySQL	-	-	-	-
PostgreSQL	2308	300	329	584

The aim of Task 1 was extraction of objects that have all attributes from the given list. The length of the list was s = 1, 10, 50 and 200. For TextFile implementation it was just reading all objects and searching for attributes. Every object found was saved in another file. That is why results are almost equal for different values of *s*.

The number of objects to be read and copied was more important for other implementations, because they used filters to read only objects fulfilling the condition. When *s* is low (like *s*=1) much more objects is returned than for big values of *s* (like *s*=200). One may guess that execution time should be lower for bigger *s*. and it is true for most MongoDB implementations. However it occurs that more complicated condition (200 or-like conditions for *s*=200) has also impact on execution time – especially for PostgreSQL. That is probably why *s*=200 result of PostgreSQL is worse than lower ones.

It was impossible to perform this test on MySQL database because performance of INSERT INTO … SELECT command in MySQL engine is very poor.

The results show that MongoDB beats opponents in all trials with s bigger than 1.

Table 3

Results of Task2 for test100k2k sample [s]

Type	*s*=1	*s*=10	*s*=50	*s*=200
TextFile	76	75	76	76
MongoDB	1	7	14	24
MySQL	15	19	23	31
MySQL_I	18	87	214	317
PostgreSQL	63	90	95	566
PostgreSQL_I	24	35	115	575

Task 2 was calculation how many objects have all attributes from the list. As in the previous task the results of TextFile implementation are similar regardless of probe and number of attributes. MongoDB proves to use count() operator very effective and is clearly the best for every experiment. MySQL is significantly slower than MongoDB but faster than TextFile and PostgreSQL. Interesting fact is that MySQL performs even 10 times worse when it has indexes. It may be interesting point for further studies of nature of MySQL query plans. Examination of that plans show that MySQL always tries to use indexes, what occurs not practical when the query must read many rows. As for PostgreSQL, it efficiently uses indexes for low *s* but always works slower than its MySQL counterpart without indexes.

Once again MongoDB is the fastest storage for all trials.

Table 4

Results of Task3 for test100k2k sample [s]

Type	*s*=1	*s*=10	*s*=50	*s*=200
TextFile	76	76	76	77
MongoDB	3	24	113	507
MySQL	15	19	25	36
MySQL_I	1	4	13	54
PostgreSQL	35	83	201	568
PostgreSQL_I	39	39	103	579

Task 3 was the calculation of the averages for given attributes. As it was explained in previous part MongoDB doesn't support aggregation functions directly and it must be programmed in JavaScript language. Such a function works fine for low values of *s* but has problems with $s > 10$. This time MySQL uses indexes very efficiently and is the fastest storage for all trials. MongoDB and PostgreSQL results are similar for bigger values of s with MongoDB being faster for $s = 1$ and $s = 10$.

Task 4 was comparison of all objects to a given example. Regardless of number of attributes of example object (*s*) it was always table scan through all rows and attributes. Therefore results are similar for every *s* value. It is not a surprise that lowest level method – TextFile – performs best in this trial. Other storages introduce some unnecessary layers so cannot be better. Interesting fact is that MongoDB is only a fraction worse than TexFile and four times faster than the best relational storage (MySQL).

Table 5

Results of Task4 for test100k2k sample [s]

Type	*s*=1	*s*=10	*s*=50	*s*=200
TextFile	105	105	105	105
MongoDB	118	120	121	122
MySQL	446	496	496	496
MySQL_I	776	809	812	810
PostgreSQL	823	826	824	825
PostgreSQL_I	2026	2028	2035	2016

10. Conclusions

The paper presented results of tests performed on three different data storages. The aim of work was to compare how these storages behave in classic mining tasks. As "classic" tasks four tasks were chosen:

- find and unload objects having a specific set of attributes,
- count the number of objects having a specific set of attributes,
- calculate average values of specific set of attributes,
- find object the most similar to the example object.

The results showed that for objects with big and changing number of attributes classic relational engines performance suffers from the necessity of vertical storage in key-value format. Characteristics of such mining data (big number of attributes, rare selections with well defined conditions and changing number of attributes) makes it inconvenient for relational engines and makes it difficult to perform fast.

On the other hand plain file format doesn't offer good performance when data selection is necessary. The file is read sequentially so every query needs to read the whole file to return correct results.

That is why it seems logical to consider document-oriented database when choosing the storage for data used in mining or machine learning tasks. It proved to be stable and usable for every task and seems to be a golden mean.

Of course the tests presented in this work don't cover all possible usages. Further work may include experiments how storages cope with textual data. Another option may be analyses of more complex objects definitions – not just simple type attributes – like graphs in [26].

Only one example of document-based database was tested in the experiment. It could be interesting to compare results with other similar solutions like CouchDB [1] or Apache Jackrabbit [2].

BIBLIOGRAPHY

1. Anderson J.: CouchDB. The Definite Guide. O'Reilly, 2010.
2. Apache Jackrabbit, http://jackrabbit.apache.org/.
3. Caraciolo M.: Map Reduce with MongoDB and Python. Artifical Intelligence in Motion blog, http://aimotion.blogspot.com/2010/08/mapreduce-with-mongodb-and-python.html
4. Codd E. F.: A relational model of data for large shared data banks. Commun. ACM, 1970.
5. Deshpande A., Madden S.: MauveDB: Supporting Model-based User Views in Database Systems. SIGMOD, 2006.
6. Džeroski S., Lavrač N.: Relational data mining. Springer Verlag, Berlin, Heidelberg 2001.
7. Extensible Markup Language (XML), W3C domain, http://www.w3.org/XML.
8. Hand D. J., Mannila H., Smyth P.: Principles of Data Mining. MIT Press, 2002.
9. Holsheimer M., Kersten M., Mannila H., Toivonen H.: A Perspective on Databases and Data Mining. KDD, 1995.
10. Imieliński T., Virmani A.: MSQL: A Query Language for Database Mining. Data Mining and Knowledge Discovery, Kluwer Academic Publishers, 1999.
11. Introducing JSON (Javasript Object Notation), http://www.json.org.
12. JSON in Java, http://www.json.org/java/index.html.
13. Kasprowski P., Ober J.: Eye movements in biometrics. Biometric Authentication Workshop, European Conference on Computer Vision ECCV'2004, Lecture Notes in Computer Science, Springer, Prague 2004.
14. Meo R., Psaila G., Ceri S.: A New SQL-like Operator for Mining Association Rule. Proceedings of the 22nd VLDB Conference, India, 1996.
15. MongoDB document-oriented storage, http://www.mongodb.org.
16. MySQL web page, http://www.mysql.com.
17. NOSQL Databases, http://nosql-database.org.
18. Oram A.: MongoDB experts model the move from a relational database to MongoDB. O'Reilly Community, http://broadcast.oreilly.com/2010/04/mongodb-experts-model-the-move.html.
19. Ordonez C., Pitchaimalai S. K.: Fast UDFs to compute sufficient statistics on large data sets exploiting caching and sampling. Data & Knowledge Engineering, Vol. 69, No. 5, 2010, p. 383÷398.
20. Ordonez C.: Integrating K-Means Clustering with a Relational DBMS Using SQL. IEEE Transactions On Knowledge And Data Engineering, Vol. 18, No. 2, 2006.

21. Ordonez C.: Programming the K-means Clustering Algorithm in SQL. KDD, 2004.
22. PostgreSQL database system web page, http://www.postgresql.org.
23. Raedt L. D.: A perspective on inductive databases. ACM SIGKDD Explorations Newsletter, 2002.
24. Sarawagi S., Thomas S., Agrawal R.: Integrating association rule mining with relational database systems: Alternatives and implications. SIGMOD, 1998.
25. Suresh L., Simha J. B.: Novel and Efficient Clustering Algorithm Using Structured Query Language. Proceedings of the 2008 International Conference on Computing, Communication and Networking, 2008.
26. Vicknair C.: A Comparison of a Graph Database and a Relational Database, ACM Proceedings of the 48th Annual Southeast Regional Conference, New York 2010.
27. Zhang T., Ramakrishnan R., Livny M.: BIRCH: An Efficient Data Clustering Method for Very Large Databases SIGMOD, 1996.

Wpłynęło do Redakcji 31 stycznia 2012 r.

Omówienie

Przechowywanie danych używanych w procesach data mining staje się, w miarę zwiększania się ilości dostępnych danych, coraz większym problemem. Tradycyjne podejścia, takie jak przechowywanie danych w plikach tekstowych lub bazach relacyjnych, przestają wystarczać przy dużych wolumenach danych. Artykuł analizuje możliwość przechowywania danych w nierelacyjnej bazie danych MongoDB. Dokonano porównania czasu działania różnych implementacji tych samych czterech, elementarnych algorytmów, dla danych przechowywanych w pliku tekstowym, bazie MySQL, bazie PostgreSQL oraz nierelacyjnej bazie danych MongoDB (tabele 2 – 5). Rezultaty wskazują, że zastosowanie MongoDB jest efektywne i może być rozważane jako realna alternatywa w praktycznych zastosowaniach.

Address

Paweł KASPROWSKI: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, pawel.kasprowski@polsl.pl.