Estimation of execution time of data-intensive out-of-core processes

TAMÁS SCHRÁDI
Schradi.Tamas@aut.bme.hu

ÁKOS DUDÁS
Dudas.Akos@aut.bme.hu

SÁNDOR JUHÁSZ
Juhasz.Sandor@aut.bme.hu

Budapest University of Technology and Economics
Department of Automation and Applied Informatics
Magyar tudósok krt. 2, 1117 Budapest
HUNGARY

Abstract: In this paper we investigate various out-of-core methods aiming to process large datasets efficiently using state-of-the-art, personal computers. A dataset is considered large in case it does not fit into the main memory. First an eager method will be shown to demonstrate the incapability and inefficiency of direct in-memory processing of huge amounts of data. Afterward two out-of-core extensions will be introduced that use the secondary storage to overcome the difficulties caused by the limited memory. The Periodic Partial Result Merging algorithm operates with smaller chunks, which fit in the main memory and continuously propagates the results on the secondary storage. The K-way Merge technique follows a similar principle, but it separates the processing and the merging phases. The two proposed methods proved to be suitable to process large datasets efficiently in a fault tolerant way. A comparative evaluation of the out-of-core algorithms and a novel model for estimation of their execution time will also be given. The goodness of the model will be validated by comparing its estimation to the results of practical measurements.

Key–Words: execution time estimation, K-way merging technique, out-of-core processing, Periodic Partial Result Merging

1 Introduction

Out-of-core method is a common name for a specific group of methods that are prepared to deal with datasets that do not fit in the main memory. These methods use secondary storages to overcome the limited size of the main memory. Although the size of the main memory has been increasing in the last decades there are challenging tasks where out-of-core methods are applicable. There are several fields (enterprise data processing, web log analysis, scientific data processing or computer graphics) where it is not hard to encounter datasets which require an amount of memory that exceeds the size of the main memory for processing.

Our research is motivated by real-life task dealing with web logs. The interaction of users with web servers is stored in log files. Due to efficiency purposes these log files store elementary data belonging to a single interaction of a user. With various, cookie based technique the users can be separated and identified, afterwards a profile can be derived from the stored data which describes the behaviour of any given user [1, 7, 15, 16]. Temporal profiles can be derived which describe temporal behaviour, counting all the interactions of any given user, the distribution of clicks per week and per day. It is not the specific profiles that are important for content providers, but the typical groups of users with the similar browsing behavior provide the valuable knowledge for them. From the point of view of practical applications typical user profiles can be used to personalize web sites or using a predictor system new site can be recommended to the users based on their profile [2, 3, 4, 5, 6].

To have these profiles from the elementary log entries aggregation has to be performed. The input of the data preparation task is the raw log files. In a real project these raw log files can be vast, in case of a real web log belonging only to a single month of some Hungarian portals contains more than 6 billion elementary entries (nearly 180 GB). The input file consists of an identifier and a timestamp of the interaction. The aggregation has to be performed based on the identifiers. When a matching identifier is found the specific profile update is executed (increasing the total number of clicks, updating the respective values in the distributions). The main task is to read all the records from log, create a central storage containing the profiles and for every read record update the respective profile.

The vast cardinality of the data to be processed causes increased execution time for this transforma-
tional step. So optimization of algorithms, controllability and execution time predictability are crucial from the point of view of performance. The models of different out-of-core algorithms show the possible optimizations to speed-up the whole aggregation. The models are validated on real data sustaining the correctness of models.

The organization of this paper is as follows: Section 2 presents the state-of-the-art scientific approaches in this field. Section 3 describes and analyses out-of-core methods, their execution time analysis compared to execution complexity of other approaches and identifies the major factors influencing the execution time. Section 4 presents execution time results of algorithms measured on real datasets compared to the estimation of our model. The last section summarizes our work and presents possible further working directions.

2 Related work

As previous section depicted the task to be solved requires more memory than the size of the available main memory. To overcome the difficulties caused by the limited memory various approaches can be taken into consideration: sampling, compression and out-of-core methods are possible approaches in this field.

Sampling means a possible solution of the presented problem. Because of the nature of our task sampling is not a suitable solution. Using sampling techniques we get the only a subset of the processed data, while the task is to create exact profiles based on the whole dataset. A sampling could cause a strange distortion in the processed data.

Compression means another possible approach: instead of storing data in a raw form, during the processing the data has to be compressed and uncompressed, when it is needed. There is no guaranty that the compressed data will fit in main memory. The theoretical lower bound of compression [18] supports this idea. Beside this, another factor has to be taken into consideration, namely the execution time of the compression, decompression, which can add a significant overhead.

Out-of-core methods can be a well-scalable solution if datasets several times larger than the size of main memory has to be processed. Out-of-core methods make the processing possible even in a memory limited environment by the usage of secondary storages (e.g. hard disks) in order to overcome the constraint of limited memory. These methods follow a partitioning principle, which means that the original dataset is processed in smaller blocks, resulting partial processed sets, from which the global result can be obtained. The partitioning principle is applicable in case of aggregation datasets with high cardinality: using smaller blocks and persisting the result on secondary storage, there will be available main memory for the next blocks.

In out-of-core literature there are several techniques for generation of global result from the partial results: the global result of some problems can be generated as the union of the partial result sets, as presented in [8, 9, 10]. For other problems a merging can be the applicable technique to determine the global dataset from partial results [11, 12]. In other cases a more complex algorithm has to be performed to derive the global result [13].

There are approaches which solve the memory limited issues using secondary storages. In this paper the performance analysis of these methods will be discussed and an essential factor of the out-of-core methods can be observed even in conceptual phase. Regarding the up-to-date computer architecture a crucial factor determining execution time can be pointed out: accessing a secondary storage lasts significantly longer than accessing the same data being actually held in main memory. This factor influences the efficiency of processing, resulting that in an out-of-core method the number of I/O instructions should be kept at a minimum level. This requirement of minimal I/O instructions is essential from another point of view as well: the raw datasets are generated in an automated way, continuously, thus it is needed to avoid the extrusion of the raw data. This could be done by assuring that the procedural steps have linear time complexity, which in general cannot be satisfied. But keeping the I/O instructions at the lowest level the performance of the processing will be still efficient to avoid data extrusion.

Based on the two previous points the core-efficiency of the out-of-core methods depends on whether they read only constant times the input dataset or not.

Before applying an out-of-core method the block size has to be chosen: the successively, equal-sized partitioning is a trivial and practically sound method [11, 12, 14]. But according to [9] a sophisticated partitioning approach can have performance increasing effect. The carefully chosen size is an important performance determining factor, because with it the consumed memory can be controlled. An eager, in-memory algorithm will be presented in this paper to demonstrate the undesirable behavior when the main memory reaches its physical bounds.
3 Out-of-core methods in aggregation based tasks

As first, but mainly theoretical approach could be if the virtual memory management of the operating system would be used, because it addresses a similar issue. Other works and our previous work show that this approach is not suitable for this specific data-intensive task. The eager method demonstrates the weaknesses of the in-memory approach and shows an argument for out-of-core methods. Optimizing algorithms, first the main components contributing to the execution time of an out-of-core method have to be detected. The execution time of the preparation task can be separated in three main steps: reading the data, saving the data and the data processing itself.

\[ c_{\text{total}} = c_{I/O} + c_{\text{proc}} = (c_{\text{read}} + c_{\text{save}}) + c_{\text{proc}} \quad (1) \]

The formula shows that the cost of processing the log files consists of two basic cost: the I/O costs and the procedural costs. The I/O cost can be further divided to read and save costs. Beside this theoretical consideration the practical consideration of accessing different memories (main memory, secondary storage) support the idea of differentiation between the costs. Because of the longer access time of secondary storages reducing disk complexities is crucial for an efficient processing.

3.1 Eager method

In order to see the weaknesses of an in-memory algorithm an eager processing method will be presented first. This method is not only important because demonstrates the incapability of the eager approach, but its out-of-core extension will be used by other algorithms. Although the name of the algorithm suggests that this approach is not a sophisticated one, still some facts have to be taken into consideration to build an efficient core approach. During the aggregation one specific element among millions of others have to be found, thus a very fast searchable container is needed. This container will be held in the main memory and this will be refreshed continuously. A well-scaled hash-based container makes the search fast (O(1)). The eager method works as follows: at every procedural step the hash-based container is updated and when the processing is done, the hash-based container in the main memory is saved to a persistent storage (hard disk).

```plaintext
while (!EOF(logfile)) do readEntry(logFile);
    if container.contains(entry.key) then
        update(container[entry.key, entry.info]);
        container.Insert(entry.key, entry.info);
    else
        saveToFile(Container)
    end if
end while
```

Assuming that our dataset contains \( n \) records, partitioned equally-sized (a block contains \( m \) records), the number of blocks created with partitioning is \( s = \lceil \frac{n}{m} \rceil \). A further assumption is that from an \( x_i \) sized dataset the aggregation will create a compressed dataset with size of \( \alpha_i x_i \), where \( \alpha \ll 1 \). All the calculations presented in this paper are based on the number of records to be processed \( n \), on the size of blocks \( m \).

Investigating the disk complexity of the eager method, all \( s \) pieces of blocks have to be read from hard disk and write the aggregated dataset back to it, expressed by the following formula:

\[ c_{\text{disk}} = \sum_{i=1}^{s} (m + \alpha_i m) \approx (1 + \alpha) n \quad (2) \]

The first part of the sum represents the reading phase of all the records and the second part of this sum is for the aggregated count of records. Here is introduced the \( \alpha \) the average compression factor.

Similarly, the procedural complexity can be calculated based on the following expression

\[ c_{\text{proc}} = \sum_{i=1}^{s} f(m) = \frac{f(m)}{m} n \quad (3) \]

where \( f(m) \) is a function expressing the cost of processing of an \( m \) sized block (the values for this function can be calculated from the actual aggregation). As the formulas suggests this is a fast method, but its applicability is limited due to immense memory need of the algorithm, bounded only by the size of the processed dataset.

3.2 Periodic Partial Result Merging

The Periodic Partial Result Merging algorithm processes only smaller sized datasets at the same time in the main memory, creating a local result on the storage [15]. After processing the first dataset the result is saved to the storage, while in other steps the actual partial result is merged with the existing one from the disk. In this approach the local results are propagated during the phases of processing, and they are merged after finishing the processing of a block. After the last merging the resulting dataset will be the global result. The processing of a data block is done according to the eager method, but here a block...
fits in the main memory. An essential step in whole processing is the merging phase. In order to elaborate an efficient working version of the algorithm, an ordering is needed, defined on processed datasets, this ensuring a minimal additional execution time, caused by merging.

\[ \text{while} (\text{!EOF(logfile)}) \text{ do} \]
\[ \quad \text{while} (!\text{A block is read}) \text{ do} \]
\[ \quad \text{readEntry(logFile);} \]
\[ \quad \quad \text{if} \ \text{container.contains(entry.key)} \text{ then} \]
\[ \quad \quad \quad \text{update(container[entry.key, entry.info]);} \]
\[ \quad \quad \text{else} \]
\[ \quad \quad \quad \text{container.Insert(entry.key, entry.info);} \]
\[ \quad \quad \end{if} \]
\[ \quad \end{while} \]
\[ \text{mergeToFile(Container);} \]
\[ \text{resetBlockCounter();} \]
\[ \text{end while} \]

\[ c_{\text{disk}} = \sum_{i=1}^{s} m \left(1 + \alpha_i(2i - 1)\right) \approx n + \frac{\alpha}{m} n^2 \quad (4) \]

In the processing costs beside the cost of aggregation of an \(m\) sized block itself the merging step has its own influence as well: at the processing of the \(i\)-th block the first block was already merged \(i\) times, the \((i-1)\)-th block \(i - 1\) times and so on. This implies the \(i\) factor in the second part of the sum. In this equation we introduce a new constant \(\beta\) which makes processing and merging costs equatable.

\[ c_{\text{proc}} = \sum_{i=1}^{s} \left(\frac{f(m)}{m} + i \cdot m \cdot \alpha_i \cdot \beta\right) \approx \]
\[ \approx \left(\frac{f(m)}{m} + \frac{\alpha \cdot \beta}{2}\right) \cdot n + \frac{\alpha \beta}{2m} \cdot n^2 \quad (5) \]

\subsection{3.3 K-way Merge Technique}
The K-way Merge Technique follows the partitioning principle too, but besides propagating the results at every procedural step, separates the processing from merging, being two different phases of the processing. As first step the algorithm processes all the partitioned blocks, according to eager method. It is essential to remark that the blocks are partitioned so, that the processing can be done in the main memory. After processing a block the resulting dataset is saved to the persistent storage. When all the blocks are processed, the merging phase will be done on partial results saved to disk, containing processed elements. This means a \(k\)-way merge among the elements, resulting at the root of the merging tree the global result.

\[ \text{while} (!\text{EOF(logfile)}) \text{ do} \]
\[ \quad \text{while} (!\text{A block is read}) \text{ do} \]
\[ \quad \text{readEntry(logFile);} \]
\[ \quad \quad \text{if} \ \text{container.contains(entry.key)} \text{ then} \]
\[ \quad \quad \quad \text{update(container[entry.key, entry.info]);} \]
\[ \quad \quad \text{else} \]
\[ \quad \quad \quad \text{container.Insert(entry.key, entry.info);} \]
\[ \quad \quad \end{if} \]
\[ \quad \end{while} \]
\[ \text{mergeToFile(Container);} \]
\[ \text{resetBlockCounter();} \]
\[ \text{end while} \]

The disk costs consists of the different parts: the first part is the reading of the whole dataset and the last one writing to the disk the resulting dataset. In any step between these steps a processed dataset has to be written to the disk and before applying the next merge has to be read from the disk. This is the cause of factor 2 in front of the summation sign, and in the summation we have a formula expressing the size of the dataset actually processed in the \(i\)-th merging step.

\[ c_{\text{disk}} = s \cdot m + 2 \sum_{i=1}^{[\log_k s] - 1} \left[\frac{s}{k^i} \cdot \alpha_i \cdot m \cdot k^i + \right. \]
\[ \left. + \alpha_{[\log_k s]} \cdot m \cdot k^{[\log_k s]} \right] \]
\[ c_{\text{disk}} \approx 2\alpha \log_k n + (1 + \alpha (1 - 2 \log_k m)) n \quad (6) \]

The processing costs can be expressed as the cost of aggregating the whole dataset for the first time and afterwards the cost of aggregation at every phase of the merge.

\[ c_{\text{proc}} = s \cdot f(m) + \sum_{j=1}^{[\log_k s]} \left[\frac{s}{k^j} \alpha_j \cdot m \cdot k^j \cdot \right. \]
\[ \left. \cdot f\left(\frac{s}{k^j}\right) \alpha_j \cdot m \cdot k^j \right] \]
\[ c_{\text{proc}} \approx \alpha \cdot n \cdot \log_k n + \left(\frac{f(m)}{m} - \alpha \cdot \log_k m\right) n \quad (7) \]

\section{4 Test results and estimations}
The experiments were carried out with the following configuration: CPU: Intel Pentium 4 at 3200 MHz, memory: 4 GB, operating system: Microsoft Windows Server 2003. The algorithms, presented in this paper were implemented in C++.
Figure 1 depicts the execution time of Periodic Partial Result Merging algorithm, with different sizes of blocks. As the theoretical background showed, it is quadratically proportional to the number of entries to be processed. The graphs clearly supports that the effect of the quadratic member can be reduced by choosing a bigger size of block.

Another important factor, influencing execution time is the number of blocks. Based on the measured execution time values let us try to estimate the observed execution time function. This function depends on the number of elements to be processed (n) and on the size of blocks (m): $f(m,n)$. According to theoretic considerations the function describing the execution time complexity of aggregation has the following form:

$$f(m,n) = a_1 + a_2 \cdot n + a_3 \cdot n^2 + a_4 \cdot \frac{1}{m} +$$
$$+ a_5 \cdot \frac{n}{m} + a_6 \cdot \frac{n^2}{m} + a_7 \cdot m \quad (8)$$

The coefficients in the equation (8) were determined using least squares approximation technique. The approximation assigns the following values to the coefficients: (-11.8176, 3.0919, 0.00045, -59.25476, 4.2016, 0.0182, 2.2008).

In order to see the correctness of the approximation is presented on Figure 2. As this graphs shows there is no significant difference between the approximated and measured values, the percent error of the approximation remains below 5% after an initial phase. The average percent error for various block sizes is 2.404575%.

The presented approximation principle is used in the case of K-way Merge Technique as well. In this case the theoretical results indicate the following execution time function:

$$g(k, n) = b_1 + b_2 \cdot n + b_3 \cdot \log_k m +$$
$$+ b_4 \cdot n \cdot \log_k n \quad (9)$$

In this case the size of processed chunks is the same for every k values, thus the m has a fixed value. The impact of various merging was investigated as well. Using least squares technique we have calculated the following values for the coefficients (2.8001, 4.6588, 13.3975, 0.0714).

Figure 3 shows the percent error of execution time approximation in the case of K-way Merge Technique. As it is clearly visible, after a short initial phase the percent error remains under 5% for each approximated k values (these are reasonable values). The average error of this approximation is nearly 0.8899%.

5 Conclusion

Based on theoretical considerations and on experimental results on real datasets the number of I/O instruction is a crucial factor determining the execution
time of an out-of-core method. An efficient out-of-core processing method uses as few I/O instructions as possible.

The effect of the quadratic member in disk and procedural complexity of Periodic Partial Result Merging can be reduced by choosing a big block size. Block size controls the size of data processed together in the main memory at the same time. It is very important to have full control over the memory consumption to avoid unmanageable execution time the eager method suffered from. In case of K-way Merge Technique looking only at the formula a conclusion can be made: the more ways are used at merging the faster the processing will be. In this method the number of levels in merging tree should be kept minimal, but other influencing factor is the number of files processed together at the same time.

In both of the algorithms the compression factor has its influence on disk- and execution time complexity, but this is a given factor of the dataset. To ensure efficient processing the compression factor should be close to 0, thus reducing the effect of quadratic members in execution time and disk complexity. (Most of the datasets fulfill this prerequisite, because the aggregation itself causes a significant size reduction).

The given models proved to be suitable to describe the execution time complexity of the different approaches. In the case of the Periodic Partial Result Merging the average error in under 3% supporting the correctness of the theoretical analysis of the proposed approach. In the case of k-way merge the execution time determining factors require a more complex analysis. The analysis of the parallel version of methods can be a further work, based on [17].

Acknowledgements: This work is connected to the scientific program of the “Development of quality-oriented and harmonized R+D+I strategy and functional model at BME” project. This project is supported by the New Széchenyi Plan (Project ID: TÁMOP-4.2.1/B-09/1/KMR-2010-0002).

References:


