Improvements to Efficient Retrieval of Very Large Temporal Datasets with the TravelLight Method

Alexandre Valle de Carvalho[a,b], Marco Amaro Oliveira[a,b], Artur Rocha[a,b]
alexandre.valle@fe.up.pt, mao@inescporto.pt, artur.rocha@inescporto.pt

[a]INESC TEC, Campus da FEUP,
R. Dr. Roberto Frias, 4200-465 Porto, Portugal
[b] DEI/FEUP,
R. Dr. Roberto Frias, 4200-465 Porto, Portugal

A considerable number of domains deal with large and complex volumes of temporal data. The management of these volumes, from capture, storage, search, transfer, analysis and visualization, provides interesting challenges. One critical task is the efficient retrieval of data (raw data or intermediate results from analytic tools). Depending on the user's task, the data retrieved may be too large to fit in the local memory and, even when it fits, the time taken to retrieve the data may be too long. Previous work proposed the TravelLight method which reduced the turnaround time and improved interactive retrieval of data from large temporal datasets by exploring the temporal consistency of records in a database. The underlying main idea is that, at any given moment, a request is sent for data intersecting the temporal focus of interest which is not already marked as loaded in the database. To do this, the proxy module keeps track of loaded data at the database level, which generates time overheads from select and deselect operations. The current work propose an improved version that adopts a new paradigm focused in time intervals instead of solely in data items. This paradigm shift enables the separation of the proxy module from any particular temporal data source, as it is autonomous and efficient in the management of retrieved data. Our work also demonstrates that the overheads introduced by the new paradigm are smaller than prior overall overheads, further reducing the turnaround time. Results are reported concerning experiments with a temporally linear navigation across two datasets of one million items. With the obtained results it is possible to conclude that the approach presented in this work, built over the original TravelLight method further reduces turnaround time thus enhancing the response of interactive tasks over very large temporal datasets.

Keywords: large datasets, temporal data, time-oriented data, data retrieval, self-balancing interval tree

I. INTRODUCTION

Over the last decade, for most domains of knowledge, the size of and complexity of data volumes has increased to a point where traditional methods for storage, search, transfer, analysis and visualization do not suffice. Some of the first domains that had to tackle the challenges related to size of datasets were related to complex scientific problems, such as astrophysics. Nowadays, with the advent of social networks and easier and faster methods to communicate, a considerable number of sources produces massive amounts of data, sometimes encompassing complex phenomena that require analytic tools and effective visualization techniques in order to depict them. Supporting these two activities, other (critical) operations such as data storage and data transfer are necessary. The latest is particularly important considering data visualization, where the user can interactively change the temporal focus of interest (TFI). Frequently, the goal of interactivity conflicts with the dimension of the dataset being visually explored. Hence, the objective is to reduce turnaround time, which is the measure of the time taken between the submission of a request for data and the availability of such data. Previous techniques for retrieval of temporal data, such as Dynamic Queries [1, 2], Temporal Brushing and Temporal focusing [3] did not address these problems.

Considering the direct retrieval of data, the turnaround time is equal to the data retrieval time, because the availability of data requires data to be entirely retrieved from a data source. However, with respect to temporal data, there is room for acceleration and potentially considerable improvements by exploring the temporal consistency of data between requests. As an example, consider the user is interested in performing exploratory visual data analysis of a very large temporal dataset. Among other tools, the user can change the TFI, for instance, to observe the evolution of a complex phenomenon.

Considering temporal data, it may happen that several requests for data with distinct TFI may retrieve data items that hold for both intervals. In this case, the turnaround time can be reduced by not retrieving data items, from the data source, when it is acknowledged that they are already available. This measure of similarity is designated as temporal consistency [4]. Moreover, a data item that exists for most of the dataset timespan is a good candidate for high temporal consistency while a data item temporally happening at a particular instant is not (except when the TFI temporally intersects it).

Previous work [4] addressed the problem of how to efficient and interactively travel across very large temporal datasets based on the hypothesis that it is possible to make use of the temporal consistency of temporal data by preventing the retrieval of data items that hold for several TFI and that have been made available from previous load operations, while keeping memory manageable. To achieve this, the proposed retrieval algorithm directly communicated with the source database and required that a temporary relation was created from the original SQL query with an extra attribute which was used to specify, at any given time, which records were made available. Hence, at the database level, a request for data would consider records temporally intersecting the TFI which were not marked as already loaded. On load, every selected record would be marked as loaded. Each data item unloaded from the cache also would require to be marked in the database as not loaded.

As compared to loading the entire subset of data temporally intersecting each TFI (disregarding temporal consistency) the TravelLight method proved to be more efficient.

The current work adds to previous work significant improvements in what concerns both the retrieval algorithm and memory management algorithms and which also involves a change in the cache data structure. For the remainder of this document this is designated by improved version.

The original method is focused in managing data items and does not keep track of the TFI for which data items are retrieved, while the improved version keeps track and manages the TFI time intervals and corresponding data items

Considering the amount of memory available to the cache, the aim of the improved version is to fill the cache with uncompromised time intervals, that is, time intervals for which it is known the subset of data is fully available. As requests for data are being performed and TFIs (with their data items) are being loaded, interval merge and split operations (also considering the corresponding data items) are being performed in order to keep interval/data items information updated (see section V).

The benefits from this paradigm shift - from items to time intervals - are the following: (1) as the proxy module is now able to know for which time intervals the full subset of data items is available, the request for data from the data source is performed only if the TFI partially intersects or does not intersect the intervals that are currently fully loaded; (2) a request for data explicitly defines for which time intervals data should be retrieved and for which time intervals it should not, (3) when the cache is full and an unload of data items is required the intervals containing the items being unloaded are compromised and discarded or reduced to the extent where data is not compromised. These operations introduce overheads but, as it can be observed in the experiments performed, it is still faster as compared to explicitly marking records as selected/deselect, in the database.

The current work also demonstrates that the overheads introduced by the data management oriented to time intervals are smaller than prior overheads resulting from the retrieval algorithm from the original method, further reducing the turnaround time.

In the next section the original TravelLight method is briefly described. Section III presents the improved TravelLight version while section IV details the corresponding algorithm. Section V focuses on the new data structure supporting the cache and interval management. Next, section VI describes the experiment performed with a prototype implementing the algorithm and the proposed cache data structure. Section VII presents and discusses the results achieved. Finally, in Section VIII, the main conclusions and future work is disclosed.

II. THE TRAVELLIGHT METHOD

The previous work [4] proposed a method based on two premises: (1) data is retrieved by temporal queries evaluated and executed by a TDBMS, and (2) a subset of the original dataset, resulting from the first temporal query, is persisted in the TDBMS as a temporary relation. Based on these premises, the TravelLight method has the following general overview: (a) keep loaded data items in the client-side memory as the TFI changes; (b) identify the loaded data items that temporally intersect the current TFI (temporal consistency); (c) request data items from the TDBMS whose valid-time intersects the current TFI and which are not yet loaded in the client-side memory. Thus, the load of a smaller number of items from the TDBMS means a decrease in the data retrieval time and the reduction of the time for data to become available to the client application (turnaround time).

In [4] the knowledge about which items are already loaded in the client-side memory, to be used in (c), is achieved by keeping and considering an additional attribute, named *state*, in the temporary relation containing the entire result set from (2). By considering attribute *state* in the query defined in (c) the query execution is performed faster as the query does not consider the entire data set as a candidate for TFI. At that time, several other alternatives to having the additional attribute were tested - such as the explicitly enumeration, from the client, of each row identifier not to be considered in (c). These alternatives proved not to be viable with large datasets, as it turned impossible to execute SQL commands involving the explicit enumeration of the entire set of row identifiers.

The TravelLight instantiates in an independent proxy module, acting as a proxy for data, composed of a linear data structure (designated by cache) and two algorithms: a retrieval and a memory management algorithm: the first manages data retrieval from the TDBMS - task (c) – while the last manages the cache structure, data load and unload – tasks (a) and (b).

A client application using the proxy module as a component can request data to the module and wait for data to become available. The results achieved in [4] demonstrated that it is possible to minimize turnaround time by exploring the temporal consistency of data through the TravelLight method.

III. IMPROVEMENTS ON THE TRAVELLIGHT METHOD

Despite the good results previously obtained, further improvement has been identified, to the process of data retrieval and memory management. These improvements result from the observation of drawbacks resulting from the initial premises, namely the fact that the method directly executes commands in the database:

- Performance of first temporal query: the fact that a temporary relation needs to be persisted may take a long time for large datasets;
- Append to the persisted temporary relation of an additional *state* column and corresponding index: as previously described in section II, this column provides knowledge of which items are already loaded in the client-side memory. Yet, the index creation time is not negligible;
- Update of *state* column: selection and deselection of data items into/from the client-side memory requires an update of the *state* column of the corresponding records;
- Concurrent access: the previous items show that concurrent access from multiple clients may be complex due to the fact that each client will require its own temporary persisted relation.

The identified drawbacks raised the hypothesis that there is room to make improvements that may further reduce the turnaround time but, most importantly, rendering this method independent from the data sources.

In order to validate this hypothesis we considered:

1) With respect to work in [4], minor changes in the cache supporting data structures, in order to speed up the reprojection and management operations. For the remainder of this document this is designated by modified version.
2) Significant changes that resulted from a paradigm shift: the knowledge of loaded items is performed through time intervals, and managed by the proxy module, instead of through selected data item at the data source (improved version). Due to this significant changes, the improved version is detailed in the next section.

IV. ALGORITHM

The Algorithm 1 is performed every time there is a request for data from the client. Consider the following variables: (a) *tfi*, which contains the time interval representing the TFI; (b) *cache*, which is the cache structure and management functions; (c) *retrieval*, which is an implementation of the retrieval functions.

When invoked, Algorithm 1 first determines if the TFI intersects any of the *current* cached intervals (lines 02-03). If true, it calculates which cached items do intersect the TFI (*hits*). Next, the algorithm checks if the TFI is entirely contained in the set of *current* cached intervals (line 05) and if so, the entire subset of cache *hits* is already available and therefore there is no need to retrieve additional data. In this case, the subset of cache *hits* is returned (line 06). If the current TFI partially intersects or it does not intersect the current TFI, the procedure continues by calculating the set of intervals for which data is *required* to be retrieved (line 09). Next, it queries and *loads* items by calling the *retrieval* of items that temporally intersect the *required* set of intervals and which, at the same time, do not intersect the *current* set of intervals already fully loaded in cache (line 10). Next, the cache is enlarged if the number of the already loaded cache *hits* plus the items to *load* is larger than the current cache size (lines 11-14). Here, the enlargement considers a factor of 120%[1] of the internal growth required. Next, the algorithm calculates if the free space available in the cache is enough to encompass the load of new items (line 15). If not (line 16), a strategy is applied to retrieve the minimum set of intervals (*toFree*) from the cache that contains at least a minimum of *toRealeaseItems* number of items (line 17). The strategy will be applied considering all cached items that are not in the *hits* list of items. All items in the cache that are fully contained in the *toFree* set of intervals will be deleted as well as the corresponding intervals released from the cache (line 18). After releasing the necessary space the loaded items and respective intervals are added into to the cache. Finally, a coalescing of intervals is performed, in order to keep a minimal representation for the set of cached intervals corresponding to cached items. The algorithm ends with the return of all the items which intersected the current TFI (hits + load, line 21).

Algorithm 1 relies on five functions namely, *getAllIntervals*, *getIntervals*, *delete* and *insert*. These functions are directly related with the data structure described in Section V. The first returns the *current* set of cached intervals (used in line 02). *getIntervals* calculates the set of intervals that contain the *toReleaseItems* number of items which are most distant from the current TFI (line 17). Each *toFree* interval passed to *releaseItemsAndIntervals* (line 18) is removed from the set of cached intervals. Moreover, every cache item contained in a *toFree* interval is removed from the cache of items. In the cache of items, removed items are placed in the free list to be reused, preventing memory allocation the next time items are loaded. Finally, the *addItemsAndInterval* function loads every retrieved item into the cache of items (by making use of the items in the free list). It also appends the current TFI into the current set of cached intervals and performs a coalescing of the resulting set.

[1] This value was set based on empirically observations with the tested datasets.

Input: *tfi*, a time interval representing the current TFI.
Output: an array of items intersecting the TFI.

```
01.  Item[] hits;
02.  Interval[] current = cache.getAllIntervals()
03.  if (tfi intersects current) {
04.    hits <- cache.query(tfi)
05.    if (tfi containedin current) {
06.      return hits;
07.    }
08.  }
09.  Interval[] required = tfi – current
10.  Item[] loaded = retrieval. queryAndLoad(required,
current)
11.  int newSize = hits.size() + loaded.size()
12.  if (newSize < cache.size()){
13.    cache.growTo(newSize)
14.  }
15.  int toReleaseItems = loaded.size() – cache.free()
16.  if (toReleaseItems > 0) {
17.    Interval[] toFree=cache.getIntervals(toReleaseItems,
                                tfi)
18.    cache.delete(toFree)
19.  }
20.  cache.insert(loaded, tfi)
21.  return hits + loaded
```

Algorithm 1: improved TravelLight algorithm.

V. DUAL RED-BLACK AUGMENT INTERVAL TREE STRUCTURE TO SUPPORT CACHE OF TEMPORAL DATA

In order to efficiently support the functions described in the last paragraph of the previous section, some data structures were tested, namely implementations of interval tree [5], interval tree supported by AVL instant trees [6], red-black augment interval tree [7, 8]. The goal was to have these functions built over efficient point and interval insert, query and delete operations performed over an ordered set of intervals. The term efficient refers to a desirable complexity of O(log n) and also a linear memory space allocation. The interval tree was abandoned, due to the lack of balancing in insert and delete operations. Several self-balancing interval tree implementations were tested which claimed to have O(log(n)) complexity. The following has been verified: (a) the implementation in Gephy [8] completes query and delete operations in time O(log n) but insert operations are less efficient; (b) the AVL instant tree implementation provided in [java-algorithms-implementation] was used to develop a balanced interval tree supported by two AVL instant trees. It performed query, insert and delete operations in time O(2.log(n)); (c) the red-black augmented interval tree provided in [7] completed queries in time O(k.log(n)) (where n is the number of intervals stored in the tree and k is the size of the result set from the query) and deletes and inserts in time O(log(n)). All implementation consumed linear memory space in the number of intervals stored in the tree. However, implementation (b) consumed more space as each interval was referenced in two AVL instant trees. Based on the previous evaluation a decision was taken to develop a new data structure that augments the red-black interval tree implementation provided by Stern [7]. This data structure uses the basic insert, query and delete operations provided by the original implementation to support further operations performed over an interval tree, with respect to the management of a cache of items and of a set of cached intervals, presented in Figure 1, which support the Algorithm 1.

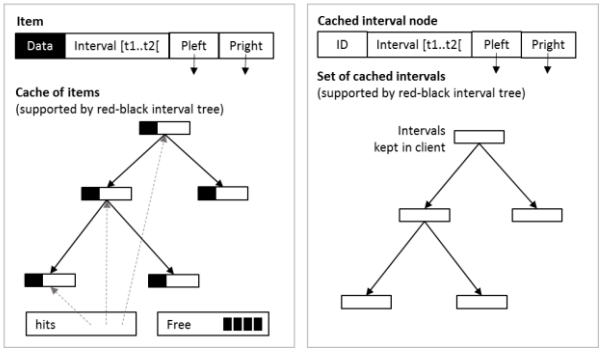

Figure 1: Dual red-black augment interval tree structure for cache of items (left) and cache of intervals (right).

VI. EXPERIMENTS

In order to evaluate the modified version, but mostly the improved version, an experiment was designed and performed that allows comparison with results achieved with the original TravelLight presented in previous work [4].

For these experiments the same dataset 1 of [4] was used. It is a time-oriented balanced dataset composed of 1 million events, occurring over a 100 year time span, starting in 1900. The average timespan of each event is 5 years. Also, the same navigation test scenario of [4] was adopted. It simulates the user behavior of linear travel across the dataset time, from start to end: the TFI starts at 1900-01-01 and has a 4 year span. For each request, the current TFI in the client application is displaced 1 month towards the future (1200 TFI changes).

For both the modified version and the improved version the following operations should be considered when performing experiments. The measurement of time (in milliseconds) associated with these operations provide an adequate solution to evaluate the method efficiency.

- Reprojection: to find which of the items that are already available in the cache hold for the current TFI (line 04 of algorithm 1);
- Retrieval: to perform three operations with respect to the data source:
  a) Deselection of a set of previously selected items: for the TravelLight method proposed in [4], an unload of items by the module requires the deselection of the corresponding data in the data source. In order to do so, a request for data may carry a list of item IDs that are to be deselected if the module decided to recycle items in cache. The improved method, detailed in algorithm 1, does not encompass this operation;
  b) Selection of currently required items: in the TravelLight method proposed in [4] this operation retrieves unselected items that temporally intersect the TFI. Next it marks these items as selected. With the improved TravelLight method this operation just retrieves items that temporally intersect a set of computed intervals (lines 09-10 in algorithm 1);
  c) Transfer of items: the transfer of commands for operations details in a) and b) and the transfer of resulting data from the data source to the module.
- Management: to manage the cache space (cache growth, release of items) in order to accommodate the load of new items. (lines 12-18 in algorithm 1);
- Insert: to load the items retrieved from the data source into the cache once they arrive in the module (line 20 in algorithm 1);

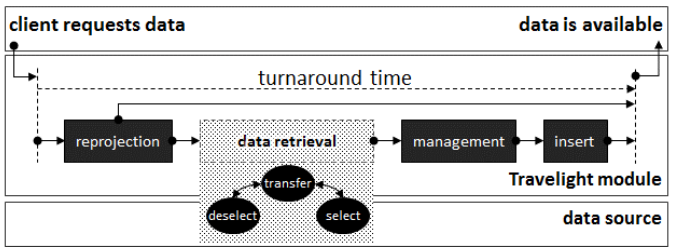

Figure 2: operations required to have data available.

Figure 2 shows the sequence of these operations and that, for complete temporal consistency of data with respect to a given TFI, the required data is already available in the cache and the turnaround relies on the temporal reprojection of data. The turnaround time is inversely proportional to temporal consistency, i.e. higher temporal consistency leads to lower turnaround times. When no complete temporal consistency of data is found then operations of data retrieval, cache management and insert are required.

As for the metrics Total Number of Records (required for the current TFI), Records Loaded From DB and Cache Hits, the test dataset and the results achieved are the same as the ones found in [4].

The experiments were conducted using two computers connected through a corporate Ethernet 100Mbps LAN. The experiment environment changed with respect to [4]: the TravelLight module is developed in Java and Google Web Toolkit and the test client application is automatically coded to Javascript and runs in a web browser. The Oracle database system was also replaced by a PostgreSQL database system. For this new environment the achieved results are reported, both for the modified version and for the improved version (implementing Algorithm 1), in the next section.

VII. RESULTS AND ANALYSIS

This section present the average results achieved from 10 repetitions of the experiment previously described.

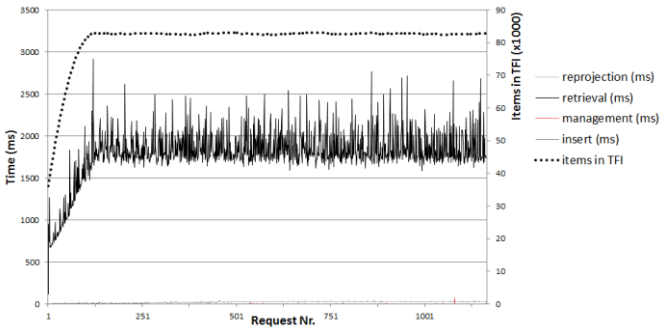

Figure 3: average times for the modified version.

Figure 3 show the non-stacked average times obtained with the modified version. This result shows that most of the time (average 1.809 seconds) is spent in the retrieval of data (data selection, data deselection and data transfer) and the times required to perform reprojection, cache management and data insert into the cache are much smaller, with an average bellow 50 milliseconds.

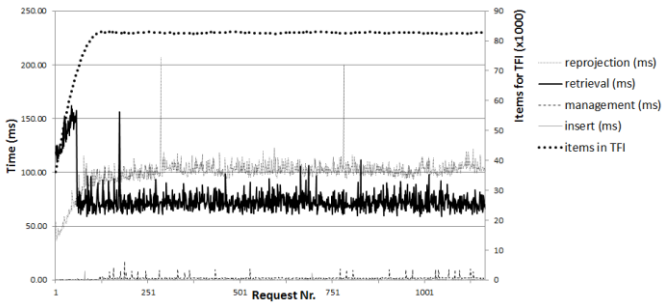

Figure 4: average times for improved version.

Figure 4 shows the non-stacked average times obtained with the improved version (Algorithm 1). This result shows that most of the time is spent in reprojection and data retrieval while the management and data insert into the cache result in much smaller times, average bellow 50 milliseconds. It is important to observe that, when comparing Figure 4 with Figure 3, the improved version shows much smaller retrieval times, but larger reprojection times.

Analysis: in the modified version, the data source is required to keep track of the data items that are currently loaded into the client. In order to keep this information synchronized update commands are required to select and deselect items. By performing these commands, the data retrieval time is considerably larger, as demonstrated by the results. With the improved version, the data source does not keep track of currently loaded items. Instead, through the data structure presented in Figure 1 the proxy module keeps track of time intervals for which it holds the complete subset of items. By reducing the tasks performed next to the data source the improved version achieves a considerable reduction of retrieval times, despite the fact that in the proxy module, the reprojection time is increased.

Table 1: Comparison of average times (in ms).

	direct retrieval [4]	TravelLight [4]	modified version	improved version
turnaround	9561	624	1831.76	178.89
reprojection	-	28	20.83	99.44
retrieval	9561	469	1809.36	77.38
management	-	127	1.06	1.67
insert			0.51	0.40

Generally, this tradeoff proved to greatly increase efficiency of turnaround time. This tradeoff is depicted in the three right columns in Table 1, where the turnaround time is considerably reduced when comparing with both the TravelLight method [4] and the modified version. Table 1 presents the average results achieved in the experiment performed in [4] and achieved in the current experiment. With respect to the insert operation, it is not displayed in column 2 because, in [4], this value was added to the management time. In what concerns the retrieval time, a significant difference between the TravelLight [4] and the modified version can be depicted, which is considered to be a consequence of the change in the testing environment, but which also affects the results with the improved version. Despite lower performance due to the testing environment, the improved version demonstrated significant improvements when compared to the others. In what concerns the reprojection time, as expected, the improved version is almost 3 times higher when compared with other TravelLight versions. However, this overhead is a necessary consequence of greatly reducing the retrieval times. Finally, with respect to the management time, a considerable improvement can be noticed between the TravelLight [4] and the modified version, due to the minor changes reported in Section III, which are now within the same range as the improved version.

Figure 5 reports a distinct perspective of the results achieved, which concerns the average turnaround time achieved per number of data items holding to their corresponding TFIs. As it can be depicted in Figure 5, there is a large number of TFI requests for which the number of returned data items falls into interval 82k-83.5k. This is observed in the item frequency histogram displayed as dotted lines in Figure 3 and 4. To improve readability of the results, Figure 6 presents a detail of this interval.

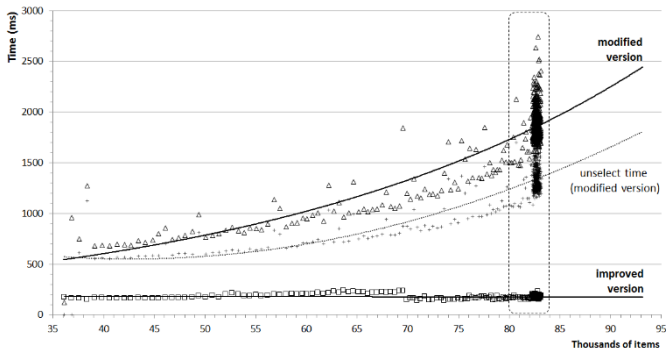

Figure 5: turnaround times per number of items holding for TFIs.

Both figures 5 and 6 display the trend lines based on the achieved results with the modified version and with the improved version of the TravelLight method. Moreover, there is an additional trend line which concerns the deselect time of the modified version, which demonstrates the relevance of this operation to the performance of the modified version.

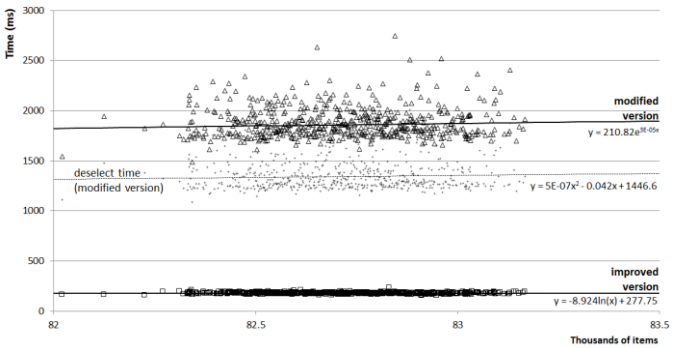

Figure 6: detail about results presented in Figure 5 with respect to interval 82000-83500 data item holding for TFIs.

Analysis: the direct relation that exists between the retrieval time and the turnaround time for the modified version supports the hypothesis presented in section III that the overhead introduced by a more complex reprojection and management operations in the improved version compensates the reduction of the retrieval time in the method.

VIII. CONCLUSION

The original TravelLight method presented in [4] validated the hypothesis that, for interactive tasks in large volumes of temporal data, it is possible to minimize turnaround time by exploring the temporal consistency of data for balanced datasets and for linear navigation through the time. The current work addresses solutions for drawbacks identified after the work in [4] and described in section III, being the most critical the ability to detach the method from the data source. To address this issue while further increasing performance, a paradigm shift has been studied and tested, where the knowledge of loaded items is performed through time intervals, and managed by the proxy module, instead of through selected data items, at the data source. In this paper we called this alternative, improved version. A prototype implementing the Algorithm 1 (Section IV) has been used in the experiment detailed in section V. As described, one concern was adopt the same test scenario detailed in [4] with similar metrics. This allowed to compare the original method results with the results achieved with both the modified version and the improved version. Based on the results achieved one can conclude that in the modified version, in order to keep track of currently loaded items, update commands are required to select and deselect items. Large retrieval times result from this type of strategy. With the improved version, by adopting the proposed data structure, presented in Figure 1, the proxy module keeps track of time intervals for which it holds the complete subset of items. By reducing the tasks performed at the data source the improved version achieves a considerable reduction of retrieval times and, consequently, of the turnaround time, despite the fact that in the proxy module, the reprojection time is higher compared to those achieved with both the original and the other versions.

There are still open questions regarding this method, such as the study of the behavior with unbalanced datasets and with distinct profiles of user interaction.

ACKNOWLEDGMENT

Acknowledgements go to COMPETE Program and the FCT, within project FCOMP-01-0124-FEDER-022701.

REFERENCES

[1] C. Ahlberg, C. Williamson, and B. Shneiderman, "Dynamic queries for information exploration: an implementation and evaluation," *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM, 1992, pp. 619-626.

[2] B. Shneiderman, "Dynamic Queries for Visual Information Seeking," *IEEE Software*, IEEE Computer Society Press, 1994, pp. 70-77.

[3] M. Monmonier, "Strategies for the visualization of geographic time-series data," *Cartographica The International Journal for Geographic Information and Geovisualization*, 1, UT Press, 1990, pp. 30-45.

[4] Alexandre Valle de Carvalho, Marco Amaro Oliveira, Artur Rocha, "Retrieval of very large temporal datasets for interactive tasks," in Information Systems and Technologies (CISTI), 2013 8th Iberian Conference on, Lisbon, Portugal, 2013, pp. 1 - 7.

[5] K. J. Dolan, "Interval Tree Implementation in Java," https://github.com/kevinjdolan/intervaltree, [2013, 2010].

[6] J. Wetherell, "java-algorithms-implementation: Algorithms and Data Structures implemented in Java," https://code.google.com/p/java-algorithms-implementation/, [2013, 2010].

[7] K. Stern, "software-and-algorithms: Neat algorithm implementations in Java.," 2010.

[8] G. Project, "Gephy Toolkit," http://gephi.org/docs/toolkit/org/gephi/data/attributes/type/IntervalTree.html, 2010].