International Journal of Research (IJR) Vol-1, Issue-9, October 2014 **ISSN 2348-6848**

Retroactive Data Structure

Nikhil & Vipin Shukla
Department of Information and Technology Dronacharya College of Engineering
Gurgaon, India
Nikhil.16540@ggnindia.dronacharya.info Vipin.16562@ggnindia.dronacharya.info

Abstract—

Retroactive data structure is a new data structuring paradigm in computer science introduced by Demaine, Iacono and Langerman. It supports efficient modifications to a sequence of operations that have been performed on the structure. Retroactive arbitrary insertion, deletion or updating an operation can occur at any point in time in the past. A data structure is fully retroactive if it supports queries and updates to current and past version. Here, in this paper, we focus on the different definitions related to retroactivity, its Comparison to persistence. We have also tried to analyse their performance and portray their utility through real life examples and applications.

Keywords – modifications; updating; retroactivity; persistence

Introduction

Basically a data structure consists of two operations invoking and revoking i.e., inserting and deleting (respectively), Insert(x) operation inserts ‘x’ value in the data structure, whereas Delete(x) operation removes ‘x’ value from the data structure. The ability to review thesequence of operations performed on a data structure is often extremely important and helpful. For instance, if incorrect value isinserted or deleted at a particular instance in pasterroneously. It is important tochange the mistaken information and to find all the decisions based on this information and to review them efficiently. Now the question arises,How to revitalize that data structure with correct values in present? In general there are two ways to regenerate data structure. The first and in most existing systems, the only way to support these changes is to rollback the state of the system to before the time in question and then re-compute all of the operations from the modifications to the present. But, this is considered as wasteful, time consuming solution, inefficient, and often unnecessary. The second way appeared when Demaine, Iacono, and Langerman introduced and developed the idea of retroactive data structures, which are data structures that efficiently support modifications to the historical sequence of operations that have been performed on the structure. These modifications can take the form of retroactive insertion, deletion or updating an operation that was performed at some time in the past on data structure.

Definitions

Any data structure can be reformulated in a retroactive setting. In general the data structure involves a series of updates and queries made

over some period of time. Let U = [u_{t1}, u_{t2}, u_{t3}, ...,u_{tm}] be the sequence of update operations from t_1 to t_m such that $t_1 < t_2 < ... < t_m$. The assumption here is that at most one operation can be performed for a given time t.

Partially Retroactive

We define the data structure to be partially retroactive if in addition to supporting updates and queries operations on the current time (present state),*it also* supports insertion and deletion of operations at the past as well. Thus for partially retroactive we are interested in the following operations:

- Insert(t, u): Insert a new operation u into the list U at time t.
- Delete(t): Delete the operation at time t from the list U.

Given the above retroactive operations, a standard insertion operation would now the form of Insert(t, "insert(x)"). All retroactive changes on the operational history of the data structure can potentially affect all the operations at the time of the operation to the present. For example if we have $t_{i-1} < t < t_{i+1}$, then Insert(t, insert(x)) would be place a new operation, *op*, between the operations op_{i-1} and op_{i+1}. The present state of the data structure (i.e.: the data structure at the current time) would then be in a state such the operations op_{i-1}, *op* and op_{i+1} all happened in a sequence, as if the operation *op* was throughout all time there. The retroactive changes on the operational history of the data structure strongly affect all existing operations between the modification time and the present time.

Fully Retroactive

The partial retroactivity definitions only takes control of half of the idea of retroactivity. The ability to insert or delete update operations in the past, and to view the effects at the present time. Easily we can travel back in time to alter the past, but we cannot directly observe the past.We define the data structure to be fully retroactive if in addition to the partially retroactive operations it can also perform queries about the past. Similar to how the standard operation insert(x) becomes Insert(t, "insert(x)") in the partially retroactive model, the operation query(x) in the fully retroactive model now has the form Query(t, "query(x)").

Retroactive Running times

To retroactive data structures running time depends on `m` which is the total number of updates applied in the structure(retroactive or not) , `r` which is the number of updates before which the retroactive operation is to be performed and `n`which is the maximum number of elements present in the structure at any single time.

Automatic Retroactivity

A natural question regarding automatic retroactivity with respect to data structures is whether or not there is a general technique which can convert any data structure into an efficient retroactive counterpart. one simple approach is to perform a rollback method on all the changes made to the structure preceding to the retroactive operation that is to be applied.Here we store as secondary information all changes to the data structure made by each operation in such a way that every change could be reversed. Once we have rolled back the data structure to the suitable state we can

then apply the retroactive operation to make the change we wanted. Once the change is made we must then reapply all the changes we rolled back before to put the data structure into its new state. While this can work for any data structure, it is often inefficient and wasteful especially once the number of changes we need to roll-back is large. To create an efficient retroactive data structure we must take a look at the properties of the structure itself to determine where speed ups can be understood. Thus there is no general way to convert any data structure into an efficient retroactive counterpart. Erik D. Demaine, John Iacono and Stefan Langerman prove this.

Comparison to persistence

At first glance the idea of a retroactive data structures is related at a high level to the persistence data structures since they both take into account the dimension of time . The key difference between persistent data structures and retroactive data structures is how they handle the element of time.In persistent data structures, each version is treated as an unchangeable archive. Each new version is dependent on the state of existing versions of the structure. However, because existing versions are never changed, the dependence relationship between two versions never changes.A persistent data structure maintains several versions of a data structure and operations are performed on one version to produce another version of the data structure.It always preserves the previous version of itself when it is modified. Such data structures are effectively perpetual, as their operations do not update the structure in-place, but instead always produce a new updated structure.Thus, the persistence model is useful for upholding archival versions of a structure, but inappropriate for when changes must be made directly to the past state of the structure. On the other hand in retroactive data structure changes are made to the past versions.Because of the interdependence ofversions, a single change can fundamentally cause a ripple of changes of all later versions.

Some applications of Retroactive Data Structures

In the real world there are many places where one would like to modify(insert and delete) a past operation from a sequence of operations. Some of the possible applications are:

- *Error correction*: Sometimes the data is entered mistakenly which should be corrected. Also the secondary effects of the incorrect data be removed.
- *Recovery*: Assume a hardware sensor was damaged but is now repaired and working correctly also the data is able to be read from the sensor. We would like to be able to insert the data back into the system as if the sensor was never damaged in the first place.
- *Bad data*: When dealing with large systems, particular those involving a large amount of automated data transfer, it is not uncommon. For example suppose one of the sensors for a weather network malfunctions and starts to report garbage data or incorrect data. The ideal solution would be to remove all the data that the sensor produced since it malfunctioned along with all the effects the bad data had on the overall system.

- *Handling the past*: Modifying the past can be beneficial in the cases of damage control and retroactive data structures are designed for intentional manipulation of the past.

Retroactive Union-Sameset: Recovery of Weather- Forecasting Data

In places where, data pertaining to weather is taken from different weather stations and is finally reported to a central computer. Various stations are placed in small groups, thus an average of the data is computed. Small groups are then combined to form larger groups, and now an average of the data is taken and so-on until we are left with only single group. Final data evaluated is, thus, analysed and various statistics are drawn out of it.

1) Problem 1: Suppose, we come to know that one of the weather station got crashed at time 't' in the past.

Normal Solution: Rollback to the point where station got malfunctioned. The solution has linear intricacy and is inefficient, especially in the cases where the number of stations affected is expressively less than the total number of stations. Therefore, we come up with the retroactive solution.

Retroactive Solution: The stations which would have been affected by malfunctioning of that station, only data of those needs to be retroactively modified. That particular group of affected stations can be identified by retroactive query, sameset(x,y).

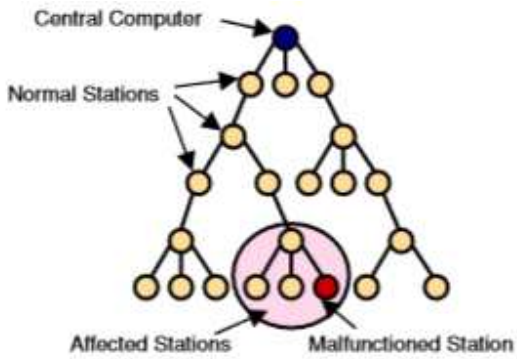

Figure 4. Group of Weather-Stations with one Malfunctioned

Thus, data can be recovered and it would indicate as if the malicious operation never occurred.

2) Problem 2: Suppose, we come to know that one of the weather station was missed to be considered before.

Normal Solution: Rollback to the point where station was missed to be considered. The solution has linear intricacy and is ineffective, especially in the cases where the number of stations affected is significantly less than the total number of stations. Therefore, we come up with the retroactive solution.

Retroactive solution: The stations whose data would have been affected by neglecting that station, only data of those need to be retroactively modified. Retroactive operation can be used to identify that particular group of affected stationsi.e insert (Union(x,y),t).

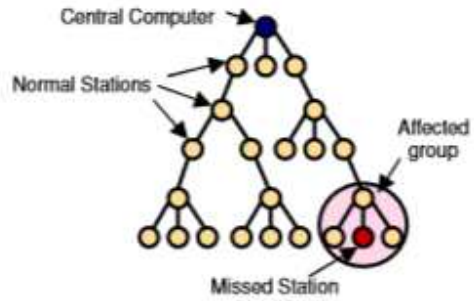

Figure 5. Group of Weather-Stations with one Missed

The data from the missed station can be used to evaluate final data again by applying retroactively adding the station into the group of stations. Thus, data can be analyzed again and it would indicate as if that station was already under consideration.

References:

[1]. Demaine, Erik D; John Iacono and Stefan Langerman (2007). "Retroactive data structures".

[2]. Sylvain Conchon, Jean-Christophe Filliatre, “A Persistent Union-Find Data Structure”, Workshop on ML 2007

[3]. Suneeta Agarwal, Prakhar panwaria. “Implementation, Analysis and Application of Retroactive Data Structures”.

[4]. Kanat Tangwongsan, Guy Blelloch. “Active Data Structures and Applications to Dynamic and Kinetic Algorithms”.

[5]. J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. Journal of Computer and System Sciences, 38(1):86–124, 1989.

[6]. A. Fiat and H. Kaplan. Making data structures confluently persistent. In Proc. 12th Ann. Symp. Discrete Algorithms, pages 537–546, Washington, DC, January 2001.