Upsortable: Programming Top-K Queries Over Data Streams

Julien Subercaze, Christophe Gravier, Syed Gillani
Abderrahmen Kammoun, Frédérique Laforest
Univ Lyon, UJM Saint-Etienne
Laboratoire Hubert Curien UMR 5516
F-42023, Saint-Etienne, France

ABSTRACT

Top-k queries over data streams is a well studied problem. There exists numerous systems allowing to process continuous queries over sliding windows. At the opposite, non-append only streams call for ad-hoc solutions, e.g. tailor-made solutions implemented in a mainstream programming language. In the meantime, the *Stream* API and lambda expressions have been added in Java 8, thus gaining powerful operations for data stream processing. However, the Java *Collections Framework* does not provide data structures to safely and conveniently support sorted collections of evolving data. In this paper, we demonstrate `Upsortable`, an annotation-based approach that allows to use existing sorted collections from the standard Java API for dynamic data management. Our approach relies on a combination of pre-compilation abstract syntax tree modifications and runtime analysis of bytecode. `Upsortable` offers the developer a safe and time-efficient solution for developing top-k queries on data streams while keeping a full compatibility with standard Java.

1. INTRODUCTION

Stream data processing systems have drawn the attention of the database community for more than a decade [1, 3, 10]. Numerous systems have been developed to handle continuous queries in the frame of real-time applications. The sliding-window paradigm is well-suited for processing the large amount of real-time data in standard real-time monitoring applications [9, 4]. This paradigm underlies the vast majority of existing data stream processing systems. Among the capabilities of such systems, top-k querying within sliding windows has been widely covered [12, 15, 11]. On the data structure side, there is a vast body of work on approximate evaluation of frequent items, top-k and cardinality for stream processing [5, 13, 6].

The sliding window paradigm covers the needs for major monitoring applications, but one size does not fit all and more complex analytics have requirements that cannot be met by this paradigm. If the data expiration is not linear with the time systems based on sliding-windows fall short [8]. Therefore these real-world data stream processing applications require ad-hoc developments with standard programming languages.

Programming languages have also evolved to answer the need for data stream processing. Be it with Domain Specific Languages [2, 16, 17], language extensions [14, 7] or with evolutions of standard API like *Stream* for Java, this field demonstrated many advances in the last few years. However, the existing data structures of these languages have been designed for static data processing and their correct use with evolving data is cumbersome – top-k query processing requires maintaining sorted collections. We show that maintaining sorted collections of dynamic data is particularly error-prone and leads to hard-to-detect bugs. In this demo, we tackle the issue of maintaining dynamically sorted collections in Java in a safe and transparent manner for the application developer. For this purpose, we developed an annotation-based approach called `Upsortable` – a portmanteau of update and sort – that uses compilation-time abstract syntax tree modifications and runtime bytecode analysis. `Upsortable` is fully compatible with standard Java and is therefore available to the greatest number of developers [1].

2. THE CASE FOR UPSORTABLE

The standard Java *Collections* API contains three implementations of sorted data structures: the *java.util.TreeSet* backed by a Red-Black tree, the *java.util.PriorityQueue* that implements a priority heap, and for thread-safety purpose, the *java.util.concurrent.ConcurrentSkipListSet* implements a concurrent variant of Skip List. These structures especially implement `add` and `remove` primitives, as well as methods to navigate within these collections. These structures are therefore well-suited for the implementation of exact top-k queries: elements are kept sorted according to either a comparator provided at the creation time of data structure or by the natural ordering of the elements. In both cases, a pairwise comparison method is used to sort the objects and this method must provide a total ordering. When dealing with data streams, the value of some fields of an object are subject to evolution and this evolution may require a reordering

[1] `https://github.com/jsubercaze/Upsortable`

Proceedings of the VLDB Endowment, Vol. 10, No. 12

within the collections this object belongs to. With the aforementioned sorted data structures – as well as third-parties Java Collections API such as Guava[2] or Eclipse Collections[3] – the developer must first remove the object from each sorted collections, update its internal fields and reinsert the object in these collections. The sorted collections may otherwise become irredeemably broken. Figure 1 depicts such an example. Hence, this `remove, update` and then *insert* process is very error-prone, especially in large code base where objects belong to different sorted collections, depending on the state of the application. Broken sorted collections are also hard to identify at runtime and may go undetected for a while. This is typical for the top-k queries, where the collections might be broken after the k-th element. The behaviour of the corrupted data structure is not predictable, it ranges from inconsistent results to wrong inserts and impossible removals – as depicted in Figure 1 where the removal of D is impossible since it cannot be reached.

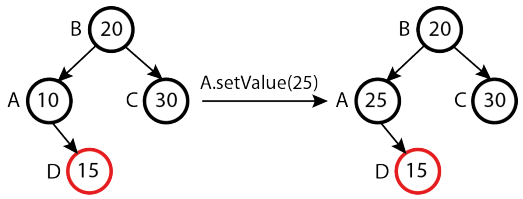

Figure 1: Example of a corrupted Red-Black tree after update of Object A via call to its setter.

To circumvent this issue, the standard solution is to rely on the *Observer* design pattern. This pattern implies that the objects must keep track of the collections they belong to. This requires to add an extra data structure within the objects to store pointers to the collections they belong to. The field setters must be updated to remove, update and insert the object (acting as the *notify* in the pattern). Using a dynamic array to store the pointer is the most compact way, however it may lead to useless remove and update if the modified field does not participate in the comparison of some sorted structures that the objects belong to. Using a Hashmap circumvents this issue by mapping fields to the structures where the object belongs and where the fields participate in the comparison. However in both cases, when dealing with millions/billions of objects that are created and destroyed during the application lifetime, this solution has a very high memory cost. Moreover, it still requires heavy modifications of the source code by the application developer who must handcraft these routines for each object definition and for each setter.

Listing 1: Annotation based solution

```
@Upsortable
public class MyObject {

    private int firstField;
    private String secondField;

}
```

[2]https://github.com/google/guava
[3]https://www.eclipse.org/collections/

3. SOLUTION OVERVIEW

Our solution proposes an alternative to the *Observer* pattern that does not require any other source code modification than adding an annotation and has a restricted memory fingerprint. The developer simply uses the `@Upsortable` annotation at the class level to declare that the internal fields are subject to modification and that the sorted collections it belongs to must be dynamically updated – such as depicted in Listing 1. Our framework performs all the required updates to maintain the collections correctly sorted when setters update values in the object fields.

The underlying idea of our solution is that in real-time applications the number of sorted collections is very small compared to the number of objects that are sorted within these collections – dozens against millions in practice. We leverage this imbalance to devise an approach that does not require to add any extra data structure to the objects definition. Instead of linking objects to the collections they belong to, as in the *Observer* pattern, a global map links each field definition to the list of collections where it participates in the comparison process.

To relieve the developer from the burden of implementing this process, our framework consists of two parts: a transparent source code injection during the compilation phase and an encapsulation of the standard API sorted collections to automatically manage the global collection.

3.1 AST modifications

The Java compilation is a two-steps process. The first step parses and compiles the source code and the second one processes the annotations. The Lombok project[4] has demonstrated the feasibility of modifying and recompiling the Abstract Syntax Tree (AST) during the second step, allowing annotations to transparently inject source code. Our framework, based on Lombok, injects setters methods for the classes annotated *@Upsortable*. The pseudo code of the setter method is given in Algorithm 1. The setter retrieves the sorted collections associated to the current field name – obtained via reflection – and performs the remove, update, insert operations. The algorithm keeps track of the sets the current object participates in (by contract, *remove()* returns true if the object was present). As a consequence, we are guaranteed to insert the updated object in the correct collections. Usage of WeakReference is detailed in Section 3.3. Figure 2 depicts the source code injection via AST modification during the annotation processing phase.

3.2 Bookkeeping

To keep track of the mappings between the fields names and the sorted collections, we encapsulate the creation of the sorted collections using the static factory pattern.

Listing 2: Collection instantiation with upsortable

```
//Without upsortable
TreeSet<MyObject> mySet = new
    TreeSet<>(comparator);

//With upsortable
UpsortableSet<MyObject> mySet =
    Upsortables.newTreeSet(comparator);
```

[4]https://projectlombok.org/

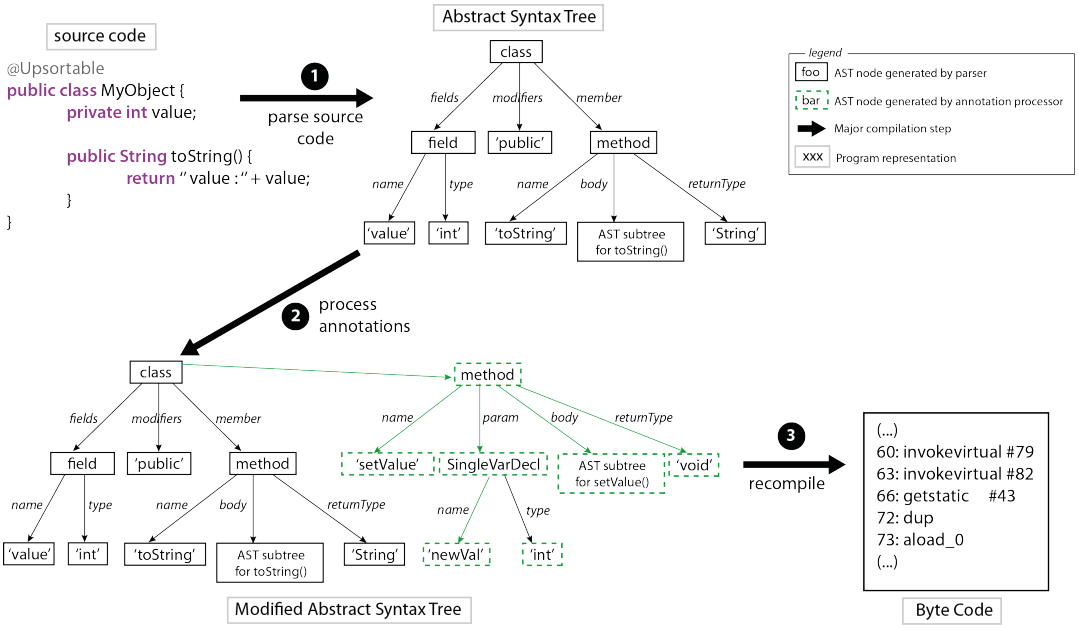

Figure 2: Upsortable Abstract Syntax Tree modifications at annotation processing time.

Algorithm 1: Injected Setter code during annotation processing

```
   Input: newValue: the new value of the field
   // Fails fast if unchanged
1  if this.field == newValue then
2  |  return
3  end
   // List of references to the collections concerned by
      this field
4  refsList ← refMap.get(currentFieldName);
   // Remove this from the collections, remove cleaned
      references
5  participatingCollections = newArrayList();
6  for ref ∈ refsList do
7  |  if ref is cleaned then
8  |  |  remove from refList
9  |  else
10 |  |  if ref.deref().remove(this) then
11 |  |  |  participatingCollections.add(ref.deref())
12 |  |  end
13 |  end
14 end
   // Update the value
15 this.field ← newValue
   // Reinsert in the right collections
16 for collection ∈ participatingCollections do
17 |  collection.add(this)
18 end
```

We created a class called *Upsortables* that exposes static methods to create sorted structures backed by the standard Java API ones: *TreeSet*, *ConcurrentSkipList* and *PriorityQueue*. These static factory methods require the usage of comparator for the creation of sorted collections, disallowing the usage of natural ordering. The comparator implements per definition a *compare(MyObject o1, MyObject o2)* method. The static factory methods analyze the content of the *compare* method via runtime bytecode analysis in order to extract the fields of *MyObject* that participates in the comparison. For this purpose, we use Javassist, a common bytecode manipulation library. The extracted field names are then associated to the sorted collection that is being created in the global map. For performance reasons, we provide two versions of this global collection, one being thread-safe, the other not. On the developer point-of-view, besides the usage of the annotation, the sorted collection instantiation is the only modification, albeit minor, that is required to use `Upsortable`. Listing 2 depicts the minor changes that this encapsulation implies. The burden on the developer side is therefore very limited and does not bring any particular difficulty.

3.3 Garbage Collection

Sorted collections may be created and deleted during the lifecycle of the application. Our framework shall therefore not interfere with the lifetime of these collections and shall especially not prevent them from being collected by the garbage collector (GC). To prevent the Hashmap that maps fields definitions to the Upsortable collections to hold a reference to these collections that would prevent their collection by the GC, we use a *WeakReference*. Contrarily to the soft references, weak ones do not interfere with the garbage collection of the objects they refer to. The injected setters' code takes care of removing weak references that have been cleaned up by the garbage collector. By relying on the *ListIterator*, we are able to both process valid references and remove cleaned ones in a single iteration over the list of weak references.

3.4 Discussion

The *Upsortable* approach offers a convenient and safe solution to manage dynamically sorted collections. Naturally, safety and convenience have a performance impact. Keeping track of the relation between fields and sorted collections in

Upsortable has a very limited memory fingerprint – especially compared to the *Observer* design pattern – and the CPU impact is also limited. Since we leverage the imbalance between the number of objects and collections, this leads to a very few useless removes (a $O(log(n))$ operation for three data structures) and has a very limited impact of several percents ($< 5\%$) of the runtime in the practice, depending on the input data.We show the attendee the impact of the framework on a real-world application in the second scenario of our demonstration.

4. DEMONSTRATION

In this demo, we will showcase i) the impact of `Upsortable` on development time in the frame of an illustrative scenario, where annotated class instances participate to a single collection; ii) the usage of `Upsortable` in a complex scenario with real-world data in a large code base.

Annotating a class field using `Upsortable`: For this demo, we provide a Java project that consumes a stream of temperatures measurements issued by one hundred different temperature sensors. Each temperature value streamed to the system is represented in memory as instances of the class `TemperatureSensor`, which holds two fields: a sensor identifier (a unique `String`) and the current temperature value for this sensor (a `float`). In this application, any new sensor value supersedes the previous one. The functional objective of the system is to deliver, at each new received value, the ten sensors with the greatest values (top-10 query over all sensors). We showcase how to effectively use the `Upsortable` annotation in order to answer such a query over stream with as little development time as possible. That is by adding the `Upsortable` annotation and adding the temperature sensors values to an `UpsortableSet` – using the set that encapsulates Java collections API `TreeSet`. Would the attendee prefer to use another collection such as a `ConcurrentSkipList` or a `PriorityQueue`, we then showcase the easiness of changing the underlying data structure. Besides being a first scenario demonstrating the practical interest for `Upsortable`, the attendees will be shown that only two lines of code and an annotation (`@Upsortable`) are required to implement the top-k query over data stream.

Non-appendable data stream processing. The aim of the second demo program for `Upsortable` is to provide a data stream processing scenario over highly heterogeneous data and more complex continuous queries. This program is part of our answer to the DEBS 2016 Grand Challenge [8] that was selected as finalist runner-ups. In this application, the underlying scenario addresses the analysis metrics for a dynamic (evolving) social network graph. The item within the stream are social events of four kinds: new *friendship* between users, a new *post* created, a new *comment* posted in response to a *post*, or a user declares a *like* on a post. This query is of complex nature since instances belong to several collections that must be continuously tracked to be updated when constituting instances are updated. The continuous query that the system must answer is the identification of the posts that currently trigger the most activity in the social network. Posts expiration is triggered when their scores reach zero, which is not bound to a sliding window but actually to the activity of the social network. The total score of an active post is computed as the sum of its own score plus the score of all its related comments. A decreasing factor is applied to both posts and comments – older events results in lower weights. Since posts lifespan cannot be predicted, this application showcases a non append-only application [12]. The attendee will gain a deeper understanding of the power of `Upsortable` where instances are candidates belong to several collections – and this ownership to different collections can change over time. The main issue tackled here ensues from events that can become obsolete and updated frequently – when a *like* or a *comment* is produced for a given post, this updates its score hence its ranking in the continuous top-k query data structure. We showcase that this previously prone-to-error and time-consuming task is a case where `Upsortable` shines.

5. REFERENCES

[1] D. J. Abadi et al. Aurora: a new model and architecture for data stream management. *VLDBJ*, pages 120–139, 2003.

[2] J. Bosboom et al. StreamJIT: A commensal compiler for high-performance stream programming. In *OOPSLA*, pages 177–195, 2014.

[3] S. Chandrasekaran et al. Telegraphcq: continuous dataflow processing. In *SIGMOD*, pages 668–668. ACM, 2003.

[4] G. Chen et al. Realtime data processing at facebook. In *SIGMOD*, pages 1087–1098. ACM, 2016.

[5] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, pages 58–75, 2005.

[6] M. Durand and P. Flajolet. Loglog counting of large cardinalities. In *ESA*, pages 605–617, 2003.

[7] P. Eugster and K. Jayaram. EventJava: An extension of Java for event correlation. In *ECOOP*, pages 570–594. Springer, 2009.

[8] V. Gulisano et al. The DEBS 2016 grand challenge. In *DEBS*, pages 289–292, 2016.

[9] S. Kulkarni et al. Twitter heron: Stream processing at scale. In *SIGMOD*, pages 239–250, 2015.

[10] J. Meehan et al. S-store: Streaming meets transaction processing. *PVLDB*, pages 2134–2145, 2015.

[11] A. Metwally et al. Efficient computation of frequent and top-k elements in data streams. In *ICDT*, pages 398–412, 2005.

[12] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD*, pages 635–646, 2006.

[13] N. Ntarmos, P. Triantafillou, and G. Weikum. Counting at large: Efficient cardinality estimation in internet-scale data networks. In *ICDE*, page 40, 2006.

[14] G. Schueller and A. Behrend. Stream fusion using reactive programming, linq and magic updates. In *FUSION*, pages 1265–1272, 2013.

[15] M. A. Soliman et al. Top-k query processing in uncertain databases. In *ICDE*, pages 896–905, 2007.

[16] X. Su et al. Changing engines in midstream: A Java stream computational model for big data processing. *PVLDB*, pages 1343–1354, 2014.

[17] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, pages 179–196, 2002.