Timeline Filters Design

Varun Saxena

This document would cover design for timeline filters.

1. Background

Data stored in the backend store by Application timeline server can be huge. It is not feasible to return such large amounts of data back to client. Moreover, clients would generally be interested in only a subset of data. So ATSv2 needs to provide an effective mechanism for the client to filter out entities to get only data of interest. In ATSv1, this was achieved via created time window, primary filters and secondary filters(based on other info field). Both primary and secondary filters looked for a key-value match and complex SQL type logical operations for query were not supported.

As ATSv2 treats configurations, metrics, info, events, relations, etc. as first class entities, we can support a wider variety of filters viz. config filters, metric filters, info filters, event filters, relation filters, etc. If any of these filters does not match, entity(flow/flow run/application/generic entity) is not returned in response.
Moreover, the intention is to support complex filter expressions with support for logical operators such as OR and AND to group filter clauses together.
Also, for things such as metrics, looking for a key value match is not useful and comparison operators such >=, <=,>,<,==,!= make more sense.
Additionally, client may only be interested in specific metrics and configs in a single query and as number of metrics and configs can be huge, it may be useful to specify which metrics or configs to retrieve from ATS reader.

Hence, keeping these use cases in mind, we need to devise a way to represent these filters both within ATS reader and in REST API interface. Next few sections will touch upon the design of timeline filters.

2. Design and Implementation details.

As discussed earlier, we support metric filters, config filters, info filters, relation filters and event filters in ATSv2.
As both configurations and info are key-value pairs, for config/info filters we just need to check for key-value pair being equal or not equal. Also, there can be alternate views on what not equal to means. If config or info key is not found, and comparison operator is not equal to, should entity be returned or not returned ? We will hence let client decide and support both cases.

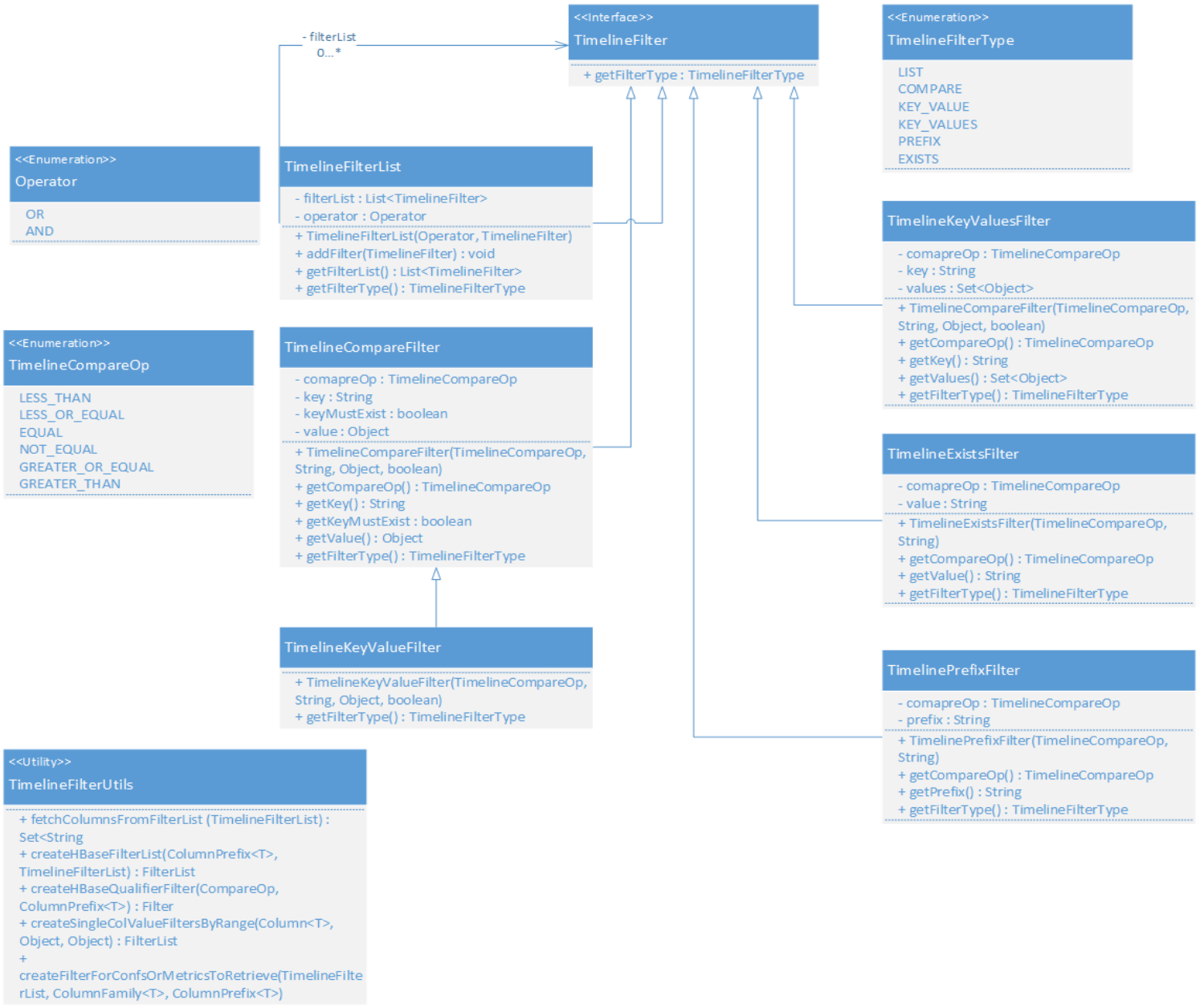
Metrics on the other hand are numerical values(only longs supported in ATSv2) which can either be counters or gauges and hence do not have fixed values. For metrics therefore, metric filters cannot merely check for equality(equals/not equals). We need to perform comparison of values for a given metric. Thus, we need to support comparison operators namely <,>,<=,>=,== and !=.
Event filters are designed to check only existence of an event.
Relation filters though check whether for an entity type, given entity IDs' are related to or relate to the entity being matched. Here entity can be treated like a key and entity IDs'as values because this is the way relations are stored in TimelineEntity object(as Map<String, Set<String>>).

We will also support specifying which metrics or confs to retrieve. In case of configs and metrics, if we retrieve all of them, this can become a lot of data even though the user doesn't need it. So we need to provide a way to query only a set of configs or metrics.
Moreover, as number of configs and metrics can be big, asking client of specify exact names of metrics of configs to retrieve may not be feasible.
In Hadoop, a lot of configs share a common prefix (yarn.resourcemanager for RM, yarn.nodemanager for NM, mapreduce.map for mappers, mapreduce.reduce for reducers and so on) and client is likely to get configurations with common prefix.
Same goes for metrics.
So we will support specifying prefixes for configs / metrics to retrieve.

Based on these considerations, below classes have been designed to represent filters.

a) *Main Classes*

Below is a class diagram representing the classes involved in filter implementation.



TimelineFilter : This is the interface which each of the filter classes implement. This interface also contains an enumeration TimelineFilterType which defines the type of filter.

***TimelineFilter$TimelineFilterType* :** The filter types decided are LIST, COMPARE, KEY_VALUE, KEY_VALUES, PREFIX and EXISTS. More on them in filter implementation classes.

***TimelineFilterList* :** Implementation of TimelineFilter that represents an ordered list of filters(of type TimelineFilter) which will be evaluated with a specified boolean logical operator

TimelineFilterList.Operator.AND or TimelineFilterList.Operator.OR. Since you can use TimelineFilter lists as children of TimelineFilter lists, you can create a hierarchy of filters to be evaluated. This means you can add another TimelineFilterList to a TimelineFilterList. The intention basically is to support a filter expression such as below:

(((Filter1 AND Filter2 AND Filter3) OR (Filter4 OR Filter5)) AND (Filter6 AND Filter7))

***TimelineCompareFilter* :** This filter is used for filtering by comparison. It has associated compare ops'specified by TimelineCompareOp enum. Supported types are LESS_THAN, LESS_OR_EQUAL, EQUAL, NOT_EQUAL, GREATER_THAN and GREATER_OR_EQUAL.

This filter has a key, value and a compare op. We also have a boolean flag named keyMustExist. This flag has meaning only if compare op is TimelineCompareOp.NOT_EQUAL. For all other compare ops', this flag is treated as true. If true, this means filter will not match if key does not exist even if compare op is NOT_EQUAL. If false, filter would match and we would return the entity if compare op is NOT_EQUAL.

This filter class is used to represent metric filters.

***TimelineKeyValueFilter* :** This filter extends TimelineCompareFilter and is used to match key-value pairs. Its exactly similar in behavior to TimelineCompareFilter except that the supported compare ops' are only EQUAL and NOT_EQUAL.

This filter class is used to represent config and info filters.

***TimelineKeyValuesFilter* :** This filter is used to match a key and multiple values. It contains a key, compare op and a set of values. Only the values specified in the filter need to match the values in entity. Support compare ops' are EQUAL and NOT_EQUAL.

This filter class is used to represent relation filters(relatesto/isrelatedto).

***TimelineExistsFilter* :** This filter is used to check for existence/non-existence of a value. Supported compare ops' are EQUAL and NOT_EQUAL.

This filter class is used to represent event filters. If an event does not exist (and compare op is EQUAL) or an event exists (and compare op is NOT_EQUAL), entity would be dropped from the response.

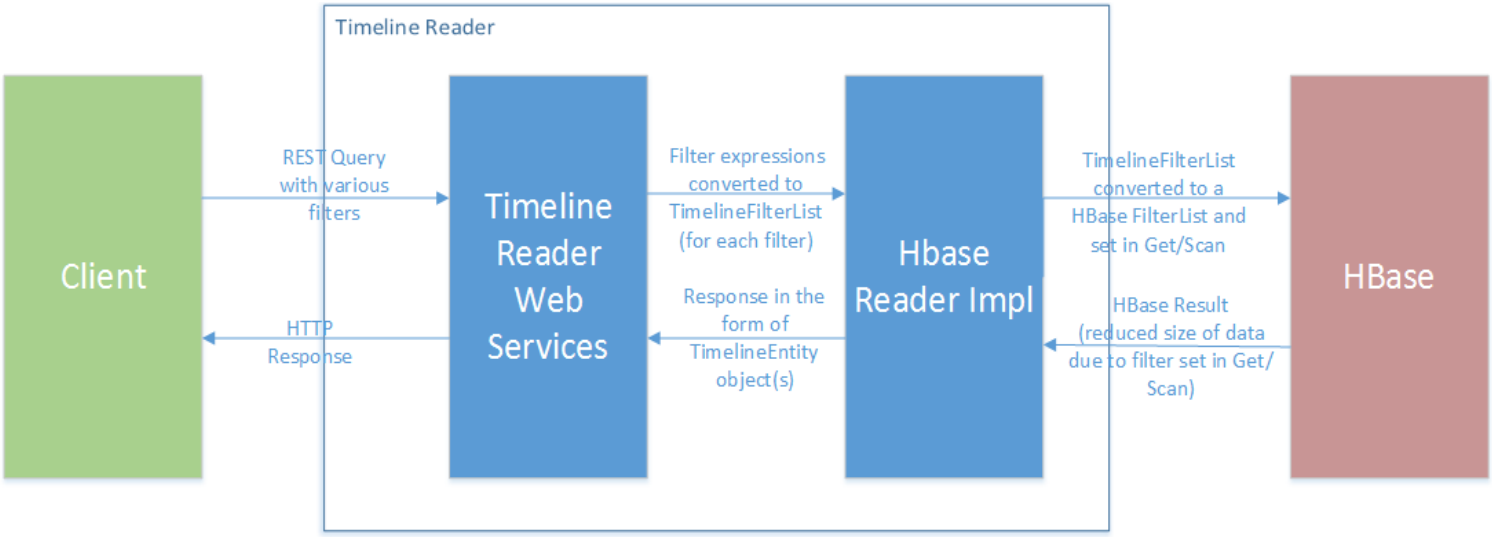
***TimelinePrefixFilter* :** This filter is used to match prefixes. Supported compare ops' are EQUAL and NOT_EQUAL.

This filter class is used to represent configs to retrieve and metrics to retrieve. The config or metric is compared against the prefix specified and if the prefix is not the same (and compare op is EQUAL) or prefix is same (and compare op is NOT_EQUAL), that config/metric would not be sent back in response. Kindly note this is not used to filter out an entity but rather decides what is to be sent within each entity.

For example: If metric prefix is HDFS_ and compare op is EQUAL, all metrics starting with HDFS_ would be sent back in response. If compare op is NOT_EQUAL, all metrics except those starting with HDFS_ would be sent back in response.

b) ***HBase Reader implementation and HBase Filters***

HBase has its own filter representation and one of the key reasons for coming up with our filter representation was to have the ability to convert timeline filters to corresponding HBase filters and pass them on in Get/Scan. This can reduce the amount of data being transferred over the wire while fetching records from HBase backend.



As can be seen above, when a REST query arrives, TimelineReaderWebServices will convert the filter expressions for various filters to a TimelineFilterList(for each filter). This will then be passed to storage layer. If its HBase storage implementation, HBase implementation will convert these Timeline filters encapsulated in a TimelineFilterList to corresponding HBase filters(if possible) and encapsulate them in a HBase FilterList. This filter list is then passed on to HBase Get/Scan. This greatly reduces the amount of data transferred over wire.

Below explains how each of the timeline filters are converted to corresponding HBase filter depending on entity type.

Application Entity/Generic Entity :

Metric filters, represented by TimelineCompareFilter(wrapped in TimelineFilterList(s)) and Info/Config Filters represented by TimelineKeyValueFilter(wrapped in TimelineFilterList(s)) will be converted to SingleColumnValueFilter. Please note keyMustExist flag is set to SingleColumnValueFilter #setFilterIfMissing() to decide whether non-existence of a key would mean whether a row is dropped or not.

Event filters, represented by TimelineExistsFilter(wrapped in TimelineFilterList(s)) cannot be directly converted to a HBase filter because of the way events are stored. Events are stored with column qualifier as e![event_id]=[event_timestamp]={event_info_key} which means we cannot create a SingleColumnValueFilter from the event ids'specified in event filters. For this, if events are not specified in fields to retrieve, we create QualifierFilter filters for each event id specified in event filters to fetch only those events which are required to match event filters. The filters are then matched locally in timeline reader.

Relation filters, represented by TimelineKeyValuesFilter cannot be directly converted to a HBase filter either because of the way relations are stored. Column qualifier is `r![entitytype]` or `s![entitytype]` and associated value is stored as a list entity ids' separated by = i.e. like, `entityid1=entityid2=entityid3.` For this, similar to events, if relations (relatesTo / isRelatedTo) are not specified in fields to retrieve, we create QualifierFilter filters for each entity type specified in relation filters to fetch only those relations which are required to match relation filters. The filters are then matched locally in timeline reader.

When it comes to metrics to retrieve and confs to retrieve, they are represented by TimelinePrefixFilter. These filters are converted to QualifierFilter with BinaryPrefixComparator to fetch only those configs/metrics which are required.

FlowRun Entity :

When it comes to fetching metrics for flow runs, we have a custom flow run coprocessor which performs summation of multiple metric values for a metric written to the backend(by apps belonging to the flow run) before returning them to timeline reader.

This means we cannot create filters and set them in HBase Scan/Get as filtering is done before coprocessor is run. This means we have to apply metric filters locally.

So similar to case with events and relations above, we fetch all the metric IDs' from metric filters and create QualifierFilter(s) so that we fetch only those metrics which are required for matching metric filters locally.

c) *Filter Representation at REST Layer:*

Filters are received as optional query parameters by various REST endpoints(for querying apps, entities, runs,etc.).

- **Metric filters** – Expression for representing metric filters will be as under :
  (((<expr>[<op> <expr>…..]) <op> (<expr>[<op> <expr>…..])) <op> (<expr>[<op> <expr>…..]))
  Here, <op> is a logical operator which can be either "AND" or "OR". This will directly convert to TimelineFilterList$Operator.
  Each <expr> would be transformed into single/multiple TimelineCompareFilter objects and wrapped inside a TimelineFilterList.
  Different <expr> or expressions can be combined together with <op>. Brackets("(" and ")") are used to club together different logical expressions. Square bracket is not part of the representation and merely denotes that inside within a single opening and closing bracket, multiple <expr> can be combined using <op>, typically same <op> .
  If brackets are not specified, operators will be parsed left to right with a new TimelineFilterList created with the old filter list wrapped inside it, whenever <op> changes.

  <expr> in turn is of the form :
  <key> <compareop> <value>

Here, <key> is metric ID and <value> is the value which will be used to compare against the metric value.
<key> corresponds to TimelineCompareFilter#key and <value> to TimelineCompareFilter#value.

<compareop> transforms directly to TimelineCompareOp. There are 7 compare ops' supported.

- ✓ gt – Equivalent to TimelineCompareOp.GREATER_THAN
- ✓ ge – Equivalent to TimelineCompareOp.GREATER_OR_EQUAL
- ✓ lt – Equivalent to TimelineCompareOp.LESS_THAN
- ✓ le – Equivalent to TimelineCompareOp.LESS_OR_EQUAL
- ✓ eq – Equivalent to TimelineCompareOp.EQUAL
- ✓ ne – Equivalent to TimelineCompareOp.NOT_EQUAL. TimelineCompareFilter#keyMustExist will be set to false. Entity would be returned if key or metric id does not exist.
- ✓ ene – Equivalent to TimelineCompareOp.NOT_EQUAL. TimelineCompareFilter#keyMustExist will be set to true. Entity would not be returned if key or metric id does not exist.

Example :

((metric1 lt 40 OR metric2 gt 80) AND (metric3 eq 10)) OR (metric4 lt 5 AND metric5 ne 4 OR metric6 ene 7)

Please note that all the URL unsafe characters including spaces have to be properly encoded by client.

- **Config and Info filters** – Expression for representing config and info filters is exactly same as metric filters. Only difference being that only 3 <compareop> are supported, namely, eq, ne and ene. And, the corresponding filter is TimelineKeyValueFilter instead of TimelineCompareFilter.

- **Event filters** – Expression for representing event filters will be as under :
(([!](<value>[,<value>,.....]) <op> [!](<value>[,<value>,...])) <op> [!](<value>[,<value>,.....]))
Here also, <op> is a logical operator which can be either "AND" or "OR". This will directly convert to TimelineFilterList$Operator.
Each <value> here means event ID and would go to TimelineExistsFilter#value. A comma separated list of event IDs' would be transformed into single(if no comma) or multiple TimelineExistsFilter objects and wrapped inside a TimelineFilterList with Operator AND.

"!" here means NOT. A comma separated list of events inside an opening and closing bracket pair with a "!" before opening bracket ("(") means the events should not exist for an entity to match. The TimelineCompareOp in each of these TimelineExistsFilter(s) here would be set to NOT_EQUAL.

If there is no "!" before opening bracket, it means the corresponding TimelineCompareOp will be EQUAL.

Basically a sub-expression such as (event1,event2,event3) or event1,event2,event3 would transform to a TimelineFilterList with Operator "AND" and containing following filters :
TimelineExistsFilter (event1, TimelineCompareOp.EQUAL),
TimelineExistsFilter (event2, TimelineCompareOp.EQUAL)
and
TimelineExistsFilter (event3, TimelineCompareOp.EQUAL)

If the subexpression is !(event1,event2,event3) would transform to a TimelineFilterList with Operator "AND" and containing following filters :
TimelineExistsFilter(event1, TimelineCompareOp.NOT_EQUAL),
TimelineExistsFilter (event2, TimelineCompareOp.NOT_EQUAL)
and
TimelineExistsFilter (event3, TimelineCompareOp.NOT_EQUAL)
Please note brackets cannot be omitted if we want compare op to be NOT_EQUAL. The comma separated list of values(events) must be within brackets with a "!" preceding opening bracket.

Example :

((((event1,event2,event3) AND !(event4,event5)) OR event6,event7) AND !(event8))

- **Relation filters** – Expression for representing relation filters is same as event filters except that the value is treated differently.
  Basic expression is same as event filters. But the <value> for relation filters is further represented as :
  <entity_type>:<entity_id>[:<entity_id>:<entity_id>…]
  It is a colon separated list of entity type and IDs'. Spaces are not allowed in expression above.
  The corresponding filter constructed for relation filters is TimelineKeyValuesFilter.

  Example :

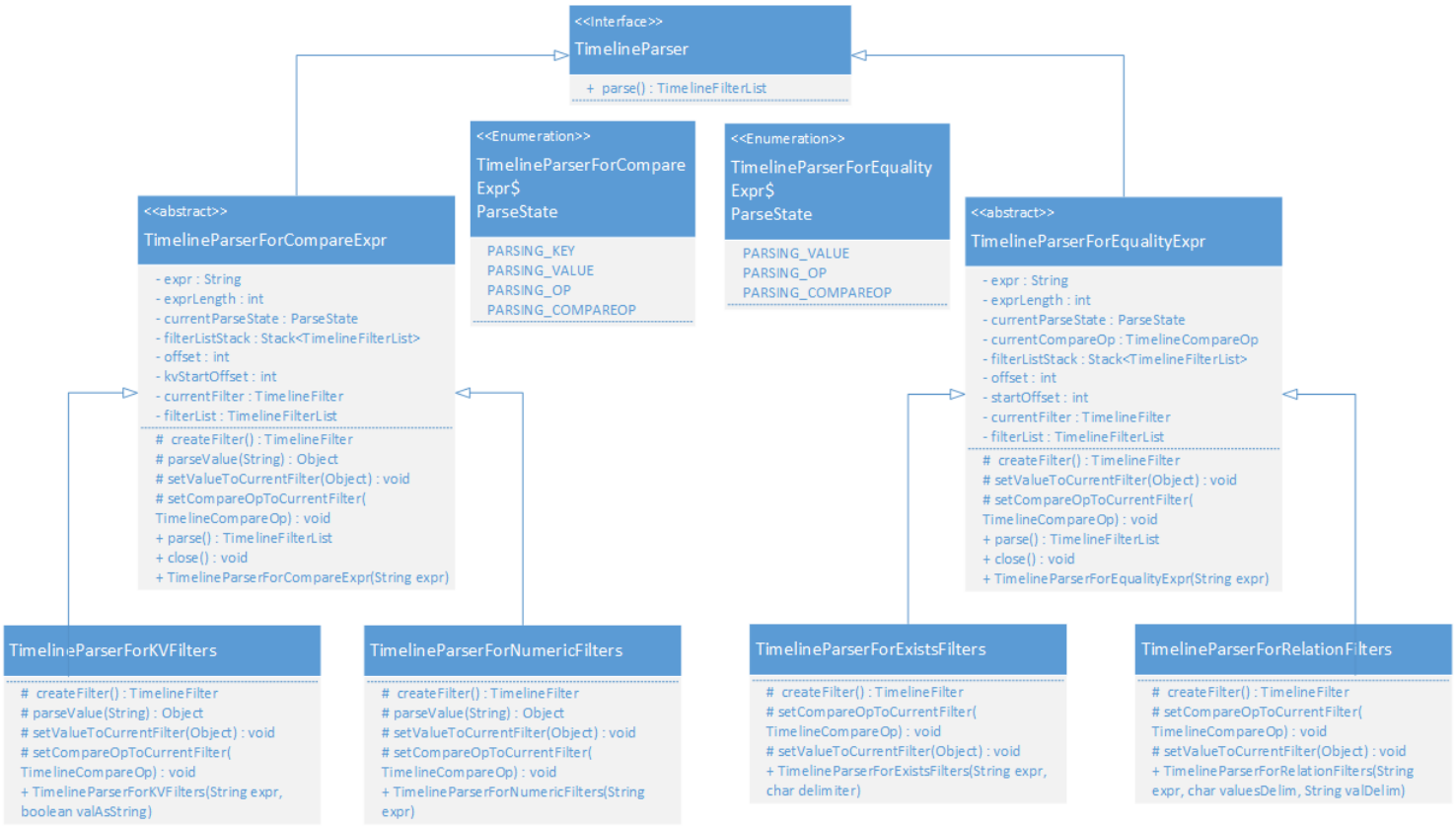
  ((((type1:entity11:entity12:entity13,type2:entity21,type3:entity31:entity32) AND

  !(type4:entity41,type5:entity51)) OR type6:entity6,type7:entity71) AND !(type2:entity25))

- **MetricsToRetrieve/ConfsToRetrieve** – Expression for representing metrics / configs to retrieve is as under :
  [!](<value>[,<value>,…..])
  Here we have no <op>. "!" has same meaning as event filters i.e. its considered as compare op of NOT_EQUAL.

If “!” is not specified, compare op will be EQUAL. Please note brackets are not required if compare op is intended to be EQUAL.
<value> represents a config prefix if expression is for confs to retrieve.
If expression is for metrics to retrieve, <value> represents a metric prefix.
Each value is transformed into TimelinePrefixFilter and then wrapped inside a TimelineFilterList.
If compare op is NOT_EQUAL, TimelineFilterList will be constructed with Operator AND, and if compare op is EQUAL, TimelineFilterList will be constructed with Operator OR.

Parsing of these expression is further handled by below classes.



TimelineParserForCompareExpr parses expression of the format :

(((<expr>[<op> <expr>.....]) <op> (<expr>[<op> <expr>.....])) <op> (<expr>[<op> <expr>.....]))

where,

<expr> = <key> <compareop> <value>

TimelineParserForEqualityExpr parses expression of the format :

(([!](<value>[,<value>,.....]) <op> [!](<value>[,<value>,...])) <op> [!](<value>[,<value>,.....]))

Above 2 classes are both abstract classes which parses the expression. These classes are further extended by concrete implementations to provide custom implementations for abstract methods which determine which TimelineFilter to create for a specific filter and how to handle values and compare ops'.

TimelineParserForKVFilters extends TimelineParserForCompareExpr and is used for parsing config and info filters. This class creates TimelineKeyValueFilter for config/info filters. Only EQUAL and NOT_EQUAL compare ops' are supported for config/info filters.

TimelineParserForNumericFilters extends TimelineParserForCompareExpr and is used for parsing metric filters. This class creates TimelineCompareFilter for metric filters. While setting value to filter, this class checks if value is numeric.

TimelineParserForExistsFilters extends TimelineParserForEqualityExpr and is used for parsing event filters. This class creates TimelineExistsFilter for event filters.

TimelineParserForRelationFilters extends TimelineParserForEqualityExpr and is used for parsing relation filters. This class creates TimelineKeyValuesFilter for relation filters. For relation filters, while a call is made from TimelineParserForEqualityExpr to set value, the value is reinterpreted in the format specified in the section of relation filters.

3. Related JIRAs'.

- ✓ YARN – 3863 : Support for TimelineFilter framework and implementation of all the filters.
- ✓ YARN – 3862 : Support for metrics and configs to retrieve.
- ✓ YARN – 4447 : Representation of complex filter expressions at the REST Layer (Section 2 (c) above).
- ✓ YARN – 5011 : Support metric filters for flow runs.