UMP Alert Engine

Status

PENDING Open for comments, suggestions and questions.

Requirements

As alert engine of monitoring platform, there are below requirements need to be met:

1. Be able to easily onboard metric stream & policy, without user submit code (or low-level implementation detail like storm topology).
2. Be able to dynamically add/remove metrics/events in-to/out-of existing topology. Given UMP as platform, it indicates from source (data-process output or well-formed user input), there might be one or more topics that dynamically changed at runtime.
3. Be able to dynamically scale metric/event and policy in topologies. As a platform, user doesn't need to care about the detailed infrastructure load. The alert engine need to scale out the user's computation with its own knowledge & algorithm, possibly with minimum of user's input.
4. Be able to support dynamic stream join. These streams might come from one data source, or different data sources. One datasource (topic in KAFKA world) might be identified as different event stream based on field of stream's schema(e.g. "metricName"). The alert engine need to provide this stream multiplex capability.
5. Be able to route the alerts to different target.

SLA Requirements

1. Rule creation does not need to happen immediately.(Immediately here means seconds)
    a. When a user dreams up a new rule, create it. It will eventually be provisioned and start functioning. Eventually will be defined later, but measured in many minutes to hours.
2. Major changes to the rules under the direction of the Customer are allowed to glitch the system at the moment of change.
    a. By glitch, we will keep this open, but it can mean loss of windows, loss of data, even missed alerts.
    b. However the coordination of the changes between the DR stacks of Sherlock shall be such that, within reason, during the transition period, if one of the rules should fire, then there will be a firing; generally the old rule will continue to function on the unmodified stack till the new rule is up and functioning on the modified stack; then one can modify the other stack.
3. Major changes to the rules can be staged in time. That is, there is no immediacy (e.g. within seconds or minutes) for the rule to actually take effect.
4. Minor changes to the rules must simply happen immediately (e.g. within seconds, ideally not more than 1 minute). Minor changes are things like: "SEC is in crisis mode, and some alert is spamming the console, silence it immediately"; "IT just did a change and the baseline threshold just altered and our dashboards have lit up, but are OK, so let's alter the threshold".
    a. Not minor: "I have decided that Bollenger Bands are not good, and would like to change it to week over week." This is major.
5. The following is not even something I think is appropriate for a production system: "I would like to experiment with lots of different rules, and need changes to happen quickly so my experimentation does not take very long" - this is really for a simulator, and not a production,

Tier 1, system.

Not a requirement:

- Input stream normalization, this is supposed to be done in pre-process module. Thus no user transformation logic of input to submit to alert engine.

Goal

To accomplish the requirements, the design goal of UMP alert engine listed below.

- **Dynamic stream consuming**
- **Conditional Window-based stream re-order**
    - As the time order of incoming stream is not guaranteed, while many alert cases require basic time-order (e.g. hdfs user command composition), so alert engine resort data sequence in time window based way ***when required*** before evaluation.
- **Metadata-Driven Event Routing:**
    - **Stream Balancing:** alert engine will route events only to the policy where they are expected, so that the events stream and policy could be linearly scaled.
    - **Dynamic GroupBy:** Dynamic partition of stream is key for scalability of platform service. Natively storm only support fixed grouping when a topology is defined, and group-by is defined only between instances of given bolt. To provide more flexibility group-by and support scale between bolts, alert engine supports dynamic group-by between bolts (through named stream).
        - A group-by of stream could be one field, or even composition of fields. Alert engine will generate the group-by key based on the requirement of alert policy.
    - **Stream Join:** Storm topology can't be changed during runtime, but as a platform, users may require to define policies to join/correlate different streams by ad-hoc, in such case , alert engine will support stream multiplex and join.
- **Alert Evaluation Engine**
    - Alert engine defines extensible evaluation interface and has Siddhi CEP as default implementation.
- **Metadata and Coordinator (Zookeeper + AS)**
    - **Resource Scheduler:** Cluster/Topology/Worker resource management/balance/optimization.
    - **Policy Lifecycle Management:** Policy CRUD Coordination.
- **Extensible Publish**

NOT in goal of this blue print:

- Pre-aggregator(pre-aggregator to generate specific variable to alert on) is an important part of alert engine. But in this blueprint, it's not detailed covered yet. To achieve MVP, this blue print doesn't treat pre-aggregator as first priority, and rely on CEP to provide most capability of aggregation.

Terms

- Stream Receiver (Spout)
- Stream Router (GroupBy Bolt)
- Policy Evaluator (Alert Bolt)
- Alert Publisher (Publish Bolt)
- Stream Source
- Stream
- Schema
- Policy
- Event
- Metadata

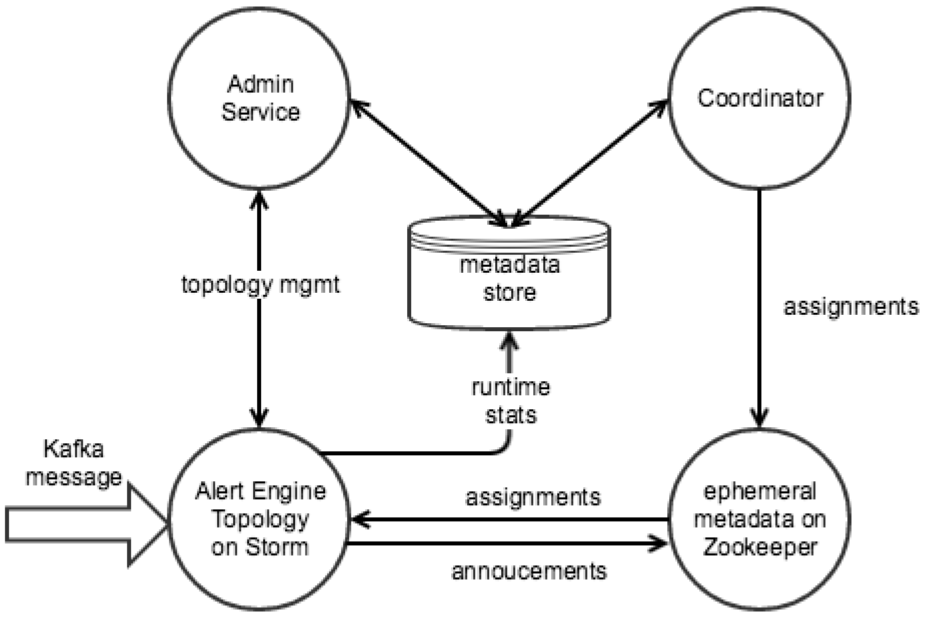
Proposed Design

High Level Diagram

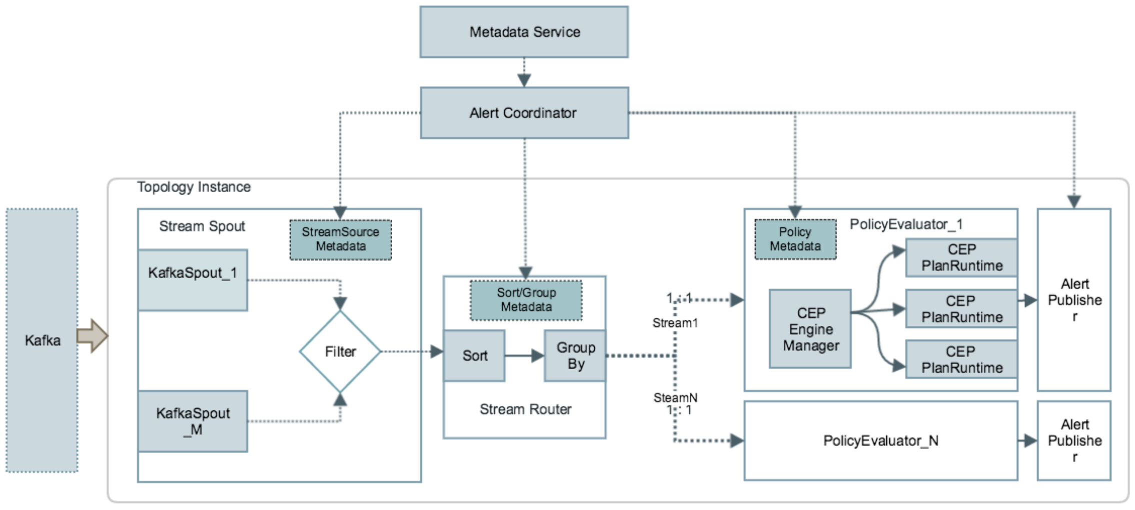
From high level view, alert engine is metadata-driven topologies runs on storm to do stream alerting. It includes multiple modules to work together

- Admin Service - Provide the API for metadata management & topology management. In this blue print, we define the API that will be used for alert engine, but admin service serves as wider purpose in whole monitoring platform.
    - Metadata store is a implementation detail of the admin service API.
- Alert Engine Topology on Storm - The storm topologies that runs the alert logic on top of storm. The storm topology will generic, not specific to any user created policy.
- Coordinator - The scheduler of the alert engine topology. It would be a backend scheduler that respond to new policy on-boarding, and

resource allocation, also it expose some internal API for administration.
- Zookeeper - used as communication between coordinator and alert engine.



Alert Engine Topology



Generic Alert Topology

Stream Receiver

A stream receiver(spout) is responsible to read from different topics on-demand driven by the topic metadata from Coordinator service.

One topology has one spout, one spout has N physical storm executors. So M topologies will have M*N physical storm executors. The M*N executors participate load balance of input stream processing.

Case 1: Semantic group-by key[1] is used all the way from source to CEP worker.

Coordinator can assign X executors out of M*N executors to participate input stream processing. Each of the X spout executors is equally taking the same amount partitions for one topic. If # of policies is very big, Coordinator should be able to replicate multiple the above assignment, with each assignment handling one portion of policies. In this case, the stream data is partitioned from the source[2].

Case 2: Randomized partition

Because of randomized partition, each topology should read all partitions for one topic because each group has data from all topic partitions. To achieve scalability, Coordinator should pick up M candidate topologies and assign X% of groups into each topology, so spout in this topology should be able to filter groups so that this topology should only consume part of input stream. In this case, the stream data is partitioned only when it comes into the spout(means the traffic are duplicated on network layer)[3].

If there are multiple group-by types, Coordinator should do the same thing for each group-by type. (groupby field1 and groupby field2 are 2 different group-by types)

If # of policies are very big one group-by types, Coordinator should replicate multiple assignments, with each assignment handling one portion of policies.

[1] This could done that user could declare a group-by key at the stream source.

[2][3] There are discussion on these part. This blueprints values both idea of randomize and semantic group by of source.

[3] This is done by the filter inside the spout before steam data sent to the router. Definitely this filter logic could also sit in group-by bolt, place in spout could help to reduce IO traffic, while this introduces the complexity for spout to identify different group of the same stream data.

Stream Router

The responsibility of group bolt is to :

- Sort the events come in with a time window if the alert policy is required time series data with ordering.
- Group the data, and send the data to different policy evaluator.

Policy Evaluator

The responsibility of policy evaluator is to get the data from upstream bolt, send to specify policy evaluation runtime(Siddhi plan runtime).

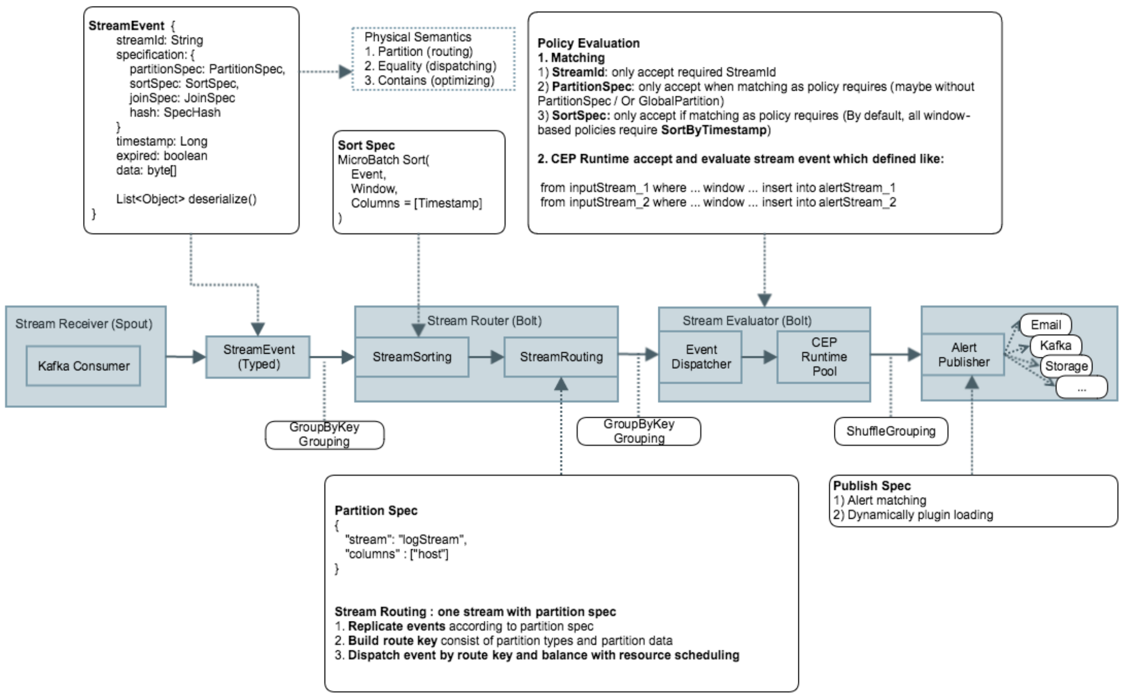
Each bolt will start one CEP engine while might have multiple policy runtime running.

Alert Publisher

The responsibility of publisher is to publish the output of policy evaluator.

- The publish target could be extensible, like in a kafka topic is an typical way for consumer like "Insight" module to consume, while emails could be important for service/operation team; SEC integration is also a must to have.
- There will be control metadata to control how to send the alert (like de-dup, suppress)

Alert Topology Detail Diagram



```
StreamEvent {
        streamId: String
        specification: {
            partitionSpec: PartitionSpec,
            sortSpec: SortSpec,
            joinSpec: JoinSpec
            hash: SpecHash
        }
        timestamp: Long
        expired: boolean
        data: byte[]

        List<Object> deserialize()
}
```

Sort Spec
```
MicroBatch Sort(
    Event,
    Window,
    Columns = [Timestamp]
)
```

Policy Evaluation
1. Matching
1) **StreamId**: only accept required StreamId
2) **PartitionSpec**: only accept when matching as policy requires (maybe without PartitionSpec / Or GlobalPartition)
3) **SortSpec:** only accept if matching as policy requires (By default, all window-based policies require **SortByTimestamp**)

2. CEP Runtime accept and evaluate stream event which defined like:

```
from inputStream_1 where ... window ... insert into alertStream_1
from inputStream_2 where ... window ... insert into alertStream_2
```

Partition Spec
```
{
  "stream": "logStream",
  "columns" : ["host"]
}
```

Stream Routing : one stream with partition spec
1. **Replicate events** according to partition spec
2. **Build route key** consist of partition types and partition data
3. **Dispatch event by route key and balance with resource scheduling**

Publish Spec
1) Alert matching
2) Dynamically plugin loading

Possible Challenges
The design option of support multiple group-by semantics on certain stream by duplicating the stream events, might introduce challenge of larger data amount pressure. To address the redundancy of traffic, there are couple of tuning options.

- **Combination Subset:** If some policy group by <C1,C2, ... ,Cn>, then we don't need to duplicate for those policies which group by combination in sub set of it.
- **Derived Combination:** No need to duplicate if one column can be derived from another column.
- **Group Similar Policies:** group the policies into same JVM (AlertBolt), which care about same streams or similar group-by combinations.
- **Compress Combination with Dictionary:** As the group-by combination are duplicated String, it will work well to reduce size with dictionary.
- **Streaming BloomFilter / FuzzyRowFilter:** In fact we only use the group-by combinations for partition key and check equality, so we don't even need to keep the original size. It's similar to algorithm like
    - https://en.wikipedia.org/wiki/Bloom_filter
    - https://en.wikipedia.org/wiki/Approximate_string_matching

Alert Metadata

The metadata should be stored and maintained by control layer(admin server etc.). See Alert metadata

Centralized Coordinator

The responsibility of coordinator is to decide the "given a policy, which cluster which topology and which logic bolt to run on", and "given a message/event, where should it be routed to". The route table of "Stream Spout"/"Stream Router"/"PolicyEvaluator" is calculated here, and spout/bolt would fetch the calculated route table, apply the route table dynamically. Zookeeper is used to do the coordination.

The coordinator should be able to provide coordination to help scale with the following factors.

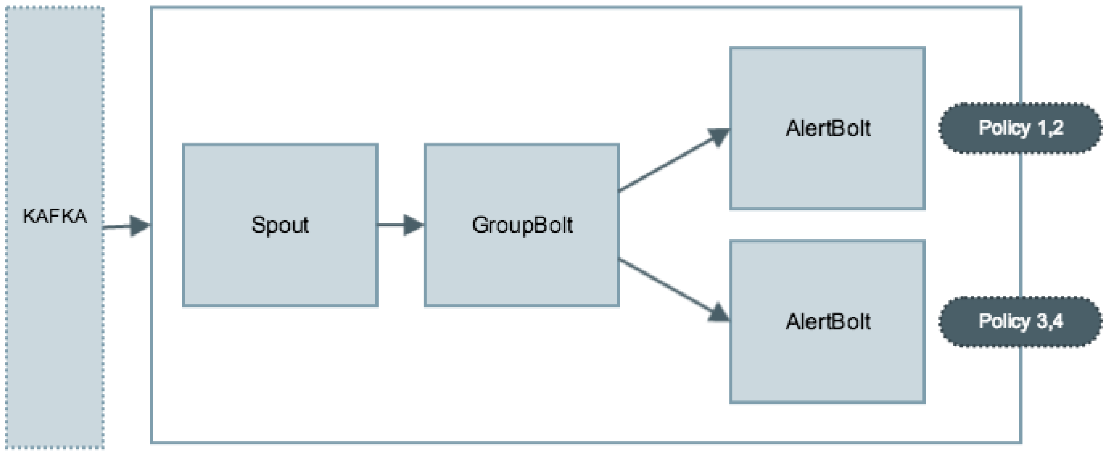
- Scale # of policies per input stream
- Scale with # of message per second for one input stream

Scale Pattern

Explain the scale pattern by case:

Note: the following policies are considered to have the same group-by semantics. Different group-by semantics may have very different

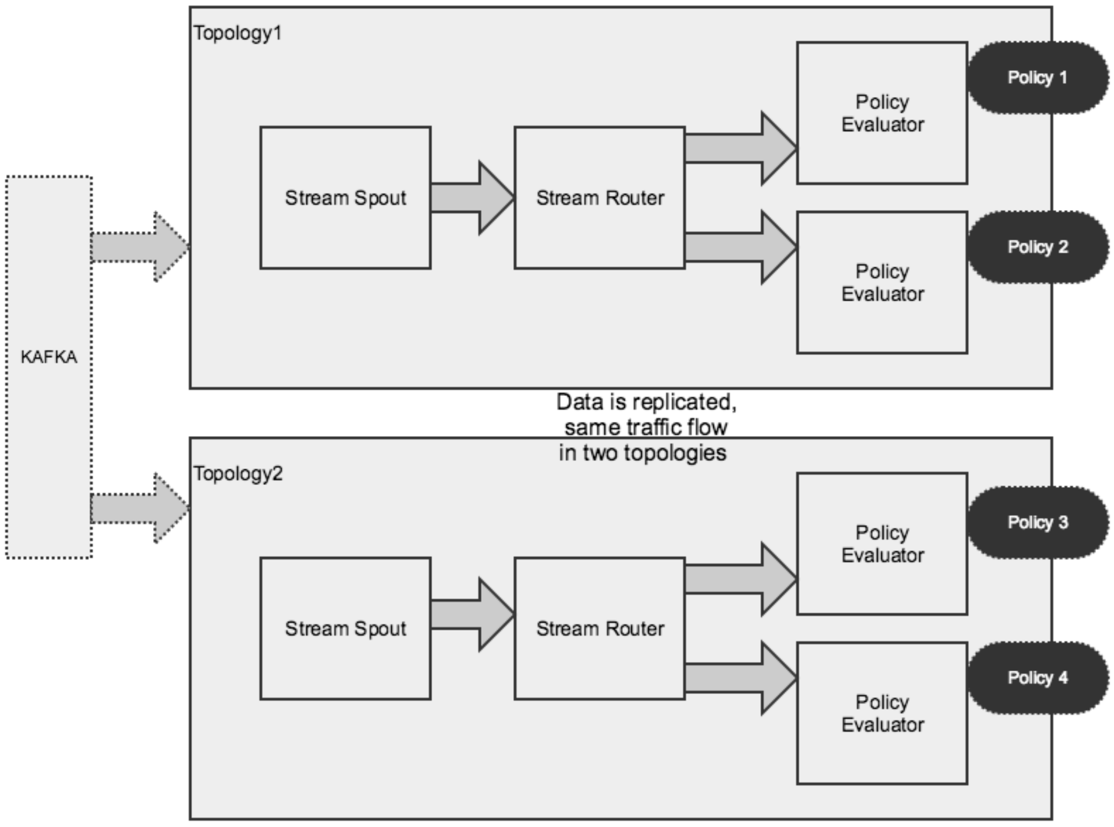
cardinality so scale is very different too.



Original Distribution (Naively all in one single topology). Here the policies 1,2,3,4 are the policies for same input stream.

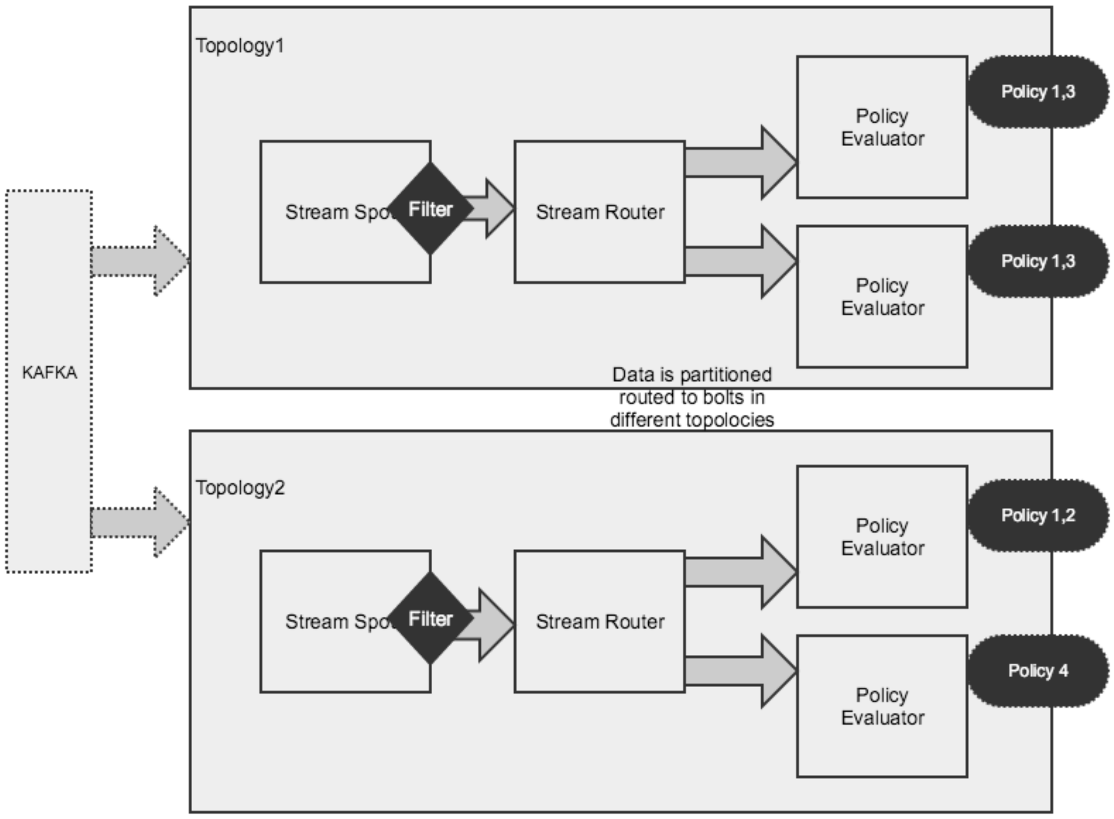
Scale # of policies per input stream

In this case, a given stream will have many consumer. By given fixed topology size and parallelism, one topology might not able to handle all the policies, while one single policy is ok to process its input data.



Scale # of policies, by distributing policy evaluators across different topologies. [1]

Scale # of message per second for one input stream



Scale # of message tuples per second (Traffic)

Say policy evaluator has fixed parallelism N.

Case 1: Cross bolt grouping in-side single topology

For schedule like policy3:

If the parallelism of policy evaluator not able to provide parallelism, then the policy is also scheduled to another policy evaluator in the same topology. This increased the parallelism to 2*N, and the stream router would have the grouped by between bolts. (Overcome the storm single bolt level grouping).

Case 2: Cross topology grouping

For schedule like policy1:

Furthermore than policy3, policy1 requires more parallelism, then it will also be scheduled on policy evaluator of another topology *with traffic also routed to that topology.* This increased the parallelism to 3*N (Overcome the fixed total parallelism of single topology).[2]

NOTE:

1. in this case, target policy 1 and policy 3 must be able to function correctly when they work on part of the partition, that's called shard-able policies.
2. The filter **NEED** to take the policy group information into consideration.

    In implementation, a route table that take the bolts of Topology1 and Topology2 into consideration will be built and push/pull to the stream spout/router. When stream router in topology1 find a message should be routed to topology2 it simply drop the message, since stream router in Topology2 should take care of that data.

[1][2] This kind of cross topology grouping comes with trade off: the traffic is duplicated from source to different topology, which might increase the load of the data source. So the coordinator would chose to expand inside topology first. And there are also tuning options as described in alert engine section.

Policy Runtime Behavior

Summary of observation of policy running of above scale pattern:

- One policy might run on multiple logic bolt inside an topology. (UMP alert engine control the concurrency of the policy)
    - A policy would need a parallelism that alert engine built for each policy (alert engine would have sys-level default value, and furthermore dynamic value to be more intelligent).
- One policy might run on different logic bolt cross topologies, for those topologies which are supposing to consume the same input. (#Scale of policy case)
- One policy might run on different logic bolt cross topologies, for those topologies which are supposing to consume different part of traffic (#Scale of single big traffic stream)
- Policies consume the same stream will be co-located scheduled as much as possible.

Coordinator is able to make bolts in different topology as workset for a given policy, and built route table to be applied by stream router.

Algorithm

The coordinator algorithm is a schedule algorithm(https://en.wikipedia.org/wiki/Bin_packing_problem).

The policies are grouped by the monitored stream(s) that it requires.

- A monitored stream is a stream with specific group by manner, here the monitored stream have a great description "Stream composition" at http://riemann.io/concepts.html.
- A monitored streams is served by a couple of bolts. This is called work slots. Ideally, work slots could be part of topology bolts, or all of given topology's bolts, or even cross topology bolts.

```
For each given policy, the greedy steps are
1. Find the same work slots that already serve the given data stream without exceed
the load
2. If no such work slot found, create a work slot in avaiable topology to served for
the given stream.
4. If no such topology available, Create a new topology and locate the policy
5. Generate policy assignments (polity -> work slot map)
5. Route table generated after all policies assigned
```

Zookeeper Directory

- /ump/alert/announcements
- /ump/alert/assignments

The topology nodes(bolts, spout) will listen to the ZK directory, reload the metadata when there are metadata/assignments change.

Topology Management

Topology Management is a functionality that could be reused between modules. SherlockConfig already provide storm config reading. This mgmt functionality need to be built based on existing sherlock-config support.

Topology mgmt would provide below features that required when coordinator try to scale out the computation.

- provide alert topology submit/stop capability.

This would be api that start the general alert topology use storm API. Don't require customized topology to be supported.

- detect topology start/stop event.

The service would periodically detect the topology state using storm interface (like Storm UI showing). And post the state to ZK path /ump/alert/announcements to notify the coordinator scheduling.

- provide statistics information for different topology.

Topology statistics will provide detail of policy running. Initially, we start from what we saw in Storm-UI, every bolt/spout would have "load" status.

Topology Mgmt APIs

HTTP Method	URL	Payload	Description
GET	/alert/topologies	N/A	List all topologies
POST	/alert/topologies	`{ "clusterName": "phx-storm1", "configTemplate": "middle-storm-template-id" }`	Spawn a new alert topology
DELETE	/alert/topologies/{topo-id}	N/A	Shutdown the given topology
GET	/alert/topologies/{topo-id}/load	N/A	Provide the statistics information for given topology.

Extensions

By default, UMP alert engine uses CEP engine (WSO2 Siddhi) as first class citizen for policy evaluator. CEP engine can handle a lot of use cases in that it naturallys support filtering or window based aggregation against stream data. But it is also possible to extend UMP alert engine to support more advanced use cases.

Thanks to UMP alert engine's capability, when we implement a new policy evaluator, we don't need worry about how data schema is defined, how data is partitioned and how multi-tenant data is handled etc. We only need implement PolicyStreamHandler interface and handle each incoming event. It is policy evaluator's responsibility to manage its own state if applicable.

The possible extensions of policy evaluator are no data alert, absence alert, percentile alert, and even machine learning based policy evaluator.

No data alert

Given N entities which continuously send out events, if M entities (M << N) don't send out events for a specific time, e.g. 1 hour, then alert.

The proposed solution is:

- Maintain time window of WINDOW_PERIOD, e.g. 1 hour
- Upon each incoming event, slide the window if some events in this window expire
- Compare distinct entities in current window (WIRI) with distinct entities which should be (WISB), if some entities exist in WISB but not in WIRI, then alert
- WISB can be provided or dynamic. For provided WISB, user should specify full set of entities. For dynamic WISB, user does not need specify full set of entities. Alert engine will use last WIRI as WISB

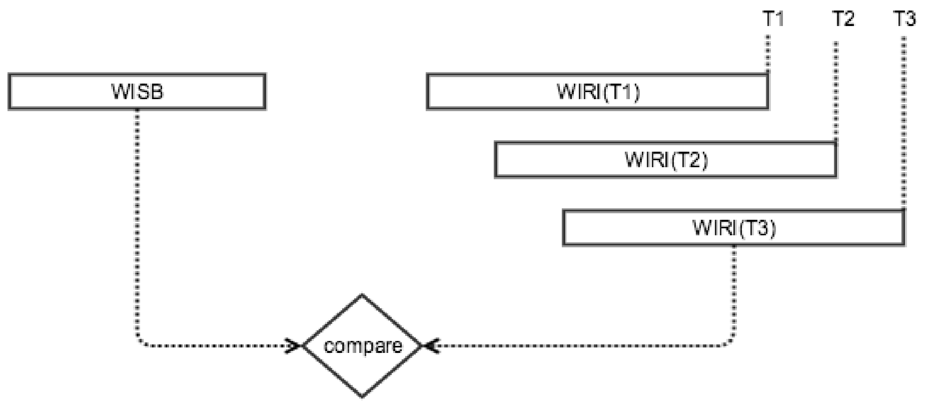
Note:

1) The assumption is that data always come in frequently to trigger set timestamp for time slide window. (To remove this assumption, we may need a separate background thread to trigger timestamp)

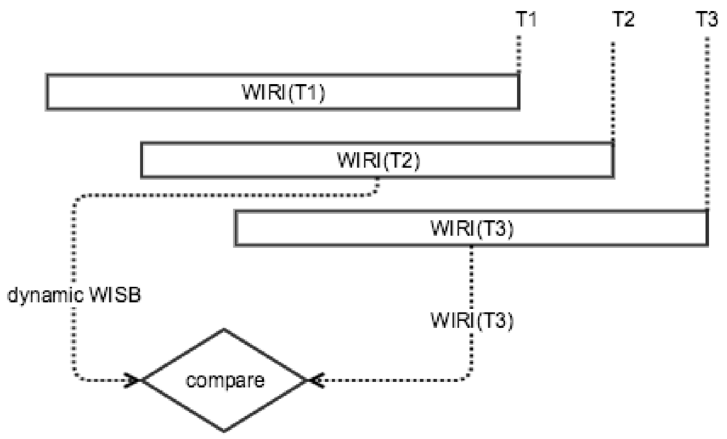
2) this may not be working for the use case where the whole topic does not have any data. As the above solution needs incoming event to trigger window sliding.

3) how about one node is in SA check status

Provided WISB



Dynamic WISB



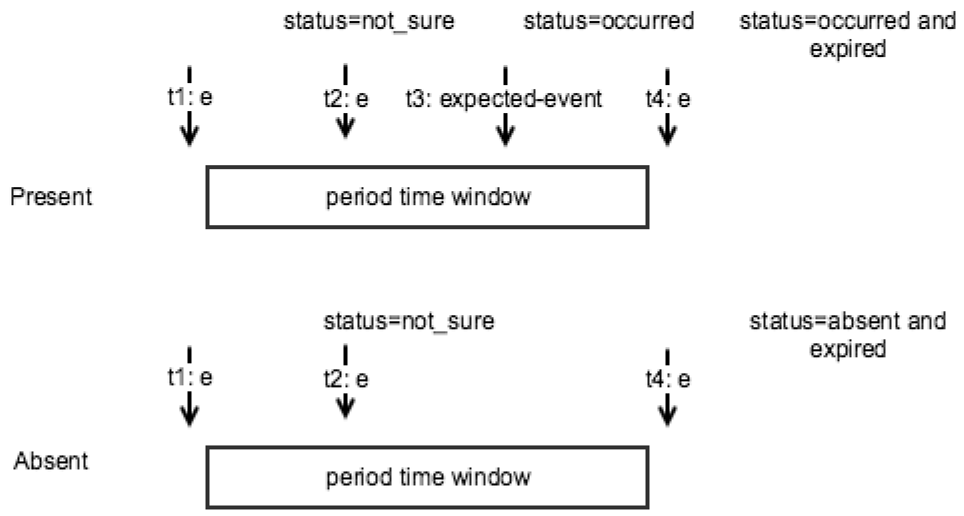
Absence Alert

Something must happen within a specific time range within a day. For example one type of job must be running from 1PM - 2PM every day.

This type of alert is similar to but different from No Data Alert in that there is fixed time window in Absence Alert while there is slide time window in No Data Alert. So space complexity for Absence Alert is O(1) while space complexity for No Data Alert is O(window), window is size of slide time window.

Note:

1) The assumption is that data always come in frequently to trigger set timestamp for time window.



SLA alert

Some task must be completed within SLA. For example when user submits a spark job, SLA can be specified for that job, e.g. 1 hour SLA.

This type of alert does not need customized implementation as the task's status can be fed into UMP alert engine periodically, we can easily use Siddhi CEP engine to check whether the task is completed at specified time.

Alert State

See sub-page Alert state and transition

Alert De-duplication/Aggregation

Alert de-duplication/aggregation will provide a topology graph based on alert de-duplication. This is be part of alert analysis wise, another topology will be used to consume the output of alert engine, and combine with the dependency graph Dependency Service REST API.

Details TBD