|  |
| --- |
| Oleksiy Dyagilev  7-29-2014 |

|  |
| --- |
| EPAM Systems |
| GigaSpaces XAP and Storm integration |
|  |

| **Rev** | **Date** | **By Whom** | **What** |
| --- | --- | --- | --- |
| 1 | 2014-08-7 | Oleksiy Dyagilev | Initial version |
| 2 | 2014-08-7 | Oleksiy Dyagilev | Feeder params changed |
|  |  |  |  |

# Revision history

Table of Contents

[Revision history 1](#_Toc395176775)

[Introduction 3](#_Toc395176776)

[Storm in a Nutshell 3](#_Toc395176777)

[Spouts 4](#_Toc395176778)

[Storm Spout 4](#_Toc395176779)

[Trident Spout 6](#_Toc395176780)

[Trident State 9](#_Toc395176781)

[Trident Read-Only state 10](#_Toc395176782)

[Storm bolts 11](#_Toc395176783)

[Illustrative example: Real-time Google Analytics 11](#_Toc395176784)

[High-level architecture diagram 12](#_Toc395176785)

[Google Analytics Topology. High level overview. 13](#_Toc395176786)

[Top urls topology branch 13](#_Toc395176787)

[Active users topology branch 14](#_Toc395176788)

[Page view time series topology branch 15](#_Toc395176789)

[Geo topology branch 16](#_Toc395176790)

[Building the Application 17](#_Toc395176791)

[Deploying in development environment 17](#_Toc395176792)

[Deploying in development environment with embedded Storm 18](#_Toc395176793)

[One-click deployment to cloud 18](#_Toc395176794)

[Monitoring XAP and Storm 18](#_Toc395176795)

# Introduction

Real-time processing is becoming very popular, and Storm is a popular open source framework and runtime used by Twitter for processing real-time data streams.  Storm addresses the complexity of running real time streams through a compute cluster by providing an elegant set of abstractions that make it easier to reason about your problem domain by letting you focus on data flows rather than on implementation details.

Storm has many use cases: realtime analytics, online machine learning, continuous computation, distributed RPC, ETL, and more. Storm is fast: a benchmark clocked it at over a million tuples processed per second per node. It is scalable, fault-tolerant, guarantees your data will be processed, and is easy to set up and operate.

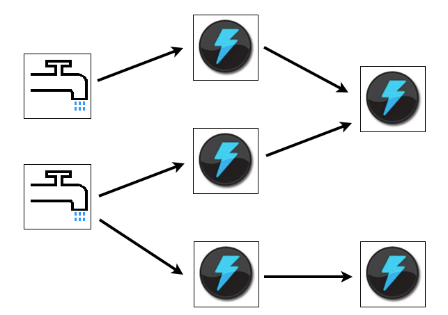
This pattern integrates XAP with Storm. XAP is used as stream data source and fast reliable persistent storage, whereas Storm is in charge of data processing. We support both pure Storm and Trident framework.

As part of this integration we provide classic Word Counter and Twitter Reach implementations on top of XAP and Trident.

Also, we demonstrate how to build highly available, scalable equivalent of Realtime Google Analytics application with XAP and Storm. Application can be deployed to cloud with one click using Cloudify.

# Storm in a Nutshell

Storm is a real time, open source data streaming framework that functions entirely in memory.  It constructs a processing graph that feeds data from an input source through processing nodes.  The processing graph is called a "topology".  The input data sources are called "spouts", and the processing nodes are called "bolts".  The data model consists of tuples.  Tuples flow from Spouts to the bolts, which execute user code. Besides simply being locations where data is transformed or accumulated, bolts can also join streams and branch streams.  Storm is designed to be run on several machines to provide parallelism.  Storm topologies are deployed in a manner somewhat similar to a webapp or a XAP processing unit; a jar file is presented to a deployer which distributes it around the cluster where it is loaded and executed.  A topology runs until it is killed.



Beside Storm, there is a Trident – a high-level abstraction for doing realtime computing on top of Storm. Trident adds primitives like groupBy, filter, merge, aggregation to simplify common computation routines. Trident has consistent, exactly-once semantics, so it is easy to reason about Trident topologies.

Capability to guarantee exactly-once semantics comes with additional cost. To guarantee that, incremental processing should be done on top of persistence data source. Trident has to ensure that all updates are idempotent. Usually that leads to lower throughput and higher latency than similar topology with pure Storm.

# Spouts

Basically, Spouts provide the source of tuples for Storm processing.  For spouts to be maximally performant and reliable, they need to provide tuples in batches, and be able to replay failed batches when necessary.  Of course, in order to have batches, you need storage, and to be able to replay batches, you need reliable storage.  XAP is about the highest performing, reliable source of data out there, so a spout that serves tuples from XAP is a natural combination.

*data stream*

*stream producer*

*spout*

*stream processing*

*Storm*

*XAP datagrid*

Depending on domain model and level of guarantees you want to provide, you choose either pure Storm or Trident. We provide Spout implementations for both – XAPSimpleSpout and XAPTranscationalTridentSpout respectively.

## Storm Spout

XAPSimpleSpout is a spout implementation for pure Storm that reads data in batches from XAP. On XAP side we introduce conception of stream. Please find SimpleStream – a stream implementation that supports writing data in single and batch modes and reading in batch mode. SimpleStream leverages XAP’s FIFO(First In, First Out) capabilities.

*SimpleStream.java, FIFO support*

*stream producer*

*spout*

*write() and writeBatch()*

*readBatch()*

*XAPSimpleSpout.java*

SimpleStream works with arbitrary space class that has FifoSupport.OPERATION annotation and implements Serializable.

Here is an example how one may write data to SimpleStream and process it in Storm topology. Let’s consider we would like to build an application to analyze the stream of page views (user clicks) on website. At first, we create a data model that represents a page view

@SpaceClass(fifoSupport = FifoSupport.OPERATION)

public class PageView implements Serializable {

private String id;

private String page;

private String sessionId;

[getters setters omitted for brevity]

}

Now we would like to create a reference to stream instance and write some data.

SimpleStream<PageView> stream = new SimpleStream<>(space, new PageView());

stream.writeBatch(pageViews);

The second argument of SimpleStream is a template used to match objects during reading.

If you want to have several streams with the same type, template objects should differentiate your streams.

Now let’s create a spout for PageView stream.

public class PageViewSpout extends XAPSimpleSpout<PageView> {

public PageViewSpout() {

super(new PageViewTupleConverter(), new PageView());

}

}

To create a spout, we have to specify how we want our space class be converted to Storm tuple. That is exactly what TupleConverter knows about.

class PageViewTupleConverter implements TupleConverter<PageView> {

@Override

public Fields tupleFields() {

return new Fields("page", "session");

}

@Override

public List<Object> spaceObjectToTuple(PageView pageView) {

return Arrays.<Object>asList(pageView.getPage(), pageView.getSessionId());

}

}

At this point we have everything ready to build Storm topology with PageViewSpout.

Config conf = new Config();

conf.put(ConfigConstants.XAP\_SPACE\_URL\_KEY, "jini://\*/\*/space");

conf.put(ConfigConstants. XAP\_STREAM\_BATCH\_SIZE, 300);

TopologyBuilder builder = new TopologyBuilder();

builder.setSpout(“pageViewSpout”, new PageViewSpout());

ConfigConstants.XAP\_SPACE\_URL\_KEY is a space URL

ConfigConstants. XAP\_STREAM\_BATCH\_SIZE is a maximum number of items that spout reads from XAP with one hit.

## Trident Spout

XAPTranscationalTridentSpout is a scalable, fault-tolerant, transactional spout for Trident, supports pipelining. Let’s discuss all its properties in details.

For spout to be maximally performant, we want an ability to scale the number of instances to control the parallelism of reader threads.

There are several spout APIs available that we could potentially use for our XAPTranscationalTridentSpout implementation:

* [IPartitionedTridentSpout](https://github.com/apache/incubator-storm/blob/master/storm-core/src/jvm/storm/trident/spout/IPartitionedTridentSpout.java): A transactional spout that reads from a partitioned data source. The problem with this API is that it doesn’t acknowledge when batch is successfully processed which is critical for in memory solutions since we want to remove items from the grid as soon as they have been processed. Another option would be to use XAP’s lease capability to remove items by time out. This might be unsafe, if we keep items too long, we might consume all available memory.
* [ITridentSpout](https://github.com/apache/incubator-storm/blob/master/storm-core/src/jvm/storm/trident/spout/ITridentSpout.java): The most general API. Setting parallelism hint for this spout to N will create N spout instances, single coordinator and N emitters. When coordinator issues new transaction id, it passes this id to all emitters. Emitter reads its portion of transaction by given transaction id. Merged data from all emitters forms transaction.

For our implementation we choose [ITridentSpout](https://github.com/apache/incubator-storm/blob/master/storm-core/src/jvm/storm/trident/spout/ITridentSpout.java) API.

*stream producer*

*emitter*

*emitter*

*emitter*

*space partitions*

*XAPTransactionalTridentSpout*

*every emitter reads its portion*

*merged data forms transaction*

*PartitionedStream*

There is one to one mapping between XAP partitions and emitters.

Storm framework guarantees that topology is high available, if some component fails, it restarts it. That means our spout implementation should be stateless or able to recover its state after failure.

When emitter is created, it calls remote service ConsumerRegistryService to register itself. ConsumerRegistryService knows the number of XAP partitions and keeps track of the last allocated partition. This information is reliably stored in the space, see ConsumerRegistry.java.

5

2

1

3

4

*ConsumerRegistry, number of partitions - 5*

*last allocated partition*

Remember that parallelism hint for XAPTranscationalTridentSpout should equal to the number of XAP partitions.

The property of being transactional is defined in Trident as following:

* batches for a given txid are always the same. Replays of batches for a txid will exact same set of tuples as the first time that batch was emitted for that txid.
* there’s no overlap between batches of tuples (tuples are in one batch or another, never multiple).
* every tuple is in a batch (no tuples are skipped)

XAPTranscationalTridentSpout works with PartitionedStream that wraps stream elements into Item class and keeps items ordered by ‘offset’ property. There is one PartitionStream instance per XAP partition.

1

2

3

4

5

6

*WriterHead {offset: 6}*

*ReaderHead {offset: 5}*

*BatchMetadata{batchId:2, offset:2, count:3}*

*Item {offset: 1}*

Stream’s WriterHead holds the last offset in the stream. Any time batch of elements (or single element) written to stream, WriterHead incremented by the number of elements. Allocated numbers used to populate offset property of Items. WriterHead object is kept in heap, there is no need to keep it in space. If primary partition fails, WriterHead is reinitialized to be the max offset value for given stream.

ReaderHead points to the last read item. We have to keep this value in the space, otherwise if partition fails we won’t be able to infer this value.

When spout request new batch, we take ReaderHead, read data from that point and update ReaderHead. New BatchMetadata object is placed to the space, it keeps start offset and number of items in the batch. In case Storm requests transaction replaying, we are able to reread exactly the same items by given batchId. Finally, once Storm acknowledges that batch successfully processed, we delete BatchMetadata and corresponding items from the space.

By default, Trident processes a single batch at a time, waiting for the batch to succeed or fail before trying another batch. We can get significantly higher throughput and lower latency of processing of each batch – by pipelining the batches. You configure the maximum amount of batches to be processed simultaneously with the “topology.max.spout.pending” property.

Operations with PartitionedStream are encapsulated in remote service – PartitionedStreamService.

Here is an example how to use XAPTransactionalTridentSpout:

Config conf = new Config();

conf.setMaxSpoutPending(20);

conf.put(ConfigConstants.XAP\_SPACE\_URL\_KEY, "jini://\*/\*/space?groups=XAPTransactionalTridentSpoutTest-test");

TridentTopology topology = new TridentTopology();

TridentState wordCounts = topology

.newStream("spout1", spout)

.each(new Fields("sentence"), new Split(), new Fields("word"))

.groupBy(new Fields("word"))

.persistentAggregate(stateFactory, new Count(), new Fields("count")).parallelismHint(16);

The full example that demonstrates usage of XAPTransactionalTridentSpout to address classic Word Counter problem can be found in XAPTransactionalTridentSpoutTest.

# Trident State

Trident has first-class abstractions for reading from and writing to stateful sources. Details are available on the Storm wiki [site](https://storm.incubator.apache.org/documentation/Trident-state).

In Trident topology that is persisting state via this mechanism, the overall throughput is almost certainly constrained by the performance of the state persistence.  This is a good place where XAP can step in and provide extremely high performance persistence for stream processing state.

*spout*

*stream processing*

*Trident*

*XAP datagrid*

*Trident State, XAPState.java*

*Readonly Trident State*

*XAPReadOnlyState.java*

XAP Trident state implementation supports all state types – non-transactional, transactional and opaque. All you need to create a Trident state is configure space url and choose appropriate factory method of XAPStateFactory class:

Config conf = new Config();

conf.setMaxSpoutPending(20);

conf.put(ConfigConstants.XAP\_SPACE\_URL\_KEY, "jini://\*/\*/space ");

StateFactory stateFactory = XAPStateFactory.transactional();

TridentState wordCounts = topology

.newStream("spout1", spout).parallelismHint(16)

.each(new Fields("sentence"), new Split(), new Fields("word"))

.groupBy(new Fields("word"))

.persistentAggregate(stateFactory, new Count(), new Fields("count")).parallelismHint(16);

The full example can be found in TridentWordCountTest.

## Trident Read-Only state

Trident Read-Only state allows to lookup persistent data during the computation.

Consider [Twitter Reach example](https://storm.incubator.apache.org/documentation/Trident-tutorial.html). Reach is the number of unique people exposed to a URL on Twitter. To compute reach, you need to fetch all the people who ever tweeted a URL, fetch all the followers of all those people, unique that set of followers, and that count that uniqued set.

XAP is a good candidate to store reference data such as tweeted url and followers. You can easily create XAP read-only state with XAPReadOnlyStateFactory. The following example demonstrates how to create a read-only state for TweeterUrl and Followers classes. The input arguments that Trident pass to stateQuery() are used as space ids.

The full example can be found in TridentReachTest.

TridentState tweetedUrls = topology.newStaticState(XAPReadOnlyStateFactory.byIds(TweetedUrl.class));

TridentState followers = topology.newStaticState(XAPReadOnlyStateFactory.byIds(Followers.class));

topology.newDRPCStream("reach", drpc)

.stateQuery(tweetedUrls, new Fields("args"), new MapGet(), new Fields("tweetedUrls"))

.each(new Fields("tweetedUrls"), new ExpandTweetersList(), new Fields("tweeter"))

.shuffle()

.stateQuery(followers, new Fields("tweeter"), new MapGet(), new Fields("followers"))

.each(new Fields("followers"), new ExpandFollowersList(), new Fields("follower"))

.groupBy(new Fields("follower"))

.aggregate(new One(), new Fields("one"))

.aggregate(new Fields("one"), new Sum(), new Fields("reach"));

Another option to create XAP read-only state is to use SQL query. In this case stateQuery’s input arguments are used as SQL parameters:

SQLQuery<Person> sqlQuery = new SQLQuery<>(Person.class, "name = ? AND age > 30").setProjections("age");

TridentState state = topology.newStaticState(XAPReadOnlyStateFactory.bySqlQuery(sqlQuery));

The full example can be found in SqlQueryReadOnlyStateTest.

# Storm bolts

If pure Storm suits better your needs, most likely you will want to read/write data from bolts to persistent storage. For instance, imagine you are processing stream of data and would like to present computation result on UI. So the final bolt in your topology pipeline should write result to XAP which can then be accessed from anywhere. For this purpose we created XAPAwareRichBolt and XAPAwareBasicBolt that have a reference to space proxy. All you need is to configure space url and extend XAP aware bolt.

Example:

public class TotalActiveUsersBolt extends XAPAwareBasicBolt {

public void execute(Tuple tuple, BasicOutputCollector collector) {

ActiveUsersReport report = … ; // compute

space.write(report);

}

….

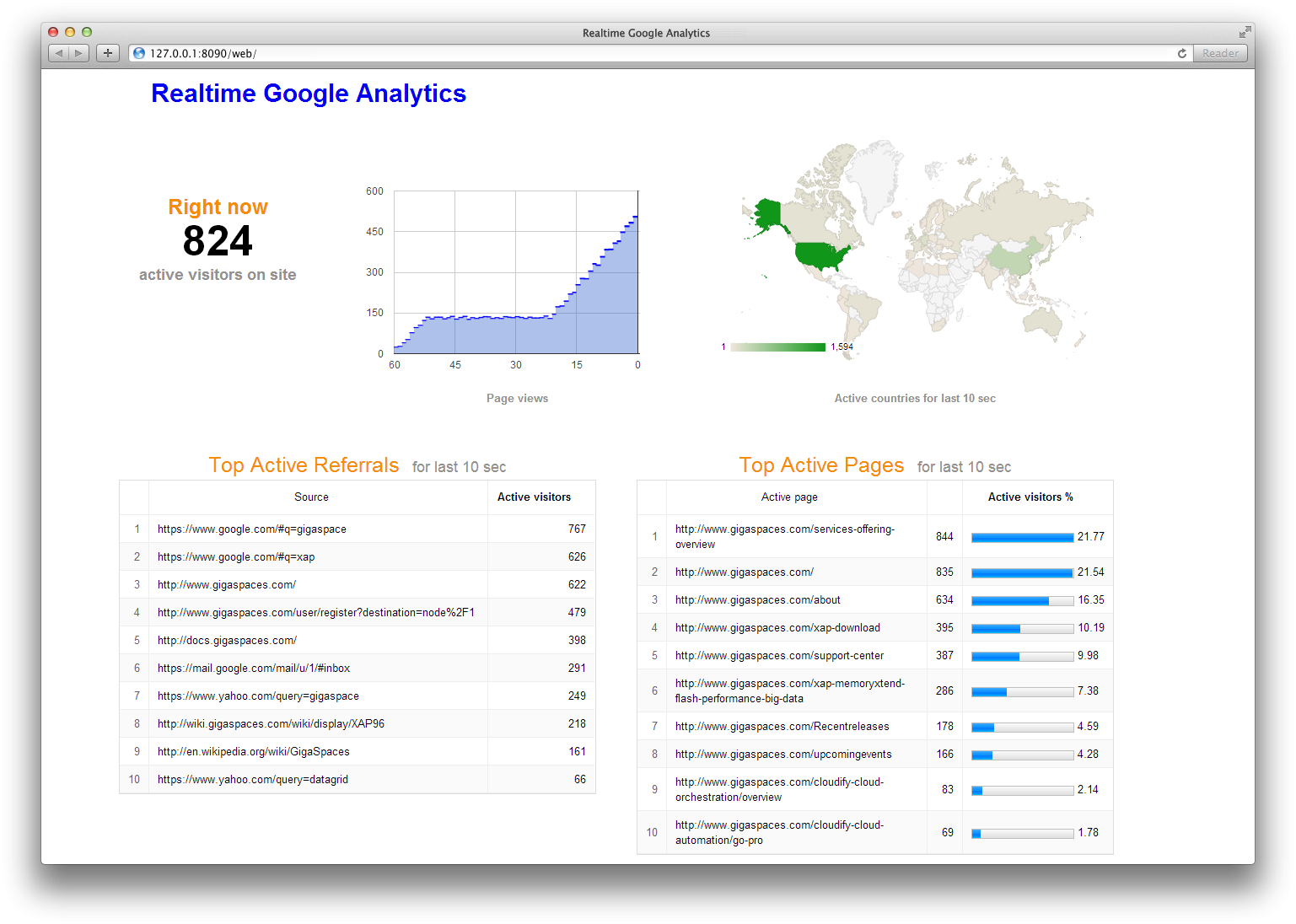
}

# Illustrative example: Real-time Google Analytics

In this section we demonstrate how to build highly available, scalable equivalent of Real-time Google Analytics application and deploy it to cloud with one click using Cloudify.

Real-Time Google Analytics allows you to monitor activity as it happens on your site. The reports are updated continuously and each page view is reported seconds after it occurs on your site. For example, you can see:

* how many people are on your site right now
* dynamic of page views during last minute
* users geographic locations
* traffic sources that referred them
* which pages or events they're interacting with



## High-level architecture diagram

*PageView stream*

*PageView feeder*

*spout*

*stream processing*

*Storm*

*XAP space*

*reports*

*Browser*

*Web PU*

*Web PU*

*Web PU*

PageView feeder is a standalone java application that simulates users on the site. It continuously sends PageView json to rest service endpoints deployed in XAP web PU. PageView looks like this

{

“sessionId”: “sessionid581239234”,

“referral”: “https://www.google.com/#q=gigaspace”,

“page”: “http://www.gigaspaces.com/about”,

“ip”: “89.162.139.2”

}

Rest service converts JSON documents to space object and writes them to the stream. Stream is consumed by Storm topology which performs all necessary processing in memory and stores results in XAP space. End user is browsing web page hosted in Web PU that continuously updates reports with AJAX requests backed by another rest service. Rest service reads report from XAP space.

We use pure Storm to build topology. There are several reasons why we don’t use Trident for this application. We are tolerant to page views loss if some Storm node fails. We don’t need exactly-once processing semantic. Instead, we want to maximize throughput and minimize latency.

## Google Analytics Topology. High level overview.

*PageView spout*

*PageView spout*

*PageView spout*

*active users branch*

*page view time series branch*

*top referrals branch*

*top urls branch*

*geo branch*

*XAP* *Space*

PageView spout forks five branches, each branch calculates its report and can be scaled independently. The final bolt in the branch writes data to XAP space. In the next sections we take a closer look at branches design.

## Top urls topology branch

Top urls report displays top 10 visited urls for the last ten seconds. Topology implements distributed rolling count algorithm. The report is updated every second.

*UrlRollingCountBolt*

*spout*

*UrlRollingCountBolt*

*UrlRollingCountBolt*

*IntermediateRankingsBolt*

*IntermediateRankingsBolt*

*TotalUrlRankingsBolt*

*spout*

*group by (url)*

*rolling count*

*sliding window 10 sec*

*(url, count), group by url*

*emitted every 2 sec*

*maintains sorted list [(url, count)]*

*local top 10 of [(url, count)]*

*emitted every 1 sec*

*calculate total top 10*

*write report to XAP*

8

11

5

2

9

*Cyclic buffer per url*

*5 slots = 10 sec sliding window / emitted every 2 sec*

*head advances every 2 sec*

Tuples flow from spout to UrlRollingCountBolt grouped by ‘url’. UrlRollingCountBolt calculates rolling count with sliding windows of 10 seconds for every url. Sliding windows is basically a cyclic buffer with a head pointing to current slot. When bolt receives new tuple, it finds a sliding window for this tuple and increments the number in current slot. Every two seconds UrlRollingCountBolt emits the sum of sliding window for every url, then sliding windows advance and head points to the next slot.

The url and its rolling count flow to IntermediateRankingsBolt which maintains pair of (url, count) in sorted by count order and emits its top 10 urls to the final stage. TotalUrlRankingBolt calculates the global top 10 urls and writes report object to XAP space. The primitives to implement rolling count algorithm can be found in [storm-starter](https://github.com/apache/incubator-storm/tree/master/examples/storm-starter) project.

Top referrals topology branch is identical to top urls one. The only difference in is that we calculate ‘referral’ rather than ‘url’ tuple field.

## Active users topology branch

Active users report displays how many people on the site right now. We assume that if user hasn’t opened any page for the last N seconds, then user has left the site. Users are uniquely identified by ‘sessionId’ tuple field. For demo purpose N is configured to 5 seconds, though it should be much longer in real life application.

*PartitionedActiveUsersBolt*

*spout*

*PartitionedActiveUsersBolt*

*PartitionedActiveUsersBolt*

*TotalActiveUsersBolt*

*spout*

*group by (session)*

*maintains Map[session, lastSeenTime]*

*emits map size*

*local users count*

*emitted every 1 sec*

*maintains Map[source\_task, count]*

*calculate total users count*

*write report to XAP*

Tuples flow from spout to PartitionedActiveUsersBolt grouped by ‘sessionId’. For every sessionId PartitionedActiveUsersBolt keeps track of the last seen time. Every second it removes sessions seen last time earlier than N seconds before and then emits the number of remaining ones.

TotalActiveUsersBolt maintains a map of [source\_task, count] and emits the total count for all sources. Report is written to XAP.

## Page view time series topology branch

Page view time series report displays the dynamic of visited pages for last minute. The chart is updated every second.

*PageViewCountBolt*

*spout*

*PageViewCountBolt*

*PageViewCountBolt*

*PageViewTimeSeriesBolt*

*spout*

*shuffle*

*simple counter*

*local count*

*emitted every 1 sec*

*sliding window counter*

*write report to XAP*

PageViewCountBolt calculates the number of page views and passes local count to PageViewTimeSeriesBolt every second. PageViewTimeSeriesBolt maintains a sliding window counter and writes report to XAP space.

## Geo topology branch

Geo report displays a map of users’ geographical location. Depending on the volume of traffic from particular country, country is filled with different colors on the map.

*GeoIpBolt*

*spout*

*GeoIpBolt*

*GeoIpBolt*

*CountryRollingCountBolt*

*spout*

*shuffle*

*convert IP address to country*

*country*

*sliding window counter*

*emits every 1 sec*

*write report to XAP*

IP address converted to country using [MaxMind GeoIP database](http://dev.maxmind.com/). The database is a binary file loaded into GeoIPBolt’s heap. GeoIpLookupService ensures that it’s loaded only once per JVM.

## Building the Application

1. [Download](http://www.gigaspaces.com/LatestProductVersion) and [install](http://wiki.gigaspaces.com/wiki/display/XAP95/Installing+GigaSpaces) XAP
2. Install Maven and the GigaSpaces Maven plug-in
3. The application source can be found under INSERT\_PATH\_HERE
4. Build the project by running

mvn clean install

## Deploying in development environment

1. Follow this [documentation](https://storm.incubator.apache.org/documentation/Setting-up-a-Storm-cluster.html) to install and run Zookeeper, Nimbus, Supervisor and optionally Storm UI.
2. Start a [Grid Service Agent](http://wiki.gigaspaces.com/wiki/display/XAP95/The+Grid+Service+Agent) by running the gs-agent.sh/bat script
3. Deploy space and Web PU by running the following from project root folder:

cd google-analytics

mvn os:deploy

1. Add apache-storm-0.9.2-incubating/bin to your $PATH
2. Run the following to deploy topology to Storm cluster

storm jar ./storm-topology/target/storm-topology-1.0-SNAPSHOT.jar com.gigaspaces.storm.googleanalytics.topology.GoogleAnalyticsTopology google-analytics 127.0.0.1

1. Run feeder

java -jar ./feeder/target/feeder-1.0-SNAPSHOT.jar 127.0.0.1

1. Open browser <http://localhost:8090/web/> to view Google Analytics UI
2. To undeploy topology run

storm kill google-analytics

## Deploying in development environment with embedded Storm

1. To run topology in embedded Storm you don’t need to install Zookeeper and Storm. Follow all steps from previous section except deployment to Strom.
2. Open google-analytics/storm-topology/pom.xml and change scope of storm-core artifact from ‘provided’ to ‘compile’.
3. Rebuild the project
4. Run storm topology

java -jar ./storm-topology/target/storm-topology-1.0-SNAPSHOT.jar

Alternatively you can GoogleAnalyticsTopology from your IDE.

## One-click deployment to cloud

## Monitoring XAP and Storm