[[1]](#footnote-1)

Druid: NoSQL based Analytics on Real-Time Data

Ujwal Sadhu, Prof. Jeongkyu Lee, Department of Computer Science, University of Bridgeport, Bridgeport, CT

*Abstract*— Druid is an open source, distributed analytics data store that powers user-facing data applications, provides fast queries on data in Hadoop, and helps you glean insights from streaming data. Real-Time data is the data that is being generated in the current moment. The data generated by the Wikipedia edits or the Twitter tweets or the data from business can be used here.

*Index Terms*—Druid, NoSQL, Analytics, Real-Time data

# INTRODUCTION

D

ATA Analytics (DA) is the science of examining raw with the purpose of drawing conclusions about that information. This study is used in many industries to allow companies and organization to make better business decisions and in the sciences to verify or disprove existing models or theories.

The combination of DA with the power of NoSQL have been implemented since the beginning as the factors to be considered are numerous for a relational database. Here we are using a NoSQL database called Druid.

Druid is an open source data store designed for real-time exploratory analytics on large data sets. The system combines a column-oriented storage layout, a distributed, shared-nothing architecture, and an advanced indexing structure to allow for the arbitrary exploration of billion-row tables with sub-second latencies. At a high-level, Druid collects event data into segments via real-time nodes.  The real-time nodes push those segments into deep storage.  Then a master node distributes those segments to compute nodes, which are capable of servicing queries.  A broker node sits in front of everything and distributes queries to the right compute nodes.

Druid is a system built to allow fast ("real-time") access to large sets of seldom-changing data. It was designed with the intent of being a service and maintaining 100% uptime in the face of code deployments, machine failures and other eventualities of a production system. It can be useful for back-office use cases as well, but design decisions were made explicitly targeting an always-up service.

# Architecture Of Druid

Druid is architected as a grouping of systems each with a distinct role and together they form a working system. The name comes from the Druid class in many role-playing games: it is a shape-shifter, capable of taking many different forms to fulfill various different roles in a group.

Each of the systems, or components, described below also has a dedicated page with more details. You can find the page in the menu on the right, or click the link in the component's description.

The node types that currently exist are:

1. Historical nodes are the workhorses that handle storage and querying on "historical" data (non-realtime). Historical nodes download segments from deep storage, respond to the queries from broker nodes about these segments, and return results to the broker nodes. They announce themselves and the segments they are serving in Zookeeper, and also use Zookeeper to monitor for signals to load or drop new segments.
2. Coordinator nodes monitor the grouping of historical nodes to ensure that data is available, replicated and in a generally "optimal" configuration. They do this by reading segment metadata information from metadata storage to determine what segments should be loaded in the cluster, using Zookeeper to determine what Historical nodes exist, and creating Zookeeper entries to tell Historical nodes to load and drop new segments.
3. Broker nodes receive queries from external clients and forward those queries to Realtime and Historical nodes. When Broker nodes receive results, they merge these results and return them to the caller. For knowing topology, Broker nodes use Zookeeper to determine what Realtime and Historical nodes exist.
4. Indexing Service nodes form a cluster of workers to load batch and real-time data into the system as well as allow for alterations to the data stored in the system.
5. Realtime nodes also load real-time data into the system. They are simpler to set up than the indexing service, at the cost of several limitations for production use.

This separation allows each node to only care about what it is best at. By separating Historical and Realtime processing, we separate the memory concerns of listening on a real-time stream of data and processing it for entry into the system. By separating the Coordinator and Broker, we separate the needs for querying from the needs for maintaining "good" data distribution across the cluster.

The following diagram shows how queries and data flow through this architecture, and which nodes (and external dependencies, discussed below) are involved

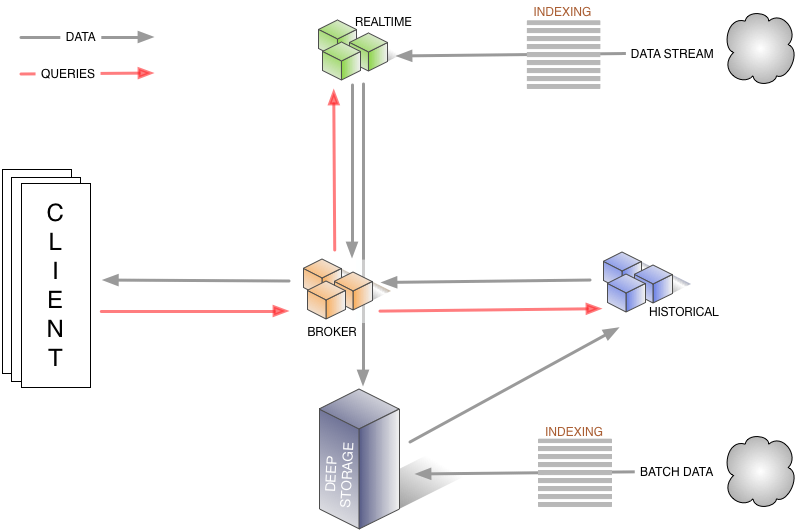


Fig.1 Data Flow in druid

All nodes can be run in some highly available fashion, either as symmetric peers in a share-nothing cluster or as hot-swap failover nodes.

Aside from these nodes, there are 3 external dependencies to the system:

1. A running ZooKeeper cluster for cluster service discovery and maintenance of current data topology
2. A metadata storage instance for maintenance of metadata about the data segments that should be served by the system
3. A "deep storage" LOB store/file system to hold the stored segments

Fully deployed, Druid runs as a cluster of specialized nodes to support a fault-tolerant architecture where data is stored redundantly and there are multiple members of each node type. In addition, the cluster includes external dependencies for coordination (Apache ZooKeeper), storage of metadata (MySQL), and a deep storage facility (e.g., HDFS, Amazon S3, or Apache Cassandra).

The following diagram illustrates the cluster's management layer, showing how certain nodes and dependencies help manage the cluster by tracking and exchanging metadata:

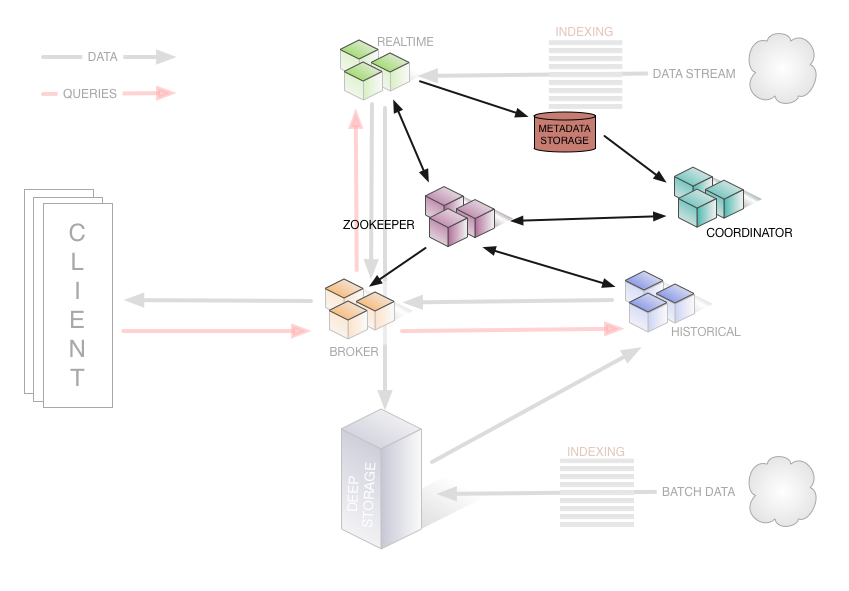


Fig.2 External dependencies

The main Functions to be discussed are:

1. Data Ingestion
2. Querying the data

## Data Ingestion

Druid supports streaming (real-time) and file-based (batch) ingestion methods. The most popular configurations are:

1. Files - Load data from HDFS, S3, local files, or any supported Hadoop filesystem in batches. We recommend this method if your dataset is already in flat files.
2. Stream push - Push a data stream into Druid in real-time using Tranquility, a client library for sending streams to Druid. We recommend this method if your dataset originates in a streaming system like Kafka, Storm, Spark Streaming, or your own system.
3. Stream pull - Pull a data stream directly from an external data source into Druid using Real-time Nodes.

## Querying the data

Queries are made using an HTTP REST style request to queryable nodes (Broker, Historical, or Real-time). The query is expressed in JSON and each of these node types expose the same REST query interface. For normal Druid operations, queries should be issued to the broker nodes.

Druid's native query language is JSON over HTTP.

Druid's native query is relatively low level, mapping closely to how computations are performed internally. Druid queries are designed to be lightweight and complete very quickly. This means that for more complex analysis, or to build more complex visualizations, multiple Druid queries may be required.

Druid has numerous query types for various use cases. Queries are composed of various JSON properties and Druid has different types of queries for different use cases. The documentation for the various query types describe all the JSON properties that can be set.

1. Aggregation Queries
   * Timeseries
   * TopN
   * GroupBy
2. Metadata Queries
   * Time Boundary
   * Segment Metadata
   * Datasource Metadata
3. Search Queries
   * Search

### Timeseries

These types of queries take a timeseries query object and return an array of JSON objects where each object represents a value asked for by the timeseries query.

An example timeseries query object is shown below:

{

"queryType": "timeseries",

"dataSource": "sample\_datasource",

"granularity": "day",

"descending": "true",

"filter": {

"type": "and",

"fields": [

{ "type": "selector", "dimension": "sample\_dimension1", "value": "sample\_value1" },

{ "type": "or",

"fields": [

{ "type": "selector", "dimension": "sample\_dimension2", "value": "sample\_value2" },

{ "type": "selector", "dimension": "sample\_dimension3", "value": "sample\_value3" }

]

}

]

},

"aggregations": [

{ "type": "longSum", "name": "sample\_name1", "fieldName": "sample\_fieldName1" },

{ "type": "doubleSum", "name": "sample\_name2", "fieldName": "sample\_fieldName2" }

],

"postAggregations": [

{ "type": "arithmetic",

"name": "sample\_divide",

"fn": "/",

"fields": [

{ "type": "fieldAccess", "name": "sample\_name1", "fieldName": "sample\_fieldName1" },

{ "type": "fieldAccess", "name": "sample\_name2", "fieldName": "sample\_fieldName2" }

]

}

],

"intervals": [ "2012-01-01T00:00:00.000/2012-01-03T00:00:00.000" ]

}

### TopN

TopN queries return a sorted set of results for the values in a given dimension according to some criteria. Conceptually, they can be thought of as an approximate GroupByQuery over a single dimension with an Ordering spec. TopNs are much faster and resource efficient than GroupBys for this use case. These types of queries take a topN query object and return an array of JSON objects where each object represents a value asked for by the topN query.

TopNs are approximate in that each node will rank their top K results and only return those top K results to the broker. K, by default in Druid, is max(1000, threshold). In practice, this means that if you ask for the top 1000 items ordered, the correctness of the first ~900 items will be 100%, and the ordering of the results after that is not guaranteed. TopNs can be made more accurate by increasing the threshold.

A topN query object looks like:

{

"queryType": "topN",

"dataSource": "sample\_data",

"dimension": "sample\_dim",

"threshold": 5,

"metric": "count",

"granularity": "all",

"filter": {

"type": "and",

"fields": [

{

"type": "selector",

"dimension": "dim1",

"value": "some\_value"

},

{

"type": "selector",

"dimension": "dim2",

"value": "some\_other\_val"

}

]

},

"aggregations": [

{

"type": "longSum",

"name": "count",

"fieldName": "count"

},

{

"type": "doubleSum",

"name": "some\_metric",

"fieldName": "some\_metric"

}

],

"postAggregations": [

{

"type": "arithmetic",

"name": "sample\_divide",

"fn": "/",

"fields": [

{

"type": "fieldAccess",

"name": "some\_metric",

"fieldName": "some\_metric"

},

{

"type": "fieldAccess",

"name": "count",

"fieldName": "count"

}

]

}

],

"intervals": [

"2013-08-31T00:00:00.000/2013-09-03T00:00:00.000"

]

}

### GroupBY

These types of queries take a groupBy query object and return an array of JSON objects where each object represents a grouping asked for by the query. The performance will be substantially better. If you want to do an ordered groupBy over a single dimension, please look at TopN queries. The performance for that use case is also substantially better. An example groupBy query object is shown below:

{

"queryType": "groupBy",

"dataSource": "sample\_datasource",

"granularity": "day",

"dimensions": ["country", "device"],

"limitSpec": { "type": "default", "limit": 5000, "columns": ["country", "data\_transfer"] },

"filter": {

"type": "and",

"fields": [

{ "type": "selector", "dimension": "carrier", "value": "AT&T" },

{ "type": "or",

"fields": [

{ "type": "selector", "dimension": "make", "value": "Apple" },

{ "type": "selector", "dimension": "make", "value": "Samsung" }

]

}

]

},

"aggregations": [

{ "type": "longSum", "name": "total\_usage", "fieldName": "user\_count" },

{ "type": "doubleSum", "name": "data\_transfer", "fieldName": "data\_transfer" }

],

"postAggregations": [

{ "type": "arithmetic",

"name": "avg\_usage",

"fn": "/",

"fields": [

{ "type": "fieldAccess", "fieldName": "data\_transfer" },

{ "type": "fieldAccess", "fieldName": "total\_usage" }

]

}

],

"intervals": [ "2012-01-01T00:00:00.000/2012-01-03T00:00:00.000" ],

"having": {

"type": "greaterThan",

"aggregation": "total\_usage",

"value": 100

}

}

### Time Boundary Queries

Time boundary queries return the earliest and latest data points of a data set.

The grammar is:

{

"queryType" : "timeBoundary",

"dataSource": "sample\_datasource",

"bound" : < "maxTime" | "minTime" > # optional, defaults to returning both timestamps if not set

}

### Segment Metadata Queries

Segment metadata queries return per segment information about:

1. Cardinality of all columns in the segment
2. Estimated byte size for the segment columns if they were stored in a flat format
3. Number of rows stored inside the segment
4. Interval the segment covers
5. Column type of all the columns in the segment
6. Estimated total segment byte size in if it was stored in a flat format

Segment id

{

"queryType":"segmentMetadata",

"dataSource":"sample\_datasource",

"intervals":["2013-01-01/2014-01-01"]

}

### Data Source Metadata queries

Data Source Metadata queries return metadata information for a dataSource. These queries return information about:

1. The timestamp of latest ingested event for the dataSource. This is the ingested event without any consideration of rollup.

The grammar for these queries is:

{

"queryType" : "dataSourceMetadata",

"dataSource": "sample\_datasource"

}

### Search Queries

A search query returns dimension values that match the search specification.

{

"queryType": "search",

"dataSource": "sample\_datasource",

"granularity": "day",

"searchDimensions": [

"dim1",

"dim2"

],

"query": {

"type": "insensitive\_contains",

"value": "Ke"

},

"sort" : {

"type": "lexicographic"

},

"intervals": [

"2013-01-01T00:00:00.000/2013-01-03T00:00:00.000"

]

}

# Result

Analytics for Wikipedia edits is setup and the visualization of data is done using pivot. The Result is as in the pictures.

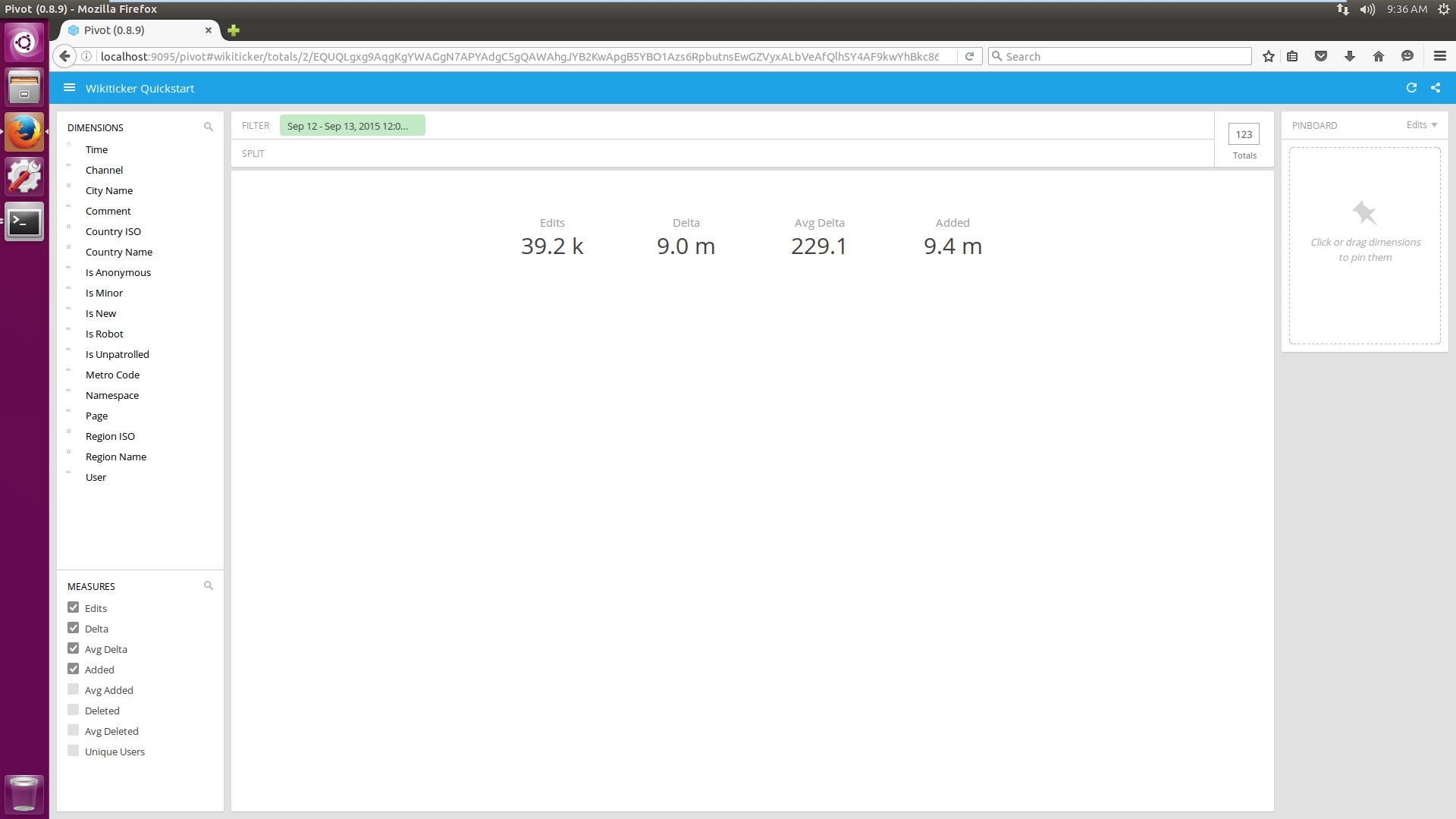


Fig.3 Visual data of the no of edits done in the Wikipedia

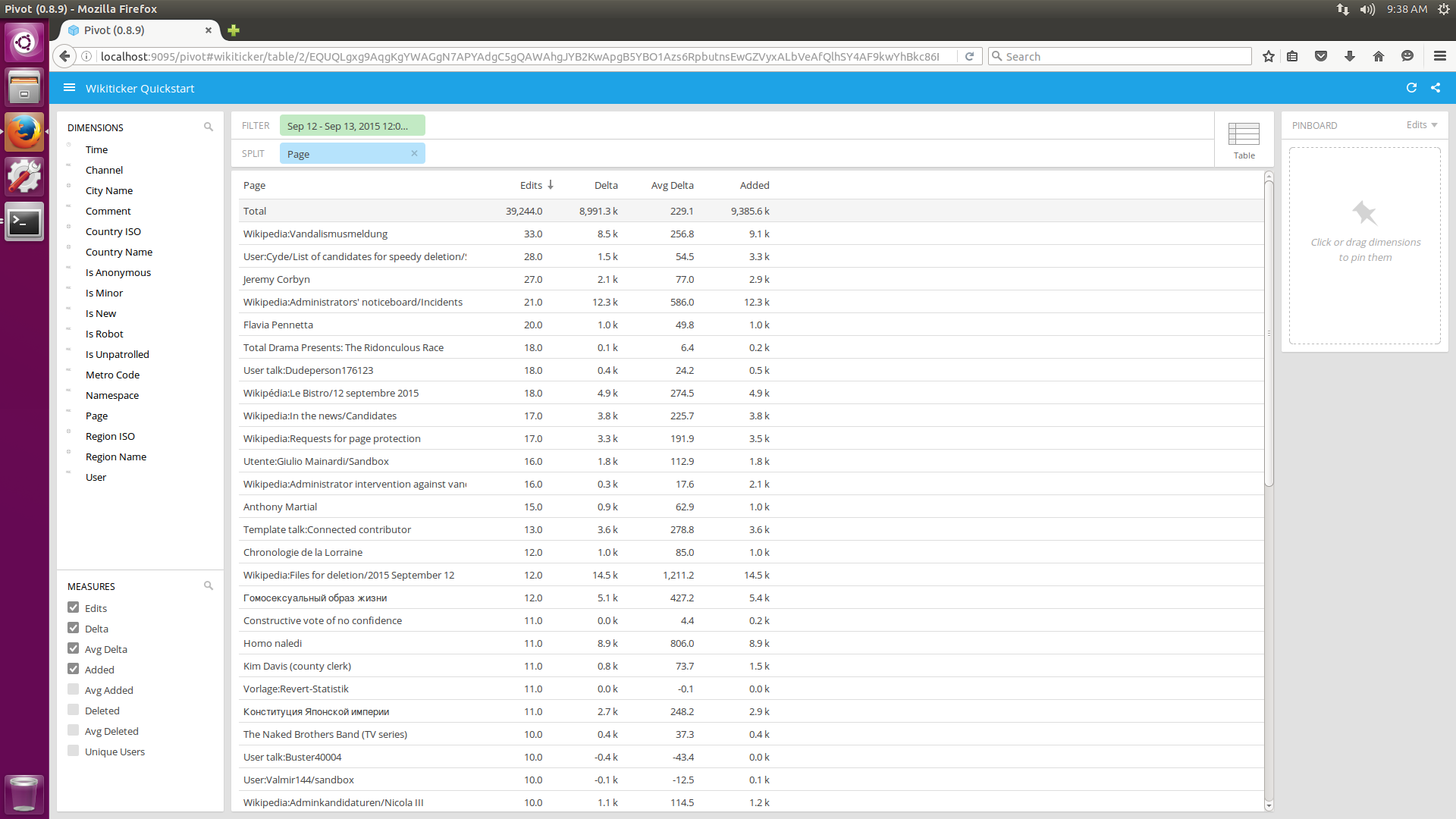


Fig.4 Visual data of the no of edits done per page in the Wikipedia

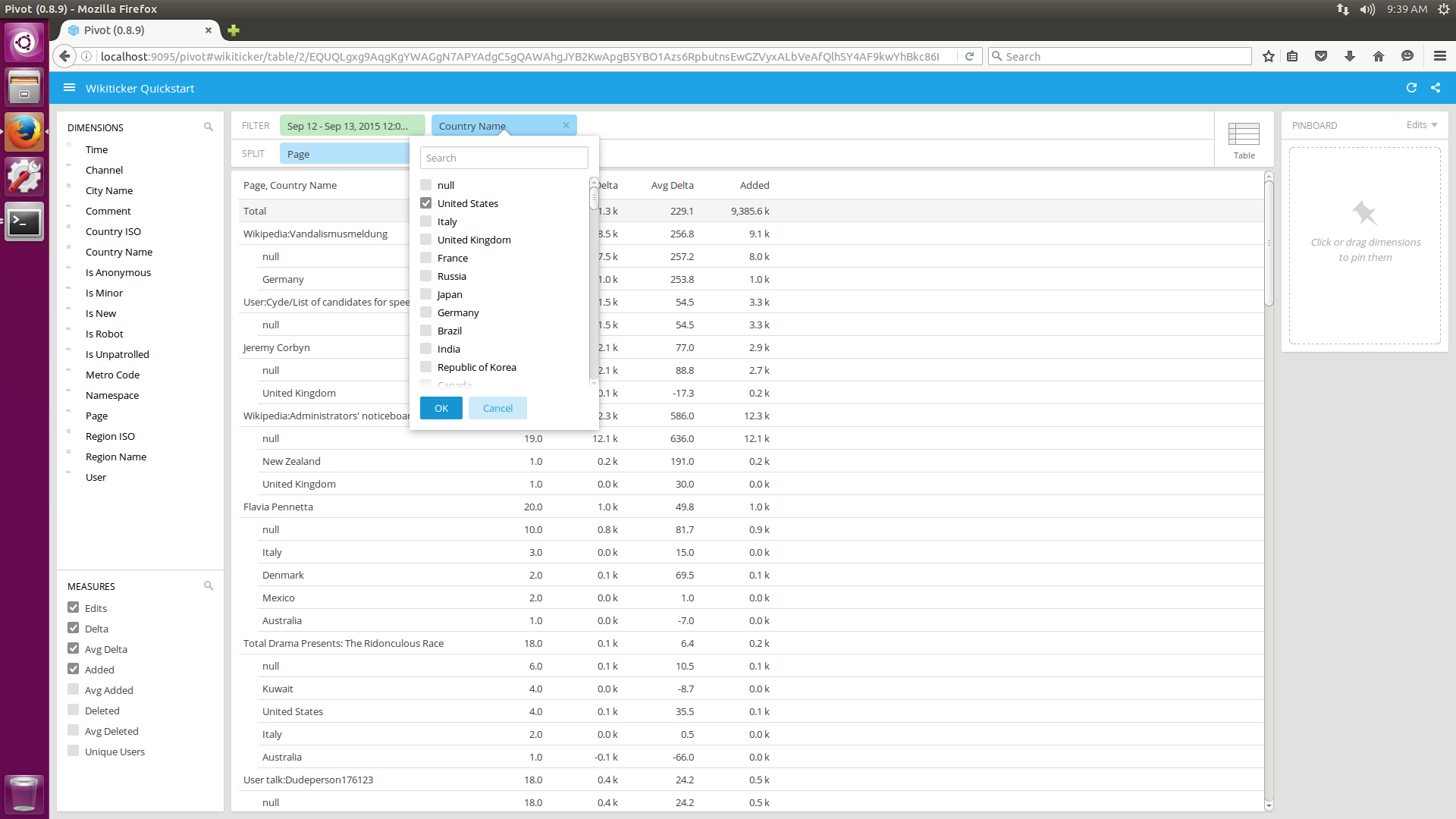


Fig.5 Visual data of the no of edits done per page in a particular country in the Wikipedia

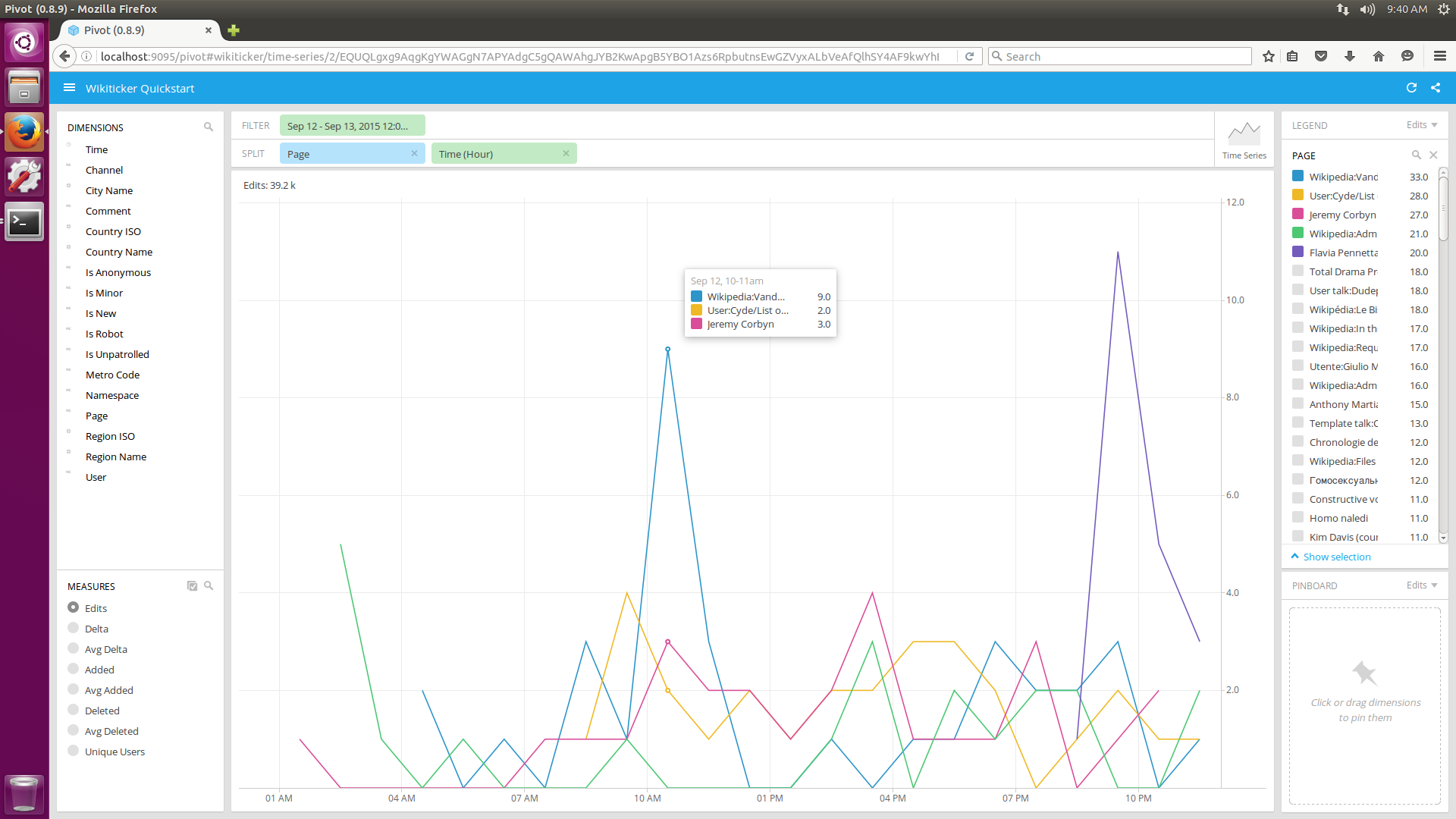


Fig.6 Visual data by time series of the no of edits done per page in a particular timings in the Wikipedia

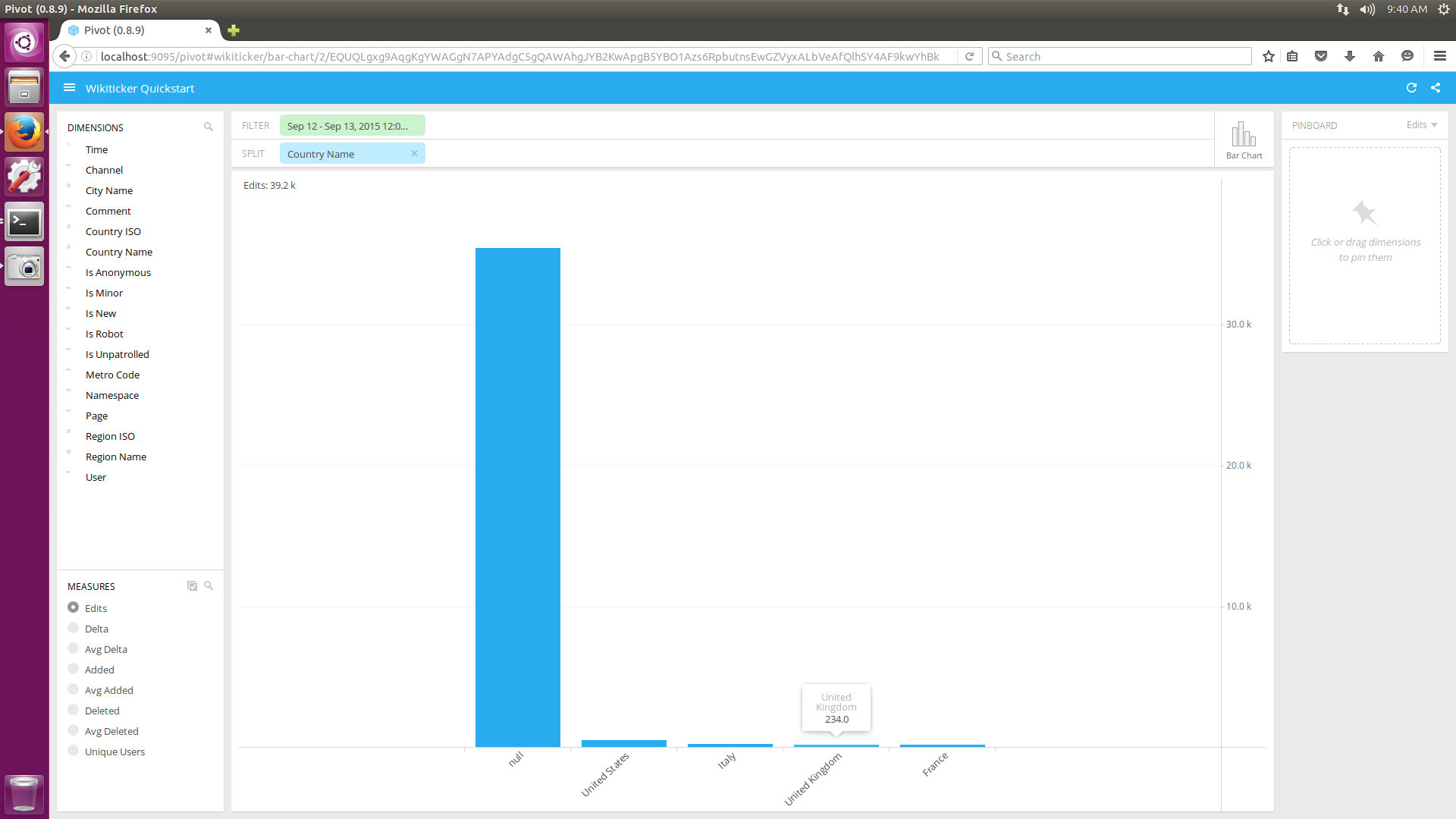


Fig. 7. Visual data by bar chart of the no of edits done per country in the Wikipedia

1. This paragraph of the first footnote will contain the date on which you submitted your paper for review. It will also contain support information, including sponsor and financial support acknowledgment. For example, “This work was supported in part by the U.S. Depart­ment of Com­merce under Grant BS123456”.

   The next few paragraphs should contain the authors’ current affiliations, including current address and e-mail. For example, F. A. Author is with the National Institute of Standards and Technology, Boulder, CO 80305 USA (e-mail: author@ boulder.nist.gov).

   S. B. Author, Jr., was with Rice University, Houston, TX 77005 USA. He is now with the Department of Physics, Colorado State University, Fort Collins, CO 80523 USA (e-mail: author@lamar.colostate.edu).

   T. C. Author is with the Electrical Engineering Department, University of Colorado, Boulder, CO 80309 USA, on leave from the National Research Institute for Metals, Tsukuba, Japan (e-mail: author@nrim.go.jp). [↑](#footnote-ref-1)