TECHNISCHE UNIVERSITÄT ILMENAU
Department of Databases and Information Systems

Research Seminar

Summer Semester 2013

Analysis and Correlation of events in Big Data

Music Category

Nikhil Rane
Matrikel 51384
Rajneesh Dewan
Matrikel 49719

Prof. Dr.–Ing. Kai-Uwe Sattler

Ilmenau, October 12, 2013

Contents

1 Introduction

[Whi12] Big data is the term for a collection of data sets so large and complex that it becomes difficult to process using on-hand database management tools or traditional data processing applications. The challenges include capture, curation, storage, search, sharing, transfer, analysis, and visualization. The trend to larger data sets is due to the additional information derivable from analysis of a single large set of related data, as compared to separate smaller sets with the same total amount of data, allowing correlations to be found to "spot business trends, determine quality of research, prevent diseases, link legal citations, combat crime, and determine real-time roadway traffic conditions." [App10] [Van10]

Event correlation is a technique for making sense of a large number of events and pinpointing the few events that are really important in that mass of information. Event correlation usually takes place inside one or several management platforms which comprises data from various sources.[EC] Since long time, event correlation is being used in the fields of telecommunications, industrial process control, network management, systems management, IT service management, and now also in Databases collecting varied information. Similar to other fields, long accumulated large amounts of data is analysed, correlated and inferred for interesting facts. These facts are of high value to study current markets, requirements, and a number of other factors.

In this paper, we are targetting data collected from an active website, eventful.com. The input data and its structure are explained in next section. On an abstract level, our ideas unfold interesting facts inferred from data over millions of records. These facs are worth for business planners to serve better and farther than their current expansions.

2 Data Structure

The input data is taken from an active website eventful.com which catalogues information about various events occurring all over the world in various categories. The data is available in both: database and CSV files. Information acquired in this way is distributed in various tables like eventcategory, events, eventtag, venues, tags, etc. Similarly, dumps for these tables are available in separate CSV files.

We required data about events in *music category* only, hence we used the file *events_music.csv*. The file has records separated by **pipe (**| delimiter with following fields:
id title **start_time stop_time** venue_name **latitude longitude** description category recur_string created modified owner url **tags venue_id** gotvenue

We did not require all of them; instead, we only used the ones in bold. Due to lot of bad data in file, we also had to filter it. We did this in 2 steps:

1. Prior to running job:
    - Remove unnecessary newline characters (in description field)
    - Insert blank (" ") strings wherever field was empty (||, |"|, |""|)
    - Make sure a new line in input file always starts with "E0-" which denotes the start of an event information.
2. While running job:
    - Parse given line of information for an event
    - Check if required fields are present (start_time, stop_time, latitude, longitude) and correctly parsed
    - If faulty record is found, filter it

We executed our algorithm on data ranging from years 2010 to 2013. In this range we had over 2 million records as input to our system.

3 Frameworks And Utilities

We primarily used VIM Editor & SED utility for pre–filtering and Apache Hadoop for processing data. Some features like Distributed Cache were also used. These are explained briefly below:

3.1 VIM Editor

One of the most advanced Linux text editors:

3.2 SED (stream editor) Utility

It is a Unix utility that parses and transforms text, using a simple, compact programming language. It has strong support for regular expressions and is notably used as substitution command. [SED]

3.3 Apache Hadoop

As stated on official website, Apache Hadoop software library is a framework that allows for distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. It avoids depending on hardware to deliver high-availability,; instead the library itself is designed to detect and handle failures at the application layer, hence delivering a highly-available service on top of a cluster of computers. [Had13]

3.4 Distributed Cache

It distributes application-specific, large, read-only files efficiently. It is a facility provided by the MapReduce framework to cache files (text, archives, jars and so on) needed by applications. The framework will copies necessary files to the slave node before any tasks for the job are executed on that node thus implementing replication and avoiding any bottlenecks. Its efficiency stems from the fact that the files are only copied once per job. [DS-13]

4 Implementation: Planner Idea

We implemented two ideas each targeting different forms of event correlation. The first idea is a simple holiday planner whereas the second one deals with association rules.

Planner is a simple utility useful for a bunch of people to hang around at a common place at a common time. Planner uses little data about 'persons' involved and whole 'events' database to find events which could be of interest for the users.

4.1 Input Data Preparation

As shown in Fig. 4.1, there are 3 persons P1, P2 and P3 who are interested in searching events of their interest. Planner takes input the events data from *events_music.csv* file described in section 2 and a *personData.txt* file consisting of following information separated by **pipe** (|) delimiter.

4.1.1 Name

Any string helpful to distinguish between persons for user.

4.1.2 Latitude & Longitude

Values for latitude and longitude of current location of the person.

4.1.3 StartTime & EndTime

Date and time (optional) range for which events have to be searched.

4.1.4 Radius

Amount of distance (in Kilometers) the person is ready to move.

4.1.5 Likes

Tags/genres of music which this person likes.

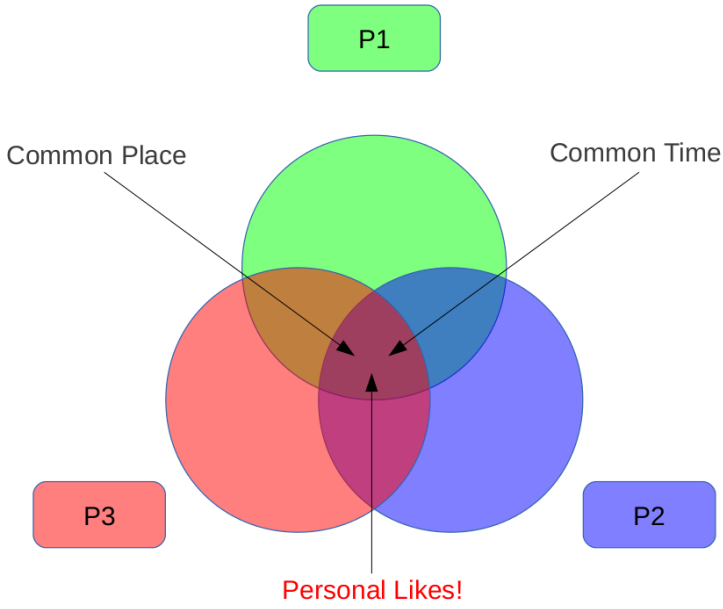

Figure 4.1: Planner

The *Likes* field is something which corresponds to one event correlation. This field can be generated by pulling and aggregating user–specific data from social networking websites like ***Facebook***. This field may differ for each user based on his/her interests. We tried pulling data from our Facebook accounts and aggregating them. The two ways of doing this are explained in next section.

4.2 Algorithms

Here we describe our algorithms for both: pulling data from Facebook and Planner Map–Combine–Reduce jobs.

4.2.1 Generating Likes field

Genres which are liked by a user can be extracted from Facebook using two ways:

Using Graph API

Using a valid OAUTH token with permissions and user's id (numeric), we can fetch all music related pages the user likes. Querying for each of this page provides its own genres. The data returned in this API is in JSON format which will again need to parsed. Also, the data for each page spans from 50 to 100 lines. Considering the number of persons and pages each person likes, a Map–Reduce job would be more feasible to pull and aggregate this genre information.

Using FQL Query

As opposed to Graph API, FQL queries are similar to SQL ones; except that FQL have restrictions like it does not support joins. However, getting data using FQL is far easy than Graph API. For instance, following query provides only the genres column which a user likes, in JSON format.

SELECT genre FROM page WHERE page_id IN
(SELECT page_id FROM page_fan WHERE uid=<numeric_user_id>AND profile_section="music");

Parsing this returned JSON data is easy and less expensive considering Graph API.

4.2.2 Planner Map–Combine–Reduce

Planner is implemented in three Map–Combine–Reduce jobs. Each of them is described below.

1. Find Candidate Events

This job finds all events which are candidate events that all persons can attend. It takes as input *events_music.csv* and persons data *personData.txt* files.

Algorithm 1 Find Candidate Events

```
1: Parse event information
2: for all persons in personData.txt do
3:     calculate distance
4:     if (distance_is_within_radius) AND (person.startTime <= event.startTime < person.endTime)
       then
5:         set isCandidate = true
6:     else
7:         set isCandidate = false
8:         break
9:     end if
10: end for
11: if (isCandidate == true) then
12:     output(venue_ID, eventInfo)
13: end if
```

Output

A list of events which are within each person's radius and start & end times. A sample output is as below.
E0-001-039140724-7|...|alternitive,...,metal,hardrock,pop,reverbnation|V0-001-000477808-3
E0-001-039179661-9|2011-05-20 10:00:00|...|festivals_parades|V0-001-001051854-3
E0-001-039239961-7|...|cincinnatiusacom,concert,music|V0-001-001051854-3
E0-001-037377806-2|39.491188|-84.32839|...|concert,reverbnation|V0-001-003699774-0
E0-001-036995779-6|...|39.491188|-84.32839|...|concert,music|V0-001-003699774-0

2. Find Trend

This job finds trend for venues which are found in previous job. The venues are extracted and matched with events file to get their tags. This job takes as input the events (*events_music.csv*) and output of previous job.

Algorithm 2 Find Trend

```
1: Parse output from previous job to create a candidate_venue_list
2: Parse event information from input (events_music.csv)
3: if (candidate_venue_list.contains(event.venue_ID) then
4:     for all tags in event.tags do
5:         output(venue_ID+tag,1)
6:     end for
7: end if
8: Count occurrence of tag in the Reducer
```

Output

A file with *venue_ID*|*tag*|*popularity (count)* values. For instance,
V0-001-000477808-3|alternitive|1
V0-001-000477808-3|cool|1
V0-001-001051854-3|festivals_parades|1
.
.
.
V0-001-003699774-0|concert|7
V0-001-003699774-0|health|2
V0-001-003699774-0|reverbnation|7
V0-001-004810906-5|concert|1
V0-001-004900812-5|concert|14

3. Rank Results

The task of this job is to generate files for each person and one *System* file. These files contain event information obtained in first job, which are ranked based upon tags. The *System* file contains events ranked as per the *trend* which was aggregated in second job. Other files pertaining to *each user* have events ranked based upon each individual's *likes* field. The input for this job is persons data (*personData.txt*) and output of first & second jobs.

Algorithm 3 Rank Results

```
1: Parse output of second job and generate top_tag_list with values <venue_ID, topTag>based on tag occurrences
2: Parse event information from output of first job
3: topTag = topTagList.get(event.venue)
4: if (event.tags.contain(topTag)) then
5:     output("SYSTEM|TOP", event)
6: else
7:     output("SYSTEM|LOW", event)
8: end if
9: for all persons in personData.txt do
10:     for all tag in event.tags do
11:         if (person.likes.contains(event.tag) then
12:             output(person.name+"|TOP", event)
13:             break
14:         end if
15:     end for
16:     if (none_of_the_tags_matched) then
17:         output(person.name+"|LOW", event)
18:     end if
19: end for
20: To separate files, extract System/Person name and send it as filename to Hadoop API
```

Output

Files for each person and one *System* file with variedly ranked events!

System

E0-001-039140724-7|...|alternitive,...,metal,hardrock,pop,reverbnation|V0-001-000477808-3
E0-001-039239961-7|...|cincinnatiusacom,concert,music|V0-001-001051854-3
E0-001-037377806-2|39.491188|-84.32839|...|concert,reverbnation|V0-001-003699774-0
E0-001-036995779-6|...|39.491188|-84.32839|...|concert,music|V0-001-003699774-0

P1

E0-001-039140724-7|...|alternitive,...,metal,hardrock,pop,reverbnation|V0-001-000477808-3
E0-001-036476457-1|...|39.5393|-84.4433|...|concert,music,reverbnation|V0-001-004810906-5
E0-001-036184100-0|...|cincinnati,concert,harcore,metal,music|V0-001-003699774-0
E0-001-036995779-6|...|39.491188|-84.32839|...|concert,music|V0-001-003699774-0

Notice the events are ranked differently. This can also be observed in files for other users.

5 Implementation: Association Rules Idea

The discovery of interesting correlation relationships among huge amounts of event records can help in many business decision-making processes such as catalog design, cross-marketing, and customer consuming behavior analysis. Frequent itemset mining leads to the discovery of associations and correlations among the 28 event categories in large relational data sets of eventful.com which have happened on the same date and at the same venue. With massive amounts of event data continuously being collected and stored, many customers and business individuals are becoming interested in mining such patterns from the database. These patterns can be represented in the form of **association rules**. From our research point of view, our interest is the information that category of event which tend to happen at the same time and at the same venue with other type events in the following association rule:

"Which groups or sets of events are likely to happen on a same date and on a same venue?" For example, *festival_parades ->music [support 2%,confidence 0.60].*

Rule **support** and **confidence** are two measures of rule interestingness. They respectively reflect the usefulness and certainty of discovered rules. A support of 2% for the Rule above means that 2% of all the events under analysis show that festival_parades and music have occured together. A confidence of 0.60 means that 60% of the events which is festival_parades also happened simultaneously with music events. Typically, association rules are considered interesting if they satisfy both a minimum support threshold and a minimum confidence threshold. These thresholds can be a set by users or domain experts. Additional analysis can be performed to discover interesting statistical correlations between associated items.

In terms of applying MapReduce programming for implementing mining mechanism for discovering association rules out of historical data of event table can be a great source of divergence of conventional Apriori algorithm. The total time complexity may be same as apriori algorithm but the data parallelism helps in great deal with reducing the number of transaction and reducing the number of candidates to 2^w from 2^d where d is the number of unique event category and w is average width of the event category set or group; width in a sense the number of event category in a set. We concluded our project with result of four MapReduce jobs to ultimately implement association rules from events table.

5.1 First & Second Map Reduce

The first and the second MapReduce program is actually the cleaning and extraction process of the data table named event by parsing and grouping by our desired attributes in order to get the count for the transaction rows to simplify the size of input.

5.1.1 First Map

The input for the First Map gets divided by string lines which gets processed line by line and then each line gets parsed by the delimiter "|" into 17 column of data for attribute. But we only have interests on the 3rd and the last one which are starting date of the event and the venue id. As per our concern we want all the event category to gather under the same reduce function which has the same starting date and the same venue, so we have made the key output for the first Map function as the concatenated string of starting date of the event and the venue id.

Text key = new Text (starting_date + venue_id)

As we have to get the event categories under the same starting_date and venue_id, we make the value part of the output of the first Map function the name of the event category. Each output carries one key and one event category.

Map Output (key: starting_date + venue_id, value: event_category)

Naturally the sorting part after the first Map reduce function doesn't serve any purpose for us but the shuffling makes the event category to come as list of values for the same starting date + venue key.

5.1.2 First Reduce

The input for the Reduce functions are the key which is concatenated string of starting_date and venue_id and the list of values of event categories. For example,
For key: 19-01-2013v0-34989203940
Values: <art, music, festival_parade>

As our next goal is to group by the key which is the set of event categories, we have to put the event categories in a string line such a way so that the order of the event categories is unique for all the other same set of event categories on different date and venue. The easiest way to make a unique order out of the array of events is to do a sorting on the array of event category set. For example,
<art, music, festival_parades>=>after sorting string array and concatenation ->art.festival_parades.music

In this way the set for event category of art, festival_parades and music will always remain unique which is also appropriate for being a key for the output of the our first reduce function. As for the value of the output of the first reduce function can be an IntWritable object of number 1 because just as for the same reason of word count the key of event category set represents only the one frequency of occurrence on a date and a particular venue.

5.1.3 Results of the first MapReduce

We get all the set or group of event categories happened on a same date and venue with a value of 1. This is a significant transformation of normal transaction row of the event table where each single row represented only one event but now each row of the first MapReduce output represents one to many rows of the event table as a set. A snap shot of the output is as such,

art.festival_parades.music 1
art.music 1
festival_parades.music 1
art.music 1
art.music 1
festival_parades.music 1

5.1.4 Second Map function

The second map function gets the result from the first MapReduce as it's input but the Map function doesn't have enough computation to do. So, it just a necessary part for the next reduce function which is necessary to have all the value of ones grouping by the key of different event category sets.

So, the output of the second Map is actually the same as the final result of the first MapReduce.

5.1.5 Second Reduce function

The job of the second reduce function is kind of similar to traditional word count program. Values of ones are grouped by the key and summation and count of the ones then made as output. Key remains the same as the input for the second Reduce function.

art.festival_parades.music <1>
art.music <1, 1, 1>
festival_parades.music <1, 1>

From the previous output result we can show the exact output we can expect as below,

art.festival_parades.music 1
art.music 3
festival_parades.music 2

The count we compute inside our second reduce function is not enough to take a decision to omit event category set which has a count below threshold of minimum support count. Like if minimum support count is 1, we cannot discard the event category set **art.festival_parades.music** because in this set there are other category set which are not added with the existing one.

If **art.festival_parades.music** set has occurred once so is the set of **festival_parades.music** and that one value has not yet added with the count or sum value of the existing **festival_parades.music**

set which is 2 in the 3rd line of output. So, the final output of the second MapReduce is not really a support count for the event category sets because the counts are still exclusive with other event sets.

5.2 Exclusive & Inclusive Support Count

Here, I refer the normal support count as the inclusive support count because it represents the support count for the entire possible event category set occurred on a same date and same venue inside the event table. But our output for the first MapReduce program needs further computation to generate actual support Count which is being considered as one of our main goals for the project.

Currently, we have the output as exclusive support count which doesn't consider the sub set of a unique set of event category. For example,

If Music + Art happened on a same date and at a same venue 100 times and only music but not art happened 10 times then actually music happening on the same date and time regardless of other event happened also or not is 100 plus 10 times. So 10 times music happening on a date and time is exclusive support count.

So exclusive support count:
Musc + Art ——-(on a same date and venue)————>100
Music not Art—-(on other date and venue————>10
Art not Music—-(on other date and venue)————>05

Actual support count:
Musc + Art ——(on a same date and venue)————->100
Music ————-(on other date and venue)————>10 + 100 = 110
Art————————(on other date and venue)———->05 + 100 = 105

5.3 Third MapReduce

So the third MapReduce actually makes further manipulation of the existing output data from the first and second MapReduce in order to discover the actual support count. But it significantly adds another complexity to our association rule generation program by introducing the combination generation function which is responsible to generate 1 to k combination from any k number of unique item set. This combination function is recursive so there is a significance chance of memory heap overflow. So it is to my attention to convert the recursive function into non recursive one later.

5.3.1 The combination generation function

In our program the function actually gets a string array as a set and it outputs a list of string with all the combination possible from length one to the size of the array which are theoretically sub sets of the given set of event categories. With an example it's function should be clear;

A set of {a, b, c} with length 3 produces 2^3 subsets of the parent set as unique combination possible.

Function Combination (input:{a, b, c})

Output->
a
ab
abc
ac
b
bc
c

5.3.2 Third Map function

The third map function is responsible for finding all the combination for a given key which is a set of event categories. This is necessary to make the support count from exclusive to inclusive. As per our previous discussion about the exclusive support count from 100 times of **art.festival_parades.music** event set we get also **art.music** also 100 times which need to be added with existing 10 number of occurrences of **art.music** exclusively on other date and place.

For this purpose we have to find all the combinational subsets of **art.festival_parades.music** and make them key with value of the parent set's number of occurrence. It is rather simple to show with the example above with a event set of {a,b,c}

So, if we have {abc} = 100 row from our last output. A further subset of number 2^3 needs to become output of the third Map function. E.g.
(a, 100)
(ab, 100)
(abc, 100)
(ac, 100)
(b, 100)
(bc, 100)
(c, 100)

5.3.3 Third Reduce function

Third Reduce function has the simple tasks as the second Reduce function of summing up the value of occurrence for each key of event category set. As for all the key they represent all the

unique event set possible from the given event table and no need to generate more candidate key than these. Finally the output can be called the support count of all the possible candidate key which is event category set occurred on a same date and place. So as for the example above only “a” set might have many counts from many parent event category sets. Like, inside the reduce function
a 100 from abc = 100
a 12 from acd = 12
a 10 from a = 10

b 100 from abc = 100
c 100 from abc = 100
c 12 from acd = 12
d 12 from acd = 12

Output->a 100+12+10 = 122
b 100
c 100+12=112
d 12
.
.
.

5.4 Fourth MapReduce

This final MapReduce is our main program to generate the association rules for the purpose of finding the possiblity of occurrence of one or more events given that other events have occurred on the same date and the place. Best can be described by an example of event set {a,b,c} with support 100.

5.4.1 Fourth Map function

If we want to know the confidence of {a} ->{b,c} which is actually shows the possibility of occurrence of {b,c} given that {a} has a support count 110 (which is obviously bigger than the support count of {a,b,c}), we have calculate the fraction of support count of {a,b,c} and support count of {a}. That is 100/110. Note that we don’t need to know the support count for {b,c} to find the confidence of {a} ->{b,c}. What we need is support count of {a,b,c} and {a} and what key set we want to associate, that is {b,c}.

So, we made the key the set of event categories which is given; that is {a}. For the value we take concatenation of string {b,c} + 100 (which is the support count of the {a,b,c})

So in the Map function when we get a set of event categories, we apply the same combination function to get all the subset of the parent set. For example of a parent set {a,b,c} with a support count 100 we get all the below combination for output,

Key	Value
a	b.c.100
a.b	c.100
a.b.c	100
a.c	b.100
b	a.c.100
b.c	a.b.100
c	a.b.100

This output will also include support count of set
{a} 110,
{b} 100,
{c} 112,
{d} 12
.....And so on.

5.4.2 Fourth Reduce function

As per the output of the fourth Map function the Reduce function will get key as a set of event categories but for list of values it will get all the necessary data and possible associated event sets whose confidence needs to be calculated. Like for the key {a} below value list will be found,
a 110
a bc.100

Above the only numeric value 110 is the support count of {a} but bc.100 is the indication to find the confidence of a ->bc which is calculable from 100(support count of {abc}). All we need to do in generating the output of the reduce function is to create key like below by cutting substring from the value.

Key
a->a
a->bc

and value corresponding to keys will be the fraction of support counts of parent set and key itself. Like,

Key	Output Value
a->a	110/110 = 1
a->bc	100/110 = 0.90

6 Conclusions

- Apache Hadoop is an extremely powerful framework especially for data crunching in projects like Wolframalpha and Enigma. Due to auto and high scalability, Hadoop re-organizes itself for to use resources wisely. Failure detection and actions avoid manual maintenance.
- As demostrated in Planner idea, external data can act as a catalyst in event correlation with Hadoop. Events are not only information which are recorded from past but are constanly generated; for instance, when a user likes a page (*on Facebook*) or posts any photos on Pinterest (*location information*).
- Considering Association rules idea, in first and second MapReduce
  - N >>N (counting transaction grouping by)
- In third MapReduce
  - N = Total number of unique itemset (Summing transaction value grouping by)
  - Complexity: $(N * \sum^{\mathrm{w}} C_{\mathrm{i}})$ where i is 1->w
- And in fourth MapReduce
  - Complexity: same as third. So total $(N * \sum^{\mathrm{w}} C_{\mathrm{i}})$ where i is 1->w

Bibliography

[App10] Data, data everywhere, the economist, 2010.

[DS-13] http://hadoop.apache.org/docs/stable/mapred_tutorial.html#DistributedCache, 2013.

[EC] http://en.wikipedia.org/wiki/Event_correlation.

[Had13] Apache hadoop. http://hadoop.apache.org/, 2013.

[SED] http://en.wikipedia.org/wiki/Sed.

[Van10] Ashley Vance. Start-up goes after big data with hadoop helper, new york times blog, 2010.

[Whi12] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media / Yahoo Press, 2012.