A Comparison of Big Data Application Programming Approaches: A Travel Companion Case Study

Pei Guo, Jianwu Wang, Zhiyuan Chen
Department of Information Systems
University of Maryland, Baltimore County
Baltimore, MD, USA
{peiguo1, jianwu, zhchen}@umbc.edu

***Abstract*—With advances of big data technologies, there are many possible ways to program for each big data application. A challenge is to know the differences of the program approaches and decide which programming approach is the best for a particular big data application. In this paper, we use vehicle travel companion as a case study to explore four different programming approaches, including Spark RDD (with GroupBy or Join), Spark SQL with Hive and Hive on Hadoop, and tune the programmed big data applications. Our experiments show that the execution time of one programming approach could be more than 100-fold longer than that of another for the same application logic, which verifies that programming approach decision is important. We also explain the reasons for the differences. The findings could be applied to the selection of programming approach for other big data applications.**

Keywords—Big Data; Spark; Hive; SQL; Programming Comparison; Benchmarking; Travel Companion

I. Introduction

As big data becomes more prevalent, more and more platforms/systems and programming models are available to build big data applications. Even only using one platform, big data application developers still have many programming options to choose from. For instance, as a popular open-source big data framework, Spark [1] supports both resilient distributed dataset (RDD) based and structured query language (SQL) based programming. Many new database systems, including Hive [2], MongoDB [3], and HBase [4], support direct SQL query and integration with big data platforms like Spark and Hadoop [5]. On the one hand, the abundance of options provides application developers with more choices. On the other hand, it poses difficulties and challenges to knowing the advantages and disadvantages of every possible programming solution, especial which solution is the best for the application. There have been many big data benchmarking studies on comparison of different big data platforms [17-22]. Yet, to our best knowledge, there still lacks comparison of different big data programming approaches for a specific application, especially for real-world applications that are more complex than the programs used in most of the current benchmarking studies.

In this paper, we address the above challenge via a case study for travel companion calculation. Travel companion calculation is to find groups of vehicles with similar moving trend within a period of time. It is a hot topic and there is a rich literature on discovering travel companion [6-11], including ours [10, 11]. Discovering travel companions has wide range of application areas such as transportation management, protecting certain special vehicles and military surveillance [9]. In crowded metropolitan area such as New York, Beijing, and Tokyo, the dense traffic flow contains huge number of travel companions of every vehicle. To compute travel companions, we need to develop a big data application which is able to process millions of records quickly.

This paper explores and compares four programming approaches for our travel companion calculation application. Because the application logic could be expressed using SQL-like statements, it can either be implemented through Spark SQL or pure SQL statements on database systems like Hive and Impala [26]. The programming approaches we evaluated include Spark RDD, Spark SQL with Hive and Hive on Hadoop. Our experiments in Azure cloud environment [15] show different programming approaches could be over 100 times different in performance. Further, by investigating the possible reasons of the performance differences, we summarize some findings that could be applicable to many other big data applications.

The contributions of the paper are: 1) using a real-world case study, we demonstrate how a big data application could be programmed in multiple approaches; 2) our experiments of the case study verify that programming approach could be an important factor for their performance differences; 3) by examining the experiment results, we summarize several general findings on how to choose the best programming approach for a specific application.

The rest of the paper is organized as follows. In Section II, we briefly introduce the technologies we applied in this paper. Section III describes two applications of our case study, the travel companions for all vehicles and for one target vehicle. In Section IV, different programming approaches for the application are introduced and explained. Section V describes the experiments, results and findings. Section VI discusses related works. In Section VII, our conclusion and future work are discussed. Also, an appendix of the HiveQL we used in the experiments can be found at the end of the paper.

978-1-5386-2715-0/17/$31.00 ©2017 IEEE

II. Background

A. Spark

As a popular open-source big data framework, Apache Spark [1] provides both application programming interface (API) to build scalable big data applications, and execution engines to run the applications with implicit data parallelism and fault-tolerance support. It also provides a collection of elements partitioned across the nodes, called RDD, to implement parallel operation on a cluster with many compute nodes. RDD follows and extends the MapReduce programming model [12]. Developers need to write imperative/procedural programs using RDD operations to process their data accordingly. Spark programs have a number of tunable parameters including data serialization, the level of parallelism, memory usage of reduce tasks, data locality [13].

B. Spark SQL

Spark SQL is a module of Spark to process structured data [14]. Compared to Spark RDD API, Spark SQL contains more information on data structure and offers much tighter integration between relational and procedural processing, through a declarative DataFrame API that integrates with procedural Spark code. Moreover, it has a built-in and highly extensible optimizer called Catalyst [17], which makes it easy to add composable rules, control code generation, and define extension points. Eventually, the physical plan generated by Catalyst will be executed on Spark RDDs. Spark SQL also supports reading and writing data stored in database systems such as Apache Hive.

C. Hive

Apache Hive [2] is an open-source data warehousing solution built on top of Apache Hadoop. Hive supports queries expressed in a SQL-like declarative language - HiveQL, which are compiled into MapReduce jobs that are executed using Hadoop. In addition, HiveQL enables users to plug in custom map-reduce scripts into queries. HiveQL support Hive data definition language (DDL) and data manipulation language (DML) similar to SQL. Hive stores file into Hadoop Distributed File System (HDFS) [5]. Hive supports Hive index table and Hive partition table for query performance improvements. Hive index table is to improve the speed of querying a table of specific column. Normal queries with predicates like 'WHERE tab1.col1 = 10' will load the entire table or partition and process all the rows. But with index for col1, only a portion of the file needs to be loaded and processed. Hive partition table is also a way to improve Hive query speed by only querying a portion of data. It divides a table into related parts based on the values of partitioned columns such as date, city, and states.

D. Azure and HDInsight

Azure is a comprehensive set of cloud services from Microsoft that developers and IT professionals could use to build, deploy, and manage applications through global network of datacenters. HDInsight [16] is a managed open-source big data analytic service on Microsoft Azure through virtual clusters. The software packages of HDInsight clusters are provided by Hortonworks Data Platform (HDP) [25]. There are different types of clusters like Hadoop, Spark, Hbase [4]. In each type of cluster, various Hadoop components, including Hadoop, Spark, Hive, HBase, Storm [24], are provided and can be deployed easily through shell or web user interface.

III. Application

The aim of our application is to calculate vehicle travel companions from real traffic data which is taken by city road cameras on crossroads. The real traffic data that we are using is called Automatic Number Plate Recognition (ANPR) data [11]. Nowadays, almost every road crossing has installed Traffic Enforcement Cameras in many cities in China. During peak hours, each camera steadily takes pictures of passing vehicles with one second interval. The information that the cameras get includes plate number and passing time, which is automatically recognized and transmitted to traffic management department's central data center continuously.

The raw traffic dataset D is in the format of (c, v, t), where c represents camera ID, v represents vehicle plate number, and t is the timestamp of vehicle v passing by camera c. Two vehicles are identified as a travel companion pair if they pass through some cameras that the number of camera is greater or equal to the camera threshold δ_c and their timestamp differences for every camera they passed are within a given time threshold δ_t. For instance, given input $\delta_c = 2$ and $\delta_t = 300$ seconds, suppose v_1 passes camera c_1 at 9:00 and v_2 passes camera c_1 at 9:02, then v_1 passes another camera c_2 at 9:10, and v_2 also passes camera c_2 at 9:14. Since their passing-by differences are within 5 minutes for c_1 and c_2, $(<v_1, v_2>, \{(c_1, <9{:}00, 9{:}02>), (c2, <9{:}10, 9{:}14>)\})$ is a travel companion.

Our case study includes two specific travel companion calculation applications: 1) all exact vehicle companion pairs (companions-for-all-vehicles), and 2) all exact companions of one specific vehicle (companions-for-one-vehicle). Companions-for-all-vehicles output contains all the vehicle pairs which pass cameras within given time threshold. Whereas companions-for-one-vehicle calculation has one more input variable v^*, which is the target vehicle that we are aimed to find out all of its travel companions.

IV. Application Programming

We programed both companions-for-all-vehicles and companions-for-one-vehicle on the following four approaches: Spark RDD with *groupByKey* function (Spark GroupBy), Spark RDD with *join* function (Spark Join), Spark SQL with Hive database programming, and HiveQL manipulation on Hadoop. The two sub-sections explain how we programmed the two applications in every approach respectively.

A. Companions-for-all-vehicles Application Programming

a) Spark RDD groupByKey based Programming

Algorithm I shows how this application is programed using the *groupByKey* function.

Algorithm I. AllCompSparkGroupBy

Input: Dataset D, camera number threshold δ_c, time threshold δ_t

Output: $(<v_x, v_y>, \{(c_1, <t_{x1}, t_{y1}>), (c_2, <t_{x2}, t_{y2}>), \ldots, (c_k, <t_{xk}, t_{yk}>)\})$ where $k \geqslant \delta_c$, $| t_{xi} - t_{yi} | \leqslant \delta_t$, $i = 1, 2, \ldots k$

```
1. map $(c, v, t)$ in $D$ into key-value pair $(c, (v, t))$
2. get all vehicles passing by each camera $(c, List<v, t>)$ by $groupByKey()$
3. filter $(c, List<v, t>)$ by condition: size of $List<v, t> \geqslant 2$
4. enumerate $List<v, t>$ for each camera to get all companion pairs: $\{c, (<v_x, t_x>, <v_y, t_y>)\}$
5. for each companion pair, if $| t_x - t_y | \leqslant \delta_t$
6.   get vehicle pair: $\{<v_x, v_y>, (c, <t_x, t_y>)\}$
7. group results based on vehicle pair to get companion: $cp = (<v_x, v_y>, \{(c_1, <t_{x1}, t_{y1}>), (c_2, <t_{x2}, t_{y2}>), \ldots, (c_k, <t_{xk}, t_{yk}>)\})$
8. output $cp$ if $k \geqslant \delta_c$
```

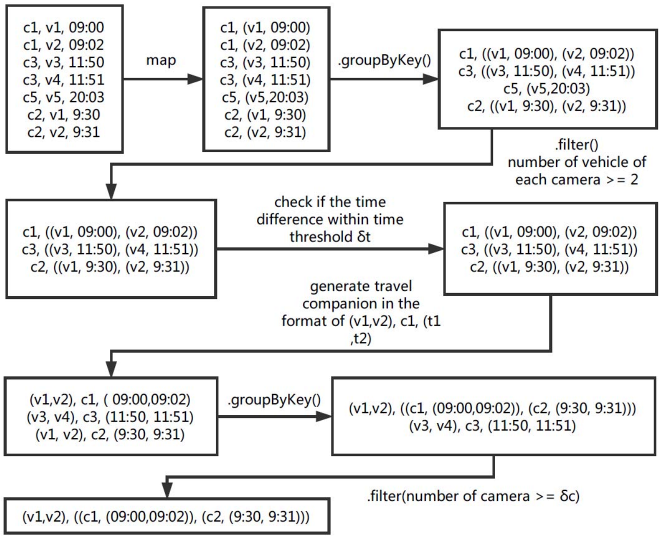
c1, v1, 09:00
c1, v2, 09:02
c3, v3, 11:50
c3, v4, 11:51
c5, v5, 20:03
c2, v1, 9:30
c2, v2, 9:31
map
c1, (v1, 09:00)
c1, (v2, 09:02)
c3, (v3, 11:50)
c3, (v4, 11:51)
c5, (v5,20:03)
c2, (v1, 9:30)
c2, (v2, 9:31)
.groupByKey()
c1, ((v1, 09:00), (v2, 09:02))
c3, ((v3, 11:50), (v4, 11:51))
c5, (v5,20:03)
c2, ((v1, 9:30), (v2, 9:31))
.filter()
number of vehicle of each camera >= 2
c1, ((v1, 09:00), (v2, 09:02))
c3, ((v3, 11:50), (v4, 11:51))
c2, ((v1, 9:30), (v2, 9:31))
check if the time difference within time threshold δt
c1, ((v1, 09:00), (v2, 09:02))
c3, ((v3, 11:50), (v4, 11:51))
c2, ((v1, 9:30), (v2, 9:31))
generate travel companion in the format of (v1,v2), c1, (t1 ,t2)
(v1,v2), c1, (09:00,09:02)
(v3, v4), c3, (11:50, 11:51)
(v1, v2), c2, (9:30, 9:31)
.groupByKey()
(v1,v2), ((c1, (09:00,09:02)), (c2, (9:30, 9:31)))
(v3, v4), c3, (11:50, 11:51)
.filter(number of camera >= δc)
(v1,v2), ((c1, (09:00,09:02)), (c2, (9:30, 9:31)))

Fig. 1. Example of Algorithm I, with $\delta_t = 300$ seconds and $\delta_c = 2$.

In Spark RDD programming, *groupByKey* function groups the values for each key in the RDD into a single sequence [1]. For example, if *groupByKey* function is called on a dataset of key-value pairs *(key, value)*, it returns a dataset of *(key, List<value>)* pairs based on *key*. The inputs of the algorithm include dataset D, camera number threshold δ_c, and time threshold δ_t. At the beginning, in line 1, each record d of D (camera c, vehicle v, and timestamp t) is transformed into $(c, (v, t))$ key-value pairs using a map function. In line 2, all $(c, (v, t))$ pairs are grouped by key c, to get all (v, t) under same camera c. Then, in line 3, if the size of $List<(v, t)>)$ in the grouped $(c, List<(v, t)>)$ is less than 2, which means that if the camera has fewer than 2 vehicles passing by, then this record is eliminated. For the remaining $(c, List<(v, t)>)$ key-value pairs, in line 4, the algorithm generates all exact companions with same c such as $(c, \{<v_x, t_x>, <v_y, t_y>\})$. Then, as in line 5, it checks the companion pairs $(<v_x, t_x>, <v_y, t_y>)$ to make sure that their timestamp difference is under δ_t, i.e., $| t_x - t_y | \leqslant \delta_t$. After having all the conformed exact companions, in line 6, the next step is to transform the data structure to $(<v_x, v_y>, \{c, <t_x, t_y>\})$. In line 7, the algorithm does another *groupByKey* with key as vehicle pair (v_x, v_y); and in line 8, it filters the companion pairs with the number of c no less than camera threshold δ_c to generate companions for all vehicle that correspond to the input requirements. An example with $\delta_c = 2$ and $\delta_t = 300$ seconds with some sample data is shown in Fig. 1.

b) Spark RDD join based Programming

Algorithm II. AllCompSparkJoin

Input: Dataset D, camera number threshold δ_c, time threshold δ_t

Output: $(<v_x, v_y>, \{(c_1, <t_{x1}, t_{y1}>), (c_2, <t_{x2}, t_{y2}>), \ldots, (c_k, <t_{xk}, t_{yk}>)\})$ where $k \geqslant \delta_c$, $| t_{xi} - t_{yi} | \leqslant \delta_t$, $i = 1, 2, \ldots k$

```
1. map $(c, v, t)$ in $D$ into key-value pair $p = (c, (v, t))$
2. get all $<v, t>$ under each camera in the format of $\{c, (<v_x, t_x>, <v_y, t_y>)\}$ by self join $p.join(p)$
3. filter $(c, \{<v_x, t_x>, <v_y, t_y>\})$ by $| t_x - t_y | \leqslant \delta_t$ and remove repeat results
4. map filtered $(c, \{<v_x, t_x>, <v_y, t_y>\})$ to get $(<v_x, v_y>, \{c, <t_x, t_y>\})$
5. group results by vehicle pair to generate vehicle companions: $cp = (<v_x, v_y>, \{(c_1, <t_{x1}, t_{y1}>), (c_2, <t_{x2}, t_{y2}>), \ldots, (c_k, <t_{xk}, t_{yk}>)\})$
6. output $cp$ if $k \geqslant \delta_c$
```

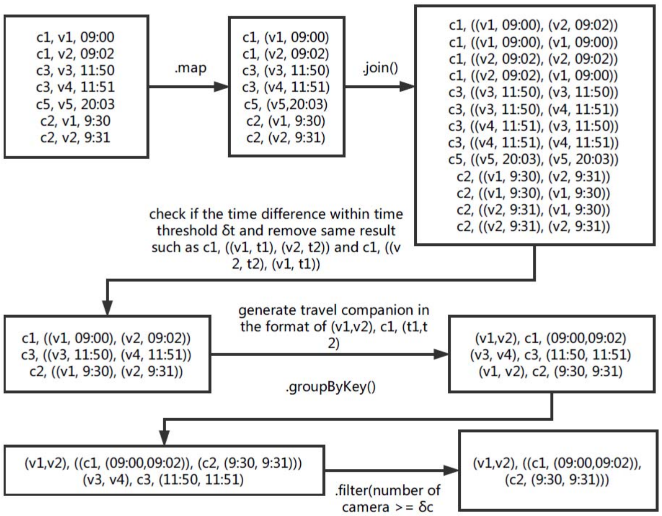
c1, v1, 09:00
c1, v2, 09:02
c3, v3, 11:50
c3, v4, 11:51
c5, v5, 20:03
c2, v1, 9:30
c2, v2, 9:31
.map
c1, (v1, 09:00)
c1, (v2, 09:02)
c3, (v3, 11:50)
c3, (v4, 11:51)
c5, (v5,20:03)
c2, (v1, 9:30)
c2, (v2, 9:31)
.join()
c1, ((v1, 09:00), (v2, 09:02))
c1, ((v1, 09:00), (v1, 09:00))
c1, ((v2, 09:02), (v2, 09:02))
c1, ((v2, 09:02), (v1, 09:00))
c3, ((v3, 11:50), (v3, 11:50))
c3, ((v3, 11:50), (v4, 11:51))
c3, ((v4, 11:51), (v3, 11:50))
c3, ((v4, 11:51), (v4, 11:51))
c5, ((v5, 20:03), (v5, 20:03))
c2, ((v1, 9:30), (v2, 9:31))
c2, ((v1, 9:30), (v1, 9:30))
c2, ((v2, 9:31), (v1, 9:30))
c2, ((v2, 9:31), (v2, 9:31))
check if the time difference within time threshold δt and remove same result such as c1, ((v1, t1), (v2, t2)) and c1, ((v 2, t2), (v1, t1))
c1, ((v1, 09:00), (v2, 09:02))
c3, ((v3, 11:50), (v4, 11:51))
c2, ((v1, 9:30), (v2, 9:31))
generate travel companion in the format of (v1,v2), c1, (t1,t 2)
(v1,v2), c1, (09:00,09:02)
(v3, v4), c3, (11:50, 11:51)
(v1, v2), c2, (9:30, 9:31)
.groupByKey()
(v1,v2), ((c1, (09:00,09:02)), (c2, (9:30, 9:31)))
(v3, v4), c3, (11:50, 11:51)
.filter(number of camera >= δc
(v1,v2), ((c1, (09:00,09:02)), (c2, (9:30, 9:31)))

Fig. 2. Example of Algorithm II, with $\delta_t = 300$ seconds and $\delta_c = 2$.

In Algorithm 2, Spark *join* function is used to generate the output of companions-for-all-vehicles instead of *groupbyKey* function. In Spark *join* function, two RDDs will join into one RDD, which contains all value pairs with same keys in the input RDDs [1], e.g., if two key-value pairs $(key, value_1)$ and $(key, value_2)$ joins in spark, for every key, it returns a dataset of $(key, (value_1, value_2))$ tuples with all pairs of elements. Line 1 in Algorithm II is the same with line 1 in Algorithm I. Then in line 2, a self *join* by key c is executed to every $(c, (v, t))$. The joined results are in the format of $(c, \{<v_x, t_x>, <v_y, t_y>\})$. In line 3, the algorithm removes the repeated results to get desirable filtered key-value pairs $(c, \{<v_x, t_x>, <v_y, t_y>\})$ in which $v_x \neq v_y$, and $| t_x - t_y | \leqslant \delta_t$. After the filtering step, in line

4, it generates all the conformed *$(<v_x, v_y>, \{c, <t_x, t_y>\})$*, then in line 5, group the results by key of vehicle companion pair such as *(v_x, v_y)* and apply a filter in line 6 to get the pairs with the number of camera in the vehicle companion is large or equal to camera threshold δ_c to generate companions-for-all-vehicles that correspond to the input requirements. Fig. 2 illustrates how Algorithm II works with with $\delta_t = 300$ seconds and $\delta_c = 2$.

c) Spark SQL and Hive based Programming

In this programming approach, we connect Spark SQL to Apache Hive database by creating a spark session to execute SQL DDL and DML. SQL code can be found in Appendix A. Below are steps:

1. Find all records where each camera has more than one vehicle passing through (view 1 in Appendix A).
2. Find all camera, vehicle and time by applying time threshold δ_t (view 2 in Appendix A).
3. Generate vehicle pair, camera and time pair specifically (view 3 in Appendix A).
4. Find all companions by applying camera threshold δ_c (Final insert Appendix A).

d) Hive on Hadoop Programming

We execute the same SQL code to generate companions-for-all-vehicles. The SQL code is stored in script on HDFS, and executed in Hive command line using Hadoop MapReduce engine. The output is written into a file stored in HDFS.

B. Companions-for-one-vehicle Application Programming

a) Spark RDD groupByKey based Programming

```
Algorithm III. OneCompSparkGroupBy
Input: Dataset D, camera number threshold $\delta_c$, time threshold
  $\delta_t$, target vehicle $v^*$
Output: $(<v_x, v^*>, \{(c_x, <t_{x1}, t^*_1>), (c_2, <t_{x2}, t^*_2>), \ldots, (c_k, <t_{xk}, t^*_k>)\})$ where $k \geqslant \delta_c$, $|t_{xi} - t^*_i| \leqslant \delta_t$, $i = 1, 2, \ldots k$
1. map $(c, v, t)$ in D into key-value pair $(c, (v, t))$
2. get all vehicles passing by each camera $(c, List<v, t>)$ by
   groupByKey( )
3. filter $(c, List<v, t>)$ by condition: size of $List<v, t> \geqslant 2$
4. enumerate $List<v, t>$ for each camera to get all companion
   pairs: $(c, \{<v_x, t_x>, <v^*, t^*>\})$
5. for each companion pair, if $|t_x - t_y| \leqslant \delta_t$ and ($v_x = v^*$ or $v_y$
   $= v^*$)
6.   get vehicle pair $(<v_x, v^*>, \{c, <t_x, t^*>\})$
7. group results based on vehicle pair to get companion: $cp =$
   $(<v_x, v^*>, \{(c_x, <t_{x1}, t^*_1>), (c_2, <t_{x2}, t^*_2>), \ldots, (c_k, <t_{xk}, t^*_k>)\})$
8. output $cp$ if $k \geqslant \delta_c$
```

Algorithm III applies Spark *groupByKey* function to calculate the travel companions based on the target vehicle v^*. Besides the input v^*, the other inputs of this algorithm include the inputs of companions-for-all-vehicles as Dataset *D*, camera threshold δ_c, and time threshold δ_t. Line 1-4 in this algorithm are the same with those in Algorithm I to get filtered all vehicle companions. In line 5 and 6, the algorithm checks not only the timestamp differences are all under time threshold δ_t, but also $v_x = v^*$ or $v_y = v^*$ to make sure that the result contains the target vehicle v^* to generates all the conformed *$(c, \{<v_x, t_x>, <v_y, t_y>\})$*. And next step in line 6 is to transform the data structure of conformed pairs to *$(<v_x, v^*>, \{c, <t_x, t^*>\})$*, and do another *groupByKey* in line 7 with key as vehicle pair *(v_x, v^*)*, and filter the pairs in line 8 that the number of *c* is no less than camera threshold δ_c to finally generate the travel companions for one vehicle in the format of *$(<v_x, v^*>, \{(c_x, <t_{x1}, t^*_1>), (c_2, <t_{x2}, t^*_2>), \ldots, (c_k, <t_{xk}, t^*_k>)\})$*.

b) Spark RDD join based Programming

Algorithm IV applies Spark *join* function to calculate the travel companions based on the target vehicle v^*. Its inputs and outputs are same with Algorithm III. The algorithm is same with Algorithm III in line 1 to get all *(v, t)* tuples under same *c*. Next, in line 2, it generates all records of target vehicle by checking every record with $v = v^*$ to get all the pairs *$p\ (c, (v^*, t))$*. In line 3, another *RDD $p_1\ (c, (v, t))$* in which $v \neq v^*$ is also generated to perform the join by *p_1.join(p)* and get result *$(c, \{<v_x, t_x>, <v^*, t^*>\})$* in line 4. After the *join* operation, in line 5, *$(c, \{<v_x, t_x>, <v^*, t^*>\})$* is filtered by $|t_1 - t_2| \leqslant \delta_t$ and repeat results are also removed. Then, in line 6, it maps the filtered *$(c, \{<v_x, t_x>, <v^*, t^*>\})$* to get all conformed companions containing target vehicle v^* in the format of *$(<v_x, v^*>, \{c, <t_x, t^*>\})$*, and execute a *groupByKey* function to group the companions by vehicle pairs in line 7, following by applying a filter of check if the number of camera in the companion is large or equal to the camera threshold δ_c in line 8 and get the output for companions-for-one-vehicle.

```
Algorithm IV. OneCompSparkJoin
Input: Dataset D, camera number threshold $\delta_c$, time threshold
  $\delta_t$, target vehicle $v^*$
Output: $(<v_x, v^*>, \{(c_x, <t_{x1}, t^*_1>), (c_2, <t_{x2}, t^*_2>), \ldots, (c_k, <t_{xk}, t^*_k>)\})$ where $k \geqslant \delta_c$, $|t_{xi} - t^*_i| \leqslant \delta_t$, $i = 1, 2, \ldots k$
1. map $(c, v, t)$ in D into key-value pair $(c, (v, t))$
2. get key-value pair $p = (c, (v^*, t))$ by filter $v = v^*$
3. get key-value pair $p_1 = (c, (v, t))$ that $v \neq v^*$
4. get $(c, \{<v_x, t_x>, <v^*, t^*>\})$ by $p_1$.join(p)
5. filtering $(c, \{<v_x, t_x>, <v^*, t^*>\})$ by $|t_x - t^*| \leqslant \delta_t$ and
   remove repeat results
6. map filtered $(c, \{<v_x, t_x>, <v^*, t^*>\})$ to get $(<v_x, v^*>, \{c,$
   $<t_x, t^*>\})$
7. group results based on vehicle pair to get companion: $cp =$
   $(<v_x, v^*>, \{(c_x, <t_{x1}, t^*_1>), (c_2, <t_{x2}, t^*_2>), \ldots, (c_k, <t_{xk}, t^*_k>)\})$
8. output $cp$ if $k \geqslant \delta_c$
```

c) Spark SQL and Hive Based Programming

Similar to the companions-for-all-vehicle, in this approach, we use Spark SQL to execute SQL codes in 4 steps. And the SQL code can be found in Appendix B.

1. Find all cameras that the target vehicle passing through (view 1 in Appendix B).
2. Find all vehicles right before or after the target vehicle passing the camera within time threshold δ_t (view 2 in Appendix B).
3. Generate vehicle pair, camera and time pair specifically (view 3 in Appendix B).

4. Find all companion by applying camera threshold δ_c (view 4 and final select in Appendix B).

d) Hive on Hadoop Programming

In this approach, we directly execute the same SQL code in Appendix B to calculate companions-for-one-vehicle.

V. EXPERIMENTS

The experiments are conducted on a real ANPR dataset collected in Beijing, China. The dataset involves one day recognized vehicle plate information from 2012-11-01 00:00:00 to 2012-11-01 23:59:59. It includes 801328 unique vehicles and 927 unique cameras. The one-day dataset is 191.2 MB in size and contains 4.4 million records. We also tried with one-hour subset from the one-day dataset which size is 1.1 MB and has 40 thousand records. The cloud environment in the experiment is Azure HDInsight 3.6 Spark cluster with HDP 2.6, Spark 2.1.6, Apache Hive 1.2.1, Apache Hadoop and Yarn 2.7.3 on Linux. The virtual cluster contains two head nodes and two worker nodes.

Questions we want to answer through the experiments are as follows.

a) How different programming approaches will affect the same application's performance?

b) Is the performance ranking/order of different programming approaches the same for different applications?

c) Will different input data sizes affect the execution performance ranking/order of different programming approaches?

d) What are the general rules of selecting programming approaches?

e) Will tuning of other parameters affect the programming approaches' performance and their ranking for each application?

To answer these questions, we conduct five experiments. Experiment 1 ran companions-for-all-vehicles using Spark Join, Spark GroupBy, Spark SQL with Hive and Hive on Hadoop in default settings and compared their performance in execution time. Experiment 2 ran companions-for-one-vehicle using the same four programming approaches as in Experiment 1. With Experiment 1 and 2, question a and b could be answered. Experiment 3 and 4 ran companions-for-all-vehicles and companions-for-one-vehicle calculation on different data sizes, respectively. These two experiments could answer question c and d. In Experiment 5, we tuned other parameters to answer question e.

A. *Experiment 1: Companions-for-all-vehicles Application in Default Settings*

In this experiment, all four programming approaches for companions-for-all-vehicles application were executed on Azure cloud HDInsight clusters on default settings with the one-day dataset and the one-hour dataset. The results are shown in Table 1 and Fig. 3. The results indicate that Spark with *groupByKey* method has the worst performance. We had to kill the execution for one-day dataset because it was still running after 600 minutes. Spark *groupByKey* function takes a lot of time in shuffling reading and writing process. Execution time using this approach is over 100 times longer than that using Spark SQL with Hive approach. Moreover, Hive on Hadoop is slower than Spark SQL and Spark Join because it does not execute on Spark engine, and it is known that Spark runs faster than Hadoop.

TABLE I. EXECUTION TIMES OF COMPANIONS-FOR-ALL-VEHICLES USING FOUR APPROACHES IN ONE-DAY AND ONE-HOUR DATASETS

Program	**Datasets**	
	One-Day Data (4.4M)	***One-Hour Data (40K)***
Spark Join	52.0 min	19s
Spark GroupBy	> 600.0 min (unfinished)	24s
Spark SQL + Hive	5.4 min	46s
Hive on Hadoop	60.0 min	173s

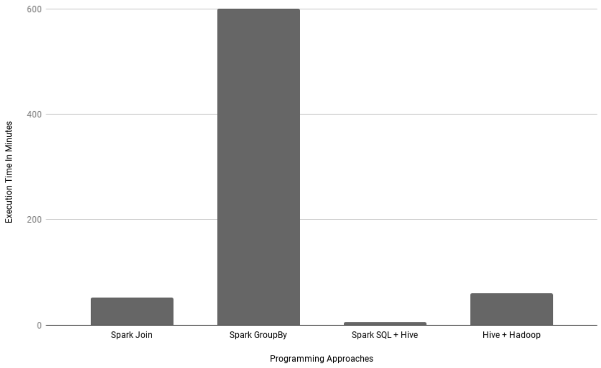

Fig. 3. Execution time comparison of companions-for-all-vehicles application for one-day dataset in default settings.

The two approaches with the best performances are Spark Join and Spark SQL with Hive. For one-day data, Spark SQL with Hive has the best performance. However, for one-hour data, Spark Join is the best option. The difference is largely due to the choice made by Spark SQL's Catalyst optimizer. For one-day data, the optimizer selects a join method (broadcast hash join) that is more efficient than the join method used in Spark Join. For one-hour data, the optimizer selects a join method (sort-merge join) that is worse than that of Spark Join. We will offer more details in next sub-section.

B. *Experiment 2: Companions-for-one-vehicle Application in Default Settings*

For companions-for-one-vehicle application on one-day dataset, the result is displayed in Table II and Fig. 4. Spark Join performs the best, followed by Spark SQL, Hive on Hadoop and Spark GroupBy. This experiment corresponds to joining two tables when one of them (for a specific vehicle) is much smaller than the other (for all other vehicles). Interestingly, Spark SQL with Hive performs best in Experiment 1 but falls behind Spark Join in Experiment 2 for one-day dataset, whereas Hive on Hadoop still falls behind and Spark GroupBy is unfinished after 120 minutes (7200s).

TABLE II. EXECUTION TIME OF COMPANIONS-FOR-ONE-VEHICLE USING FOUR APPROACHES IN ONE-DAY AND ONE-HOUR DATASET

Program	Datasets	
	One-Day Data (4.4 M)	***One-Hour Data (40K)***
Spark Join	27s	16s
Spark GroupBy	> 7200s (unfinished)	22s
Spark SQL + Hive	58s	49s
Hive on Hadoop	102s	51s

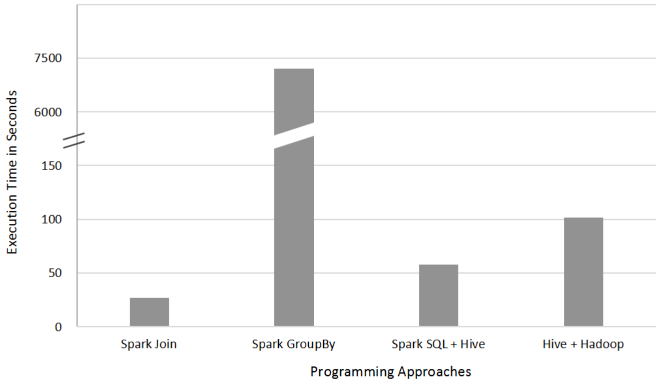

Fig. 4. Execution time comparison of companions-for-one-vehicle application for one-day dataset in default settings.

Our examination shows that the change on the ranks of Spark Join and Spark SQL on Hive performance is mainly due to the switch of the physical plan that how Spark SQL process HiveQL. In the calculation of companions-for-one-vehicle, sort merge join is used. As a comparison, in Experiment 1, Spark SQL chooses broadcast hash join to join the tables.

Fig. 5 shows the physical plan for companions-for-all-vehicles that is automatically generated by Spark SQL based on its SQL statements. In the SQL code (see Appendix A) view1 returns cameras with at least two vehicles passing by. The query plan to compute view 1 contains two table scans (one with group by and filtering) and a hash join (the bottom left part in Fig. 5). The next step is a self join on view 1 to return pairs of vehicles passing by the same camera within a time window. So the bottom right part of Fig. 5 also computes view 1 and the third hash join near top computes the self join. The remaining steps are filtering on the join results. One quick observation is that the same view (view1) is computed twice. So performance could be further improved if Spark SQL knows how to reuse results from other parts of query plans.

Fig. 6 shows the query plan for companions-for-one-vehicle application that is automatically generated by Spark SQL based on its SQL statements. The first step selects only traffic records of a vehicle with a certain license plate number. The second step selects traffic records of vehicles passing the same camera with the given vehicle within a time threshold. This is done by a sort-merge join (the bottom right part of Fig. 6). Next another sort-merge join is done to generate vehicle pair, camera, and time pairs. Finally, the method finds companion by applying camera threshold.

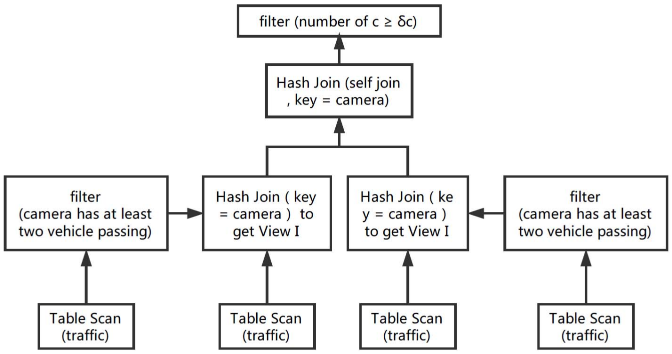
filter (number of c ≥ δc)
Hash Join (self join , key = camera)
filter (camera has at least two vehicle passing)
Hash Join (key = camera) to get View I
Hash Join (ke y = camera) to get View I
filter (camera has at least two vehicle passing)
Table Scan (traffic)
Table Scan (traffic)
Table Scan (traffic)
Table Scan (traffic)

Fig. 5. Physical plan by Spark SQL for companions-for-all-vehicles

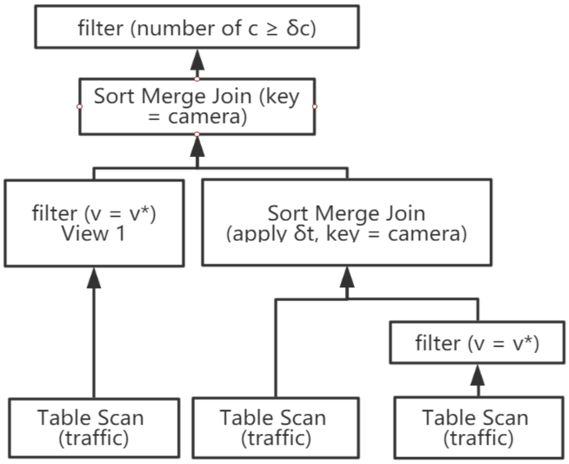

Fig. 6. Physical plan by Spark SQL for companions-for-one-vehicle

In Spark SQL, sort merge join and hash join are implemented differently. Sort merge join is executed in three steps. First, it shuffles the two join tables based on their join key, and distributes the data to the cluster for parallel execution. The second step is to sort the data on every worker node. Finally, it joins the sorted data based on the join key. However, in Spark SQL broadcast hash join, a table is broadcast to all the nodes where another table partitions located at, then these tables do hash join on each partition. In our experiment, the dataset is not sorted. So compared to the hash join, Spark SQL takes much more time in the stage of sorting the data during sort merge join. We believe this is the main reason Spark SQL performs worse than Spark Join in the companions-for-one-vehicle application.

In this experiment, the programming of Spark GroupBy and Hive on Hadoop still performs not as good as the Spark Join and Spark SQL with Hive. So in the following experiments we only used Spark Join and Spark SQL with Hive to examine the performances on different data sizes.

In conclusion, for companions-for-one-vehicle application, when the two tables in join operation have very different sizes (a.k.a. imbalanced datasets), e.g. one of them is much smaller than the other, Spark Join is the best option. Also, if both are small datasets, Spark Join is still the best one, which is the same with Experiment 1. Spark SQL + Hive is the best for large data sets and when both tables in join operation have similar sizes (in Experiment 1).

C. *Experiment 3: Companions-for-all-vehicles Application on Different Datasets*

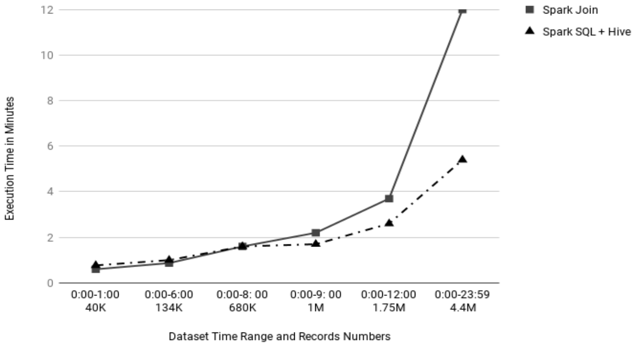

Fig. 7. Execution time of Spark Join and Spark SQL with Hive for companions-for-all-vehicles application with different input dataset sizes.

We extracted different datasets from one-day dataset by the timestamp of every records. For example, the dataset with one million records is 12 am to 9 am data. We used input of these different datasets to examine the performance of Spark Join and Spark SQL with Hive since they are the top two options from Experiment 1 and 2. As shown in Fig. 7, the result indicates that in a smaller dataset, Spark Join execution is faster than Spark SQL with Hive. However, when data size increases, the execution time of Spark Join method increases more than that of Spark SQL with Hive. As discussed in Experiment 1 and 2, Spark SQL uses broadcast hash join, which creates hash join key and hash table. When data size is small, the hash process takes more time than directly join manipulation in Spark RDD. But as data size increases, hash join can process join faster than the Spark RDD join, and the preprocessing of hash key and hash table creation could be ignored. Thus Spark SQL with Hive performs better in larger dataset. Spark SQL with Hive will be the best option only when the size of data is large enough.

D. *Experiment 4: Companions-for-one-vehicle Application on Different Datasets*

Experiment 4 varies the data sizes similar to Experiment 3. The result in Fig. 8 indicates that Spark Join performs better than Spark SQL with Hive for all data sizes in companions-for-one-vehicle application. In this experiment, Spark SQL uses sort merge join, which takes more time than direct join by Spark.

In conclusion, based on the results in this experiment, Spark Join approach performs best on joins of tables with very different sizes and joins of small datasets.

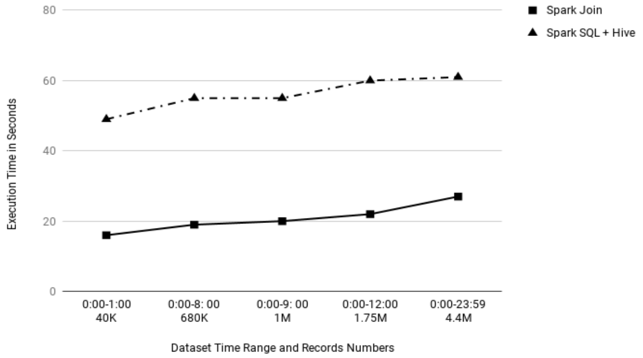

Fig. 8. Execution time of Spark Join and Spark SQL with Hive for companions-for-one-vehicle application with different input dataset sizes.

E. *Experiment 5: System Tuning of Spark, Spark SQL and Hive*

To find out whether tuning system parameters will affect application performance and the ranking of different programming approaches, we tune the following three parameters: 1) level of parallelism in Spark, 2) Hive index table and Hive partition table, and 3) *cache* method on Spark RDD and Spark SQL programming.

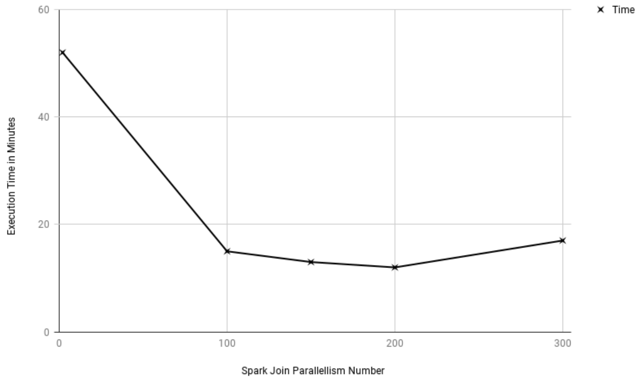

Fig. 9. Spark parallelism number tuning, on one-day dataset of companions-for-all-vehicles.

Fig. 9 shows the results for different levels of parallelism in companions-for-all-vehicles using Spark Join programming approach. We change the number of *map* tasks [13] to execute on each file, and find that this number causes huge differences in execution time when the input dataset is large. In Spark default settings, in the companions-for-all-vehicles execution, the number of tasks running on CPU on each cluster is 2. When we change the parallelism number to 200, the execution time of one-day data calculation is one third of that of the original setting. However, it is still not as fast as Spark SQL with Hive. Thus the ranking of different programming approaches in Experiment 1 is not affected. There are two other findings: 1) when data size is small, increasing level of parallelism will slow down the program; 2) when data size is large, increasing level of parallelism to an appropriate level will increase the performance; however, if number of parallelism is too large, the performance will again decrease. In our case, 200 is a proper level of parallelism.

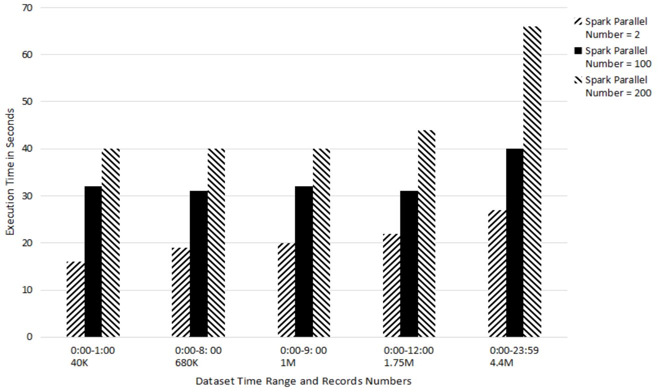

Fig. 10. Spark parallelism number tuning on different datasets for companions-for-one-vehicle.

In calculation of companions-for-one-vehicle using Spark Join function, Fig. 10 shows that increasing parallelism level of Spark does not improve the execution time. We believe that the speed decrease is due to excessive overhead for Spark to manage too many small tasks.

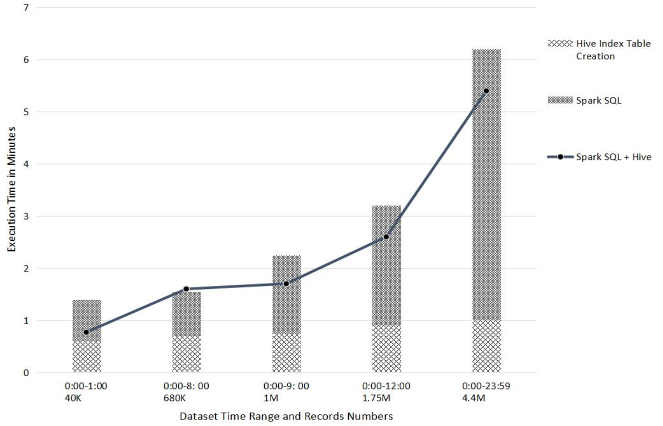

Fig. 11. Hive index table influences with different input datasets on companions-for-all-vehicles.

We created Hive index table with index on vehicle column and Hive partition table partitioned by vehicle column for companions-for-all-vehicles and companions-for-one-vehicle both using Spark SQL on Hive and Hive on Hadoop programs. The influence of Hive index table results is shown in Fig. 11 and Fig.12, respectively. In each figure, Spark SQL with Hive table without indexing program execution time is illustrated in line, and the bars include time of creating Hive indexing table and Spark SQL execution. Significantly, Spark SQL with Hive table without indexing runs faster or nearly same to that of Spark SQL with Hive indexing tables in both applications. Moreover, Hive takes more than 35 minutes to create the partition table which partitioned by vehicle, and the time it takes is definitely more than the program execution time of Spark SQL with Hive without partition table which is 5.4 minutes. The results indicate that Hive indexing and partitioning do not lead to significant improvement of performance. One possible reason is that when computing travel companion, the program still needs to check data about most vehicles (even in companions-for-one the companion vehicle could be any one), so indexing and partitioning do not help much.

We applied *cache* function on Spark Join and Spark SQL with Hive programming. But the results show that it has little effect on the performance of Spark Join and Spark SQL with Hive.

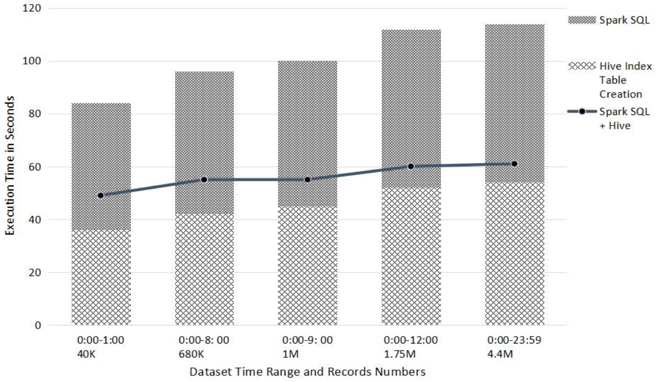

Fig. 12. Hive index table influences with different input datasets on companions-for-one-vehicle.

F. Findings from Experiments:

From the results of all the experiments, our findings can be concluded as following:

Finding 1: It is crucial to select the right programming approach because different programming approaches of the same application could have over 100 times performance differences.

Finding 2: Hive-on-Hadoop and Spark GroupBy programming approaches should be avoided because their performances are worse than others and never achieve the best performance in our experiments.

Finding 3: Either Spark Join or Spark SQL approach will achieve the best performance in all experiments.

Finding 4: Spark SQL approach performs the best on joins of large datasets and when joining tables with similar sizes.

Finding 5: Spark Join approach performs the best on small datasets and when joining tables with very different sizes.

Finding 6: It is crucial to select appropriate level of parallelism.

Finding 7: Indexing or partitioning the Hive table do not have much performance impact for this application.

VI. Related Work

There have been some comparison studies on big data analysis through different database approaches. Andrew et al. [18] compared MapReduce and parallel database management systems (DBMS) on large-scale data analysis and claimed that there are trade-offs between long data-loading time and shorter processing time of parallel DBMSs and MapReduce. Later, the comparison of traditional relational databases and NoSQL databases (MongoDB, Couchbase, Cassandra, etc.) in big data environment with key-value store implementations [20, 21] shows that NoSQL databases can be easily scaled out to fit big data environment better, and they are optimized for key-value data structures but not all of them is better than SQL databases. Basically, MongoDB and Couchbase are good choices over traditional SQL implementation.

Furthermore, there are work about comparison of database performance which includes Spark SQL. The Spark SQL paper [17] compared performance of Spark SQL against Shark and Impala and claiming that Spark SQL is faster than Shark, and competitive to Impala based on the AMPLab big benchmark. Also, a performance evaluation of HiveQL and Spark SQL using BigBench [19] finds that HiveQL performs better on Spark SQL than pure HiveQL commands. Our experiments get similar results.

There also studies on comparing different big data implementations for specific applications. MapReduce implementation on Hadoop, LEMO-MR and Twisters were compared in [22]. Our previous work [23] compared different big data programming approaches for a bioinformatics tool.

To our best knowledge, our work is unique and different from existing work in the following three aspects. First, we compare different big data programming approaches, not different big data platforms/systems in above studies. Second, there is no benchmark that combines both big data platform such as Spark and database systems. Our work considers such combinations by comparing approaches using Spark functions (*groupByKey* and *join*), Spark SQL with HiveQL, and HiveQL on Hadoop. Third, instead of using atomic/simple application jobs like WordCount and MegaSort, our work uses two real-world big data applications in the comparison. In conclusion, we believe our work can complement existing work of comparing different platforms to help developers choose not only the best platform but also the best programming approach on the platform for their applications.

VII. Conclusion and Future Work

In the era of big data, choosing a proper programming platform and approach is crucial for application developers. This paper presents a case study in travel companion calculation in which we intend to find the best big data programming approach. The case study includes two travel companion calculations: companions-for-all-vehicles and companions-for-one-vehicle. The programming approaches we chose to implement our application were Spark Join, Spark GroupBy, Spark SQL with Hive and Hive on Hadoop approach. From our experiments, we have found that the four programming approaches have very different performances, and the difference of execution time could be more than 100 times. So we have to be aware of the performance differences of different programming approaches. Moreover, either Spark Join or Spark SQL with Hive approach is the best choice among different applications and input data sets. Spark SQL approach tends to perform better on large datasets and where the application needs to join tables with similar sizes; whereas Spark Join approach tends to perform better on smaller data sets and with joins of tables with very different sizes. In conclusion, the selection of the best programming approach needs to consider both the application logic and the data characteristics.

For future work, we will also consider application approaches that use both Spark SQL and Spark Join. We will also explore whether we can generalize our current findings by considering more big data applications.

Acknowledgment

This work is supported in part by the NSF Grant #1730250: CyberTraining: DSE: Cross-Training of Researchers in Computing, Applied Mathematics and Atmospheric Sciences using Advanced Cyberinfrastructure Resources. We also thank the Microsoft Azure for Research Grant from Microsoft for Azure usage credit.

References

[1] Apache Spark. http://spark.apache.org/.

[2] Hive. http://hadoop.apache.org/hive/.

[3] MongoDB Documentation. https://docs.mongodb.com/.

[4] Apache HBase. https://hbase.apache.org/.

[5] Hadoop. http://hadoop.apache.org/.

[6] J. Gudmundsson, M. V. Kreveld, "Computing longest duration flocks in trajectory data," 14th ACM International Symposium on Geographic Information Systems, ACM-GIS 2006, November 10-11, 2006, Arlington, Virginia, USA, Proceedings. 2006:35-42.

[7] H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, H. T. Shen, "Discovery of Convoys in Trajectory Databases," Proceedings of the Vldb Endowment, 2010, 1(1):1068-1080.

[8] Z. H. Li, B. L. Ding, J. W. Han, R. Kays, "Swarm: Mining Relaxed Temporal Moving Object Clusters," Submission, 2010, 3(12):723- 734.

[9] L. A. Tang, Y. Zheng, J. Yuan, J. W. Han, "A Framework of Traveling Companion Discovery on Trajectory Data Streams," Acm Transactions on Intelligent Systems & Technology, 2013, 5(1):992- 999.

[10] Z. Zhao, W. Ding, J. Wang and Y. Han, "A Hybrid Processing System for Large-Scale Traffic Sensor Data," in IEEE Access, vol. 3, pp. 2341-2351, 2015.

[11] M. Zhu, C. Liu, J. Wang, X. Wang and Y. Han, "A Service-Friendly Approach to Discover Traveling Companions Based on ANPR Data Stream," 2016 IEEE International Conference on Services Computing (SCC), San Francisco, CA, 2016, pp. 171-178.

[12] J. Dean, S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in: Brewer E, Chen P, eds. Proc. of the OSDI. California: USENIX Association, 2004, pp. 137−150.

[13] Tuning Spark. https://spark.apache.org/docs/2.1.0/tuning.html/.

[14] Spark SQL & DataFrames. https://spark.apache.org/sql/.

[15] Microsoft Azure. https://azure.microsoft.com/en-us/.

[16] HDInsight — Hadoop, Spark, and R Solutions for the Cloud. https://azure.microsoft.com/en-us/services/hdinsight/.

[17] M. Armbrust, R. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational Data Processing in Spark," in Proc. of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 1383-1394.

[18] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, and S. Madden, "A comparison of approaches to large-scale data analysis," in Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, 0ADAD, pp. 165–178.

[19] T. Ivanov and M.-G. Beer, "Performance Evaluation of Spark SQL Using BigBench," in Big Data Benchmarking, vol. 10044, pp. 96–116.

[20] Y. Li and S. Manoharan, "A performance comparison of SQL and NoSQL databases," 2013 IEEE Pacific Rim Conference on

Communications, Computers and Signal Processing (PACRIM), Victoria, BC, 2013, pp. 15-19.

[21] A. Nayak, A. Poriya, and D. Poojary, “Type of NOSQL Databases and its Comparison with Relational Databases,” International Journal of Applied Information Systems, vol. 5, no. 4, pp. 16–19, 2013.

[22] E. Dede, Z. Fadika, M. Govindaraju, and L. Ramakrishnan, “Benchmarking MapReduce implementations under different application scenarios,” Future Generation Computer Systems, vol. 36, pp. 389–399, Jul. 2014.

[23] J. Wang, D. Crawl, I. Altintas, K. Tzoumas, V. Markl, “Comparison of Distributed Data-Parallelization Patterns for Big Data Analysis: A Bioinformatics Case Study”. In Proceedings of The Fourth International Workshop on Data Intensive Computing in the Clouds (DataCloud) 2013.

[24] Apache Storm. http://storm.apache.org/.

[25] Hortonworks Documentation. https://docs.hortonworks.com/.

[26] Apache Impala. https://impala.apache.org/.

APPENDIX

A. *SQL for companions-for-all-vehicles with $\delta_c = 2$, $\delta_t = 300$* seconds.

-- Create table and load data.

```
CREATE TABLE IF NOT EXISTS traffic(camera STRING,
vehicle STRING, time Int)
LOAD DATA INPATH '11-1.txt' INTO TABLE traffic;
```

-- Find all records that contains more than one vehicle time pair passing through a certain camera.

```
CREATE VIEW view1 AS
SELECT camera, vehicle, time
FROM traffic
WHERE camera IN (SELECT camera FROM traffic GROUP
BY camera HAVING COUNT(vehicle) >= 2);
```

-- Find vehicle pair, cam, time pair with Time Threshold δ_{t} *applied.*

```
CREATE VIEW view2 AS
SELECT a.vehicle, b.vehicle AS vehcomp, a.camera, a.time,
b.time AS timecomp
FROM view1 a, view1 b
WHERE ((a.time - b.time <= 300 AND a.time - b.time >= -
300)) AND (a.camera = b.camera);
```

-- Create final table vehpair, camera, timepair.

```
CREATE VIEW view3 AS
 SELECT CONCAT(vehicle, ',', vehcomp) AS vehpair,
camera, CONCAT(time,',',timecomp) AS timepair
 FROM view2
 WHERE (vehicle > vehcomp);
```

-- Return exact travel companions filtered by camera threshold δ_c and save results to HDFS.

```
INSERT OVERWRITE LOCAL DIRECTORY
'/allHiveHadoop'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
SROTRED AS textfile
SELECT vehpair, COLLECT_LIST(camera),
COLLECT_LIST(timepair)
FROM view3
GROUP BY vehpair
HAVING COUNT(camera) >= 2;
```

B. *SQL for companions-for-one-vehicle with $\delta_c = 2$, $\delta_t = 300$ seconds, target vehicle v^*.*

-- Create table and load data.

```
CREATE TABLE IF NOT EXISTS traffic(camera STRING,
vehicle STRING, time Int)
LOAD DATA INPATH '11-1.txt' INTO TABLE traffic;
```

-- Find cameras based on v from database.*

```
CREATE VIEW view1 AS
SELECT camera, vehicle, time
FROM traffic
WHERE vehicle = v*;
```

-- Find all vehicles from database that passed one camera in camera set right before or after v^* *passing the camera within time threshold δ_t.*

```
CREATE VIEW view2 AS
SELECT a.camera, a.vehicle, a.time
FROM traffic a, view1 b
WHERE ((a.time - b.time <= 300 AND a.time - b.time >= -
300)) AND (a.camera = b.camera);
```

-- Create the vehicle pair table.

```
CREATE VIEW view3 AS
SELECT a.vehicle vehcomp, b.vehicle, a.camera, a.time
vehcomptime, b.time
from view2 a, view1 b
where a.camera = b.camera and a.vehicle <> b.vehicle;
```

-- Create final table vehpair, camera, timepair.

```
CREATE VIEW view4 AS
SELECT concat(vehcomp,',',vehicle) AS vehpair, camera,
CONCAT(vehcomptime,',',time) AS timepair
from view3;
```

-- Groupby vehicle pair and filter by camera threshold δ_c and save results to HDFS.

```
INSERT OVERWRITE LOCAL DIRECTORY
'/allHiveHadoop'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
SROTRED AS textfile
SELECT vehpair, COLLECT_LIST(camera),
COLLECT_LIST(timepair)
FROM view4
GROUP BY vehpair
HAVING COUNT(camera) >= 2;
```