

Applying Stratosphere for Big Data Analytics

J V P S Avinash Akshay Deshpande

Abstract: Big Data refers to various forms of large information sets coming from different data sources that requires special computational platforms in order to be analyzed. Timely and cost-effective analytics over Big Data is a key ingredient for various use-cases now-a-days. Web search engines and social networks capture and analyze every user action on their sites, Telephonic information are needed by the exchange department for the call logs, retrieving the past user information can be cited as a best use-cases for varying datasets. Hadoop comes with an implementation of MapReduce which is a programming model and a popular choice for analytics over Big Data. Unfortunately, Hadoop's performance out of box leaves much to be desired, leading to suboptimal usage of resources with respect to complex operations like JOIN, CROSS, GROUPS etc. In this paper, we present a topic on StratoSphere, The Next-Gen Data Analytics Platform, where Big Data Analytics look tiny here. The basic building block of StratoSphere is MapReduce, we try to forecast various complex operators that involves less computational time than in traditional MapReduce. Using examples, we show how to formulate analytical tasks as Meteor queries and execute them with StratoSphere.

1. Big Data

We are in the Big Data era - the cost of hardware and software for storing data, accelerated by cloud computing, has enabled the collection and storage of huge amounts of data. It is proper to define Big Data in terms of 3V's – **Variety**, **Velocity** and **Volume** (Figure 1).

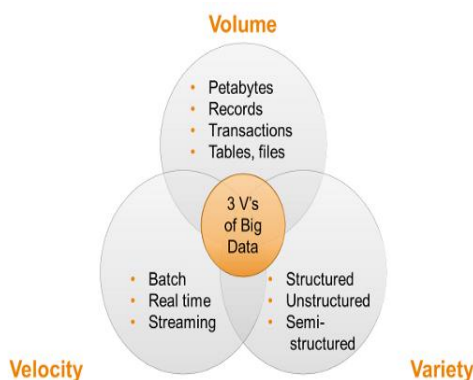


Figure 1. The three Vs of big data

Volume: Data exists in various structured and unstructured data formats consisting of videos, music's and larger images coming from various sources. These compromise of terabytes and petabytes of the storage system for archiving them. These data may grow over the time and increase in size. These big volumes indeed represent **Big Data**.

Velocity: Data is being updated every second and thereby the data growth have become very prominent. The data movement is now almost real time coming from various batch and streaming process. This high velocity data represent **Big Data**.

Variety: Data can be stored in multiple formats. The formats might include structured data like tables, unstructured data like video, and semi-structured data like xml. These variety of data represent **Big Data**.

2. Hadoop File System Metadata

Hadoop is a software framework for distributed processing of large datasets across large clusters of computers. Large datasets refer to terabytes or petabytes of data and clusters refer to hundreds or thousands of nodes. Hadoop employs a master/slave architecture for both distributed storage and distributed computation. Hadoop is based on a single programming model called MapReduce and a file system to store data called Hadoop Distributed File System (Figure 2).



Figure 2. Hadoop Compositions

Hadoop Distributed File System (HDFS) is a file system designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware.

HDFS is built around the idea that the most prominent data processing pattern is a write-once, read-many-times pattern. Hadoop does not require expensive, highly reliable hardware. It is designed to run on the clusters of commodity hardware. HDFS creates multiple replicas of data blocks and distributes them on computer nodes throughout the cluster to enable reliable computations. HDFS data block is usually 64MB or 128MB. Each block is replicated multiple times (default = 3) and stored on different data nodes.

Hadoop mainly includes five daemons: - NameNode, DataNode, Secondary NameNode, JobTracker, TaskTracker. **NameNode** manages the filesystem namespace. It maintains the filesystem tree and the metadata for all the files and directories in the tree. The namenode also knows the datanodes on which all the blocks for a given file are located.

DataNode daemon performs the main task of reading or writing a HDFS blocks. The file is broken into blocks and the NameNode will tell us which DataNode each blocks resides in. A DataNode may communicate with other DataNodes to replicate its data blocks for redundancy.

Secondary NameNode (SNN) is an assistant daemon for monitoring the state of the HDFS cluster. The SSN doesn't receive or record any real-time changes to HDFS. Instead, it communicates with the NameNode to take snapshots of the HDFS metadata at intervals.

JobTracker is an interface between our application and Hadoop. Once the job is submitted through cluster, the JobTracker determines the execution plan by determining which files to process, assign nodes to different task trackers, and monitor them.

Each **TaskTracker** is responsible for executing the individual tasks that the JobTracker assigns. This is mainly responsible for running Map Reduce over the data.

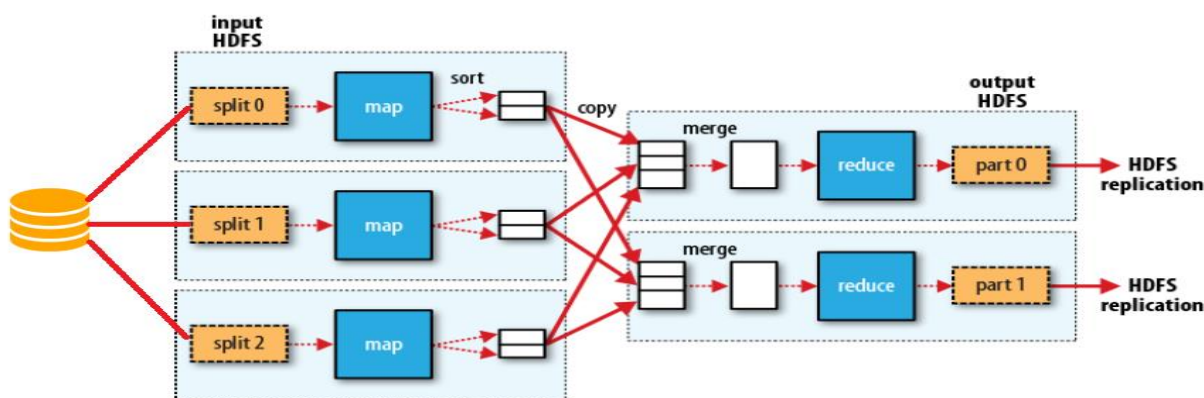


Figure 3 . A typical Map Reduce Paradigm

3. Hadoop – MapReduce

MapReduce is a programming model for parallel data processing. MapReduce works by breaking the processing into two phases: the *map* phase and the *reduce* phase. Each phase has key-value pairs as input and output. The difference between this approach and traditional ware houses are below.

	Traditional RDBMS	MapReduce
Data Size	Gigabytes	Petabytes
Access	Interactive and batch	Batch
Updates	Read and Write many times	Write once, read many times.
Structure	Static Schema	Dynamic Schema
Scaling	Non Linear	Linear

Table 1. RDBMS vs MapReduce

Table 1 demonstrates the map reduce framework. The input to the application must be a list of (key/value) pairs, list (<k1, v1>). This list of (key/value) pairs is broken up and each individual (key/value) pair, is processed by calling a map function of the mapper. The mapper transforms each of <k1, v1> pair into a list of <k2, v2> pairs. The output of all mappers are aggregated into one giant list of <k2, v2> pairs. All pairs sharing the same key k2 are grouped together into a new (key/value) pair, <k2, list (v2)> pair, <k2, list (v2)>. The framework asks the reducer to process each one of the aggregated (key/value) pairs individually. The MapReduce framework automatically collects all the pairs (<k3, v3> in below example) and writes them to file(s). The below pseudo

code explains the working of the above process for word count.

Pseudo Code for Map function		Pseudo Code for Reduce function	
Input	<k1,v1>	Input	<k2,list(v2)>
<pre>map(String filename, String document){ List<String> T = tokenize(document); for each token in T{ emit((String) token , (Integer) 1); } }</pre>		<pre>reduce(String token , List<Integer> values){ Integer sum = 0; for each value in values{ sum = sum + value; } emit((String) token , (Integer) sum); }</pre>	
Output	list(<k2>,<v2>)	Output	list(<k3,v3>)

3.1. MapReduce - Joins

Now let us discuss a simple JOIN using Map Reduce. The workflow relates the number of pages liked by users individually. We consider two data sources Users and Pages Liked. The *group key* functions like a join key in a relational database. Mappers receive data from two files and map() function is called with each record. For joining, we want the map() function to output a record package where the key is the group key for joining – user ID in this case. After map() packages each record of the inputs, data is partitioned across Reducers based on same key. Now the reduce() method will process all records of the same join key together. The function reduce() will take its input and do a full *cross-product* on the values. It feeds each combination from the cross-product into a function called combine(). It's the combine() function that determines whether the whole operation is an inner join, outer join, or others.

Join operation between one large and many small data sets can be done using Broadcast Join performed in the map-side. This process completely eliminates the need to shuffle any data to the reduce phase. All the small data sets are essentially read into memory during the setup phase of each map task, which is limited by the JVM heap. Join is entirely done in the map phase, with the large data set being input for the MapReduce job and must also be in the *left* part of any join operation. The mapper is responsible for reading all the files from the distributed cache during the setup phase and storing them into in-memory lookup tables. Mapper processes each record, joins it with all the data stored in-memory. The above steps are shown in Figure 4(a) and Figure 4(b) given below.

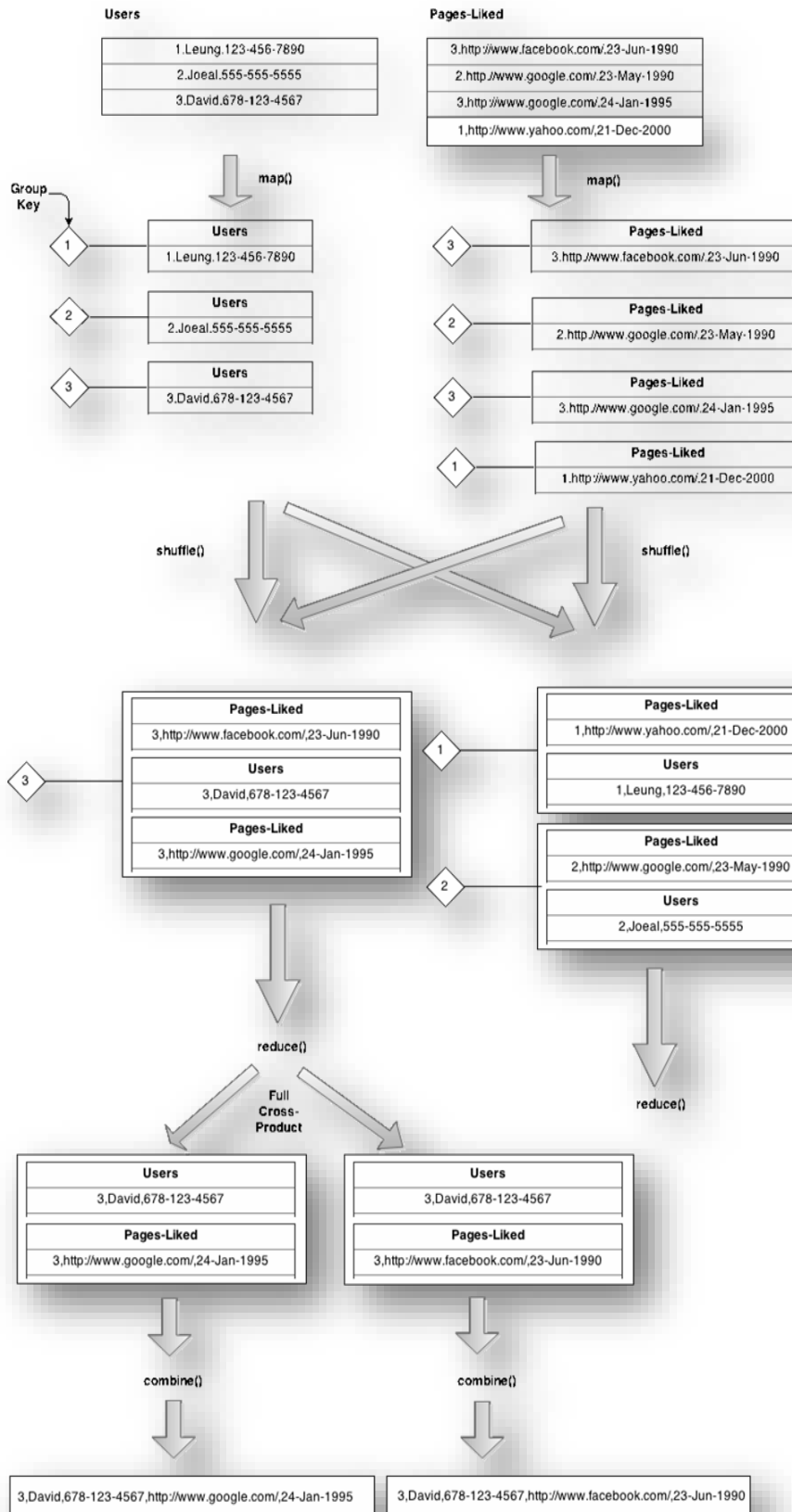


Figure 4 (a). Reduce-Side Join Using Map Reduce

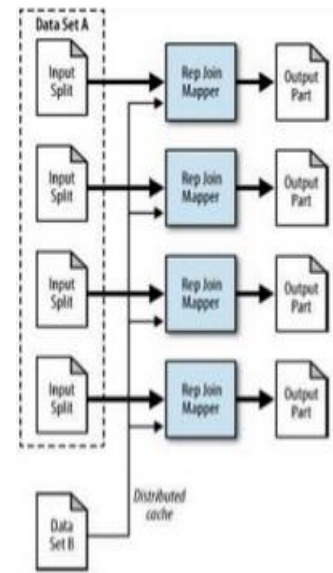


Figure 4 (b). Iterative (/Replicative) join using Map Reduce

Figure 4(a) illustrates a typical Map Reduce example for a shopping cart website where all the pages liked by the respective users need to be obtained. This includes a simple Reduce-Side Join.

Figure 4(b) describes the Join with iterative calls using Map Reduce. Mapper does a replicated join each and every time and the result is sent to Reducer.

4. StratoSphere

Stratosphere is a data analytics stack that enables the execution, analysis and integration of heterogeneous data sets, ranging from strictly structural relational data to unstructured text data and semi-structured data. Stratosphere can perform information extraction and integration, traditional data warehousing analysis, model training, and graph analysis using a single query processor, compiler and optimizer. Stratosphere uses an execution engine that includes external memory query processing algorithms and natively supports arbitrary long programs shaped as Directed Acyclic Graph (DAG). Stratosphere offers both pipeline (inter-operator) and data (intra-operator) parallelism.

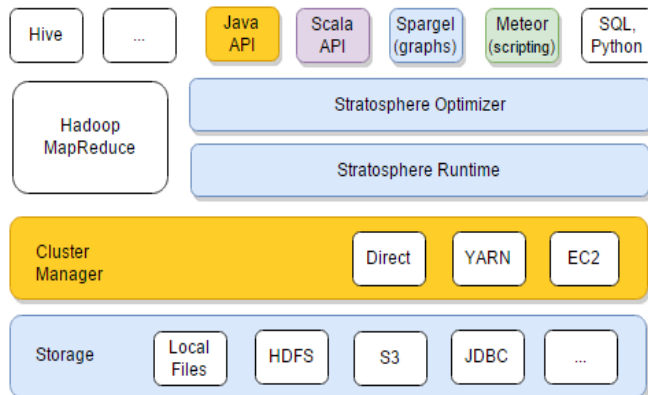


Figure 5. Stratosphere Stack

Figure 5 depicts the Stratosphere stack with its components illustrated. Stratosphere has its data located in various source formats like HDFS, Databases and local files. It can be deployed in a Hadoop/YARN Cluster forming the base for MapReduce execution. The optimizer is used to decide cost-based logical and physical plans. The jobs are queried using tools like Hive (SQL like programming on Hadoop), Java / Scala API, Spargel (Graph executions), Meteor (JAQL

like scripting language on HDFS). Now, let's discuss Stratosphere Architecture.

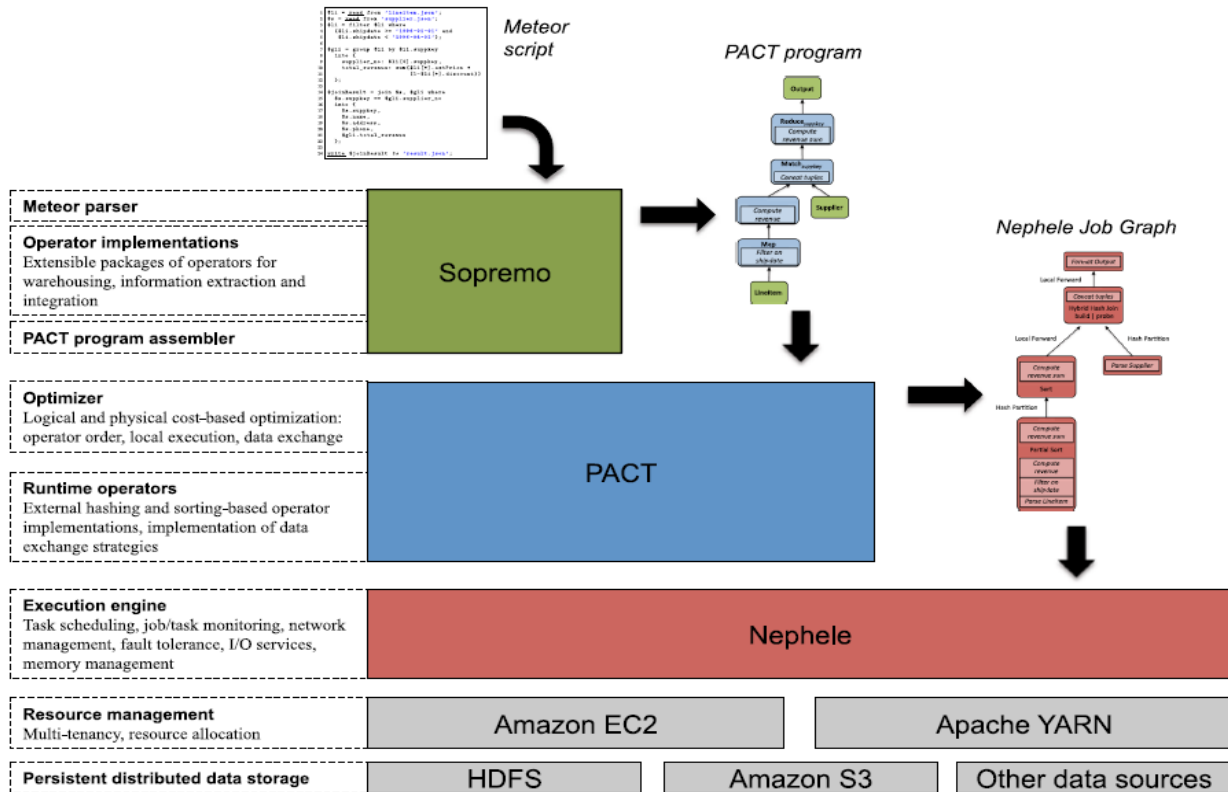


Figure 6. Stratosphere Architecture

4.1 Sratosphere Architecture

The Stratosphere architecture consists of three layers, termed the *Sopremo*, *PACT* and *Nephele* layers. Each layer is defined by its own programming model and a set of components that have certain responsibilities in the query processing pipeline. Figure 6 demonstrates the basic idea of the layers.

Sopremo is the top most layer consisting of a set of logical operators connected in a Directed Acyclic Graph (DAG), similar to a logical query plan in relational DBMSs. Programs for the Sopremo layer can be written in **Meteor**, an operator-oriented query language that uses a JSON-like data model to support the analysis of unstructured and semi-structured data. Meteor has same features and is similar to Hadoop scripting languages like PIG, JAQL.

Once a Meteor script has been submitted to Stratosphere, the Sopremo layer first translates the script into an operator plan. Moreover, the compiler within the Sopremo layer can derive several properties of a plan, which can later be exploited for the physical optimization of the program in the subjacent PACT layer.

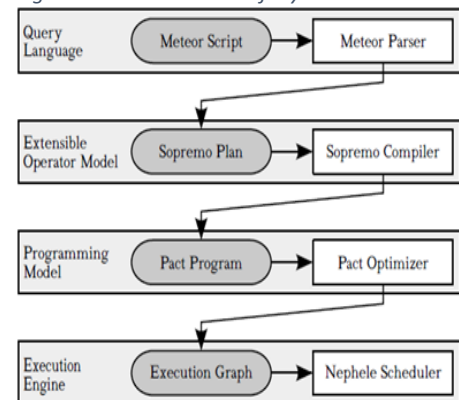
The output of the Sopremo layer, and at the same time, input to the *PACT* layer is a PACT program. PACT programs are based on the PACT programming model, an extension to the MapReduce programming model. Similar to MapReduce, the PACT programming model builds upon the idea of second-order functions, called PACTs.

Each PACT provides a certain set of guarantees on what subsets of the input data will be processed together, and the first-order function is invoked at runtime for each of these subsets. In addition with Map and Reduce features, PACT also offers additional operators which will be discussed later. Choosing the cheapest of those data reorganization strategies is the responsibility of a special cost-based optimizer, contained in the PACT layer. Similar to classic database optimizers, it computes alternative execution plans and eventually chooses the most preferable one.

The output of the PACT compiler is a parallel data flow program for *Nephele*, the third layer, Stratosphere's parallel execution engine. Similar to PACT programs, Nephele data flow programs, also called Job Graphs, are also specified as DAGs with the vertices representing the individual tasks and the edges modeling the data flows between those. However, in contrast to PACT programs, Nephele Job Graphs contain a concrete execution strategy, chosen specifically for the given data sources and cluster environment.

Nephele itself executes the received Job Graph on a set of worker nodes. It is responsible for allocating the required hardware resources to run the job from a resource manager, scheduling the job's individual tasks among them, monitoring their execution, managing the data flows between the tasks, and recovering tasks in the event of execution failures. During the execution of a job, Nephele can collect various statistics on the runtime characteristics of each of its tasks, ranging from CPU and memory consumption to information on data distribution. The collected data are centrally stored inside Nephele's master node and can be accessed, for example, by the PACT compiler, to refine the physical optimization of subsequent executions of the same task. Figure 7 short-lists the main steps of each layer.

Figure 7. Execution Plan of layers with Meteor



Stratosphere provides support for the popular Hadoop distributed file system and the cloud storage service Amazon S3, as well as for Eucalyptus. We plan to support multi-tenancy by integrating Stratosphere with resource management systems, such as Apache YARN. Moreover, Stratosphere can directly allocate hardware resources from infrastructure-as-a-service clouds, such as Amazon EC2.

4.2 Stratosphere Operators

In addition to Map and Reduce, Stratosphere also includes extra operators, described in Table 2 below.

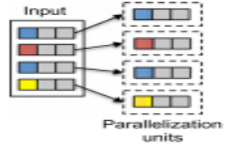
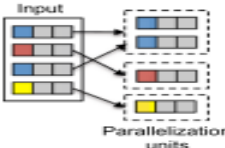
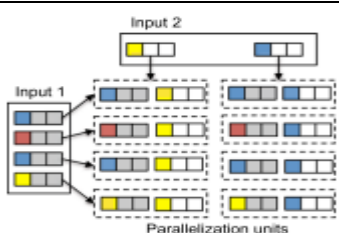
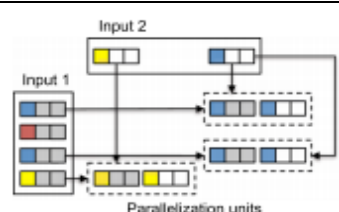
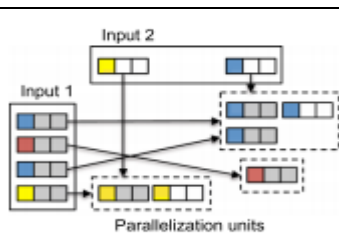
MAP	Record-at-a-time		Accepts a Single Record as Input, Emits Any number of Records, Applications:- Filters/Transformations.
	One Input		
REDUCE	Group-at-a-time		Groups the record of its input on Record Key. Accepts a list of records as Input, Emits any number of records, Applications:- Aggregations
	One Input		
JOIN	Record-at-a-time		Joins both inputs on their Record keys and non-matched records are discarded. Accepts one record of each input, Emits any number of Records , Applications:- Equi-Joins
	Two Inputs		
CROSS	Record-at-a-time		Cartesian Product of the records of both inputs. Accept one record of each input, Emits any number of records, A Very Expensive Operations.
	Two Inputs		
CO-GROUP	Group-at-a-time		Groups the record of its input on Record Key. Accepts One list of records for each input, Emits any number of records.
	Two Inputs		
UNION	Record-at-a-time	Merges two or more input data sets into a single output data set.	Follows Bag Semantics. Duplicates are not removed.
	Two Inputs		

Table 2. Basic Operators in StratoSphere

4.2 Sratosphere – The Meteor Query Language

Meteor is an operator-oriented query language that focuses on analysis of large-scale, semi- and unstructured data. Users compose queries as a sequence of operators that reflect the desired data flow. The internal data is built upon JSON, but Meteor supports additional input and output formats.

We take an example (Listing 1) where a county school-board wants to find out which of its teen-aged students carry out voluntary work and thus have appeared in recent news articles.

This example involves operators from the domains (packages) of data cleansing and information extraction. The first two lines imports packages cleansing and IE. All the operators in this packages can be accessed by the program. Line 4 and 12 reads the spreadsheet of students and news into variables **\$students** and **\$articles** with that dataset. Line 5, 6, and 7 are used to remove duplicate in students who enrolled for more than one school. This is calculated by *Levenshtein* Distance measure. Line 9 and 10 filters all students that have age less than 20. Line 14-18 joins the two variables containing the filtered datasets on person name attribute and the results are written to a file.

Figure 8 depicts the Sopremo Plan that meteor returns for our example. Sopremo is a framework to manage an extensible collection of semantically rich operators organized into packages. It acts as a target for the Meteor parser and ultimately produces an executable Pact program. Relational Operators co-exist with application-specific operators, e.g., data cleansing and information extraction operations such as *remove duplicates*. All variables in Meteor script are replaced by edges, which represent the flow of data.

Operators may have several properties, e.g., the *remove duplicates* operator has a similarity measure and a threshold as properties. The values of properties belong to a set of expressions that process individual Sopremo values or groups thereof. These expressions can be nested to form trees to perform more complex calculations and transformations. For example, the selection condition of the example plan compares the year of the calculated age with the constant 20 for each value in the data set **\$students**.

Analysis Program

Listing 1. Example of Meteor

```
1. using cleansing ;
2. using ie;
3.
4. $students = read from 'students.csv' ;
5. $students = remove duplicates $students
6.           where average ( levenshtein(name) , dateSim ( birthDay ) ) > 0.95
7.           retain maxDate ( enrollmentDate ) ;
8.
9. $teens = filter $stud in $students
10.          where (now() - $stud.birthDay).year < 20;
11.
12. $articles = read from 'news.json' ;
13.
14. $teensInNews = join $teen in $teens , $person in $articles
15.                 where $teen.name == $person.name
16.                 into {
17.                     student : $teen , articles : $person.articles [*].url
18.                 };
19.
20. write $teensInNews to 'result.json' ;
```

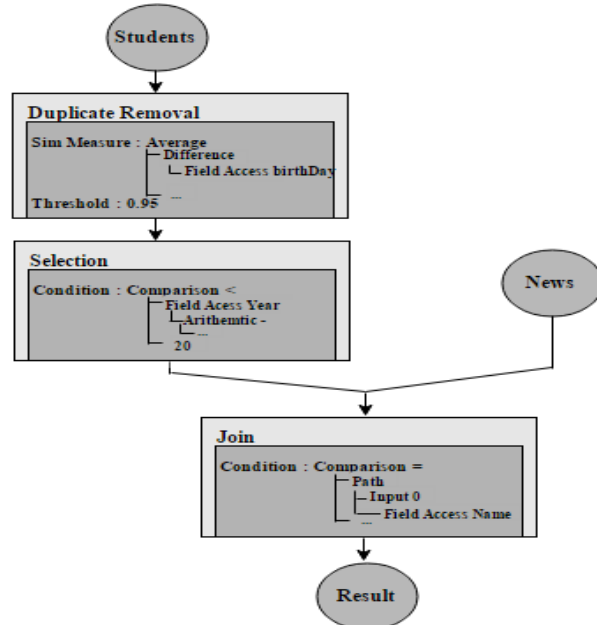


Figure 8. Sopremo Plan for Meteor Example

4.2 Application of Stratosphere to Web Log Analysis

In this demonstration, we visualize the phases in analyzing Web Log Data using Stratosphere: their specifications, optimization, execution, and their scheduling. For this analysis, we consider three datasets: - *documents* (containing URL and description), *visits* (containing IP address, URL, date, time, etc.) and *ranks* (URL, ranks). The data is loaded into HDFS/Local File System. The pseudo code for Web Log Analysis is done in JAVA and a JAR file is created. The JAR is uploaded in the query interface as shown in Figure 9. The default plan for executing the JAR is also shown in the figure.

The interface displays a list of JAR files on the left, including 'stratosphere-java-examples-0.4-hadoop2-KMeansIterative.jar', 'stratosphere-java-examples-0.4-hadoop2-KMeansSingleStep.jar', 'stratosphere-java-examples-0.4-hadoop2-TPCHQuery3.jar', 'stratosphere-java-examples-0.4-hadoop2-WebLogAnalysis.jar', 'stratosphere-java-examples-0.4-hadoop2-WordCount.jar', 'stratosphere-java-examples-0.5.2-KMeans.jar', and 'stratosphere-java-examples-0.5.2-WebLogAnalysis.jar'. The central area shows a query plan diagram with nodes like 'Filter', 'Map', 'Join', 'CoGroup', and 'Print to System.out'. The bottom section includes fields for 'Show or launch selected PACT program...' and 'Select a new PACT program to upload...', along with checkboxes for 'Show optimizer plan' and 'Suspend execution while showing plan'.

Figure 9. StratoSphere Query Interface

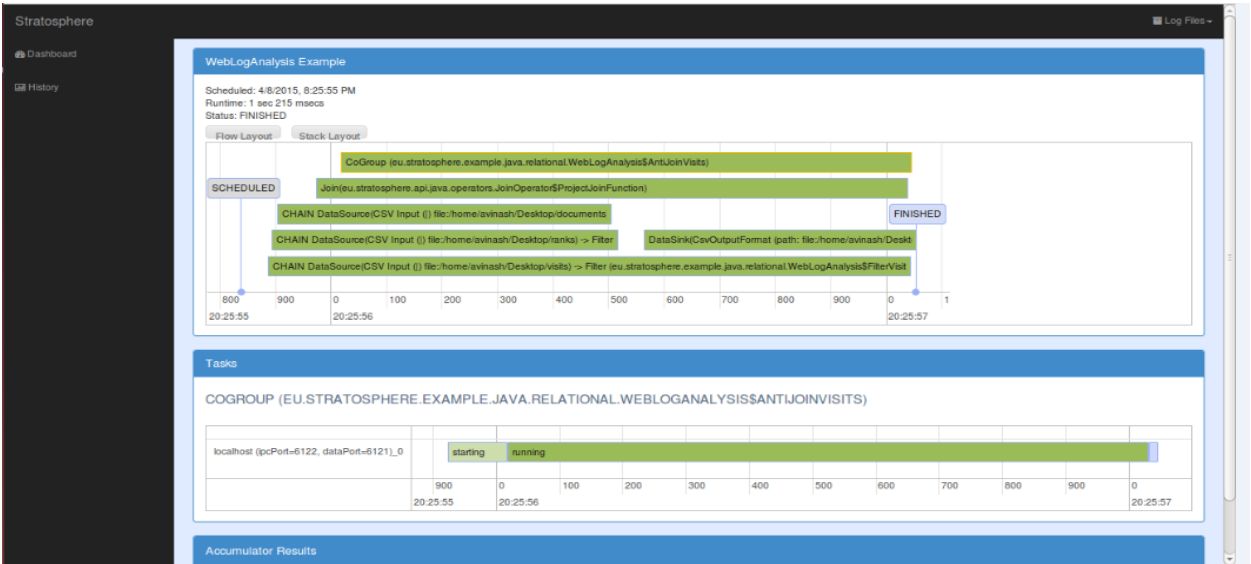
The interface also accepts the input parameters i.e., the source file locations and destination file location. After the details are entered and the job is run, the compiler generates a best plan (shown in Fig 10) based on the given conditions. Also, the Parser checks for valid file paths. The compiler also gives the entire PACT, Global Data and Local Data properties along with cost estimation.

The figure displays a detailed query plan diagram with nodes like 'CSV Input', 'Filter', 'Map', 'Join', 'CoGroup', and 'CecOutputFormat'. Below the diagram is a table with properties and cost estimates.

PACT Properties	Global Data Properties	Local Data Properties	Size Estimates	Cost Estimates
Operator: None	Partitioning: RANDOM	Order: (none)	Est. Output Size: 399.9 KB	Network: 0.0 B
Parallelism: 1	Partitioning Order: (none)	Grouping: not grouped	Est. Cardinality: 3.1 K	Disk I/O: 399.9 KB
Subtasks-per-instance: 1	Uniqueness: not unique	Uniqueness: not unique		CPU: 0.0
				Cumulative Network: 0.0 B
				Cumulative Disk I/O: 399.9 KB
				Cumulative CPU: 0.0

Figure 10. Optimizer created execution: Nephele and PACT Query Plan

After the job is submitted to the query interface, Stratosphere itself generates a Scheduler task for the current job execution. This is shown in the Figure 11 and 12 below. The task is executed in five parallel processes. The first three process involves in loading the DataSources loaded to Stratosphere. Filter conditions are applied on these data sources. The fourth process involves JOIN of three data sources and the fifth forms a CO-GROUP of these data sources and the output acts as a Data Sink.



To express this in Joins, we need two successive jobs. The first MR job preforms an inner-join, the second one an anti-join. The first Map task processes the input relations Documents d, Rankings r and carries out the specific condition to filter the record tuples. To associate a tuple with its source relation, the resulting value is augmented with a lineage tag. The subsequent reducer collects all tuples with equal key, forms two sets of tuples based on the lineage tag and forwards all tuples from r only if a tuple from d is present. The second mapper acts as an identity function on the joined intermediate result j and as a selection on the relation Visits v. Finally, the second reducer realizes the anti-join by only emitting tuples (augmented with a lineage tag 'j') when no Visits tuple (augmented with a lineage tag 'v') with an equal key is found. This makes MR Join more complex.

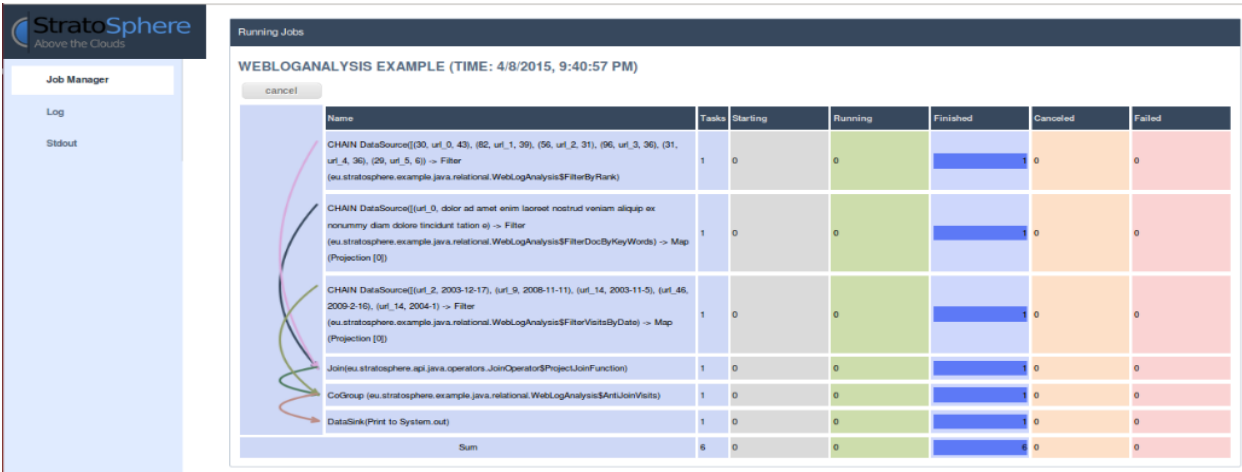


Figure 12. Job Manager of StratoSphere for Web Log Analysis

4. Feature Matrix of StaratoSphere (PACT Model) and MapReduce

The PACT programming model is a generalization of MapReduce, providing additional second-order functions, and introducing output contracts. The following show PACTs advantages over MapReduce:

- The PACT programming model encourages a more modular programming style. Although often more user functions need to be implemented, these have much easier functionality. Hence, interweaving of functionality which is common for MapReduce can be avoided.
- PACT frequently eradicates the need for auxiliary structures, such as the distributed cache which “brake” the parallel programming model.
- In MapReduce, data organization operations such as building a Cartesian product or combining pairs with equal keys must be provided by the developer of the user code. In PACT, they are done by the runtime system.
- Finally, PACT’s contracts specify data parallelization in a declarative way which leaves several degrees of freedom to the system. These are an important prerequisite for automatic optimization - both a-priori and during runtime.

Table 3 below summarizes the differences.

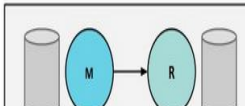

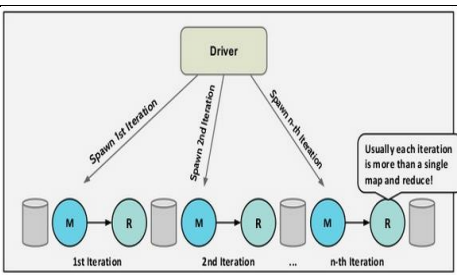
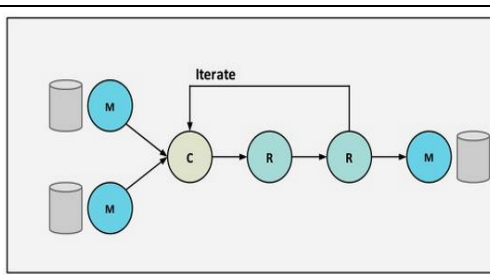
	Map Reduce	StratoSphere
Operators	Map , Reduce	Map, Reduce, Cross, Join, CoGroup, Union, Iterate, Filter, Transform, Intersect, Subtract, Replace, Pivot, Split
Compositions	Only MapReduce	Arbitrary Data Flows
Data Exchange	Batch through disk	Pipe-lined, in-memory (automatic spilling to disk)
Data Flows	 <p>Apache Hadoop MR is limited to one data flow</p>	 <p>One of many possible data flows in Stratosphere</p>
Iterations	 <p>Loop is outside the program</p> <ul style="list-style-type: none"> • Hard to program since each iteration is spawn to individual Mapper and Reducer. • Very Poor Performance 	 <p>Loop is inside the program</p> <ul style="list-style-type: none"> • Easy to program since in each iteration, a Reducer calls a Combiner to iterate. • Huge Performance gains.

Table 3. Feature Matrix of StratoSphere and MapReduce

6. Conclusions

We presented a comparison of several analytical tasks in their MapReduce and PACT implementations. We demonstrated Meteor, a declarative scripting language influenced by Jaql, to express sophisticated data flows. Sopremo, the underlying operator layer provides a modular and highly extensible set of application-specific operators. Currently, pre-defined relational operators as well as packages for IE and DC are available. We have shown that operators from these packages can be easily used together in a single Meteor program to implement advanced use cases that require functionality from both packages.

We also demonstrated the techniques used in Stratosphere to efficiently execute various operators on large data sets. Stratosphere is a very flexible data flow system with dedicated support for various complex analysis, optimizing plans and scheduling tasks. A distinguishing aspect is its abstraction for workset iterations, which allow it to be efficient over MapReduce.

7. References

- [1] Stratosphere Project. <http://stratosphere.eu/>, 2013.
- [2] Alexander Alexandrov, Dominic Batre, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. Massively Parallel Data Analysis with PACTs on Nephel. PVLDB, 3(2):1625–1628, 2010
- [3] S. Ewen, S. Schelter, K. Tzoumas, D. Warneke, and V. Markl. Iterative parallel data processing with stratosphere: An inside look. SIGMOD, 2013.
- [4] Apache Hadoop. URL: <http://hadoop.apache.org>.
- [5] DOPA Project. URL: <http://www.dopa-project.eu/dopa/uploads/DOPA-WP3-D1.pdf>
- [6] Stephan Ewen, Sebastian Schelter, Kostas Tzoumas. Iterative Parallel Data Processing with Stratosphere: An Inside Look
- [7] Alexander Alexandrov, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, Daniel Warneke. MapReduce and PACT - Comparing Data Parallel Programming Models
- [8] Christoph Boden, Volker Markl, Marcel Karnstedt, Miriam Fernandez. Large-Scale Social-Media Analytics on Stratosphere
- [9] Big Data looks Tiny from the Stratosphere. By Author Volker Markl.
- [10] Stephen Ewen, Fabian Hüske, Daniel Warneke, Kostas Tzoumas, Joe Harjung .The Stratosphere System Parallel Analytics beyond MapReduce.
- [11] Arvid Heise, Astrid Rheinländer, Marcus Leich, Ulf Leser, Felix Naumann. Meteor/Sopremo: An Extensible Query Language and Operator Model.
- [12] Stephan Ewen - Stratosphere Next-Gen Data Analytics Platform.
- [13] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag-The Stratosphere platform for big data analytics.