International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 9 Issue 06, June-2020

Compilers used in the Stratosphere Platform for Big Data Analytics

Chhandak Bagchi, Khushbu Chopra, Purva Singh, Dr. Rajasekhara Babu M
School of Computer Science and Engineering
VIT University
Vellore-632014, India

***Abstract*— Big data analytics requires processing and analysis of Terabytes or even Petabytes of data. In such situations, it is important that the data processing and analysis is carried out efficiently. In the past, many query processing languages have been developed for MapReduce systems (like Hadoop, Asterix and Spark), which helps in accomplishing these complex tasks easily in a user interactive manner. The existing systems are inefficient in executing iterative programs and promoting abstraction. The Stratosphere platform provides features like data processing, a declarative query language, automatic program parallelization and optimization, support for iterative programs and a scalable and efficient execution engine. In this paper we talk about compilers in the stack organization of the Stratosphere platform with an extensive outlook on its semantically rich operator model Sopremo and Meteor, an extensible query language embedded in the Stratosphere stack. Parallelization on the Stratosphere framework is achieved through the Nephele/PACTs system, PACTs (Parallelization Contracts) being parallel data processor which works with its execution engine, Nephele.**

Index Terms*— Abstraction, Big data, Parallel databases, Cloud Computing, Compilation, Data cleansing, Distributed systems, Graph processing, Query Optimization, Query Processing, Stack, Text mining.

I. INTRODUCTION

BY the expansion and universal acceptance of cloud computing, the cost of hardware and software for storing data has been greatly reduced. This had enabled huge collection and storage of data on the cloud. Commercial RDBMS products cannot cope with the scale and diversity of the collected data sets. This has led to the revaluation of the existing methods for managing and processing data at such a large scale, leading to new software developments.

In this paper, our main focus is on Stratosphere [4], a data analytics stack that enables the extraction, analysis, and integration of semi-related and unrelated heterogeneous data sets, ranging from strictly structured relational data to unstructured text data and semi-structured data. The Stratosphere system provides a structured query language, a compiler and an optimizer to perform information extraction and integration, traditional data warehousing analysis, model training, and graph analysis.

For relational data sets, data flow systems as a generalization of the Map-Reduce programming model [2] have gained much attention as they provide an easy way of writing scalable code for analytical tasks on huge amounts of data. However, developing data flow programs for non-relational workloads like information extraction, data mining and processing can become quite complex. Map-Reduce programming model and its implementation Hadoop has been used in various simple log file analysis, executes code in a fault tolerance manner. But when more complex operations are needed to be handled, it doesn't satisfy the design goals of the model and leads to expensive performance penalties. Hence, an alternative approach of Stratosphere is discussed in this paper.

Meteor and Sopremo [8], is an extensible query language and a semantically rich operator model for the Stratosphere data flow system. Meteor is an operator-oriented query language that emphasis on analysis of large-scale, semi- and unstructured data. Users compose queries as a sequence of operators that reflect the desired data flow. Sopremo is an execution engine that manages packages of extensive operators. This framework works as a Meteor parser that produces an executable PACT program. Sopremo provides a programming framework that allows users to define custom packages, the respective operators and their instantiations. Sopremo already contains a set of base operators from the relational algebra, plus two sets of additional operators for information extraction and data cleansing. End-users specify data analysis operations by writing Meteor queries. The input query is then parsed and translated into a Sopremo plan, a directed acyclic graph (DAG) of interconnected high level data processing operators.

PACTs (Parallelization Contracts) [3] is responsible for ensuring parallel processing and offers a low-level abstraction to user-defined functions. The programming model is based on Input and Output Contracts. Input Contracts are second-order functions which allow developers to express complex data analytical operations naturally and parallelize them independent of the user code. Output Contracts annotate properties of first-order functions and enable certain optimizations. A separation of the programming model from the concrete execution strategy: The PACT programming model exhibits a declarative character. Data processing tasks implemented as PACT programs can be executed in several ways. A compiler determines the most efficient execution plan for a PACT program and translates it into a parallel data flow program.

The extensible execution engine Nephele: Nephele [1] executes data flow programs modelled as directed acyclic graphs (DAGs) in a parallel and fault-tolerant way. Nephele is the original data processing scheme to clearly accomplish the dynamic resource allotment afforded by today's cloud computers for both, task appointing and beheading. It is a new processing skeletal that is categorically constructed for cloud computations. It allows assigning the particular tasks of a processing job to different types of virtual machines and takes care of their instantiation and termination during the job execution. Nephele is the first framework that dynamically allocates or deallocates resources from the cloud for job scheduling and execution. Nephele/PACTs system is parallel processing system that is centered around Parallelization Contracts (PACTs) and the scalable parallel execution engine Nephele.

II. RELATED WORKS

A. End-to-end big data systems

Currently there are a few systems that are being developed that will help us to advance distributed data management. Hadoop [6] with its higher-level languages Pig [22], Hive [7], and libraries such as Mahout is the most popular.

Hadoop's architecture makes use of the Hadoop Common package, which provides OS level abstractions and a file system, a MapReduce engine like MapReduce/MR1 or YARN/MR2 and finally the Hadoop Distributed File System (HDFS).

A small Hadoop cluster consists of multiple worker nodes with a single master. A slave or worker node is both a Data Node as well as a Task Tracker. The master node consists of a Name Node, Data Node, Job tracker, Task tracker.

Asterix started with a vision to create a parallel, semi-structured information management system. This has led to the creation of three reusable software layers. The bottommost layer of the Asterix software stack is a data-intensive runtime named Hyracks. Hyracks is roughly at the same level as MapReduce in implementations of higher-level data analysis languages in Hadoop such as Pig[22], Jaql[20] or Hive[7]. The topmost layer of the software stack is composed of a parallel database management system, with its own query language (AQL) for querying, describing and analyzing data and a full, flexible data model (ADM). There exists a middle layer between Hyracks and the Asterix DBMS called Algebricks. It is a model-agnostic, algebraic "virtual machine" for parallel query processing and optimization.

Spark [20] is a distributed system developed by UC Berkeley that works on memory-resident data. Spark provides programmers with an API centred on a data structure called resilient distributed dataset (RDD), which is a read-only multiset of data items distributed over a cluster of machines, which is maintained in a fault-tolerant manner.

As RDDs are available in the Spark system, it facilitates the implementation of both iterative algorithms as well as interactive/exploratory data analysis.

B. Query Languages and models for parallel data management

Conceptually it is easy to parallelize the basic operators used in relational algebra, and such parallel database systems have existed for a long time. The MapReduce paradigm of parallel programming revolutionized parallel programming and widened its scope by including more generalized user defined aggregation functions.

The PACT programming model has been developed with the goal of overcoming the problems faced during complex analytical tasks. It is based on the concept of Input Contracts, which is a generalization of the Map and Reduce functions. It is better than MapReduce in several aspects and has been discussed in detail later in section 5 of this paper.

C. Query Optimization

The Scope [19] system uses an optimizer which focuses on transformations and is based on the Cascades framework that is used to translate input scripts into execution plans that are efficient. The Scope optimizer considers many alternate plans and chooses the one that has the lowest cost among them.

Manimal [17] uses static analysis-based techniques in a MapReduce system to enable relational-style optimization. Standard MapReduce system scans through each bit of the input while Manimal only chooses to scan those bits which are important to get the output.

Stubby [18] is an automatic cost-based optimizer for MapReduce workflows. Stubby takes into consideration multiple types of optimization which can be composed together, which in turn leads to the generation of a large plan space for a MapReduce workflow.

D. Distributed dataflow execution

The principles behind parallel databases have been known and explored since the 1980s in systems like Gamma and Grace.

Gamma [11] was a relational database machine introduced in 1986. It exploits dataflow query processing techniques. The framework and prototype of Gamma demonstrated the practical implementation of parallelism and also described for the first time how it can be controlled by using a combination of hashing and pipelining algorithms on the data between various processes. The advantage that Gamma had over existing relational database system on local area network was that Gamma had no notion of site autonomy, had a centralized schema, and a single point for initiating the execution of all queries.

Grace [12] was an initial system software for parallel relational database. In this system, the execution of the operations and its data sets are encapsulated and controlled as a task. The tasks were made into autonomous objects using data stream control protocol between the various modules of the particular task. The control overheads are greatly eliminated firstly by adopting the task-level granularity for the execution and control and then by executing the operations along with the flow of its data. Grace achieves a complete data stream oriented processing.

MapReduce [10] is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key.

E. Distributive Iterative Algorithms

For a long time now various stand-alone graph processing systems or approaches to integrate iterative processing in dataflow engines have been proposed.

GraphLab [14] is a specialized framework for parallel machine learning, where programs are modelled into a graph expressing the computational dependencies of the input. Programs are expressed as update functions on vertices, which can read neighbouring vertices' state through a shared memory abstraction. Furthermore, GraphLab provides configurable consistency levels and asynchronous scheduling of the updates. Distributed GraphLab extends the shared memory GraphLab abstraction to the distributed setting by refining the execution model, relaxing the scheduling requirements, and introducing a new distributed data-graph, execution engines, and fault-tolerance systems.

A computational model, Pregel [15] has been developed for processing large-scale graphs. It is a graph processing adoption of bulk synchronous parallel processing. Programs are expressed as a sequence of iterations, in each of which a vertex can receive messages sent in the previous iteration, send messages to other vertices, and modify its own state and that of its outgoing edges or mutate graph topology. This vertex-centric approach is flexible enough to express a broad set of algorithms. Its computations are a sequence of iterations, called supersteps. The input to a Pregel computation is a directed graph in which each vertex is uniquely identified by a string vertex identifier. Each vertex is associated with a modifiable, user defined value. The directed edges are associated with their source vertices, and each edge consists of a modifiable, user defined value and a target vertex identifier. The output of a Pregel program is the set of values explicitly output by the vertices.

REX [16] is a parallel shared-nothing query processing platform that provides programmable deltas for expressing incrementally iterative computations and handles faults gracefully. REX implements a core declarative programming model, RQL, that is derived from SQL. It can directly execute arbitrary Hadoop MapReduce jobs consisting of multiple map and reduce functions, generating a RQL template. REX's RQL language provides support for standard SQL features like joins, aggregation, and nested subqueries, as well as extensions for recursion, native Java objects (including collection-typed objects), and seamlessly embedded Java user-defined code.

III. GENERAL SYSTEM ARCHITECTURE

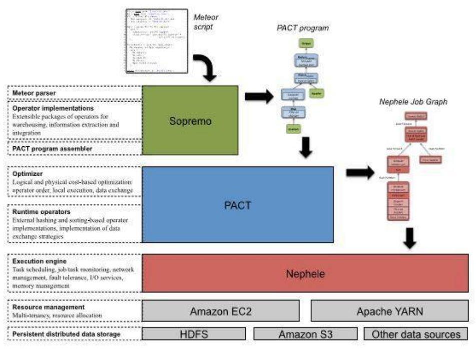
Meteor script
PACT program
Nephele Job Graph
Meteor parser
Operator implementations
Extensible packages of operators for warehousing, information extraction and integration
PACT program assembler
Sopremo
Optimizer
Logical and physical cost-based optimization: operator order, local execution, data exchange
Runtime operators
External hashing and sorting-based operator implementations, implementation of data exchange strategies
PACT
Execution engine
Task scheduling, job/task monitoring, network management, fault tolerance, I/O services, memory management
Nephele
Resource management
Multi-tenancy, resource allocation
Amazon EC2
Apache YARN
Persistent distributed data storage
HDFS
Amazon S3
Other data sources

Fig. 1. The Stratosphere Software Stack

The Stratosphere software stack [4] consists of three stages: Sopremo, PACT and Nephele layers. The output of one layer is given as input to the next layer. Each layer has its own programming model. In this section we will provide a general overview of each of the layers. We will be discussing the layers in detail in the subsequent sections. The reason why the architecture is divided into three levels is to provide the users a choice regarding the amount of declarativity of their programs. The highest level Sopremo is the most declarative layer whereas the other two layers PACT and Nephele trade declaritivity with expressiveness.

The Sopremo model is the first layer in the stack. A Sopremo program comprises a set of logical operators connected in a directed acyclic graph (DAG), like the logical query plan in a RDBMS system. Programs for the Sopremo layer are written in a language called Meteor.

After a Meteor program is submitted to Stratosphere, Sopremo first translates it into an operator plan. Moreover, the compiler within the Sopremo layer can derive several properties of the plan, which can be later exploited for the physical optimization of the program in the subsequent PACT layer.

The input of the PACT layer, is called a PACT program. Like MapReduce, the PACT programming models is based on the concept of second-order functions, called PACTs. PACTs can be reassembled into complex direct acyclic graphs (DAGs), not just fixed pipelines of jobs like MapReduce.

In PACT programs, the first-order or the user defined functions can be written in Java and the semantics of these programs are hidden from the system. This makes it more expressive than writing a program in the Sopremo model as the code is no longer bound to a specific set of operators. The PACT programs still show a certain level of declarativity as they do not define how the specific guarantees of the second-order function will be enforced at runtime. For most of the PACT input contracts, there exists different strategies to fulfil the provided guarantees with different implications on the required effort for data reorganization. Choosing the cheapest of those data reorganization strategies is the responsibility of a special cost-based optimizer, which is present in the PACT layer. It computes the cost of each execution plan and selects the one that is most preferred by the system.

The output of the PACT compiler, which is a parallel data flow program, is provided as an input to the Nephele layer. The Nephele layer is Stratosphere's parallel execution engine and is the third layer of Stratosphere's stack.

A Nephele data flow program or a Nephele *Job Graph*, much like PACT programs, are specified by DAGs with each vertex of the graph representing the individual tasks and the edges representing the data flow between them. PACT programs generate a number of strategies to solve the problem and chooses the most preferable result but Nephele Job Graphs contain a concrete execution strategy, chosen specifically for the given data sources and cluster environment. In particular, this execution strategy includes a certain degree of parallelism for each vertex of the Job Graph, concrete instructions on data partitioning as well as hints on the co-location of vertices at runtime.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 9 Issue 06, June-2020

The Nephele layer shows maximum expressiveness. But this degree of expressiveness is reached at the expense of programming simplicity.

Nephele itself executes the Job Graph on a set of worker nodes. It allocates the 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 case of failures in execution.

While a job is being executed, Nephele can collect statistics on the runtime characteristics of each of the tasks, starting from CPU and memory consumption to information on data distribution. These data are stored in the master node of Nephele's Job Graphs and can be accessed by accessing the master node.

IV. METEOR AND SOPREMO

In Stratosphere, the semantics of the user defined functions would be abstracted from the compiler and the optimizer. These functions offer an advantage over existing query languages as they can be modified by user to perform complex operations easily. Here, we treat user-defined functions as first-class operators for a data flow scripting language. This proves to be advantageous as the operator’s semantics can be accessed at compile time for data flow optimization or correcting semantic errors.

Sopremo is a semantically rich operator model, and Meteor is an extensible query language that is embedded in Sopremo. Sopremo provides a framework that allows users to easily develop and integrate extensions with the respective operators and instantiations. Meteor queries are translated into data flow programs of operator instantiations, i.e., concrete implementations of the involved Sopremo operators. Sopremo’s framework allows users to customise packages, the respective operators and their instantiations. Sopremo already contains a set of base operators from the relational algebra, plus two sets of additional operators for Information Extraction and Data Cleansing.

A. *Meteor – The Query Language*

Meteor is an operator-oriented query language that emphasis on analysis of large-scale, semi- and unstructured data. Users compose queries as a sequence of operators that reflect the desired data flow. By importing application-specific operators, users can use Meteor to process data for a wide range of applications. The internal data model is based on Json, but Meteor supports additional input and output formats. The goal of Meteor is to support a large variety of applications, each with its own specialized and customized operators which have been imported dynamically. To enable the correct specification of a query with arbitrary, dynamically imported operators, all operators in Meteor have a uniform syntax.

Additionally, to operators, Meteor allows users to define and import functions. Function definitions inside a Meteor script have the same expressive power as a new Meteor script. These functions serve to shorten a script that repeatedly uses a given sequence of operators or expressions. The meteor parser parses any given Meteor query into an abstract syntax tree and then translated it into a logical execution plan of Sopremo operators. The plan is then handled by the Sopremo layer.

B. *Sopremo – The Operator Model*

Sopremo is an execution engine that manages packages of extensive operators. This framework works as a Meteor parser that produces an executable PACT program. Sopremo consists of relational operators along with the application-specific operators, like data cleansing and information extraction operations such as remove duplicates and annotate persons. When the Meteor script is executed by Sopremo, all variables are replaced by edges in order to represent the flow of data. The various operators possess certain properties such as 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 further nested to form trees in order to perform more complex calculations and transformations.

In Sopremo, all operator instantiations have a direct Pact implementation. Therefore, there are two steps in the final compilation of the complete Sopremo plan. First, all operator instantiations in the Sopremo plan are translated into partial Pact plans. Second, the inputs and outputs of the partial Pact plans are rewired to form a single, consistent Pact plan. In order to improve the runtime efficiency of a compiled plan, the translation process is extended with two more steps: First, a Sopremo plan is logically optimized. A separate, physical optimization is performed later in the Pact layer. Second, a schema less data model is used by PACT which is crucial to reorder Pacts. For pure Pact programs, the schema interpretation is performed by Pact users in their UDFs. Nevertheless, the additional semantics of Sopremo operator allows Sopremo to deduce an efficient data layout and bridge the gap between the data model of the Pact model and the nested data model of Sopremo. Meteor or Sopremo users thus do not have to specify the data layout explicitly.

Meteor users formulate a query that is parsed into a Sopremo plan. To import packages, Meteor requests the package loader of Sopremo to inspect the packages and load the discovered operators and predefined functions. Meteor uses these packages to validate the script and translate it into a Sopremo plan. The plan is analysed by the schema inferencer to obtain a global schema that is used in conjunction with the Sopremo plan to create a consistent Pact plan. When the query is successfully executed, one or more output files are produced that are encoded in Json if not specified. In case of an error, Meteor users receive two kinds of feedback depending on the type of error. Firstly, a Meteor script may be invalid. Operators check whether their configuration is containing any error, if properties are conflicting or some prerequisites are unavailable. Secondly, Sopremo operators may contain errors. In this case, Meteor shows a detailed stack trace of the erroneous operator on the master node. That means the Sopremo plan is reconstructed at execution time and stack traces from the cluster nodes are transferred to the master node. Also, Sopremo might optionally add debug information to allow values to be traced along a script execution to ease debugging of Sopremo operators as well as Meteor scripts.

V. PACT: A MODEL FOR PARALLEL PROGRAMMING

Stratosphere provides an explicit layer called the PACT [1] layer that abstracts the process of parallelization so that the user does not have to write complex parallel code. Here we describe the PACT layer.

A. *Data Model*

PACT programs work on a flat data model. An intermediate result of one PACT program is called a data set which is an unordered collection of records. A data set is consumed by another PACT program. The semantics of the values in the record and how they are interpreted are opaque to parallel runtime operators. They can only be manipulated by the user defined function (UDFs) that process them.

Some functions grouping of attributes according to one attribute or by other types of association. To facilitate such operations, a subset of the record's field is maintained as the key. In the definition of the key, we need to include the types of the values in these fields so that the runtime operators can access the relevant fields (for sorting and partitioning) from the otherwise schema-free records [4].

B. *Operators in PACT and acyclic PACT programs*

A PACT is a second-order function that takes a data set and a first order user-defined function (UDF) as an input. A (PACT) operator consists of a second order PACT function and a concrete instantiation of the UDF. PACTs specify how the input data will be partitioned into independent subsets called parallelization units (PUs). The actual semantics of how the data will be manipulated is encapsulated in the UDFs. The PACT programming model is declarative enough to abstract away parallelization, but it does not directly model semantic information like the Sopremo layer. On the other hand, the semantic information is encapsulated in the UDF and is hidden from the system.

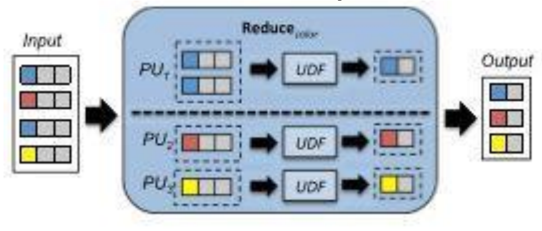

Fig. 2. A PACT operator using a Reduce PACT

Figure 2 shows a PACT operator that uses the Reduce function as its PACT. The input data set is logically grouped according to their keys (shown as different colors, which corresponds to its first attribute). Each group with a certain specific key becomes a parallelization unit (PU). Then, the UDF is applied to each PU independently. The logical division between the PUs can be achieved by various physical data partitioning schemes. Taking an example of figure 8, we can physically partition the PUs into two nodes as indicated by the thick horizontal dashed line such that PU_1 resides in node 1, and PU_2 and PU_3 reside in node 2. The logical output of the PACT program is the concatenation of the outputs of all the UDF calls.

There are five second-order functions that are implemented in the system. These are Map, Reduce, Match, Cross and CoGroup. The Map forms a PU from every record in the input. The Reduce function forms a PU of all the records in the input which have the same key value or any other attribute defined by the user. The Match, Cross and CoGroup PACTs operate on two input data sets. The PUs of the Match function are pairs of data that have the same key attribute value. The Cross function dictates that every record of the first input together with every record of the second input forms a PU, performing a Cartesian product. CoGroup generalizes Reduce to two dimensions; each PU contains the records of both input data sets with a given key. The source of records (left or right input) is available to the UDF programmer.

C. *Iterative PACT programs*

Most data analysis tasks cannot be accomplished by just one pass over the data. On the other hand, we require to make many iterations over the data repeating certain computations to make their solutions better until a convergence criterion is reached. The PACT programming model supports such iterations [22].

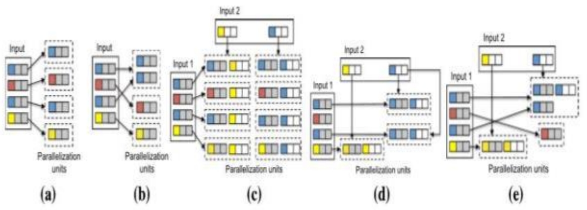

Fig. 3. The five second-order functions (PACTs) currently implemented in Stratosphere. The parallelization units implied by the PACTs are enclosed in dotted boxes. (a) Map (b)Reduce (c) Cross (d) Match (e) CoGroup

PACT offers two different declarative fix-point operators: one for Bulk iterations and the other for incremental iterations. One parallel application of the step function to all partitions of the partial solution is called a superset. Bulk iterations execute the PACT program which forms the step function iteratively in each superstep, using the entire partial solution and then re-computes the next partial solution by applying the step function, which in turn is consumed in the next iteration. The iteration halts when a user defined criterion is reached.

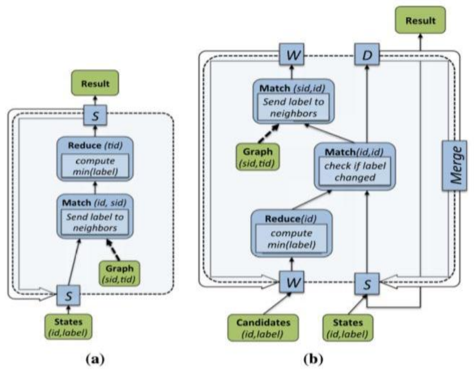

Fig. 4. An algorithm that finds the connected components of a graph as a bulk iteration and an incremental Stratosphere iteration. (a) Bulk iteration (b) Incremental iteration

In Incremental iterations, the partial solution is divided into two data sets: a solution set (S) and a working set (W). At each iteration, only the working set is consumed and selective modifications are made to the solution set elements. Thus, we are incrementally increasing the quality of the solution set rather than fully re-computing the partial solution. Using the S and W sets, the step function generates a new working set and a delta set (D) that tracks the changes that are taking place in the solution set (S) in that step, that is, it contains the set of elements that need to be updated in the solution set. The step function is then applied to the new working set.

Incremental iterations give us more efficient algorithms because not every element in the partial solution are consumed by the step function and hence not each element has to be examined in each superstep.

VI. OPTIMIZATION IN STRATOSPHERE

The optimizer in Stratosphere compiles PACT programs into Nephele Job Graphs.

A. *Overview of the Optimizer*

The optimizer itself consists of four phases. Like many relational optimizers, the optimization process is separated into a logical rewriting and a physical optimization phase.

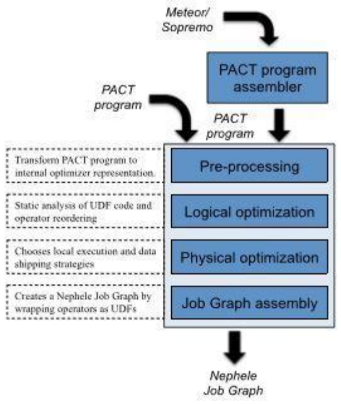
Meteor/ Sopremo
PACT program assembler
PACT program
PACT program
Transform PACT program to internal optimizer representation
Pre-processing
Static analysis of UDF code and operator reordering
Logical optimization
Chooses local execution and data shipping strategies
Physical optimization
Creates a Nephele Job Graph by wrapping operators as UDFs
Job Graph assembly
Nephele Job Graph

Fig. 5. The different program transformation phases of the Stratosphere optimizer

Before optimization, the optimizer converts a PACT program into an internal representation which is a DAG consisting of operator nodes that represent data sources, data sinks, PACT operators and internal operations.

The optimizer generates plans that are semantically equivalent in the next phase by the process of operator reordering.

The next step after operator reordering is physical optimization. The logical parallelization is defined by the second-order function of an operator. For the same second-order function, there can be several strategies for physical data shipping, such as range or hash partitioning or broadcasting, that help in providing the parallelization requirements and also multiple strategies for local physical execution, such as sort or hash based techniques. Like traditional database systems, interesting properties [9] can be used in this step. The execution plan that is finally generated is translated into a Nephele Job Graph and is submitted for execution.

B. *Reordering of Operators*

In the first stage of optimization, the PACT operators are reordered to achieve better computational speed, just like any other relational database. But unlike other relational databases, classical rewriting rules cannot be applied to PACT operators as the semantics of these operators are not known by the optimizer. For Stratosphere's optimizer, we define two sufficient rules which successive operators need to fulfil so that they can get reordered without affecting the program semantics. We use the read and write field sets to hold the information of all the fields read and written by the UDF. Hence, a write access to any record might add, remove or update any existing attribute.

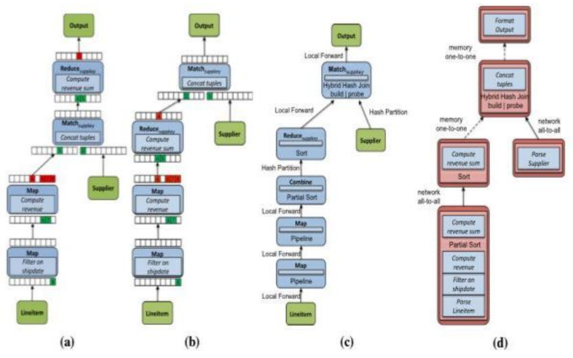

Fig. 6. (a) Original PACT program (b) Modified PACT program (c) Physical plan (d) Nephele Job Graph

The first condition for reordering checks the read and write sets of the two successive PACT operators for overlapping access patterns. For the condition to be met, only the read sets can intersect. The second condition is only applied to the group operators like Reduce and CoGroup. As group orders can have different group sizes and the semantics of the program might get altered if we reorder grouping operators of different sizes, we keep a condition that the input groups are preserved when we reorder the operators.

When the conditions are met by two successive operators to get reordered, we have to have an algorithm to reorder the operators. With a DAG, we can have multiple successors, much like a relational optimization with subexpressions. Going by the simplest solution to this problem, we split the data flow after each such operator. By doing this, we decompose our DAG into a number of trees where operator reordering is performed. After reordering, the trees are recomposed into a DAG.

C. *Physical Optimization*

Execution strategies from parallel database systems are supported by Stratosphere. These strategies include repartitioning and broadcast data transfer strategies and local execution strategies; such as sort based grouping or multiple join algorithms. In addition to execution strategies, the optimizer uses the concept of interesting properties. The optimizer keeps track of all the physical data properties such as sorting, grouping and partitioning of any given PACT operator to improve the operator's execution. The optimizer in Stratosphere chooses the optimal plan using a cost-based approach from multiple semantically equivalent plans.

The physical plans are generated by an algorithm which follows a depth-first search approach from the sink nodes of the program. As it moves down to the sources, it keeps track of the interesting properties. When the algorithm reaches a data source, it starts generating physical plans on its way back to the sinks. For each such sub-flow, the algorithm remembers the least cost plan and also the interesting properties. Keeping these in mind, the optimal plan is produced.

To correctly identify the optimal plans for arbitrary DAG flows, we investigate the program to see where the DAG branches and which binary operators joins these branches back. For the joining operators, the sub-plans that are rooted at the branching operator are seen as subexpressions and the plan candidates for that operator's inputs must have the same sub-plan for the common subexpression.

VII. NEPHELE

Nephele is the aboriginal data processing scheme to clearly accomplish the dynamic resource allotment afforded by today’s cloud computers for both, task appointing and beheading. It allows authorizing specific tasks of a processing job to distinct varieties of virtual machines and takes care of their instantiation and termination amid the job execution.

A. *Architecture of Nephele*

Nephele, based on classical master-worker principle, is a densely populated parallel data flow engine handling resource management, scheduling, fault tolerance and communication. Before deferring a Nephele compute job, the client must outset a VM that runs Job Manager. The Job Manager is answerable for scheduling client’s job. The execution of tasks is done by a group of instances which are allocated an element called Task Manager. A Task Manager accepts one or more jobs or tasks from the Job Manager one at a time, executes them and notifies the Job Manager about their conclusion or possible glitches. A client is the one who commences the request to the job manager. The job manager will then be on hold until the task from client harmonizes the process and it audits the vacancy of the server. If the server is vacant for that particular task, it allots the resources for execution and holds up till conclusion. Cloud Controller Module acts as admix between the job manager and task manager. It is also answerable for harmonizing and management of the execution and also the expedition of tasks. It checks for the availability of task managers and allocate the resource for the task to be executed. Task Manager Module holds till the final execution of task; it then executes it and circulates the complete feedback to the job manager which then goes to the client.

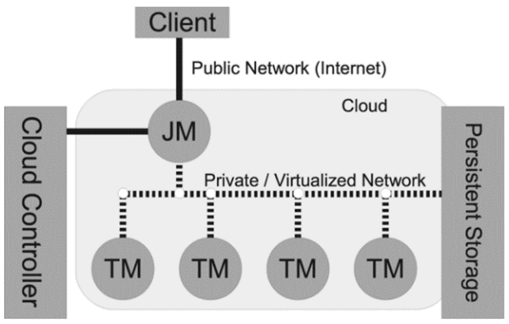

Fig. 7. Structural overview of Nephele running inside a compute cloud.

B. *Job Description in Nephele*

Nephele Jobs are conveyed as a directed acyclic graph (DAG) where each vertex symbolizes a task of the overall processing job and edges construe the communication discharge between these tasks. An Extended Nephele job is composed of three essential methods: First, the user must establish a connection to VM and commence his task. Second, the task program must be allotted to a vertex. Finally, the graph must be a connected graph. For developing the job graph, client must have some basic knowledge about features like number of subtasks per instances, sharing instances midst tasks, variety of channels and instance types for job descriptions. Once the Job Graph is defined, the user acknowledges it to the Job Manager along with the accreditations that the user has retrieved from the cloud operator.

C. *Job Scheduling and Execution*

Once an authorized Job Graph is received by user, JM converts it into Execution Graph. It defines the aligning of subtasks to instances and the communication channels between them. Execution Graph comprises of a Group Vertex where stages are utilized to avert the instance type vacancy complications in the cloud. Subtasks are termed as Execution vertex which is monitored by its parallel Group vertex. Each subtask is aligned to an Execution Instance which is represented by an ID and an instance type defining the hardware properties of the parallel VM. After giving the job to the JM, it splits the job into subtasks and appoints them into a number of Task managers as per the number of subtasks. These subtasks are reported to the TM via any type of channel as per the type of the job.

VIII. EXPERIMENTAL EVALUATION

The current version of Stratosphere is carefully evaluated against other systems for large scale data processing that are open-source. A series of experiments are conducted which compare Stratosphere with version 1.0.4 of the vanilla MapReduce engine which is shipped along with Apache Hadoop, version 0.10.0 of Apache Hive which is a declarative language and a relational algebra runtime running on Hadoop's MapReduce and also version 0.2 of Apache Giraph [5]. Giraph is an implementation of Pregel's [18] vertex centric model for graph computation that makes use of a Hadoop map-only job for distributed scheduling.

A. *Triangle Enumeration*

Triangle enumeration is a problem that illustrates the benefits of PACT compositionality. The algorithm is based on enumerating graph triangles. This algorithm is computed as two MapReduce tasks. The first reduce function is responsible for building all triads, i.e. pairs of connected edges. The second reduce function would create a join between various triads produced by first function and would remove all triads that cannot be closed by a matching edge. Even though MapReduce implements this algorithm, it is not that efficient and it is difficult to manage two reduce functions. PACT offers a way to represent triangle enumeration as a single, simple dataflow using a map, a reduce and a match function. The two approaches have been diagrammatically explained in Figure 8.

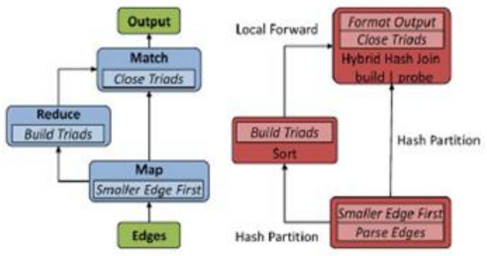

Fig. 8. Triangle Enumeration of Pokec social network graph.

Experiment of enumerating triangles for a symmetric version of Pokec social network graph, it has been observed and highlighted in that the strategy using pipelined execution in PACT offers a better result over a Hadoop implementation involving two MapReduce functions.

B. *Relational Query*

We make a comparison of Stratosphere with Hive, which compiles SQL-like queries as sequence of Hadoop MapReduce jobs.

The benefits of using Stratosphere are clear from figure 10 as it has a more general approach to optimization and specification of complex data processing programs. Hive's optimizer, has to follow the strict MapReduce pipeline and has to break up HiveQL expressions into several different MapReduce jobs. This leads to unnecessary I/O overhead as between each map and reduce phase, data has to spilled to the disk. On the other hand, Stratosphere can optimize and execute complex tasks by representing the programs as DAGs. No data is written to the disk till the output is generated because the operators and the input of the reducer function handling the revenue aggregate computation fit into memory.

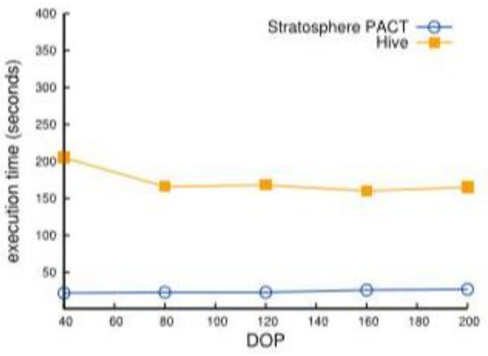

Fig. 9. Scale-out experiment with DOP ranging from 40 to 200 for the relational query.

C. *Fault Tolerance*

For a system to be efficient and reliable, it is important to make sure that it has a strong recovery mechanism and is fault tolerant. Here, a comparative study of two programs namely, Triangle Enumeration and a variant of Relational Query has been used in order to emphasize the need and efficiency of fault-tolerance mechanism checkpoints. For each of the program, a failure-free runtime has been calculated with and without checkpoints and also the total runtime post recovery when encountered a failure. For both the programs, on experimentation it has been observed that the failure is encountered approximately after 50% of the failure-free runtime and mainly in the join operator.

IX. CONCLUSION

In this paper, we presented an elaborated study of a software stack for analysing and processing big data, Stratosphere. Stratosphere has a layered structure with a high-level scripting language Meteor, an operator model, Sopremo and distributed execution engine, Nephele. Using Meteor and Sopremo, the functionality of the operators can be extended by importing packages and functions for various operations. Nephele is the next layer of the stack that acts as a distributed execution engine. The task of Nephele is to provide scalable execution, scheduling, network data transfers and fault tolerance. Stratosphere maintains a perfect balance between two of the most widely accepted programming platforms for big data that is, MapReduce and Relational databases. Analysis of Stratosphere has opened a diverse area for research. Firstly, a lot of modifications can be suggested in design, compilation and optimization of the high-level declarative languages in various domains. Also, merging the distributed data management systems with existing scalable system by adapting its algorithms and architecture will make the system more efficient.

X. REFERENCES

[1] Alexandrov A., Battré D., Ewen S., Heimel M., Hueske F., Kao O., Markl V., Nijkamp E., Warneke D.: Massively parallel data analysis with pacts on nephele. PVLDB Vol. 3, No. 2, pp. 1625–1628 (2010)

[2] Alexandrov A., Ewen S., Heimel M., Hueske F., Kao O., Markl V., Nijkamp E., Warneke D.: Mapreduce and pact - comparing data parallel programming models. In: BTW, pp. 25–44 (2011)

[3] Battré D., Ewen S., Hueske F., Kao O., Markl V., Warneke D.: Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In: SoCC, pp. 119–130 (2010)

[4] Alexandrov, A., Bergmann, R., Ewen, S. et al.: The Stratosphere platform for big data analytics. The VLDB Journal, Vol. 23, No.6, pp. 939-964 (2014)

[5] Apache Giraph. http://incubator.apache.org/giraph/

[6] Apache Hadoop. http://hadoop.apache.org/

[7] Apache Hive. http://sortbenchmark.org/

[8] Heise A., Rheinländer A., Leich M., Leser U., Naumann F.: Meteor/sopremo: an extensible query language and operator model. In: BigData Workshop at VLDB (2012)

[9] Selinger P.G., Astrahan M.M., Chamberlin D.D., Lorie R.A., Price T.G.: Access path selection in a relational database management system. In: SIGMOD Conference, pp. 23–34 (1979)

[10] Dean J., Ghemawat S.: Mapreduce: simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)

[11] DeWitt D.J., Gerber R.H., Graefe G., Heytens M.L., Kumar K.B., Muralikrishna M.: Gamma—a high performance dataflow database machine. In: VLDB, pp. 228–237 (1986)

[12] Fushimi S., Kitsuregawa M., Tanaka H.: An overview of the system software of a parallel relational database machine grace. In: VLDB, pp. 209–219 (1986)

[13] Isard M., Budiu M., Yu Y., Birrell A., Fetterly D.: Dryad: distributed data-parallel programs from sequential building blocks. In: EuroSys, pp. 59–72 (2007)

[14] Low Y., Gonzalez J., Kyrola A., Bickson D., Guestrin C., Hellerstein J.M.: Distributed graphlab: a framework for machine learning in the cloud. : Proceedings of the VLDB Endowment (PVLDB), Vol. 5, No.8, 716–727 (2012)

[15] Malewicz G., Austern M.H., Bik, A.J.C., Dehnert J.C., Horn I., Leiser N., Czajkowski G.: Pregel: a system for large-scale graph processing. In: SIGMOD Conference, pp. 135–146 (2010)

[16] Mihaylov S.R., Ives Z.G., Guha S.: Rex: recursive, delta-based data-centric computation. : Proceedings of the VLDB Endowment (PVLDB) Vol. 5, No. 11, 1280–1291 (2012)

[17] Cafarella MJ, Reaaá C. Manimal: Relational optimization for data-intensive programs. In Proceedings of the 13th International Workshop on the Web and Databases, Co-located with ACM SIGMOD (2010)

[18] Lim H., Herodotou H., Babu S.: Stubby: A transformation-based optimizer for mapreduce workflows. In: Proceedings of the VLDB Endowment (PVLDB), Vol. 5, No. 11, pp. 1196-1207 (2012)

[19] Chaiken R., Jenkins B., Larson P., Ramsey B., Shakib D., Weaver S., Zhou J.: SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. In: Proceedings of the VLDB Endowment (PVLDB), Vol. 1, No. 2, pp. 1265-1276 (2008)

[20] Beyer K.S., Ercegovac V., Gemulla R., Balmin A., Eltabakh M.Y., Kanne C.C., Özcan F., Shekita E.J.: Jaql: a scripting language for large scale semistructured data analysis. : Proceedings of the VLDB Endowment (PVLDB), Vol.4, No. 12, 1272–1283 (2011)

[21] Ewen S., Schelter S., Tzoumas K., Warneke D., Markl V.: Iterative parallel data processing with stratosphere: an inside look. In: SIGMOD (2013)

[22] Olston C., Reed B., Srivastava U., Kumar R., Tomkins A.: Pig Latin: a not-so-foreign language for data processing. In: SIGMOD Conference, pp. 1099–1110 (2008)