Programming Models and Execution Frameworks for Large Scale Data Processing

Vojin Jovanovic
LAMP, I&C, EPFL

***Abstract*—Amounts of data being processed increase every day often exceeding computing capabilities of a single computer. In order to overcome notorious difficulties of writing distributed, fault-tolerant and at the same time concise programs, frameworks like MapReduce and Dryad have been developed. These frameworks provide a constrained and low level programming interface as the trade-off for out of the box fault-tolerance and resource management in distributed programs.**

Although these frameworks solve many processing problems, in practice, it was shown that their interfaces are too low level and that writing both large programs and ad-hoc queries is very cumbersome. Recent research proposes several programming models and frameworks that increase expressiveness of MapReduce and Dryad, hide unnecessary details from the programmer and leave space for great number of automatic optimizations.

Here we present three solutions: *i)* FlumeJava, *ii)* DryadLINQ and *iii)* Nephele/PACTs. With insight from presented work we propose Distributed Collections for Scala (DCS) library that is compatible with regular Scala Collections. DCS is enriched with join operations and can be used with different large scale processing frameworks making its code portable.

***Index Terms*—Scala, MapReduce, LINQ, DryadLINQ, Nephele**

Proposal submitted to committee: September 7th, 2011; Candidacy exam date: September 14th, 2011; Candidacy exam committee: Prof. Martin Odersky, Prof. Christoph Koch, Prof. Rachid Guerraoui.

This research plan has been approved:

Date: _____

Doctoral candidate: _____
(name and signature)

Thesis director: _____
(name and signature)

Thesis co-director: _____
(if applicable) (name and signature)

Doct. prog. director: _____
(R. Urbanke) (signature)

EDIC-ru/05.05.2009

I. Introduction

LARGE amounts of data have been processed from the early beginnings of the computer era and it is well known that its reliable processing is extremely difficult. Before the massive explosion of web companies data was processed either using expensive parallel databases or custom made MPI [1] programs. Due to limitations of SQL [2], [3], the need to process unstructured data in the fault tolerant way and complexities of MPI programming, it has become necessary to make a simple framework for processing large amounts of data. In 2004, researchers from Google developed MapReduce [3] which provided a simple yet very powerful interface for reliable data processing on large computer clusters. MapReduce triggered a new wave of different frameworks that target large scale data processing. First there was Hadoop MapReduce [4] (clone of Google MapReduce) followed by Dryad [5] which provides more general programming model than MapReduce.

However, practice has shown that developing programs for more complex tasks in Dryad and MapReduce becomes awkward. Moreover, program maintenance, debugging and comprehension becomes very hard [6], [7], [8]. Furthermore, great amounts of data combined with unpredictable business requests often require writing of ad-hoc programs. Writing each program using Dryad's and MapReduce's interfaces requires much boilerplate code for defining data types, specifying serialization/deserialization of data and handling resources.

To overcome mentioned problems there have been several proposals in the research community. Some of them are in form of domain specific languages which employ extensions to SQL's stored procedure model like Pig [6] and Sawzall [9]. Although these systems hide much of program complexity from the developer they are not of interest to us due to their limitations. These languages have limited type systems, IDE support and their language is not embedded in some more widely used language. For our future work we are more interested in solutions that completely embed higher level abstractions in a statically typed and widely used programming language.

In this work we present FlumeJava [8] which treats data processing as operations on Java collections (Section II) and DryadLINQ [10] that compiles LINQ [11] statements to highly optimized cluster programs (Section III). Then we present Nephele/PACTs [12] which provides a novel programming model that enforces specification of output data properties (Section IV). In Section V we propose Distributed Collections for Scala library that is compatible with Scala Collections and

extends them with join operations. With insight from presented papers we choose core abstractions that keep the code portable.

II. FlumeJava: easy, efficient data-parallel pipelines

FlumeJava is a Java library that provides a high level interface for writing data-parallel programs. The library's main abstraction is a strongly typed parallel collection which provides interface familiar to most programmers. Its back-end optimizes and compiles collection operations to MapReduce programs that can be executed on large clusters. Since the library is written in pure Java it benefits from strong static typing, powerful IDEs, debugging and code modularization. In the next section we describe the programming interface in detail and then optimizations done in the library's back-end. FlumeJava is developed at Google and it runs on top of the Google MapReduce framework. Due to its popularity we will not explain MapReduce here but all complete system details can be found in [3].

A. FlumeJava core abstractions

FlumeJava's front-end provides two base classes and a set of operations on them. Those classes model data as collections that are abstracted through interface *PCollection<T>* or tables which are accessed through interface *PTable<K, V>*. *PTable<K, V>* is implementation-vise the same as the *PCollection<Pair<K, V>>* but the Java language does not have adequate "syntactic sugar" and type system to allow the interface with just one class.

Presented collection abstractions are strongly statically typed and the complexity of serializing and deserializing data is abstracted away from the programmer. For cases when collections need to be created from unstructured input library provides static helper methods for their instantiation. Due to their parallel processing nature the order of elements in collections is non deterministic, however library provides the ordered versions of the abstractions. These are called sequences and they have significant performance overhead compared to their unordered counterparts.

The library also provides four base data-parallel operations in the form of class methods on described data structures:

1) *parallelDo(DoFn<InT, OutT> fn)* can be applied on both *PCollection<T>* and *PTable<K, V>* and represents the second-order function that applies the *DoFn* to each element of the collection. Each application of *DoFn* can emit one or more results that produce the final result which is a new *PCollection<OutT>* containing elements of type OutT. There are also versions of *parallelDo* that can output multiple collections but they use same back-end features as single output version.
2) *combineValues(PTable<K, Collection<V>> c, CmbFn fn)* is just a special case of *parallelDo* with a more restricted interface, so it can be used in the place of MapReduce combiner. It accepts a table whose values are regular collections and a associative combining function. It outputs a *PTable<K, V>* with values being the aggregate of the input collections.
3) *flatten(List<PCollection<T>>)* is applied on a list of parallel collections of type T. It returns a *PCollection<T>* that contains a view over elements from all collections.
4) *groupByKey()* can be applied only on *PTable<K, V>*. It groups elements by key of type K and outputs a *PTable<K, Collection<V>>* with the unique key set.

With described collections being regular Java constructs they can be extended (in terms of OO programming) and modularized. By using core abstractions FlumeJava provides a derived set of basic operations. Here, we present a representative set of those operations without going into specifics: *i) count()* that returns the number of elements *ii) join()* that does equi-join on two *PTables* (there are also functions for other types of joins) *iii) top()* has the comparison function as a parameter and returns the greatest n elements. Due to data-parallel nature of computations functions that are passed to core methods must be side-effect free and must not use other data-parallel operations of class instances.

B. Deferred evaluation

When created, collection objects represent just a view of the actual distributed collection. Operations executed on them do not execute eagerly but instead create another view of the collection object which is not evaluated yet. Hence, the collection objects also carry the information about the state of the collection that can be either *materialized* or *deferred*. Internally the library represents collections and operations as the directed acyclic graph called *Execution Plan* (EP). The graph has input nodes that can be either materialized collections or other sources of data, operation nodes and output nodes that again represent either collections or other sources of data such as files. There are four types of operation nodes which directly correspond to base operations applicable on parallel collections and each node type has the same name as the operation it represents. Edges in the graph represent the actual collections that are created by the first upstream node.

For explicit materialization of parallel collections FlumeJava library provides a blocking operation *FlumeJava.run()* that starts MapReduce execution and blocks the invoking thread until its completion. The method first calls the optimizer to apply series of transformations on EP and then sends it to the MapReduce back-end for execution. When MapReduce completes and stores all the output to the file system or database FlumeJava marks all collections as materialized and returns control to the thread that originally called *FlumeJava.run()*.

However, some programs require aggregate results from one computation to be used in other parallel computations. Since the evaluation is deferred and parallel collections can not be used in other parallel operations this would not be possible without the special library construct. To fetch the contents of collections during and after pipeline execution FlumeJava provides *PObject<T>* abstraction. *PObject<T>* contains a single Java object of type T, it can be in *deferred* or *materialized* state and unlike collections can be used both in and out of the pipeline (its parallel operations). *PObject<T>* behaves in the same way as the *future* [13]. Instances of *PObject<T>* can

be constructed in several ways: *i)* method *asSequentialCollection()* creates *PObject<Collection<T>>* from all elements of *PCollection<T> ii) combine()* method of *PCollection<T>* accepts a combining function over type T as the argument and produces *PObject<T> iii) operate()* accepts a list of *PObjects* and a processing function and returns a list of new *PObjects*. *operate()* allows preprocessing of *PObjects* when they are used in the pipeline.

C. Optimizations

FlumeJava optimizer is based on combining as many operations to a single MapReduce step. This is achieved by creating **MapShuffleCombineReduce (MSCR)** blocks in the EP. MSCR is a combination of *parallelDo*, *groupByKey*, *combineValues*, and *flatten* nodes that could be executed as a single MapReduce step. An MSCR operation can have multiple input channels (each performing a map operation) and multiple output channels (each optionally performing a shuffle, an optional combine, and a reduce). MSCR generalizes MapReduce by allowing multiple reducers and combiners, by allowing each reducer to produce multiple outputs, by removing the requirement that the reducer must produce outputs with the same key as the reducer input and by allowing pass-through outputs, thereby making it a better target for the optimizer.

The optimizer applies series of five passes to the graph in order to achieve the most optimal execution plan:

1) **Sink flatten** pushes *flatten* operation down through *parallelDos* duplicating the *parallelDo* for each input of *flatten*
2) **Lift combineValues** identifies that *combineValues* is right below *groupBy*, marks that fact and further treats the *combineValues* as regular *parallelDo*
3) **Insert fusion blocks** decides if the *parallelDo*, that is in the graph topologically between two *groupBy* operations, should be fused with its predecessors or successors
4) **Fuse parallelDos** merges both consecutive and sibling *parallelDo* operations into a single *parallelDo* that takes one input and produces multiple outputs
5) **Fuse MSCRs** tries to create multiple MSCR operations from the execution plan graph. The transformation first produces the set of *related groupBy* operations. Operations *groupBy* are considered *related* if they have the same input or inputs created by the same (possibly fused) *parallelDo*. MSCR is created by fusing all related *groupBy* operations, their first adjacent nodes and their inputs and outputs.

D. Execution

When a FlumeJava pipeline is executed the library runtime analyses sizes of data sets and decides if the code is going to be run locally on the client or it will be submitted to a MapReduce cluster. Cluster environment imposes performance constraints on each execution so moderate size jobs are run locally.

This fact eases the debugging of code as users do not have to wait for long lasting cluster jobs to finish. Moreover,

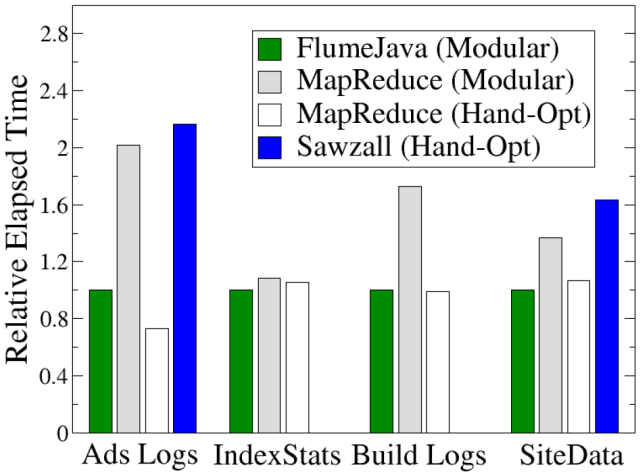

Fig. 1. Performance comparison between FlumeJava, MapReduce and Sawzall. Y-axis present relative elapsed time of different jobs.

FlumeJava tries to make execution as similar to regular Java programs. To achieve this it streams *System.out* and *System.err* to the client machine. Also, all temporary files created in the pipeline stages are automatically collected by the system. To further enhance debugging FlumeJava allows caching intermediate results and restarting jobs from a given checkpoint very quickly. This makes debugging of programs on very large data sets very effective.

E. Evaluation

User experience FlumeJava is used for large scale data processing internally at Google and has been adopted by 319 users in period of 10 months. Users are generally very satisfied with the programming interface of the library. However, users did complain about Java's syntax limitations for manipulating anonymous functions and tuples.

Optimizer effectiveness We have seen in section II-C how optimizer reduces the number of MapReduce stages. Here we present its operation on very large number of production jobs written by Google developers. On average the optimizer reduces the number of stages by factor of 5. The greatest reduction achieved reaches factor of 30.

Performance To evaluate performance of FlumeJava it was tested against a set of benchmark based on production pipelines written by its users. Benchmarks perform following computational tasks: *i)* analyzing advertisement logs (Ads Logs and *ii)* IndexStats), *iii)* Site-Data that both extract and join data from websites) and *iv* computing usage statistics from logs dumped by build tools (Build Logs). Benchmarks are written in three different ways: *i)* modular style using FlumeJava, *ii)* modular style using Java MapReduce and *iii)* hand-optimized style using Java MapReduce. Benchmarks Site-Data and Ads Logs were additionally implemented using the domain specific language Sawzall [9]. Each benchmark was run on same cluster configuration that represents production systems. In Figure 1 we see that FlumeJava has performance comparable to hand optimized MapReduce. Performance penalty of using FlumeJava ranges from 5% to 35%.

III. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language

In this section we present the DryadLINQ system developed at Microsoft Research. The system operates on top of the Dryad [5] execution engine and uses LINQ [11] as the programming model. Since LINQ and Dryad are very complex we will first present basics about them.

A. LINQ

LINQ (Language-Integrated Query) is a set of language extensions that enables unified interface for query operations not only on relational data and XML but also on other sources of data such as collections. LINQ expressions can be written using either query syntax very similar to SQL or set of imperative programming methods. In both cases the programming environment provides compile time syntax checking, strong static typing, auto completion and debugging. The base interface of LINQ expressions is *IEnumerable[T]*. Classes that extend it are enhanced by all LINQ extension methods allowing both embedded queries and method invocations to occur on their instances. Another important feature of *IEnumerable[T]* is that all LINQ computations occur lazily i.e. are calculated only when needed. LINQ supports many operations for filtering, projections, joins, ordering, grouping and aggregation. These operations are higher order extension methods that accept delegate functions or expression trees which are in .NET created from *lambda expressions*[1]. Here we will present a representative set of LINQ methods in a pseudo language similar to the Scala language:

- *Select[B](f: T => B): IEnumerable[B]* – applies a given function *f* to each element of a list, returning a list of results. This function is also known as *map* in most programming languages.
- *Where(p: T => Boolean): IEnumerable[T]* – produces the new collection containing only those elements for which the predicate *p* returns *true*.
- *GroupBy[K](key: T => K): IEnumerable[IGrouping[K, T]]* – groups elements on values produced by *key* parameter function.

LINQ also supports the *IQueryable[T]* interface which is a subtype of *IEnumerable[T]* but unlike *IEnumerable[T]* its methods operate on *expression trees* instead of *delegate functions*. Expression trees are data structure representations of lambda expressions that allow their runtime analysis, making space for better optimizations and easier translation to remote APIs like databases, web and cluster services. In .NET lambda expressions are transparently translated to both expression trees and delegate functions so users do not feel the difference.

B. Dryad system overview

Dryad is a distributed execution engine for general purpose data-parallel processing. Dryad operates in terms of jobs which are represented as directed acyclic graphs (DAGs) where edges represent data channels and vertices represent computation that reads data from input edges processes it and writes the results to output edges. This representation is purely logical and the graph can have more vertices than there are physical resources. Also, channels abstract away the details about communication leaving the simple interface for input and output. Their implementation can be in form of shared memory, TCP pipes or persistent temporary files.

[1]Lambda expressions are anonymous functions created from expressions like $x => x * x$. They are transparently compiled to either delegate methods or anonymous functions.

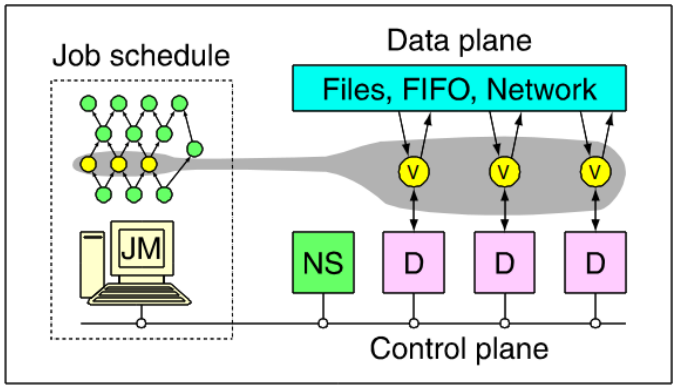

Fig. 2. The name server (NS) maintains a list of member machines. The job manager (JM) with the data from (NS) schedules running vertices (V) to available computers by using the daemon (D) as a proxy. Data from the vertices is exchanged through files, shared-memory channels or TCP pipes. The shaded space indicates vertices that are currently running.

Figure 2 explains the Dryad system. The key component is the job manager (JM) which has following roles in the cluster: *i)* optimally scheduling vertices on available machines, *ii)* re-executing failed or slow tasks thus providing fault-tolerance, *iii)* collecting statistics on processed data and *iv)* transforming the job graph dynamically for optimization purposes based on collected statistics and user supplied policies. All jobs are managed by a task scheduler which is separate from Dryad and keeps the job queue on which it enforces job policies and priorities.

C. DryadLINQ system

DryadLINQ is a library that transparently compile imperative programs in a general-purpose programming language to efficient Dryad based distributed computations. Its interface is based on LINQ but contains several extensions that enable more general computations and easier automatic optimizations. DryadLINQ uses a combination of static and dynamic optimizations to generate the most efficient code for Dryad. Strong typing, programming environment integration and declarative nature of LINQ in combination with generality of Dryad execution engine enable regular programmers to write reasonably efficient cluster computing code without worrying about complexities of distributed environment.

Although the LINQ interface is expressive the DryadLINQ system introduces several modifications in order to tackle the problems imposed by the distributed environment:

- *DryadTable* extends the *IQueryable[T]* interface and references remote data in the cluster which can be either

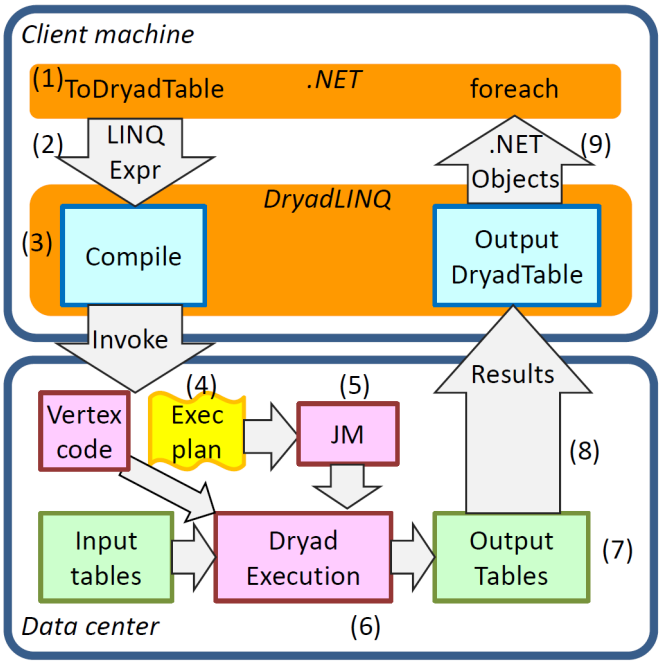

Fig. 3. Step by step diagram of DryadLINQ statement execution.

distributed files or database tables. Moreover, *DryadTable* contains meta-data that concerns data partitioning and ordering in the cluster. Instances are fetched by invoking *GetTable* on a URI or by invoking *ToDryadTable* on a DryadLINQ query.

- *DryadTable* can be re-partitioned by specific operator applying either hash-partitioning or range-partitioning
- LINQ is extended with functions *Apply* and *Fork* that have access to the whole data set and are used for computations that are not possible by regular LINQ extensions – for example sliding window computations. The difference between *Apply* and *Fork* is that *Fork* can output multiple tables.
- All expressions passed to DryadLINQ should be side-effect free but can contain references to static data.
- Annotations that give optimization hints to the compiler.

In Figure 3 we demonstrate flow of one DryadLINQ statement through the system (item number corresponds to the number in the figure):

1) Program runs and creates DryadLINQ expression object.
2) Invocation of *ToDryadTable* triggers the execution on the expression object and blocks the clients current thread.
3) DryadLINQ compiles the expression to a Dryad execution plan. Compilation decomposes the expression into subexpressions that are to be run in separate vertex, optimizes the graph, generates the code for remote Dryad vertices and generates the serialization code for data types.
4) System starts a custom DryadLINQ job manager which coordinates execution and does dynamic optimizations.
5) Job manager (JM) creates the job graph according to the plan created in Step 3.
6) Dryad executes code from vertices as directed by JM.
7) Dryad writes data to the output tables which can be of different types (distributed file system or database).
8) Job manager terminates and returns control back to client. DryadLINQ creates the local DryadTable objects that reference output tables. They may be used for further processing or explicitly returned to the clients memory as objects.
9) Client thread resumes execution by returning DryadTable. Client can access the objects directly through iterator interface of DryadTable or apply further DryadLINQ statements to it.

D. Optimizations

When DryadLINQ starts operating (Step 3 from Figure 3) it converts the LINQ expressions to the execution plan graph (EPG) in which each node is an operator and edges represent inputs and outputs. The EPG closely resembles traditional database query plans but it is slightly more general as it supports common subexpressions and multi output operations like *Fork*. The nodes of EPG are augmented with partitioning and ordering information while edges are augmented with data type that is being channeled. DryadLINQ even uses static code analysis of lambda expressions to infer whether partitioning and sorting order of data can be propagated through the graph. When the EPG is produced the system applies static optimizations on it and then submits the task to Dryad where dynamic rewritings of execution DAG are performed. Static optimizations that can be performed on the graph are:

- **Pipelining** which executes multiple operations in the same process by using memory channels.
- **Removing redundancy** which removes redundant partitioning steps from the graph.
- **Eager aggregation** which executes aggregation operations as early as possible in order to reduce network traffic in partitioning steps.
- **I/O reduction** which uses TCP-pipe and in-memory channels where possible and compressing of the sent through the channels.

For dynamic optimizations DryadLINQ uses Dryad API to mutate the execution graph. As the data flows through the system the number of machines is automatically determined by the size of the data set. For this optimization it is necessary to have runtime information. The complete graph transformations are best described in [5].

E. Evaluation

DryadLINQ is evaluated on the 240 computers each having 16 GB of memory, 4 striped 750 GB hard drives and 2 dual-core AMD Opteron 2218 HE CPUs working on 2.6 GHz, all interconnected with the classical data center network topology. Evaluation is performed on numerous applications but here we present two that are most representative. The Terasort benchmark that sorts 10 billion 100-Byte records with 10-Byte keys (10^{12} Bytes in total) has completed on specified cluster in 319 seconds. Query Q18 from Sloan Digital Sky Survey database [14] that determines effects of gravitational lens by star colors, was run both on native Dryad and DryadLINQ

for comparison of performance. The benchmark was run with different number of computers and the performance penalty for using DryadLINQ instead of Dryad is approximately 1.3 x. However the size of DryadLINQ code is only 10% the Dryad one.

IV. Nephele/PACTs: a programming model and execution framework for web-scale analytical processing

The *Nephele/PACTs* is a large scale data processing system developed for the Java Virtual Machine that provides *Parallelization Contracts* (PACTSs) as the programming abstraction and automatically optimizes them for execution on *Nephele* [15] distributed execution engine. In the following sections we describe Nephele/PACTs in greater detail but before we proceed we describe the *Nephele* system.

A. Nephele

The Nephele system [15] is a distributed execution engine that accepts incoming jobs as directed acyclic graphs (DAGs). Edges in the DAG represent communication channels that transfer data between different subprograms. Vertices of the DAG are sequential executable programs that process the data that they get from input channels and write it to output channels. There are three different communication channel types: *i)* in-memory, *ii)* network and *iii)* file channels. For persistent and reliable storage of data the system uses The Hadoop Distributed File System [16] (HDFS). The Nephele system greatly resembles the Dryad [5] described in III-B, but it does not support runtime DAG optimizations.

Unlike Dryad, which is designed for deployment on static hardware, Nephele supports dynamic allocation of physical resources in the cloud computing environment thus allowing very high levels of on-demand parallelism. Moreover, Nephele uses annotations to provide higher level constructs for interconnecting and spanning of vertices on actual physical hardware layout.

B. The PACT programming model

The PACTs program is represented as an arbitrary data-flow graph that contains input nodes, output nodes and processing nodes which are named *Paralelization Contracts* (PACTs). The PACT programming model operates on key/value data model and PACTs are its main programming abstractions. Each PACT is a composite made out of three components:

1) **User function (UF)** represents a task specific sequential user code that is executed for each parallelization unit.
2) **Input Contracts** are predefined second order functions. They divide input data in parallelization units following a specific strategy and than call the first order UF on each parallelization unit.
3) **Output Contracts** are an optional part of each PACT. They provide guarantees about the UFs behavior and thus they enable the optimizer to infer certain data properties that lead to faster execution

Figure 4 illustrates the job workflow for the PACTs equivalent of the following query.

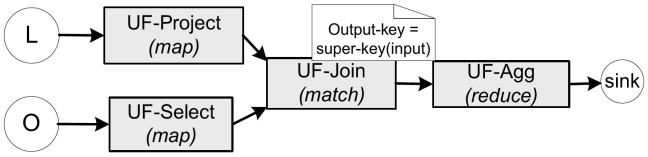

Fig. 4. PACTs program for code in the Listing 1. Inputs and outputs are marked as circles and PACTS as rectangle shapes. Each PACT defines the UF in the top, input contract in the bottom and output contract as the label outside the rectangle shape.

```
SELECT l_orderkey, o_shippriority,
       sum(l_price) as revenue
FROM orders, lineitem
WHERE l_orderkey = o_orderkey
      AND o_cust_key IN [X]
      AND o_orderdate > [Y]
GROUP BY l_orderkey, o_shippriority
```

Listing 1. Sample Join/Aggregate query used for description and evaluation.

In order to produce the code that is executable on Nephele the PACT workflow is passed through the compiler, however before we describe the compiler in subsection IV-E we first explain input and output contracts supported by the system.

C. Input Contracts

Parallel data processing relies on the fact that for processing one element of the dataset it is not needed to have the whole context hence independent parts of the data can be independently processed. We will call those parts parallelization units (PUs). Input contract is a second order function that defines how to map input into PUs and then executes the user specified function on each PU. Input contracts can be either single input contracts or a multiple input contracts. Unlike single input contracts, where one PU can contain key/value pairs from the single data set, multiple input contracts allows each PU to contain key/value pairs from all input data sets. Moreover, one input key/value pair can be mapped to multiple PUs which makes the model much more general allowing it to implement programs like sliding window computations. The system supports five predefined input contracts while the others (like the one needed for sliding window aggregations) need to be implemented by the user. Single input contracts provided by the system are:

- **Map** contract behaves the same as map phase in MapReduce model. It assigns each key/value pair to the independent processing unit thus allowing high levels of parallelism.
- **Reduce** contract assigns each key/value pair with the same key to the same PU. Therefore, it partitions input data set based on the key. It behaves the same as reduce phase in MapReduce and it also allows the use of a combiner for pre-aggregation. Reduce operation is shown on the left side of Figure 5.

Multiple input contracts defined by the system are:

- **Cross** builds a Cartesian product over its multiple inputs assigning each tuple of the product to one PU.

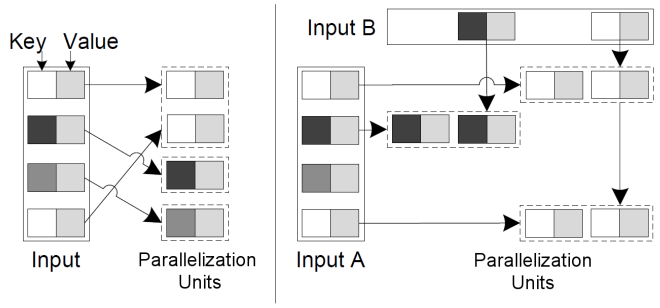

Fig. 5. Left and right images represent Reduce and Match input contracts respectively. Shapes drawn with dashed line represent parallelization units (PUs).

- **CoGroup** is similar to Reduce but it accepts multiple inputs. It assigns each key/value pair with the same key to the same PU. Hence, the user function is invoked on the data set partitioned by the key.
- **Match** assigns each key/value pair with the same key to the same PU but only if all data sets contain at least one pair with that key. Unlike CoGroup it does create an independent PU for each combination of inputs with the same key. Therefore, each PU contains exactly the same number of key/value pairs as there are inputs to the Match. Match is illustrated on the right part of Figure 5.

D. Output Contracts

Output contracts are an optional component of PACTs that specifies the properties that the user function preserves on the keys of the input. They are not enforced by the system so mistakes in specifying the output contract of the user function can lead to erroneous behavior of the program. Output contracts are only used by the compiler to infer when the PACTs preserve the sort order and partitioning of the data so it can apply optimizations to the final program. The output contracts supported by the system are:

- **Same-Key** specifies that every key/value pair processed by the UF will have the same key after the processing. This contract signals the compiler that the partitioning and sort order will be preserved.
- **Super-Key** specifies that each key/value pair that is generated by UF has a super key of the key/value pair(s) that it was generated from. The contract will preserve partitioning and partial order on the keys.
- **Unique-Key** specifies that every key that is written to output is unique in the set. Output key/value pairs are therfore partitioned and grouped by the key.
- **Partitioned-by-Key** similarly to Super-Key specifies that the output key/value pairs will keep the partitioning but the order inside partitions will not be preserved.

E. PACT Compiler

Job of the PACT compiler is to translate PACT workflows into the DAG representation suitable for execution on Nephele. Since the PACT programming model is declarative it leaves space for the compiler to choose among different execution strategies at runtime. All possible strategies together with data sizes and partitioning information are calculated against cost model in order to choose the optimal strategy. In the following paragraphs we will discuss PACT local optimizations, PACT workflow (whole program) optimizations and their translation to Nephele DAG.

Optimizing single PACTs PACT programming abstraction is of declarative nature. Programmer declares what the user function should do but the exact path of execution is not specified. This leaves space for a compiler to adopt execution strategy that is optimal for the data set being processed. For the Match contract there are two possibilities for execution strategies: *i)* broadcasting the smaller data set to all nodes and applying Hybrid-Hash-Join or *ii)* using a classical Sort-Merge-Join. The Cross contract can also be implemented using a broadcast strategy or symmetric-fragment-and-replicate [17] strategy. Using the right strategy can tremendously reduce data transfer costs.

Optimizing across PACTs However, it is not enough to only optimize individual PACTs. Significant performance improvements can be achieved when looking across different PACTs and trying the best strategy for the whole workflow. In this case data partitioning and sort order in one part of the workflow can be reused later. Therefore, PACT compiler uses the same strategies used in Selinger-style SQL optimizer [18]. Optimizer builds execution plans bottom up and prunes more expensive plans. However, if the expensive plan has *interesting properties* (in our case partitioning and sort order) the compiler might keep it. For inferring interesting properties PACT compiler uses Output Contracts.

Generating Nephele DAG The optimized PACTs workflow is then transformed into the Nephele DAG described in subsection IV-A. The user function is wrapped with input contract code which invokes the user function according to its Input Contract and writes its output. The overall execution strategy for a PACT determines the connection pattern between the subtasks, the PACT wrapping code and communication channel code.

F. Evaluation

Detailed programming model evaluation of the Nephele/-PACTs does not fit this document's frame. However, in [19], Nephele/PACTs has been compared to MapReduce against following algorithms: *i)* relational OLAP query, *ii)* representative XPath query, *iii)* K-means data clustering, *iv)* pairwise shortest paths algorithm and *iv)* Edge-Triangle enumeration. It is shown that code written in PACTs programming model is more modular, has more straightforward data flows, does not require auxiliary utilities like data cache and hides many complexities of operations like join.

In order to fairly compare performance of Nephele/PACTs with MapReduce they were run on the same hardware configuration and MapReduce was implemented on top of the Nephele system. Two queries were run: *i)* Join/Aggregation task (the query from Listing 1) and *ii)* Star-Join task. In the first query PACT performed 43% better with cold operating system caches, while it performed 110% better with warm

caches. The second query goes much more in favor of PACT where it performed 2.7 times better with cold caches and 3.7 times better with warm caches compared to MapReduce.

V. Conclusions and future work

All systems covered here take different approaches in providing high level abstractions for large scale data processing. FlumeJava shows that programmers appreciate a collections based interface. However it is not compatible with regular Java collections and Java syntax lacks constructs like lambda expressions and tuple handling. FlumeJava also shows limitations of MapReduce model. Optimizer can not apply optimizations that involve repartitioning and sorting.

DryadLINQ solves many problems of FlumeJava. It provides an interface that is compatible with the widely used LINQ and shows that graph based execution engines provide numerous optimization possibilities. However, DryadLINQ does not provide constructs that guarantee output properties of data so often the optimizer can not infer data partitioning and sort order.

Finally, Nephele/PACTs provides language abstractions that enforce the programmer to think about partitioning and output properties of user functions. This helps the compiler infer interesting properties (sort order and data partitioning) in the execution plan. Moreover, it allows dynamic resource allocation from cloud environment making space for whole new area of research. All frameworks do not support any kind of side effects in anonymous functions.

Finally, we propose Distributed Collections for Scala (DCS). Unlike FlumeJava, DCS has the same interface as regular Scala collections which are well known and appreciated by the community. It supports abstractions like *Set*, *Map*, *Seq* and their sorted counterparts. Also, Scala supports syntactic sugar for anonymous functions and collections have a very rich set of operations. In order to support distributed environments we enhance the set of operations with joins, partitioning and data sampling methods. Furthermore, we support both eager and deferred evaluation thus making DCS suitable for both interactive data analysis and batch jobs. Currently, DCS works only with MapReduce framework but we plan to make the code written with DCS portable between local execution, MapReduce, Nephele and Spark [20]. This way the programmer could use benefits from each system by using the uniform interface. Moreover, we plan to introduce the possibility to change the execution engine dynamically making the library even more flexible.

In contrast to other frameworks DCS supports side effects in anonymous functions. We support *distributed counters* which are initialized to an integer value and can be incremented from anonymous functions that are running on cluster nodes. The second type of side effects that we support are *distributed builders*. They have methods for addition of typed elements. These methods are treated specially in cluster nodes so the builders represent the distributed collections them selves. They allow us to use constructs like *foreach* which do not have a return value but can create new distributed collections through anonymous function side effects.

In order to make the code portable we will implement all DCS operations by using core abstractions. These methods need to be carefully designed for most optimal execution on all frameworks. Some frameworks will not support some features or optimizations of DCS but the code will mostly remain portable. Core abstractions will be a hybrid of FlumeJava core abstractions and Nephele/PACTs contracts. Adapting the DCS to some new framework will then require implementation of core abstractions in the back-end.

References

[1] M. Snir, S. Otto, D. Walker, J. Dongarra, and S. Huss-Lederman, "MPI: the complete reference," 1995.

[2] A. Pavlo, E. Paulson, A. Rasin, D. Abadi, D. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proceedings of the 35th SIGMOD international conference on Management of data*, 2009, p. 165178.

[3] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, p. 107113, 2008.

[4] "Hadoop." [Online]. Available: http://hadoop.apache.org/

[5] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, 2007, p. 5972.

[6] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, p. 10991110.

[7] Y. Yu, M. Isard, D. Fetterly, M. Budiu, . Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008, p. 114.

[8] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "FlumeJava: easy, efficient data-parallel pipelines," *ACM SIGPLAN Notices*, vol. 45, no. 6, p. 363375, 2010.

[9] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with sawzall," *Scientific Programming*, vol. 13, no. 4, p. 277298, 2005.

[10] Y. Yu, M. Isard, D. Fetterly, M. Budiu, . Erlingsson, P. Gunda, and J. Currey, "DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008, p. 114.

[11] "LINQ (Language-Integrated query)." [Online]. Available: http://msdn.microsoft.com/en-us/library/bb397926.aspx

[12] D. Battr, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele/PACTs: a programming model and execution framework for web-scale analytical processing," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, p. 119130.

[13] R. Halstead, "New ideas in parallel lisp: Language design, implementation, and programming tools," *Parallel Lisp: Languages and Systems*, p. 157, 1990.

[14] J. Gray, A. Szalay, A. Thakar, P. Kunszt, C. Stoughton, D. Slutz, and J. VandenBerg, "Data mining the SDSS SkyServer database," *Arxiv preprint cs/0202014*, 2002.

[15] D. Warneke and O. Kao, "Nephele: efficient parallel data processing in the cloud," in *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, 2009, p. 8.

[16] D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Project Website*, 2007.

[17] J. W. Stamos and H. C. Young, "A symmetric fragment and replicate algorithm for distributed joins," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 12, pp. 1345–1354, Dec. 1993.

[18] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price, "Access path selection in a relational database management system," in *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, 1979, p. 2334.

[19] A. Alexandrov, S. Ewen, M. Heimel, F. Hueske, O. Kao, V. Markl, E. Nijkamp, D. Warneke, M. Hovestadt, A. Kliem *et al.*, "MapReduce and PACT-Comparing data parallel programming models."

[20] Z. Matei, "Spark." [Online]. Available: http://spark-project.org/