Massively Parallel Data Analysis with PACTs on Nephele

Alexander Alexandrov, Dominic Battré*, Stephan Ewen*
Max Heimel, Fabian Hueske*, Odej Kao*
Volker Markl*, Erik Nijkamp, Daniel Warneke*

Technische Universität Berlin, Germany
* firstname.lastname@tu-berlin.de

1. INTRODUCTION

Large-scale data analysis applications require processing and analyzing of Terabytes or even Petabytes of data, particularly in the areas of web analysis or scientific data management. This trend has been discussed as "web-scale data management" in a panel at VLDB 2009. Formerly, parallel data processing was the domain of parallel database systems. Today, novel requirements like scaling out to thousands of machines, improved fault-tolerance, and schema free processing have made a case for new approaches.

Among these approaches, the map/reduce programming model [4] and its open-source implementation Hadoop [1] have gained the most attention. Developed for simple logfile analysis, map/reduce systems execute sequential user code in a parallel and fault-tolerant manner, once it has been written to fit the second-order functions map and reduce.

However, with the success of map/reduce, many projects have started to push more complex (e.g., SQL-like) operations into the programming model, thereby violating some of its initial design goals (i.e., separation of parallelization and user code) and paying significant performance penalties.

To eliminate these shortcomings we have developed the *Nephele/PACTs* [3, 7] system, a parallel data processor centered around a programming model of so-called *Parallelization Contracts* (PACTs) and the scalable parallel execution engine *Nephele*. Our system pursues the same design goals as map/reduce and is highlighted by three properties:

1. A richer programming model than map/reduce that preserves the same abstraction level: Our 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.

2. 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.

3. The flexible execution engine *Nephele*: Nephele executes data flow programs modeled as directed acyclic graphs (DAGs) in a parallel and fault-tolerant way. Job annotations enable the PACT compiler to influence the execution in a very fine-grained manner.

To highlight the expressiveness of our programming model, we have implemented a suite of diverse data processing tasks as PACT programs. We will showcase the optimization and compilation of these programs using the visual explain facility of our PACT compiler. The parallel execution of compiled programs will be visualized by a graphical interface that provides immediate feedback on the node utilization, data flow congestions and execution bottlenecks.

Section 2 gives a high-level overview of the Nephele/PACTs system and briefly explains its fundamental concepts. Section 3 discusses the demonstration in detail.

Proceedings of the VLDB Endowment, Vol. 3, No. 2

2. BACKGROUND

This section provides an overview of the PACT programming model and the architecture of the Nephele/PACTs query processor. Due to space constraints, some aspects are explained in a highly simplified manner. A detailed description of the architecture and formalized presentation of PACTs can be found in [3], together with a conclusive discussion of related work.

2.1 The PACT Programming Model

The PACT programming model is a generalization of map/reduce [4]. It is based on a key/value data model and the concept of *Parallelization Contracts (PACTs)*. A PACT consists of exactly one second-order function which is called *Input Contract* and an optional *Output Contract*. An Input Contract takes a first-order function with task-specific user code and one or more data sets as input parameters. The Input Contract invokes its associated first-order function with independent subsets of its input data in a data-parallel fashion. In this context, the well-known functions *map* and *reduce* are examples of Input Contracts. The first-order function has to implement an interface that is specific to the PACT it uses, as it is known from map/reduce.

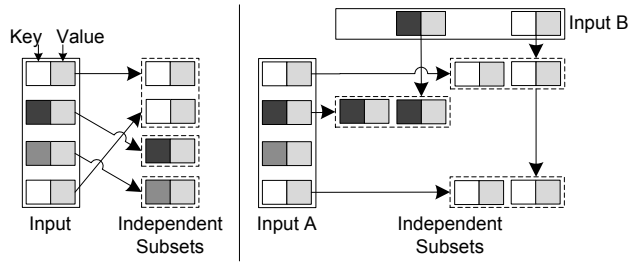

Figure 1: a) Reduce | b) Match

The programmer can attach optional Output Contracts to PACTs to denote certain properties of the user code's output data, which are relevant to the parallelization. The compiler can exploit that information and deduce in some cases, that suitable partitionings or orders exist and reuse them.

Data processing tasks are implemented by providing custom code to PACTs and assembling them to a work flow graph. In the following we list an initial set of Input Contracts and describe them shortly:

Map The *Map* contract is used to process each key/value pair independently. Every key/value pair becomes an independent subset consisting solely of itself.

Reduce The *Reduce* contract partitions key/value pairs by their keys. Every group becomes an independent subset as shown in Figure 1 a). Similar to map/reduce we allow the use of a combiner.

Cross The *Cross* contract operates on multiple inputs and builds a distributed Cartesian product over its input sets. Each element of the Cartesian product becomes an independent subset.

CoGroup The *CoGroup* contract partitions each of its multiple inputs along the key. Independent subsets are built by combining equal keys of all inputs. Hence, the key/value pairs of all inputs with the same key are assigned to the same subset.

Match The *Match* contract operates on multiple inputs. It matches key/value pairs from all input data sets with the same key. All key/value pairs within an independent subset have the same key, but in contrast to *CoGroup* each subset contains only one key/value pair from each input (cf. Figure 1 b)). Hence, the *Match* contract associates key/value pairs from its inputs like an inner join on the key, without actually joining them.

While Input Contracts are mandatory components of PACTs, Output Contracts are optional. They allow developers to guarantee certain behaviors of the user code with respect to the properties of the output data. As an example, the developer can attach the *Same-Key* Output Contract to a function that returns the same key as it was invoked with. The PACT compiler exploits these guarantees to generate more efficient data flow programs. With the mentioned *Same-Key* Output Contract, the compiler can for example deduce that any key-partitioning on the input data of the function still exists on the output data. The compiler can hence avoid unnecessary repartitioning and therefore expensive data shipping.

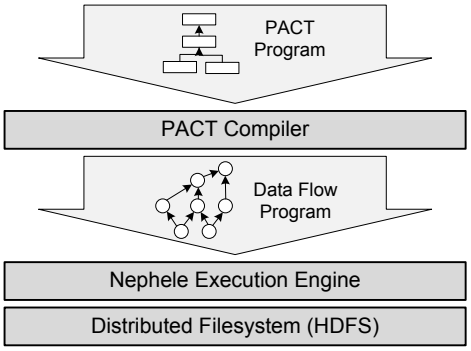

Figure 2: Architecture of the Nephele/PACTs prototype

2.2 Architecture

Our prototype has a three–tier architecture as shown in Figure 2. To execute a PACT program it is submitted to the *PACT Compiler*. The compiler translates the program into a data flow program and hands it to the *Nephele* system for parallel execution. Input/output data is stored in the distributed filesystem HDFS [1]. We will briefly introduce the Nephele system and the PACT compiler in the following.

2.2.1 Nephele

The *Nephele* system [7] executes the compiled PACT programs in a parallel fashion. Similar to systems like Dryad [5], Nephele considers incoming jobs to be DAGs with vertices being subtasks and edges representing communication channels between these subtasks. Each subtask is a sequential program, which reads data from its input channels and writes to its output channels. The initial DAG representation does not reflect parallel execution. Prior execution, Nephele generates the parallel data flow graph by spanning the received DAG. Thereby, vertices are multiplied to the desired degree of parallelism. Connection patterns that are attached to channels define how the multiplied vertices are rewired after spanning. During execution Nephele takes care of resource scheduling, task distribution, communication as well as synchronization issues. Moreover, Nephele's fault-tolerance mechanisms help to mitigate the impact of hardware outages.

Unlike existing systems, Nephele offers to annotate jobs with a rich set of parameters, which influence the physical execution. For example, it is possible to set the desired degree of data parallelism for each subtask, assign particular sets of subtasks to particular sets of compute nodes or explicitly specify the type of communication channels between subtasks. With respect to the PACT layer we leverage these parameters to translate optimization strategies of the PACT compiler into scheduling hints for the execution engine.

Currently, Nephele supports three different types of communication channels: Network, in-memory, and file channels. While network and in-memory channels allow the PACT compiler to construct low-latency execution pipelines in which one task can immediately consume the output of another, file channels collect the entire output of a task in a temporary file before passing its content on to the next task. As a result, file channels can be considered check points, which help to recover from execution failures.

Further details about Nephele can be found in [7].

2.2.2 The PACT Compiler

The PACT compiler translates a PACT program into a Nephele DAG. In contrast to map/reduce [4], our system separates programming model and execution strategy. Due to the declarative character of the PACT programming model, the PACT compiler can choose from several execution plans with varying costs for a single PACT program. In the following, we discuss some optimization opportunities and describe the generation of Nephele DAGs.

Optimizing a single PACT PACTs exhibit a declarative character. They define which independent subsets are generated from the input data and provided to separate instances of the user function, but do not define how that is actually achieved. A single Input Contract can be fulfilled by multiple execution strategies. Among the strategies we consider, several were devised by research on parallel database systems such as repartitioning, broadcasting, and symmetric-fragmentation-and-replication. As an example, the *Match* contract can be satisfied using either a repartition strategy which partitions all inputs by keys or a broadcast strategy that fully replicates one input to every partition of the other input. Choosing the right strategy can tremendously reduce network traffic and execution time.

Optimizing a PACT program Similar as in query optimization for relational DBMS, the optimal execution strategy for a PACT program cannot be found by combining the locally optimal choices for all PACTs. The evaluation of a PACT becomes significantly cheaper when existing properties of the data such as partitionings or sort orders can be exploited. Therefore, our PACT compiler works similar as a Selinger-style SQL optimizer [6] which tracks so-called interesting properties. During optimization more expensive plans are spared from pruning if they provide an interesting property which can be utilized later. The PACT compiler exploits information provided by the Output Contracts to infer that the user code preserves certain properties.

Our compiler's cost model considers data shipping costs. Starting with file size information, we use user annotations, such as factors for changing data volume and key cardinalities, to derive reasonable estimations. If reasonable estimates are impossible due to missing annotations, the compiler picks the strategy that performs best on large input sizes.

Transformation to Nephele DAGs After choosing the execution plan for a PACT program, the PACT compiler must transform it into a Nephele DAG. As described in Section 2.2.1, a Nephele DAG consists of sequential code blocks (vertices) and communication channels (edges). The compiler wraps the user function of each PACT with *PACT code* and maps it to a vertex in the Nephele DAG. The wrapping PACT code invokes the user code with an independent subset of data according to its Input Contract and receives its output. The execution strategy for a PACT is reflected in three aspects within the Nephele DAG: First, in the wiring pattern between the subtasks. Second, in the PACT code that calls the user code, and finally in the PACT code which receives and forwards the data in the preceding vertex.

3. DEMONSTRATION

To demonstrate the power and applicability of our approach we will give a live demonstration of the Nephele/PACTs prototype. Our demo will cover all steps from task implementation to result computation. First, we will show how data processing tasks are implemented as PACT programs. The demo continues with the compilation of a task. A visualization of the compiled task provides insight into the optimization capabilities of Nephele/PACTs. Finally, a graphical interface displays the progress of the task's parallel execution. In the remainder of this section we will discuss these steps in details and present a running example.

3.1 Writing a PACT Program

We have implemented a suite of data analysis tasks to demonstrate the expressive power of the PACT programming model. These tasks include selected relational OLAP queries originating from the TPC-H benchmark [2] as well as data mining tasks such as K-Means clustering and frequent-itemset mining on synthetic data sets. Furthermore, we will provide graph algorithms to analyze a previously crawled data set of linked open data. We will show how these tasks are implemented as PACT programs. The compilation and execution of each of these tasks can be shown during the demo.

For the running example we have chosen a simplified query from the TPC-H workload:

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

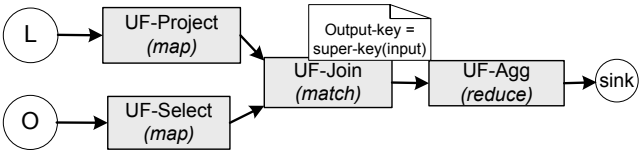

Figure 3: The example PACT program

Figure 3 shows a simplified, graphical representation of the PACT program for the query. Selection and projection are implemented using *Map* PACTs. The join is expressed using a *Match* PACT. The *Super-Key* Output Contract states that the keys produced by the user code are superkeys of the input keys. Finally, the *Reduce* PACT performs the aggregation.

3.2 Compiling a PACT Program

As emphasized in Section 2.2.2, transforming a PACT program to a Nephele DAG inherits large optimization potential. In contrast to classic relational database systems, the PACT compiler must cope with arbitrary user code with unknown semantics. We will present how the PACT compiler generates efficient data flow graphs from our suite of data processing tasks for varying degrees of parallelism. A visual explain tool shows the result of the optimization as depicted in Figure 4. The tool displays the data shipping strategy, local strategy,

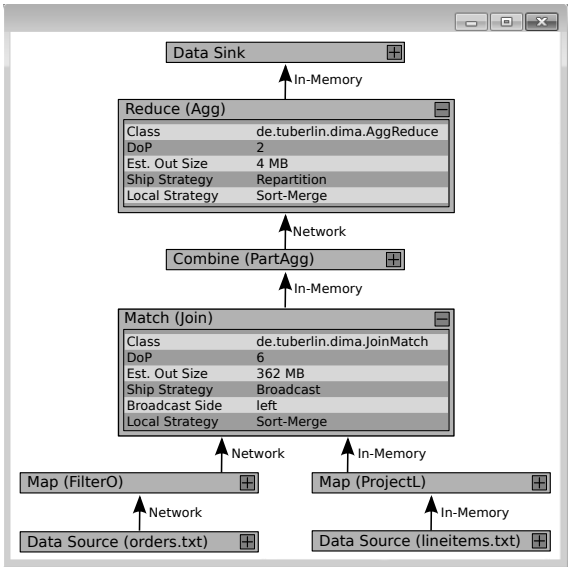

Figure 4: Screenshot of the visual explain of a compiled PACT program

degree of parallelism (DoP), and channel types. Furthermore, the estimated output sizes are shown.

Figure 4 shows the optimized data flow graph for the example query. The *Match* PACT which implements the join is realized by broadcasting the filtered *orders* relation. The *lineitem* relation is locally projected, joined, and pre-aggregated as indicated by the in-memory channels). The final aggregation is performed using a repartition strategy.

3.3 Executing a PACT Program

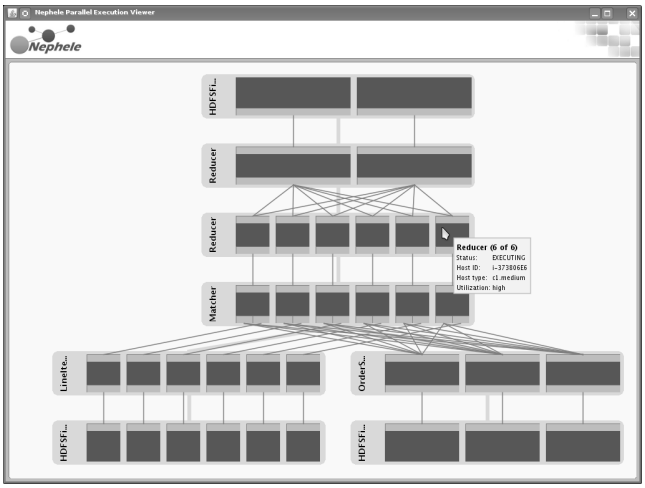

Figure 5: Screenshot of Nephele's graphical execution interface

The Nephele execution engine evaluates the Nephele DAG generated by the PACT compiler in parallel. In order to do so, it spans the received DAG to a parallel data flow and distributes the respective subtasks among the available compute resources. Our demonstration setup will feature a medium-sized cloud system which is accessible through a commodity Internet connection.

During the task execution Nephele will monitor the involved compute nodes and record statistics on the CPU and network utilization. A graphical interface will display the collected statistics. Hence, immediate visual feedback on the efficiency of the parallel execution strategies chosen by the PACT compiler is provided. Furthermore, performance bottlenecks resulting from network congestions or inappropriate degrees of parallelism can be detected.

Figure 5 shows the execution of our example task. The local pipelining of the *lineitem* relation can be observed following the wiring from the bottom left tasks up to the pre-aggregation tasks. Furthermore, the broadcast of the *orders* relation and the partitioning for the final aggregation can be identified. The tasks' degree of parallelism corresponds to the output of the compiler (Figure 4).

4. CONCLUSIONS

We will demonstrate the Nephele/PACTs query processor, a system for massively parallel data processing based on the concept of *Parallelization Contracts*. The demonstration will show the implementation, compilation, optimization, and parallel execution of PACT programs on the Nephele execution engine. A suite complex data analysis tasks will highlight the expressive power of the PACT programming model. We will showcase the usage of our system step-by-step, starting with the definition of PACT programs until their execution finishes. Visualization tools will give insight into each step of the processing of a PACT program.

Acknowledgments

We thank HP for supporting this work through an Open Collaboration Grant as well as IBM for a Shared University Research hardware grant.

5. REFERENCES

[1] Hadoop. URL: http://hadoop.apache.org.

[2] TPC-H. URL: http://www.tpc.org/tpch/.

[3] D. Battré, 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 *Symposium on Cloud Computing*, 2010.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[5] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In P. Ferreira, T. R. Gross, and L. Veiga, editors, *EuroSys*, pages 59–72. ACM, 2007.

[6] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In P. A. Bernstein, editor, *SIGMOD Conference*, pages 23–34. ACM, 1979.

[7] D. Warneke and O. Kao. Nephele: Efficient Parallel Data Processing in the Cloud. In I. Raicu, I. T. Foster, and Y. Zhao, editors, *SC-MTAGS*. ACM, 2009.