Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing

Dominic Battré
Odej Kao

Stephan Ewen
Volker Markl

Fabian Hueske
Daniel Warneke

Technische Universität Berlin
Einsteinufer 17
10587 Berlin
Germany
firstname.lastname@tu-berlin.de

ABSTRACT

We present a parallel data processor centered around a programming model of so called *Parallelization Contracts* (PACTs) and the scalable parallel execution engine *Nephele* [18]. The PACT programming model is a generalization of the well-known map/reduce programming model, extending it with further *second-order functions*, as well as with *Output Contracts* that give guarantees about the behavior of a function. We describe methods to transform a PACT program into a data flow for Nephele, which executes its sequential building blocks in parallel and deals with communication, synchronization and fault tolerance. Our definition of PACTs allows to apply several types of optimizations on the data flow during the transformation.

The system as a whole is designed to be as generic as (and compatible to) map/reduce systems, while overcoming several of their major weaknesses: 1) The functions *map* and *reduce* alone are not sufficient to express many data processing tasks both naturally and efficiently. 2) Map/reduce ties a program to a single fixed execution strategy, which is robust but highly suboptimal for many tasks. 3) Map/reduce makes no assumptions about the behavior of the functions. Hence, it offers only very limited optimization opportunities. With a set of examples and experiments, we illustrate how our system is able to naturally represent and efficiently execute several tasks that do not fit the map/reduce model well.

Categories and Subject Descriptors

H.2.4 [**Database Management**]:
Systems—*Parallel Databases*

General Terms

Design, Languages, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SoCC'10, June 10–11, 2010, Indianapolis, Indiana, USA.
Copyright 2010 ACM 978-1-4503-0036-0/10/06 ...$10.00.

Keywords

Web-Scale Data, Cloud Computing, Map Reduce

1. INTRODUCTION

The term *Web-Scale Data Management* has been coined for describing the challenge to develop systems that scale to data volumes as they are found in search indexes, large scale warehouses, and scientific applications like climate research. Most of the recent approaches build on massive parallelization, favoring large numbers of cheap computers over expensive servers. Current multicore hardware trends support that development. In many of the mentioned scenarios, parallel databases, the traditional workhorses, are refused. The main reasons are their strict schema and the missing scalability, elasticity and fault tolerance required for setups of 1000s of machines, where failures are common.

Many new architectures have been suggested, among which the *map/reduce* paradigm [5] and its open source implementation Hadoop [1] have gained the most attention. Here, programs are written as *map* and *reduce* functions, which process key/value pairs and can be executed in many data-parallel instances. The big advantage of that programming model is its generality: Any problem that can be expressed with those two functions can be executed by the framework in a massively parallel way. The map/reduce execution model has been proven to scale to 1000s of machines. Techniques from the map/reduce execution model have found their way into the design of database engines [19] and some databases added the map/reduce programming model to their query interface [8].

The map/reduce programming model has however not been designed for more complex operations, as they occur in fields like relational query processing or data mining. Even implementing a join in map/reduce requires the programmer to bend the programming model by creating a tagged union of the inputs to realize the join in the *reduce* function. Not only is this a sign that the programming model is somehow unsuitable for the operation, but it also hides from the system the fact that there are two distinct inputs. Those inputs may be treated differently, for example if one is already partitioned on the key. Apart from requiring awkward programming, that may be one cause of low performance [13].

Although it is often possible to force complex operations into the map/reduce programming model, many of them require

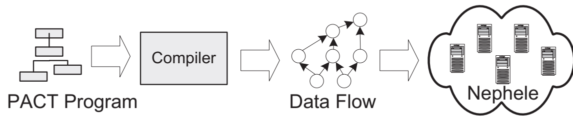

Figure 1: Compiling a program to a data flow.

to actually describe the exact communication pattern in the user code, sometimes as far as hard coding the number and assignment of partitions. In consequence, it is at least hard, if not impossible, for a system to perform optimizations on the program, or even choose the degree of parallelism by itself, as this would require to modify the user code. Parallel data flow systems, like Dryad [10], provide high flexibility and allow arbitrary communication patterns between the nodes by setting up the vertices and edges correspondingly. But by design, they require that again the user program sets up those patterns explicitly.

This paper describes the *PACT programming model* for the *Nephele* system. The PACT programming model extends the concepts from map/reduce, but is applicable to more complex operations. We discuss methods to compile PACT programs to parallel data flows for the Nephele system, which is a flexible execution engine for parallel data flows (cf. Figure 1). The contributions of this paper are summarized as follows:

- We describe a **programming model**, centered around key/value pairs and *Parallelization Contracts* (PACTs). The PACTs are second-order functions that define properties on the input and output data of their associated first-order functions (from here on referred to as "user function", *UF*). The system utilizes these properties to parallelize the execution of the UF and apply optimization rules. We refer to the type of the second-order function as the *Input Contract*. The properties of the output data are described by an attached *Output Contract*.

- We provide an **initial set of Input Contracts**, which define how the input data is organized into subsets that can be processed independently and hence in a data parallel fashion by independent instances of the UF. *Map* and *Reduce* are representatives of these contracts, defining, in the case of *Map*, that the UF processes each key/value pair independently, and, in the case of *Reduce*, that all key/value pairs with equal key form an inseparable group. We describe additional functions and demonstrate their applicability.

- We describe **Output Contracts** as a means to denote some properties on the UF's output data. Output Contracts are attached to the second-order function by the programmer. They describe additional semantic information of the UFs, which is exploited for optimization in order to generate efficient parallel data flows.

- We present rules and a **method to create an efficient parallel data flow** for a given PACT program. Note that the second-order functions have a certain declarative aspect: They describe guarantees about what data the UF receives per independent invocation (for example groups of same keys in the case of *Reduce*), but do not imply directly how to actually produce those guaranteed properties. Some of the new functions allow several different algorithms (like either partitioning, replication, or a combination of both) to produce the required properties. To exploit that declarativity, our system separates the programming model from the execution. The compiler uses a series of transformation and optimization rules to generate an efficient parallel data flow from the PACT program.

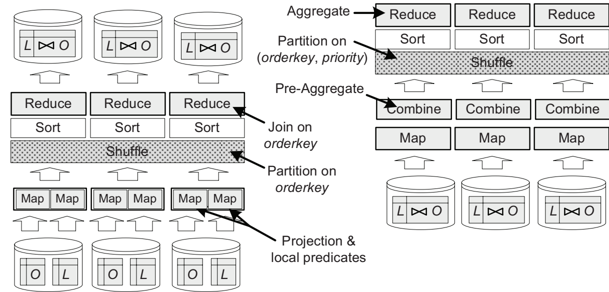

Figure 2: The sample task as map/reduce jobs

In order to illustrate the suggested approach, we discuss a motivating example in Section 2. Section 3 will briefly describe the Nephele System. Sections 4 and 5 give a formal definition of *Parallelization Contracts*, list an initial set of defined contracts, and describe strategies to parallelize and optimize the program execution. In Section 6 we present evaluation results from our proof-of-concept implementation. Related work is discussed in Section 7; Section 8 concludes the paper.

2. A SAMPLE TASK

To motivate the implications of our approach, we discuss a simple example task in this section. The example task is a simplification of TPC-H [2] query 3 (Shipping Priority Query), which we will compare in its map/reduce and PACTs implementation:

```
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
```

To express the query in map/reduce, we need two jobs: The first one performs the join, the second one the aggregation. Figure 2 shows the execution plan. The jobs are depicted as running on three nodes in parallel. Boxes of the same type next to each other (like the Reduce or Sort boxes) describe data parallel operations. The end-to-end shuffle box describes the dam, where the nodes have to repartition the data.

The first job realizes a repartition merge join, and the shuffling phase sends the whole *LINEITEM* table over the network. The group/aggregate job partitions the rows by a superkey of the previous join key. Since there is no way to express the key/superkey relationship, the framework cannot assume the pre-existing partitioning and hence cannot reuse it. Furthermore, implementations like Hadoop write the

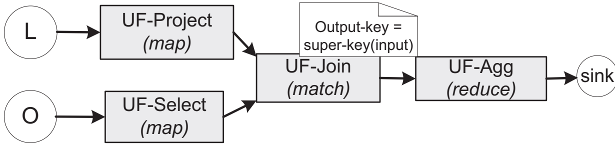

Figure 3: The sample PACT program

result of every job into the distributed file system, which additionally prohibits to reuse any partitioning. Consequently, the query requires two partitioning steps (shuffling phases). The first of those is very expensive, as it sends all rows from the *LINEITEM* table over the network.

The corresponding PACT program for the query is given in Figure 3. Complete PACT programs are work flows of user functions, starting with one or more data sources, which are files in a distributed file system, and ending with one or more data sinks. Each UF implements the interface of an Input Contract. Besides the well-known *Map* and *Reduce* contracts, the novel *Match* contract is used by the user function implementing the join (UF-join). The *Match* contract provides two key/value pairs as input for the UF. It guarantees that each pair of key/value pairs (with the same key) from the two different inputs is supplied to exactly one parallel instance of the user function. The selections and projections are implemented in UF-Project and UF-Select (*Map*), the aggregation in UF-Agg (*Reduce*).

When the compiler generates the parallel data flow for the task, it has several choices. Two of them are shown in Figure 4. Strategy (a) filters and projects both tables with the *Map* type UFs. Then, each parallel instance broadcasts its produced key/value pairs from the *ORDERS* table to all other nodes, so that each node has a the complete bag of selected rows. The rows from the *LINEITEM* table remain on their original nodes. The system can now form UF-Join's input, which is all pairs of *ORDER* and *LINEITEM* rows having the same key. Different pairs with the same key will possibly be processed in different parallel instances of UF-Join, but that is consistent with the *Match* contract, as it is with an inner join. UF-Join's output is partitioned and sorted by the grouping columns and aggregated in UF-Agg (*Reduce*). A *Combiner* is used with the *Reducer*, as it is typical for map/reduce programs [5]. Since the result of the join is a lot smaller than the *LINEITEM* table, fewer of its rows have to be sent over the network here.

Strategy (b) filters and projects both tables with the *Map* type user functions as well. It then partitions both outputs after the *ORDERKEY* and shuffles the partitions across the parallel instances of UF-Join (*Match*). This strategy is similar to the first map/reduce job. However, after the join, the plan does not repartition the result. Instead, it just performs an additional sort to have a secondary order after *SHIPPRIORITY* within the rows of a partition. The system can make that optimization, because the Output Contract of UF-Join states that the output key is a superkey of the input key (cf. Figure 3). Compared to the map/reduce plan, this strategy saves one shuffling phase. In addition, the second sort is very cheap, as it only sorts partitions with identical *ORDERKEY* on *SHIPPRIORITY*.

3. BACKGROUND: NEPHELE

We use the *Nephele* system [18] as the physical execution engine for the compiled PACT programs. Nephele executes DAG-based data flow programs on dynamic compute clouds. It keeps track of task scheduling and setting up the required communication channels. In addition, it dynamically allocates the compute resources during program execution. Moreover, Nephele provides fault-tolerance mechanisms which help to mitigate the impact of temporary or permanent hardware outages.

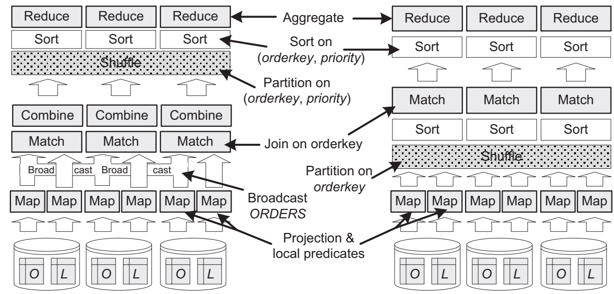

Figure 4: Two different execution strategies for the PACT program

Although in principle the presented PACT programming model could be set up to run on top of other DAG-based execution engines as well (e. g. Dryad [10]), we chose Nephele for the following two reasons:

First, Nephele offers a rich set of parameters which allow to influence the physical execution schedule of the received data flow program in a versatile manner. For instance, it is possible to set the desired degree of data parallelism for each task individually, assign particular sets of tasks to particular sets of compute nodes or explicitly specify the type of communication channels between tasks. With respect to the PACT layer we can use these parameters to translate optimization strategies of the PACT compiler into scheduling hints for the execution engine. In case the provided parameters are insufficiently specified to construct a full execution schedule, Nephele will apply default strategies to determine a reasonable degree of parallelism based on the number of available nodes and the data source.

Second, Nephele is highlighted by its ability to deal with dynamic resource allocation. The system can independently request new compute nodes from a commercial cloud like Amazon EC2 in order to match the occurring workload. Although this feature is currently only used in a very limited fashion, we plan to leverage it for load balancing and dynamic reoptimization in the future.

As shown in Fig. 1, the PACT compiler generates a DAG representing the data flow. Each vertex of the Nephele DAG is generated for one user function of the original PACT work flow (cf. Fig. 3). However, in addition to the UF code, the vertex may contain code that the PACT compiler has generated in order to fulfill the respective contracts. Moreover, the PACT compiler has added instructions on how to setup the communication channels between two sets of tasks, in case they are executed in parallel. That process is described further in Section 5.

Similar to Microsoft's Dryad, Nephele offers three different types of communication channels: Network, in-memory and file channels. While network and in-memory channels allow the 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.

4. THE PACT PROGRAMMING MODEL

The *PACT Programming Model* is a generalization of the map/reduce programming model [5]. It operates on a key/value data model and is based on so-called *Parallelization Contracts (PACTs)*. Following the PACT programming model, programs are implemented by providing task specific user code (the user functions, UFs) for selected PACTs and assembling them to a work flow. A PACT defines properties on the input and output data of its associated user function. Figure 5 shows the components of a PACT, which are exactly one *Input Contract* and an optional *Output Contract*. The Input Contract of a PACT defines, how the UF can be evaluated in parallel. Output Contracts allow the optimizer to infer certain properties of the output data of a UF and hence to create a more efficient execution strategy for a program. In this section we will first give a formal definition and present an initial set of Input Contracts. After that we discuss Output Contracts and how they can be used for optimization.

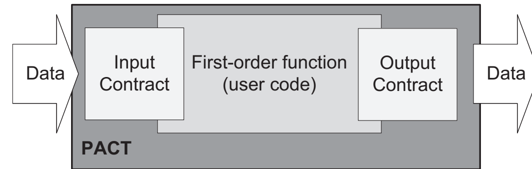

Figure 5: PACT components

4.1 Input Contracts

A key concept of parallel data processing is data parallelism. It is based on the fact that many data processing tasks do not require the context of the whole data set. Hence, subsets of the data can be independently processed. In the following, we call those subsets parallelization units (PUs). The number of parallelization units is task- and data-dependent and determines the maximum degree of parallelization. We propose *Input Contracts* as a way to describe how to generate those parallelization units, which different instances of the user function then process independently. In that sense, Input Contracts are a second-order function that calls its first-order function (the user function) on the input data following a specific strategy. That strategy corresponds to the formation of the parallelization units. Since, the Input Contract is the central aspect of a PACT, we frequently refer for example to a PACT with a *Match* Input Contract as a *Match PACT*.

For better understanding we start with the definition of *Single–Input Contracts*, which apply to user functions with a single input, and show how the well-known *Map* and *Reduce* functions fit into this definition. We define contracts for multiple inputs later on and present representatives of both types.
Note: Our formal definitions here are more general than it is required for the presented initial set of Input Contracts. We keep them general in order to support further, more complex PACTs as well.

4.1.1 Single–Input Contracts

The input of a Single–Input Contract is a set of key/value pairs $S = \{s\}$. For a key/value pair $s = (k, v)$ the function $key(s)$ returns the key k. A Single–Input Contract is defined as a mapping function[1] m that assigns each $s \in S$ to one or more parallelization units u_i. Hence, a PU u_i is a set of key/value pairs for which holds $u_i \subseteq S$. The set of all PU is denoted with $U = \{u_i\}$.
We define the mapping function m as:

$$m : s \rightarrow \{i \mid i \in \mathbb{N} \wedge 1 \leq i \leq |U|\} ,$$

and generate all $u_i \in U$ such that:

$$u_i = \{s \mid s \in S \wedge i \in m(s)\}.$$

Note: Our definition allows arbitrary, possibly non–disjunctive partitionings of the input set. Hereby, each partition becomes an independent parallelization unit. For $|m(s)| = 1, \forall s \in S$, U is a disjunctive partitioning of the input set S.

Following that definition, the operators of the well-known map/reduce programming model [5], namely *Map*, *Reduce*, and *Combine* can be expressed as Single–Input Contracts as follows:

Map The *Map* contract is used to independently process each key/value pair; consequently, the UF is called independently for each element of the input set. The following definition guarantees that each key/value pair is assigned to a single partition that solely consists of itself.

$$|m(s)| = 1 \wedge m(s_i) \neq m(s_j), \text{ for } i \neq j \quad (1)$$

Reduce / Combine The *Reduce* contract partitions key/value pairs by their keys. The formal definition assures that all pairs with the same key are grouped and handed together in one call to same instance of the UF.

$$|m(s)| = 1 \wedge (m(s_i) = m(s_j)) \Leftrightarrow (key(s_i) = key(s_j)) \quad (2)$$

As described in [5], the evaluation of *Reduce* typed functions can often be significantly improved by the use of a *Combiner*, if the UF that implements *Reduce* has certain properties. Since, the *Combine* contract is not deterministic, it can only be used as a preprocessing step before a *Reduce* function.
Similar to *Reduce*, the *Combine* contract groups the key/value pairs by their keys. However, it does only assure that all pairs in a partition have the same key, not that all pairs with the same key are in the same partition. Therefore, *Combine* can produce more than one partition for each distinct key.

$$|m(s)| = 1 \wedge (m(s_i) = m(s_j)) \Rightarrow (key(s_i) = key(s_j)) \quad (3)$$

4.1.2 Multi–Input Contracts

A Multi–Input Contract is defined over n sets of key/value pairs $S_i = \{s_i\}$, for $1 \leq i \leq n$. Again the function $key_i(s_i)$

[1] Here, we do not refer to the *map* function of the map/reduce programming model.

returns the key k of the key/value pair $s_i = (k, v)$.
A Multi–Input Contract generates from its input sets a set of parallelization units $U = \{u\}$, where a parallelization unit u is defined as $u = (p_1, \ldots, p_n)$, with $p_1 \subseteq S_1, \ldots, p_n \subseteq S_n$. For each input set S_i a set of subsets $P_i = \{p_{(i,j)}\}$ is generated by using a mapping function m_i that is defined as:

$$m_i : s_i \to \{j \mid j \in \mathbb{N} \wedge 1 \leq j \leq |P_i|\}$$

All $p_{(i,j)} \in P_i$ are generated as:

$$p_{(i,j)} = \{s_i \mid s_i \in S_i \wedge j \in m_i(s_i)\}$$

Given a set of subsets P_i for each input set S_i, the set of parallelization units U is generated as follows:

$$U = \{u \mid a(u) = true, u \in (P_1 \times P_2 \times \ldots \times P_n)\}$$

where $a(\cdot)$ is an association function that decides whether a parallelization unit u is valid:

$$a : u \to Boolean$$

Similar to our definition of Single–Input Contracts, this definition allows for an arbitrary partitioning of each individual input set. The association function a defines which parallelization units are built by combining partitions of all inputs. In the following we present an initial set of Multi–Input Contracts.

Cross The *Cross* contract is defined as the Cartesian product over its input sets. The user function is executed for each element of the Cartesian product.

$$\begin{gathered} m \text{ for each input like } \mathit{Map}\text{, see Eq. (1)} \\ a(u) = true \end{gathered} \tag{4}$$

CoGroup The *CoGroup* contract partitions the key/value pairs of all input sets according to their keys. For each input, all pairs with the same key form one subset. Over all inputs, the subsets with same keys are grouped together and handed to the user function.

$$\begin{gathered} m \text{ for each input like } \mathit{Reduce}\text{, see Eq. (2)} \\ a(u) = true \text{ if } key_1(s_1) = \ldots = key_m(s_m), \\ \text{for } u = \{p_1, \ldots, p_m\} \wedge \forall s_1 \in p_1, \ldots, \forall s_m \in p_m \end{gathered} \tag{5}$$

Match The *Match* contract is a relaxed version of the *CoGroup* contract. Similar to the *Map* contract, it maps for all input set each key/value pair into a single partition that consists solely of itself. The association rule assures that all parallelization units are valid which contain only key/value pairs with the same key. Hence, *Match* fulfills the requirements of an inner equi–join.

$$\begin{gathered} m \text{ for each input like } \mathit{Map}\text{, see Eq. (1)} \\ a \text{ identical to } \mathit{CoGroup}\text{, see Eq. (5)} \end{gathered} \tag{6}$$

4.2 Output Contracts

An Output Contract is an optional component of a PACT and gives guarantees about the data that is generated by the assigned user function. This information can be used by a compiler to create a more efficient data flow. Our initial set consists of the following Output Contracts:

Same-Key Each key/value pair that is generated by the UF has the same key as the key/value pair(s) that it was generated from. This means the function will preserve any partitioning and order property on the keys. Because functions implementing *Cross* may be called with two different keys, the contract has to specify to which input it refers.

Super-Key Each key/value pair that is generated by the UF has a superkey of the key/value pair(s) that it was generated from. This means the function will preserve a partitioning and partial order on the keys. As before, this contract needs to specify the input side when used with a *Cross*.

Unique-Key Each key/value pair that is produced has a unique key. The key must be unique across all parallel instances. Any produced data is therefore partitioned and grouped by key. This contract can for example be attached to a data source, if it is known that only unique keys are stored.

Partitioned-by-Key Key/value pairs are partitioned by key. This contract has similar implications as the *Super-Key* contract, specifically that a partitioning by the keys is given, but no order inside the partitions. This contract can be attached to a data source that supports partitioned storage. The compiler will then be able to exploit that property.

5. PARALLEL EXECUTION

The complete process of creating a parallel data flow for a PACT program can be separated into two steps which are illustrated in Figure 6. For space reasons the figure does not show data sources and sinks. We consider the PACT program presented in Section 2.

First, the compiler transforms the PACT program into a Nephele DAG. The Nephele DAG is a compact representation of a parallel data flow. It consists of vertices and edges. Each vertex contains one of the previously specified user functions wrapped with PACT code. The PACT code is responsible for preprocessing the input (e. g. to sort it) so that it meets the properties of the Input Contract the respective UF has been written against. The edges between the vertices denote the communication channels which are used to transport data between the UFs. The resulting Nephele DAG is handed to the Nephele system.

Second, the Nephele system *spans* the compact Nephele DAG and obtains a parallel data flow by creating multiple instances of each vertex. The number of parallel instances is a parameter of the vertex and initially set by the PACT compiler. Each vertex may have a different number of parallel instances. Hence, it is possible to have different degrees of parallelism for different parts of the program. In Figure 6, Vertex 3 is spanned to twice as many parallel instances as the other ones.

Creating multiple instances from a vertex requires to multiply its associated communication channels as well. The actual way of multiplying the channels again depends on the data routing strategy to fulfill the Input/Output Contract of the respective UFs and is determined by the PACT code inside the Nephele vertex. E.g. for the *Map* contract it might be sufficient to connect each parallel instance to only one parallel instance of the preceding vertex (point-wise connection pattern). Other strategies repartition the data, so that each parallel instance of the consuming vertex must be connected

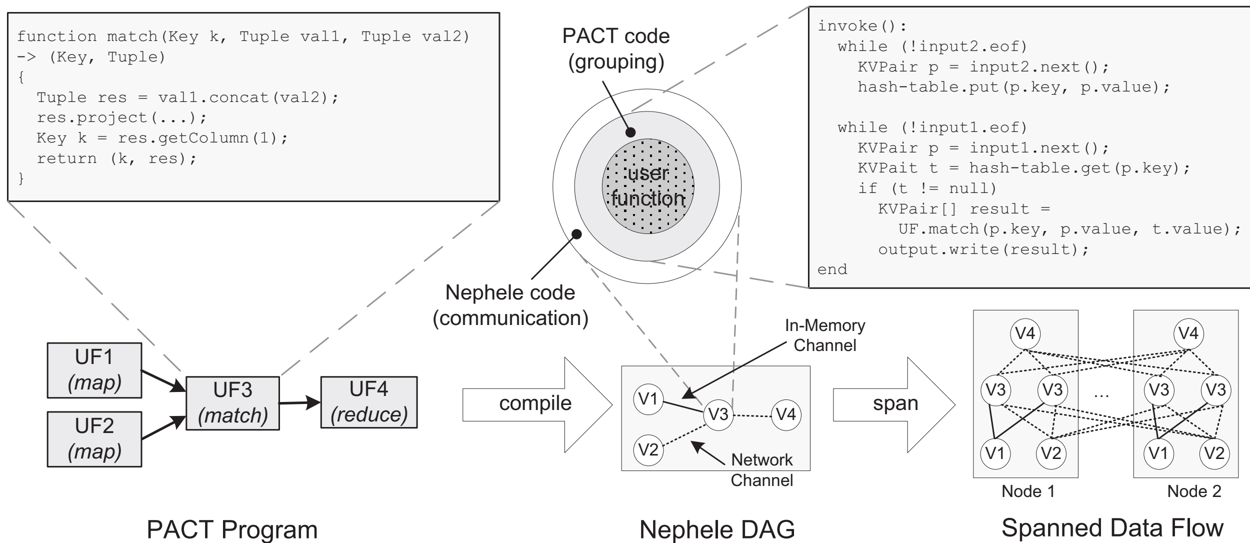

Figure 6: Compiling a PACT program

to each parallel instance of the producing vertex (complete bipartite connection pattern). The choice of the connection pattern is made by the compiler when generating the Nephele DAG. In Figure 6, the network channels between Vertices 2 and 3, as well as between 3 and 4 are multiplied with a complete bipartite connection pattern, while the channel between Vertices 1 and 3 is multiplied with a variant of the point-wise connection pattern.

In general, the transformation of the output from one user function to the input for the succeeding UF depends on three aspects: First, the connection pattern of the edge between the vertices that the functions are contained in. Second, the PACT code that wraps the first UF. Depending on its Output Contract and the Input Contract of the succeeding UF, it may postprocess the output (e. g. sort it to produce partially sorted streams) and choose how to distribute the data for further processing. Third, the transformation depends on the code that wraps the succeeding UF, which can for example merge the input streams, or hash-group the key/value pairs. Each Input/Output Contract is fulfilled by generating and setting those three parts correctly.

5.1 Parallelizing a Single Function

Parallelizing a user function implementing the **Map** contract is trivial. In the considered example, its corresponding Nephele vertex is connected via an in-memory channel with a point-wise connection pattern to the preceding vertex. No special code is needed on the preceding vertex side, and the UF wrapping code simply calls the function for each key/value pair independently.

The **Reduce** PACT states that all key/value pairs with the same key are given to one instance of the user function in one call. Hence, the input must be grouped by the key. Parallelizing such a UF requires partitioning the data on the key. Such a partitioning is in some cases already established. For that, the preceding UF's contract must be realized through a partitioning and the UF must specify an Output Contract stating that the partitioning is preserved (*Same-Key* or *Super-Key*). In the introductory example from Figure 4, plan variant (b) shows that situation. The Nephele vertex for the user function connects here to the preceding vertex with an in-memory channel and a point-wise distribution pattern.
In the case that the partitioning is not already established, we create a repartitioning step comparable to Hadoop's shuffle phase. The edge in the Nephele DAG that connects the preceding vertex to the one containing the *Reduce* user function is set to a complete bipartite distribution pattern. The preceding UF's vertex chooses the target channel by applying a hash function to the key.
The code wrapping the user function additionally sorts its partition on the key to group the input, if no sort order pre-exists. The pre-existence of a sort order can be deduced similarly as the pre-existence of the partitioning. The user function is called once per group of values with the same key, having as parameters the key and an iterator over the values. If a partitioning pre-exists, but no sorted order, and the *Reduce* contract has been given a *Combine* function, we perform an additional optimization: The wrapping code reads the key/value pairs from the input. For a new key, the value is inserted into a hash-table. If a value with the same key is already present in the hash-table, we use the combiner to merge them to one value. The *Reduce* function is called only to finalize the values. This strategy is basically a hash-grouping strategy with running aggregates. If the combiner actually produces multiple key/value pairs, we fall back to the sorted grouping strategy.

The **CoGroup** contract is similar to the *Reduce* contract, but the functions that implement it have two different inputs. Besides grouping key/value pairs from each input by the key, the groups with the same key from both inputs have to be processed together in one UF call. If only one of the inputs has a group, then an empty group is created for the other input. As for the *Reduce* contract, a partitioning according

to the key has to be established for both inputs and the partitioning scheme has to be identical for both inputs. The same rules as for the *Reduce* contract are used to deduce the pre-existence of a partitioning or to create one. The rules are applied to each input individually, which means that also a pre-existing partitioning on one input can be exploited.
The code wrapping the user function sorts both inputs after the key. It calls the user function, providing as parameters the key and two iterators over the values from the two inputs.

Match guarantees that the each two key/value pairs from the two inputs that have the same key are processed at one parallel instance of the user function. It can be realized in the same way as the *CoGroup* contract (cf. Figure 4 (b)). The fact that the contract permits multiple calls to the user functions with the same key (and a different pair of values) allows the system to use another strategy as well: When one of the inputs is significantly smaller than the other, it is broadcasted to all nodes, allowing the other input to be processed locally. To broadcast one side, its edge is parametrized to use the complete bipartite connection pattern. The output code from the preceding vertex is set to send each output key/value pair over each channel instance.
Let s be the size of the smaller input, r the size of the larger input, and p the number of nodes to be involved in executing the user function implementing *Match*. The communication costs for partitioning an input is roughly its size, yielding for a repartitioning costs between 0, if a suitable partitioning pre-exists for both inputs, and $r + s$, if both inputs have to be partitioned. The communication costs for replicating the smaller input are roughly $p * s$. Whichever strategy leads to smaller communication costs is chosen here by the PACT compiler.
The code wrapping the user function needs to build each pair of values with the same key and call the function with it. For that, we use a Hybrid-Hash-Join algorithm in the case that one of the sides was broadcasted, and a Sort-Merge-Join algorithm otherwise.

The **Cross** contract creates a Cartesian product of the two inputs and distributes it over instances of the user function. In many cases, one of the sides is very small and hence lends itself to be broadcasted, as it is done for the *Match* contract. In the case where both inputs are of similar size, we use a symmetric-fragment-and-replicate strategy [16] to span the Cartesian product across all instances. That strategy is realized by tailored distribution patterns for the input edges.

Some of the mentioned choices for parallelization depend on the input sizes. The sizes of the initial inputs is always known by the system, simply by the sizes of the files in the distributed file system. Estimates about the size of intermediate results are generally hard to make and error-prone. Without numbers for these sizes, we pick the most robust strategy, if multiple strategies are possible. The most robust one for the *Match* is the partitioning based, for the *Cross* contract the symmetric-fragment-and-replicate strategy.
In the current version, we offer three different annotations to inform the compiler about sizes to expect: The first one gives the ratio of the output size to the input size (or the larger of the two inputs' sizes). The second one gives the ratio of the output key cardinality to the input key cardinality (or the larger of the two inputs' key cardinalities). The third one states how many key/value pairs a single call to the user function produces, at most (-1 if no upper bound can be given).
In most cases, the plan does not consist of a single pipeline, but has points where the data is materialized. Such points are for example file channels, external sorts, or other materialization points used for fault tolerance. The part of the PACT program that corresponds to the data flow behind such points can be compiled after the results for these points have been materialized. In the future, we plan to make use of that fact and defer parts of the compilation until runtime. That way, the compiler can determine the size of the next input exactly.

5.2 Optimizing across PACTs

The parallelization strategies for the individual user functions cannot be chosen independently, when attempting to produce an efficient parallel data flow. As can be seen in the introductory example (Fig. 4), the decision to parallelize UF-Join (*Match*) via a broadcasting strategy implies that a partitioning step is required to fulfill UF-Agg's *Reduce* contract. Parallelizing UF-Join through a partitioning based strategy lowers the costs to fulfill UF-Agg's PACT to almost zero.
Optimizing the selection of the strategies across multiple contracts can be done as in a Selinger-style SQL optimizer [15]. Such optimizers generate plan candidates bottom-up (from the data sources) and prune the more expensive ones. So called *interesting properties* are generated top-down to spare plans from pruning that come with an additional property that may amortize their cost overhead later. Using partitioning and sort order as interesting properties, such an optimizer chooses a globally efficient plan.

6. EVALUATION AND EXPERIMENTS

This section evaluates the benefits of Nephele/PACTs over pure map/reduce, which is the closest comparable distributed data-centric programming model. We compare first how certain tasks can be expressed using only map/reduce and using the extended set of PACTs. Afterwards, we show how data flows compiled from PACT programs outperform those created from map/reduce programs.

6.1 Programming Model Evaluation

In the following, we present some examples showing how certain data processing tasks can be expressed using the PACT programming model. Although these examples are restricted to use only the set of PACTs presented in this paper, other task can be expressed by extending the programming model with additional PACTs. Tasks that can be expressed using only *map* and *reduce* have been presented in [5] and a lot of follow-up literature. We start the discussion with well–known relational operations and continue with tasks from other contexts.

Inner–Join To realize an inner–join in map/reduce one must work around the fact that the *map* and *reduce* functions have only a single input. That is typically achieved using one of two methods: The first one is to manually copy one input to all nodes (possibly using a utility such as Hadoop's Shared Cache [1]) and performing the join in the *map* function, for example by using a hash table. The second variant is to implement the

join in the *reduce* function, using a union of both inputs where each value has a tag that indicates which input it belongs to. The shuffle and sort phase co-locate and group the input by key, such that the reducer has to take its input, separate it by the tag to obtain the two original inputs, and concatenate the values pair-wise.

An inner join maps naturally to the *Match* PACT. Given that in both input sets the key/value pairs have the corresponding attributes of an equality–join predicate as key, *Match* guarantees that all joinable pairs of key/value pairs from its two inputs are provided exactly once to an instance of the user function, which only needs to implement a concatenation of the values. The *CoGroup* PACT can be used as well, however its contract is stronger than what an inner–join actually requires. While the map/reduce implementation fixes the execution strategy, the PACT optimizer can choose the strategy with the least expected costs for a given PACT program.

Outer–Join The implementation of an outer–join using map/reduce is similar to the *Reduce*-side strategy given for the inner–join. If for a certain key, the *reduce* function receives only values from one input (inputs are identified using the tag), all input tuples are concatenated with NULL values.

An outer–join can be implemented using the *CoGroup* PACT. If for a certain key, the set of values from one input is empty, the other input's values are concatenated with NULLs.

Anti–Join Realizing an anti–join using map/reduce can be done in a similar way as in the outer–join implementation. The values from the input that is to be filtered are discarded as soon as the reduce function has also values for the same key from the other input.

Similar to an outer–join, the *CoGroup* PACT can be used to implement an anti–join. The user function discards the tuples from one input as soon as the other input is not empty.

Theta–Join In map/reduce, arbitrary theta–joins can be realized following the *map*–side join strategy described for the inner–join. This strategy can cause significant communication costs in case of inputs with similar sizes. Another option is to hard-wire a symmetric-fragment-and-replicate strategy. However, this strategy causes data to be redundantly stored on the local hard disks of the mappers and a fixed degree of parallelization.

Arbitrary theta–joins can be mapped to the *Cross* PACT. It guarantees that each element of the Cartesian product is handed once to the user function which can check all kinds of join conditions. In contrast to the map/reduce implementation, the strategy to build the distributed Cartesian product required for arbitrary theta–joins is automatically chosen by an optimizer.

Pairwise Correlation Computation The input is a set of N random variables, each given as M sample points, organized as rows (*VariableID*, [sample-points]). The goal is to compute the $N \times N$ correlation matrix. An application for this problem is the correlation detection for gene expressiveness in cancer cells. Delmerico et al. [6] describe this application and give a detail description of a map/reduce implementation.

Briefly summarized, the task runs a first *map* function, which adds for each variable independently the mean and standard deviation and exchanges the sample points by their deviation from the mean. Then, the complete result is added to Hadoop's Distributed Cache and thereby replicated to all nodes. A second map takes as its input a list of incrementing numbers that correspond to the numbers of the random variables. The map function then accesses the variables in the cache and computes the i'th row of the correlation matrix, where i is the number that the function reads from its input. It can compute the row, because it has full access to all random variables through the cache.

With PACTs, the task is expressed with two functions, implementing *Map* and *Cross*. Similar as in map/reduce, the *Map* adds for each variable the mean and standard deviation and exchanges the sample points by their deviation from the mean. The *Cross* takes these results in both its inputs, computing the correlation coefficient for each encountered pair, outputting $(ID_1, ID_2, correlation)$. In comparison to the map/reduce variant, this is much more natural, since it requires neither any utilities, such as the distributed cache, nor the usage of the list of incrementing numbers as the second mapper's input.

K–Means Clustering A typical implementation of one iteration of the K–Means Clustering algorithm on Hadoop looks the following way: A set of initial cluster centers is added to the distributed cache and hence replicated to all nodes. Input to the map/reduce job is the file that contains the data points. The *map* function processes each point and computes the distance to all centers, selecting the center with the smallest distance. The key of the mapper's output is the ID of the closest center, the value is the data point's position. The reducer then receives the values grouped by cluster center and computes the new cluster center position as the mean positions represented by the values.

The PACTs implementation computes the distance of each data point to each center is computed using the *Cross* PACT. For each data point the cluster which is least distant is found by a function implementing the *Reduce* PACT. Another *Reduce* function computes for each cluster center its new position. Given the new cluster centers the algorithm can be started from the beginning. Significant optimizations can be applied, if the *Unique-Key* Output Contract is assigned to the source of the data points and the *Same-Key* contract is given to the function implementing *Cross*, referring to the input with the data points. That way, the system can generate a data flow where the first *Reduce* type function does not require a shuffling of the data and therefore, comes virtually for free.

6.2 Performance Experiments

We compare the performance of queries, which we implement both as map/reduce and PACT programs. The data flow for the queries implemented with the full set of PACTs is created from the PACT program based on the rules described in Section 5. To make a fair comparison and to restrict the

Query	Time in s (cold cache)	Time in s (hot cache)	Net traffic in MB
Query 1			
PACT BC	613	273	5318
PACT Part	739	385	46652
MR	879	568	51215
Query 2			
PACT BC	751	415	8784
MR	2027	1545	72688

Table 1: Experimental results: time and traffic

comparison to highlight the benefits of the PACT programming model over a pure map/reduce programming model, we run the map/reduce implementations in Nephele as well. The data flows for the map/reduce programs executed by Nephele perform exactly the same operations as the map/reduce implementation *Hadoop* would perform: They repartition the data between the *map* and *reduce* step according to the key and sort each partition by the key, as described in [5]. A *combiner* is used whenever reasonable, i. e. when the *reducer* performs an aggregation.

In an initial comparison, we ran the map/reduce implementations of some of the queries on Hadoop 0.19. The results were however significantly slower than both the map/reduce program on Nephele and the program based on the full set of PACTs, which can be largely attributed to the fact that Hadoop writes heavily to disk and into the distributed file system during the execution. Those operations are very expensive, but necessary for the fault tolerance and load balancing scheme. In contrast to that, Nephele writes comparatively few data due to its pipelining support.

In the following, we give two relational queries representing common analytical operations. The test data is a 200 GB set from the TPC-H benchmark suite [2], stored in the Hadoop Distributed Filesystem as CSV files. We use a block size of 64 MB and distribute the data evenly among 12 virtual machines we allocated from our local compute cloud. Each virtual machine has eight CPU cores and 24 GB of RAM. Thus, 96 dedicated CPU cores are available during our experiments. The guest operation system is Ubuntu Linux (kernel version 2.6.31-17-server). We use Sun's JVM version 1.6.0.15 to run the Nephele prototype, the PACT code, and the programs, which are written in Java. Our cloud system is hosted by commodity servers, each equipped with two Intel Xeon 2.66 GHz CPUs and 32 GB RAM. All servers are connected through regular 1 GBit/s Ethernet links and run Gentoo Linux (kernel version 2.6.30). Virtualization is provided by KVM (version 88-r1).

6.2.1 Query 1: Join/Aggregation Task

The first task with which we evaluate the system is the example task described in the introduction. The task performs a join between the *LINEITEM* and *ORDERS* tables, followed by a grouping on a superkey of the joinkey and an aggregation. Figure 2 shows the parallelization of the map/reduce program, Figure 4 the two possible data flows for the PACT program. In the following, we are going to refer to the flow (a) from Figure 4 as the *broadcasting* variant, and to flow (b) as the *partition only* variant. When executing the map/reduce plan in Nephele, we do not write the results of the first reduce function into the distributed file system, but pipeline them directly into the second map function.

The runtimes as well as the total network data volume is given in Table 1. The broadcasting variant of the PACT program (PACT BC) is roughly twice as fast as the map/reduce job on a hot system, and still takes only 70% of the its execution time on a cold system. The partition only variant (PACT Part) is still 15% faster than map/reduce with cold caches, and one third faster with warm caches. Recall that this is despite the fact that the map/reduce based data flow is exempt from the typical write operations performed after the mapper and reducer.

Figure 7 shows plots for the network and CPU utilization of all three data flows, each time for a run with a cold operating system file cache and for a run with a hot one. The generally low utilization can be attributed to the fact that the virtualized environment provides sizeable compute power, but comparatively poor disk and network bandwidths. As a comparison, the broadcasting variant of the PACT program takes less than 2 minutes on a non-virtualized cluster with 5 nodes (warm caches), where each node holds the same amount of data as the nodes in the virtualized cluster.
The shapes of the network and CPU utilization curves of the runs with cold caches are similar to those of the runs with hot caches, although the utilization is consistently lower and the wait times are higher. That indicates that the limiting factor is how fast the data can be read from disk.

The charts for the *broadcasting* variant of the PACT program show an initial peak in network and CPU utilization. During that time, the *ORDERS* table is read, filtered, broadcasted, and a hash-table is built. Afterwards, the network drops to zero while the *LINEITEM* table is read and joined against the replicated hash-table. The small peaks of network traffic at the end represent the shuffling phase before the aggregation. Since comparatively few rows are shuffled and aggregated, the utilization remains low.
The charts for the *partition only* variant show that network utilization quickly increases and remains high for the entire job. It indicates that the shuffling needs the majority of the query time. The CPU utilization drops immediately with the network utilization. Because the system can hold the sorted input from the *ORDERS* side completely in memory, it does not need to wait until all rows from the *LINEITEM* table are partitioned and sorted before starting the join. Instead, whenever the memory buffer from the *LINEITEM* side is full, it is immediately sorted, joined against the other input and processed by the user code. Recall that no shuffling is required for the aggregation, which is why the plots lack a second small peak in network traffic.
The plot for the map/reduce program is initially similar to the one of the partition only data flow. Because with a *reduce* contract, the system cannot proceed while the data is still shuffled. Instead, we see the map/reduce typical peak in CPU utilization after the shuffle phase, when the reducer merges the sorted sub-streams and runs the user code.

6.2.2 Query 2: Star-Join Task

This task performs a simple star join between a fact table and two dimensions, which is a common operation in OLAP scenarios.

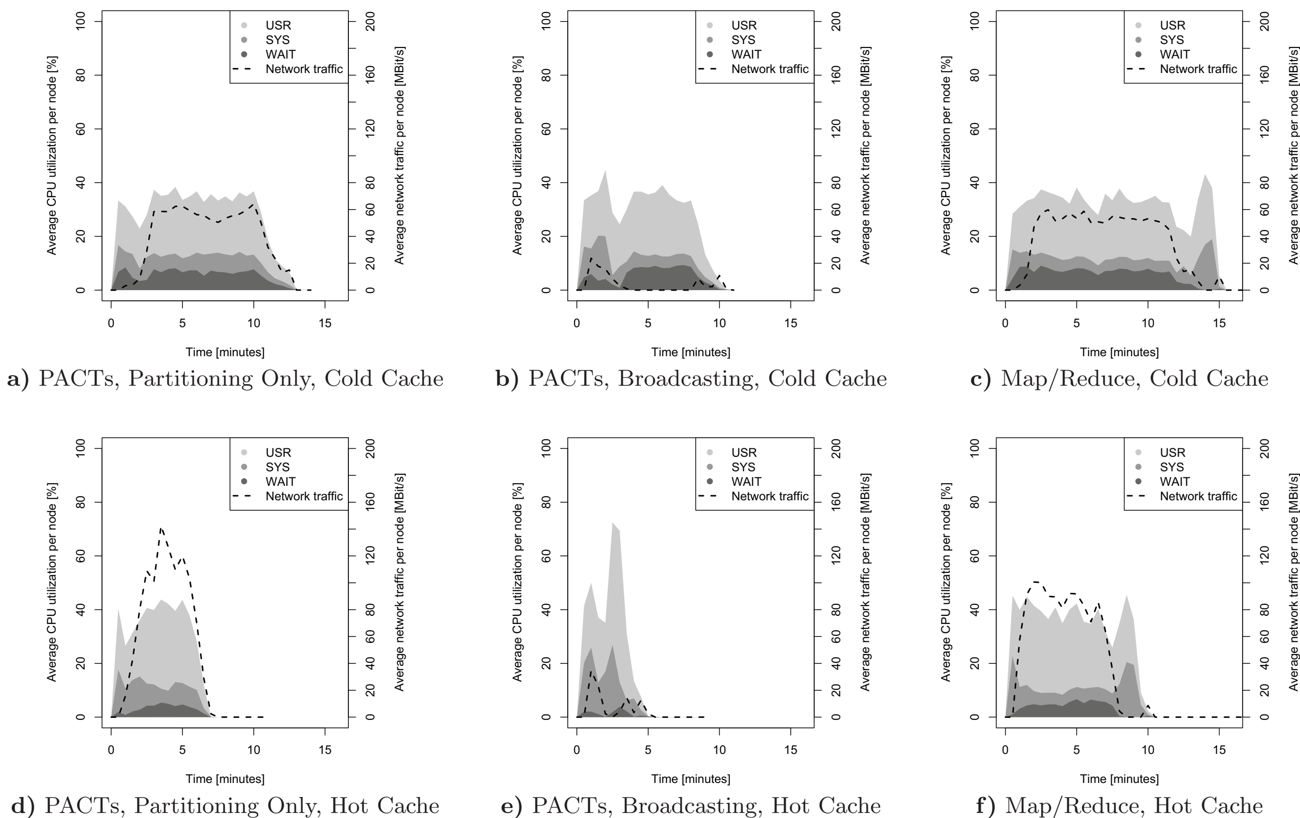

Figure 7: CPU and network utilization in performance experiments of Query 1. The figure shows the results for the PACT program run with both strategies illustrated in Figure 4 and the map/reduce program.

```
SELECT l_suppkey, l_orderkey, l_partkey,
  l_quantity, o_orderpriority,
  s_name, s_address, s_acctbal
FROM lineitem, orders, supplier
WHERE l_orderkey = o_orderkey
  AND l_suppkey = s_suppkey
  AND YEAR(o_orderdate) = 1995
  AND o_orderpriority IN ('1-URGENT','2-HIGH')
  AND s_nationkey = 5
```

We join the *LINEITEM (L)* table with the two dimensions *SUPPLIER (S)* and *ORDER (O)*. Both *S* and *O* apply local predicates with a selectivity of about 5%. From the dimensions, we select 30% and 50% of the non-key columns, from the fact table 30% of the non-key columns. For the PACT program, that information is provided to the compiler via data volume and key cardinality annotations.

In map/reduce the star-join is implemented as two *reduce*-side joins as shown in Figure 8 (a) (cf. [17]). Each *reduce*-side join is effectively a repartition-join, with merge-joining as the local join algorithm. Figure 8 (b) shows the corresponding PACT program. The join is implemented via two functions implementing the *Match* contract. For the given task and data, the compiler chooses for both functions to broadcast the smaller side to each node and to build a hash table from the smaller side, as described in Section 5.1. Effectively, that represents an asymmetric-fragment-and-replicate join (a.k.a. broadcast join), with a hash-join as the local join algorithm. Note that neither the programming of the *Reduce* nor the *Match* PACT involved implementing anything more than concatenating tuples. The sort-merge or hash algorithms are a result of the framework's internal strategies to fulfill the Input Contracts.

Table 1 shows the execution times and accumulated network traffic for both implementations of Query 2. The map/reduce implementation requires significantly more time to compute the result. We identified three main reasons for clear deviance in performance: 1) the amount of data transferred over the network. While the PACT broadcast implementation only broadcasts the filtered and projected

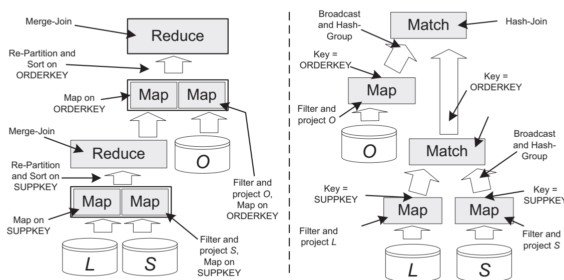

Figure 8: a) map/reduce star–join, b) PACT star–join

dimension tables (8.5 GB), the map/reduce implementation ships the huge fact table as well resulting in about 71 GB of network traffic. 2) due to the hash join the PACT implementation is able to pipeline the tuples of the fact table and hence receiving and joining tuples at the same time. In contrast, the map/reduce implementation must wait for the join until the last fact tuple was received. 3) the merge join requires both joining relations to be sorted which is an expensive operation due to the relations cardinalities.

7. RELATED WORK

Our approach is related to three families of systems: General-purpose distributed execution engines, map/reduce systems, and parallel databases.

General-purpose distributed execution engines execute parallel data flows. Their programs are graphs, where the vertices are blocks of sequential code and the edges are communication channels. Representatives for this category are Dryad [10] or Nephele [18]. Their big advantage is their generality: By putting the right code into the vertices and setting up the channels in a suitable way, these systems can model arbitrary data-parallel programs. It is possible to create data flows that describe map/reduce type processes, or SQL queries. The generality comes with the disadvantage that writing a parallel data flow is tiresome and complicated. The programmer has to hand-craft and -optimize the parallelization strategies. Automatic optimization on such a data flow is only possible to a very limited degree. That holds for both a priori and run-time optimizations. In our system, the general-purpose distributed execution engine Nephele is used to run the final data flow. We add a more abstract programming model (PACTs) and compile the programs to a suitable parallel data flow.

Our PACT programming model is very similar to *map/reduce* [5]. It builds on higher-order functions that can be automatically parallelized. There are however crucial differences between map/reduce and our PACT programming model: First, we add additional functions that fit many problems which are not naturally expressible as a *map* or *reduce* function. Second, in map/reduce systems like Hadoop [1], the programming model and the execution model are tightly coupled – each job is executed with a static plan that follows the steps map/combine/shuffle/sort/reduce. In contrast, our system separates the programming model and the execution and uses a compiler to generate the execution plan from the program. For several of the new PACTs multiple parallelization strategies are available. Third, map/reduce loses all semantic information from the application, except the information that a function is either a *map* or a *reduce*. Our PACT model preserves more semantic information through both a larger set of functions and through the annotations.

Parallel database system architectures [7, 9] are built around data parallelism as well. Their query compilers generate parallel query execution plans from an SQL query, utilizing rules that describe how the relational operations can be parallelized. In contrast to them, our approach is based on Parallelization Contracts and allows for automatic parallelization of programs, independent of their specific semantics and the data model, if the data model can be mapped to a key/value model. Underneath the PACT abstraction layer, we employ parallelization techniques known from parallel database systems (e. g. [14], [16]). In that sense, one can for example see the *Match* PACT as a de-schematized join.

SCOPE [4] and DryadLINQ [21] are similar to our approach, as they compile declarative or more abstract programs to a parallel data flow for Dryad. The first one generates the parallel data flow from an SQL query, the second one from a .NET program. Both restrict the programmer to their specific data model and set of operations (relational, resp. LINQ). They optimize the parallelization in a domain specific way during the compilation. Our system differentiates itself from them by being more generic. It can parallelize and optimize the execution independently of a particular domain by the virtue of the PACT abstraction.

An extension to map/reduce with an additional function called *merge* has been suggested in [20]. *Merge* is similar to our *CoGroup* PACT, but closely tied to the execution model. The programmer must define explicitly which partitions are processed at which instances. The work focus only on joins and how to realize different local join algorithms (Hash-Join/Merge-Join) rather than different parallel join strategies (re-partitioning, (a)symmetric-fragment-and-replicate).

HadoopDB is an extension of Hive and queried through its SQL like query language. Instead of storing the data in Hadoop's distributed file system, it governs multiple Postgres instances that store the data. It loses the generality of Hadoop, which supports arbitrary data models, but improves significantly on Hadoop's storage layer by allowing to push parts of a query closer to the data. It is orthogonal to our work, as our system still operates on a distributed file system, but provides a more efficient execution engine.

Related techniques for the optimization of parallelism beyond parallel databases are found in the languages that are put on top of map/reduce. Pig [12, 11], JAQL [3], and Hive [17] generate map/reduce jobs for Hadoop from declarative queries. Some perform domain specific optimization (like join order selection) and all of them apply simple heuristics to improve performance, such as chaining of *Map* operations. Both aspects are orthogonal to our optimization of the parallelization. Hive optimizes join parallelism by choosing between (still inefficient) *Map*-side joins and *Reduce*-side joins. These domain specific optimization techniques are tailored specifically towards equi-joins, similar to how parallel databases perform them. The alternative execution strategies for our *Match* contract encapsulate that optimization in a more generic fashion.

8. CONCLUSIONS

In this paper, we have presented a generic system for web-scale data processing. The programming abstraction for writing tasks are *Parallelization Contracts* (PACTs), consisting of Input and Output Contracts. The abstraction provided by Input Contracts is well-known from map/reduce. We have generalized this concept such that it abstracts communication patterns behind second-order functions. Tasks are executed in a flexible engine that is able to execute arbitrary acyclic data flows. We believe that the combination of those two technologies has a lot of potential for web-scale analytical processing. The programming model has the right level of abstraction for writing data processing tasks. It is generic enough to be widely applicable. Input and Output Contracts capture a reasonable amount of semantics, which are used to compile the program to an efficient data flow. The system is easily extensible, since new contracts can be added by simply setting up rules to map them to a data flow.

8.1 Open Issues and Future Work

The presented set of PACTs is useful to express relational operations or basic data mining operations. We are currently investigating additional ones for more complex data mining tasks and time series analysis. Those new PACTs will abstract communication patterns that do not partition the data into disjoint sets. Candidate patterns are a partitioning based on fuzzy key equalities (using a metric distance threshold), or a range partitioning with defined overlap. The latter one is useful for example in Sliding-Window-Aggregations. We will go on validating the PACTs against several other use cases, such as data mining, scientific analysis, or information extraction/retrieval. Furthermore, the programming model's abstraction is perfect for a system that generically adapts the degree of parallelism to the characteristics of an operator (such as CPU and I/O intensity) and distributes operations across cores or nodes accordingly.

At the side of Nephele, an open question is how to best realize fault tolerance. Between checkpointing and materializing every step, or checkpointing nothing, the optimal strategy is dependent on the task, its operations, the data, and the environment. Additionally, since Nephele may allocate its machines from a Infrastructure-as-a-Service provider, static network topology descriptions are not applicable. Hence, the incorporation of techniques for automatic network topology discovery would be very valuable.

Acknowledgments

We thank HP for supporting this work through an Open Collaboration Grant as well as IBM for a Shared University Research hardware grant that provided the hardware necessary for the evaluation of our work. We also thank Guy Lohman for suggesting the name "PACT" for the contracts, as well as the anonymous reviewers for their constructive comments and suggestions.

9. REFERENCES

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

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

[3] K. Beyer, V. Ercegovac, J. Rao, and E. Shekita. Jaql: A JSON Query Language. URL: http://jaql.org.

[4] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *PVLDB*, 1(2):1265–1276, 2008.

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

[6] J. Delmerico, N. Byrnes, A. Bruno, M. Jones, S. Gallo, and V. Chaudhary. Comparing the Performance of Clusters, Hadoop, and Active Disks on Microarray Correlation Computations. In *International Conference on High Performance Computing*, 2009.

[7] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. GAMMA - A High Performance Dataflow Database Machine. In W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, editors, *VLDB*, pages 228–237. Morgan Kaufmann, 1986.

[8] E. Friedman, P. M. Pawlowski, and J. Cieslewicz. SQL/MapReduce: A Practical Approach to Self-describing, Polymorphic, and Parallelizable User-defined Functions. *PVLDB*, 2(2):1402–1413, 2009.

[9] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An Overview of The System Software of A Parallel Relational Database Machine GRACE. In W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, editors, *VLDB*, pages 209–219. Morgan Kaufmann, 1986.

[10] 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.

[11] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic Optimization of Parallel Dataflow Programs. In R. Isaacs and Y. Zhou, editors, *USENIX Annual Technical Conference*, pages 267–273. USENIX Association, 2008.

[12] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In J. T.-L. Wang, editor, *SIGMOD Conference*, pages 1099–1110. ACM, 2008.

[13] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In U. Çetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, editors, *SIGMOD Conference*, pages 165–178. ACM, 2009.

[14] D. A. Schneider and D. J. DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. In J. Clifford, B. G. Lindsay, and D. Maier, editors, *SIGMOD Conference*, pages 110–121. ACM Press, 1989.

[15] 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.

[16] J. W. Stamos and H. C. Young. A Symmetric Fragment and Replicate Algorithm for Distributed Joins. *IEEE Trans. Parallel Distrib. Syst.*, 4(12):1345–1354, 1993.

[17] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[18] 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.

[19] C. Yang, C. Yen, C. Tan, and S. Madden. Osprey: Implementing MapReduce-Style Fault Tolerance in a Shared-Nothing Distributed Database. In *ICDE*, 2009.

[20] H. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In C. Y. Chan, B. C. Ooi, and A. Zhou, editors, *SIGMOD Conference*, pages 1029–1040. ACM, 2007.

[21] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In R. Draves and R. van Renesse, editors, *OSDI*, pages 1–14. USENIX Association, 2008.