MapReduce and PACT - Comparing Data Parallel Programming Models

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

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

Abstract: Web-Scale Analytical Processing is a much investigated topic in current research. Next to parallel databases, new flavors of parallel data processors have recently emerged. One of the most discussed approaches is MapReduce. MapReduce is highlighted by its programming model: All programs expressed as the second-order functions *map* and *reduce* can be automatically parallelized. Although MapReduce provides a valuable abstraction for parallel programming, it clearly has some deficiencies. These become obvious when considering the tricks one has to play to express more complex tasks in MapReduce, such as operations with multiple inputs.

The Nephele/PACT system uses a programming model that pushes the idea of MapReduce further. It is centered around so called *Parallelization Contracts (PACTs)*, which are in many cases better suited to express complex operations than plain MapReduce. By the virtue of that programming model, the system can also apply a series of optimizations on the data flows before they are executed by the Nephele runtime system.

This paper compares the PACT programming model with MapReduce from the perspective of the programmer, who specifies analytical data processing tasks. We discuss the implementations of several typical analytical operations both with MapReduce and with PACTs, highlighting the key differences in using the two programming models.

1 Introduction

Today's large-scale analytical scenarios face Terabytes or even Petabytes of data. Because many of the early large-scale scenarios stem from the context of the Web, the term *Web-Scale Analytical Processing* has been coined for tasks that transform or analyze such vast amounts of data. In order to handle data sets at this scale, processing tasks run in parallel on large clusters of computers, using their aggregate computational power, main memory, and I/O bandwidth. However, since developing such parallel programs bottom-up is a cumbersome and error-prone task, new programming paradigms which support the automatic parallelization of processing jobs have gained a lot of attention in recent years.

The MapReduce paradigm [DG04] is probably the best-known approach to simplify the development and parallel execution of data processing jobs. Its open-source implementation

Hadoop [Had] is very popular and forms the basis of many parallel algorithms that have been published in the last years ([VCL10, Coh09, Mah]). Compared to parallel relational databases, which have been been the predominant solution to parallel data processing, MapReduce proposes a more generic data and execution model. Based on a generic key/value data model, MapReduce allows programmers to define arbitrarily complex user functions which then are wrapped by second-order functions *map* or *reduce*. Both of these second-order functions provide guarantees on how the input data is passed to the parallel instances of the user–defined function at runtime. That way, programmers can rely on the semantics of the second-order functions and are not concerned with the concrete parallelization strategies.

However, even though the two functions *Map* and *Reduce* have proven to be highly expressive, their originally motivating use-cases have been tasks like log-file analysis or web-graph inverting [DG04]. For many more complex operations, as they occur for example in relational queries, data-mining, or graph algorithms, the functions *Map* and *Reduce* are a rather poor match [PPR+09]. A typical example is an operation that matches key/value pairs with equal keys from two different inputs. Such an operation is crucial for many tasks, such as relational joins and several graph algorithms [YDHP07]. To express it in MapReduce, a typical approach is to form a union of the inputs, tagging the values such that their originating input is known. The Reduce function separates the values by tag again to re-obtain the different input sets. That is not only an unintuitive procedure, programmer must also make explicit assumptions about the runtime parallelization while writing the user code. In fact, many implementations exploit the fact that MapReduce systems, like Hadoop, implement a static pipeline of the form *split-map-shuffle-sort-reduce*. The shuffle and sort basically exercise a parallel grouping to organize the data according to the Reduce function's requirements. Because all operations are customizable, many tasks are executed with appropriate custom split or shuffle operations. That way, MapReduce systems become a parallel process runtime that execute hard-coded parallel programs [DQRJ+10].
This clearly conflicts with the MapReduce's initial design goals. While highly customizing the behavior allows to run more complex programs, it destroys the idea of a declarative specification of parallelism, preventing any form of optimization by the system. Especially the incorporation of runtime adaptation to the parallelization methods (such as broadcasting or partitioning) requires a clear specification of the user function's requirements for parallelization, rather than a hard-coding of the method.

To overcome those shortcomings, we have devised a programming model that offers the same abstraction level as MapReduce but pushes its concepts further. It is centered around so called *Parallelization Contracts* (PACTs), which can be considered a generalization of MapReduce [BEH+10]. The PACT programming model eases the expression of many operations and makes it more intuitive. Moreover, its extended expressiveness also enables several optimizations to be applied automatically, resulting in more efficient processing.

In this paper, we give a short introduction to both programming models and compare them from the perspective of the developer writing the data analysis tasks. We present tasks from the domains of relational queries, XQuery, data mining, and graph algorithms and present their implementations using MapReduce and PACT. We discuss the differences of the two programming models and the implications for the programmer. The remainder of the paper

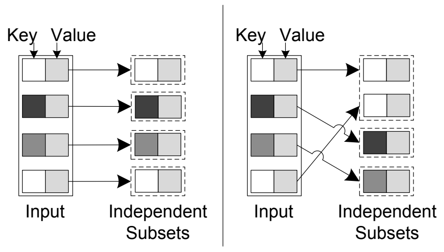


Figure 1: a) Map | b) Reduce

is structured as follows: Section 2 discusses the MapReduce and the PACT programming model in detail, highlighting key differences. Section 3 describes five typical analytical tasks and compares their MapReduce and PACT implementations. Section 4 discusses related work, Section 5 concludes the paper.

2 Parallel Programming Models

This section contrasts the parallel programming models MapReduce and PACT. Since the PACT programming model is a generalization of MapReduce, we start with a short recapitulation of MapReduce before introducing the extensions in the PACT programming model. A more thorough description of the PACT programming model can be found in [BEH$^+$10].

2.1 The MapReduce Programming Model

The MapReduce programming model was introduced in 2004 [DG04]. Since then, it has become very popular for large-scale batch data processing. MapReduce founds on the concept of data parallelism. Its data model is key/value pairs, where both keys and values can be arbitrarily complex. A total order over the keys must be defined.

The key idea of MapReduce originates from functional programming and is centered around two second-order functions, *Map* and *Reduce*. Both functions have two input parameters, a set of key/value pairs (input data set) and a user-defined first-order function (user function). *Map* and *Reduce* apply the user function on subsets of their input data set. Thereby, all subsets are independently processed by the user–defined function.
Map and *Reduce* differ in how they generate those subsets from their input data set and pass them to the attached user function:

- *Map* assigns each individual key/value pair of its input data set to an own subset. Therefore, all pairs are independently processed by the user function.
- *Reduce* groups the key/value pairs of its input set by their keys. Each group becomes an individual subset which is then processed once by the user-defined function.

Figure 1 depicts how *Map* and *Reduce* build independently processable subsets. Each subset is processed once by exactly one instance of the user-defined function. The actual functionality of the *Map* and *Reduce* operations is largely determined by their associated user functions. The user function has access to the provided subset of the input data and can be arbitrarily complex. User functions produce none, one, or multiple key/value pairs. The type of the produced keys and values may be different from those of the input pairs. The output data set of *Map* and *Reduce* is the union (without eliminating duplicates) of the results of all evaluations of the user function.

A MapReduce program basically consists of two stages which are always executed in a fixed order: In the first stage the input data is fed into the *Map* function that hands each key/value pair independently to its associated user function. The output of the *Map* function is repartitioned and then sorted by the keys, such that each group of key/value pairs with identical keys can be passed on to the *Reduce* function in the second stage. The user function attached to the *Reduce* can then access and process each group separately. The output of the *Reduce* function is the final output of the MapReduce program. As complex data processing tasks do often not fit into a single MapReduce program, many tasks are implemented using a series of consecutive MapReduce programs.

Since all invocations of the user functions are independent from each other, processing jobs written as MapReduce programs can be executed in a massively parallel fashion. In theory *Map* can be parallelized up the the number of input key/value pairs. *Reduce*'s maximum degree of parallelism depends on the number of distinct keys emitted in the map stage.

A prominent representative of a MapReduce execution engine is Apache's Hadoop [Had]. This paper focuses on the abstraction to write parallelizable programs. Therefore, we do not discuss the execution of MapReduce programs. Details can be found in [DG04].

2.2 The PACT Programming Model

The PACT programming model [BEH$^+$10, ABE$^+$10] is a generalization of MapReduce [DG04] and is also based on a key/value data model. The key concept of the PACT programming model are so-called *Parallelization Contracts (PACTs)*. A PACT consists of exactly one second-order function which is called *Input Contract* and an optional *Output Contract*. Figure 2 visualizes the aspects of a PACT. An Input Contracts 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

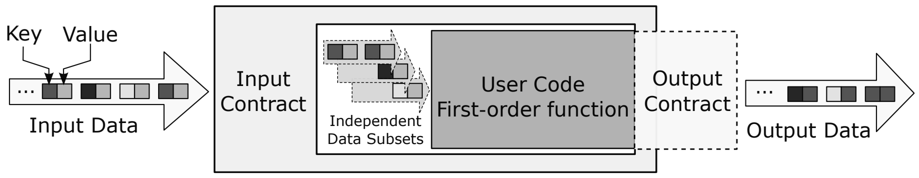


Figure 2: Parallelization Contract (PACT)

input data in a data-parallel fashion. In the context of the PACT programming model, MapReduce's *Map* and *Reduce* functions are Input Contracts. The PACT programming model provides additional Input Contracts that complement *Map* and *Reduce*, of which we will use the following three in this paper:

- *Cross* operates on multiple inputs of key/value pairs and builds a Cartesian product over its input sets. Each element of the Cartesian product becomes an independent subset.
- *CoGroup* groups each of its multiple inputs along the key. Independent subsets are built by combining the groups with equal keys of all inputs. Hence, the key/value pairs of all inputs with the same key are assigned to the same subset.
- *Match* operates on multiple inputs. It matches key/value pairs from its input data sets with the same key. Each possible two key/value pairs with equal key form an independent subset. Hence, two pairs of key/value pairs with the same key are processed independently by possibly different instances of the user function, while the CoGroup contract assigns them to the same subset and guarantees to process them together.

Figure 3 illustrates how *Cross*, *Match*, and *CoGroup* build independently processable subsets.

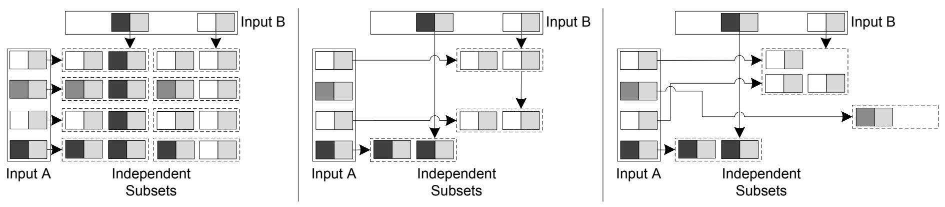


Figure 3: a) Cross | b) Match | c) CoGroup

In contrast to Input Contracts, which are a mandatory component of each PACT, Output Contracts are optional and have no semantic impact on the result. Output contracts give hints about the behavior of the user code. To be more specific, they assert certain properties of a PACT's output data. An example of such an output contract is the *SameKey* contract. When attached to a Map function, it states that the user code will not alter the key, i.e. the type and value of the key remain after the user code's invocation and are the same in the output as in the input. Those hints can be exploited by an optimizer that generates parallel execution plans. The aforementioned SameKey contract can frequently help to avoid unnecessary repartitioning and therefore expensive data shipping. Hence, Output Contracts can significantly improve the runtime of a PACT program. Currently, developers must manually annotate user functions with Output Contracts. However, automatic derivation based on static code analysis, or suggestions inferred from runtime observations, are worth exploring.

In contrast to MapReduce, multiple PACTs can be arbitrarily combined to form more complex data processing programs. Since some PACTs naturally expect multiple input data sets, the resulting data processing program is not necessarily a strict pipeline like in the MapReduce case but can yield arbitrarily complex data flow graphs. In order to implement the program, each PACT in the data flow graph must be provided with custom code (the user function). Furthermore, at least one data source and data sink must be specified.

2.3 Comparing MapReduce and PACT

For many complex analytical tasks, the mapping to MapReduce programs is not straightforward and requires working around several shortcomings of the programming model. Such shortcomings are for example the limitation to only one input, the restrictive set of two primitive functions *Map* and *Reduce*, and their strict order. The workarounds include the usage of auxiliary structures, such as the distributed cache, and custom partitioning functions. However, the necessity to apply such tricks destroys the desired property of transparent parallelization.

The PACT programming model has been explicitly designed to overcome the problems with more complicated analytical tasks. It is based on the concept of Input Contracts, which are a generalizations of the Map and Reduce functions. Compared to MapReduce, it offers several additions: First, it offers a richer set of parallelization primitives (which also include Map and Reduce). The generic definition of Input Contracts allows to extend this set with more special contracts. Second, with the concept of Output Contracts it is possible to declare certain properties of the user functions to improve efficiency of the task's execution. Thirdly, PACTs can be freely assembled to data flows, in contrast to MapReduce's fixed order of *Map* and *Reduce*. Hence, PACT avoids identity mapper or reducer which are frequently required within MapReduce implementation. All those features of the PACT programming model significantly ease the implementation of many complex data processing tasks, compared to the MapReduce approach.

MapReduce programs are always executed with a fixed strategy. The input data is read from a distributed filesystem and fed to the Map function. The framework repartitions and sorts the output of the Map function by the key, groups equal keys together and calls the Reduce function on each group. Due to the declarative character of Input Contracts, PACT programs can have multiple physical execution plans with varying performance. For example, the definition of the Match contract is such that a parallel execution of the attached user function can choose among the following strategies: 1) repartition and sort both inputs by the key (in the same way as MapReduce), or 2) broadcast one of the inputs to all instances and not transferring data from the other input between parallel instances. The choice of the execution strategy is made by the PACT compiler, which translates PACT programs to parallel schedules for the Nephele runtime. The compiler uses an optimizer in a similar fashion as a relational database and selects the strategies that minimize the data shipping for the program. We refer to [BEH$^+$10] for details.

3 Comparing Common Task Implementation

In this section we present a selection of common data analysis tasks. The tasks are taken from the domains of relational OLAP queries, XQuery, data mining, and graph analysis algorithms. For each task we give a detailed description and implementations for both programming models, MapReduce and PACTs, pointing out key differences and their implications. All of the presented PACT implementations were designed manually. While Nephele/PACT takes away much work from the programmer, the actual process of defining a given problem as a PACT program still has to be done manually. The automatic generation of PACT programs from declarative languages like SQL or XQuery is a field of research.

3.1 Relational OLAP Query

Task Even though relational data is traditionally analyzed through parallel relational database management systems, there is increasing interest to process it with MapReduce and related technologies [TSJ$^+$09]. Fast growing data sets of semi-structured data (e.g. click-logs or web crawls) are often stored directly on file systems rather than inside a database. Many enterprises have found an advantage in a central cheap storage that is accessed by all of their applications alike. For the subsequent analytical processing, declarative SQL queries can be formulated as equivalent MapReduce or PACT programs. Consider the SQL query below. The underlying relational schema was proposed in [PPR$^+$09] and has three tables: Web-page documents, page rankings, and page-visit records. The query selects the documents from the relation Documents d containing a set of keywords and joins them with Rankings r, keeping only documents where the rank is above a certain threshold. Finally, the anti-join (the *not exists* subquery) reduces the result set to the documents not visited at the current date.

```
1:  SELECT *
2:    FROM Documents d JOIN Rankings r
3:           ON r.url = d.url
4:   WHERE CONTAINS(d.text, [keywords])
5:     AND r.rank > [rank]
6:     AND NOT EXISTS
7:        (SELECT * FROM Visits v
8:         WHERE v.url = d.url AND v.visitDate = CURDATE());
```

MapReduce To express the query in MapReduce, we intuitively need two successive jobs, which are shown on the left-hand side of Figure 4: The first MapReduce job performs an inner-join (lines 1-5 of the sample query), the second one an anti-join (lines 6-8). The first Map task processes the input relations Documents d, Rankings r and carries out the specific selection (line 4-5) based on the source relation of the tuple. To associate a tuple with its source relation, the resulting value is augmented with a lineage tag. The subsequent reducer

collects all tuples with equal key, forms two sets of tuples based on the lineage tag and forwards all tuples from r only if a tuple from d is present. The second mapper acts as an identity function on the joined intermediate result j and as a selection on the relation Visits v. Finally, the second reducer realizes the anti-join by only emitting tuples (augmented with a lineage tag 'j') when no Visits tuple (augmented with a lineage tag 'v') with an equal key is found.
The implementation can be refined to a single MapReduce job. Since the key is the same for the join and anti-join, both operations can be done together in a single reduce operation. As before, the mapper forms a tagged union of all three inputs and the reducer separates the values for each key, based on their lineage. The reduce function concatenates the values from the d and r, if no value from v is found. The advantage of this implementation is that all inputs are repartitioned (and hence transferred over the network) only once.

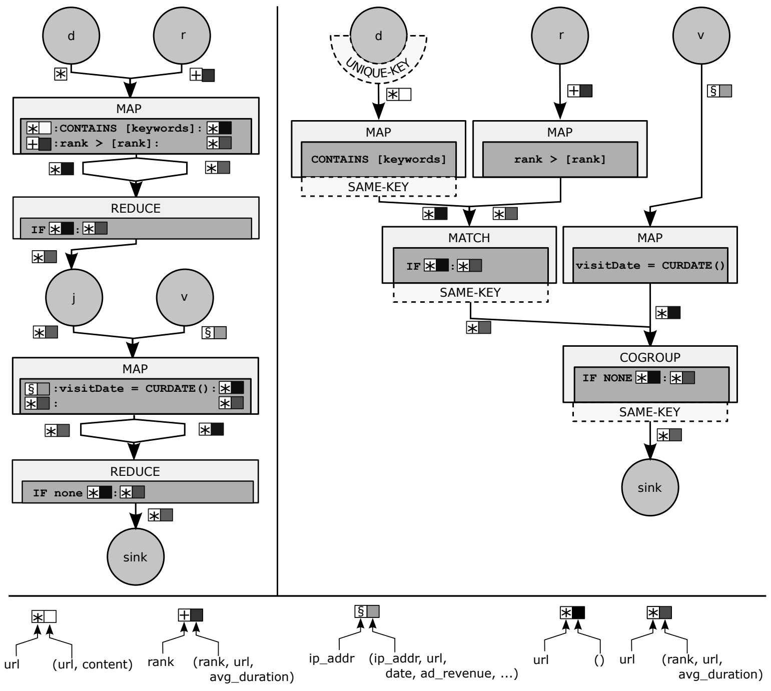


Figure 4: OLAP query as MapReduce (left) and PACT program (right).

PACT The corresponding PACT program for the query is illustrated on the right-hand side of Figure 4. The selections on the relations d, r, v are implemented by three separate user functions attached to the Map contract. The first Reduce task of the original MapReduce implementation is replaced by a Match task. Since the Match contract already guarantees to associate all key/value pairs with the same key from the different inputs pairwise together, the user function only needs to concatenate them to realize the join. The SameKey output

contract is attached to the Match task, because the URL, which has been the key in the input, is still the key in the output. Ultimately, the CoGroup task realizes the anti-join by emitting the tuples coming from the Match task, if for the current invocation of the CoGroup's user function the input of selected tuples from relation v is empty.
With the *SameKey* output contract attached to the Match, the compiler infers that any partitioning on the Match's input exists also in its output[1]. If the Match is parallelized by partitioning the inputs on the key, the system reuses that partitioning for the CoGroup and only partitions the $visits$ input there. That yields the same efficient execution as the hand-tuned MapReduce variant that uses only a single job. However, the PACT implementation retains the flexibility to parallelize the Match differently in case of varying input sizes. In future versions, even dynamic adaption at runtime based on observed behavior of the user-code might be possible.

3.2 XQuery

Task XML is a popular semi-structured data model. It has been adopted by many web-related standards (e.g. XHTML, SVG, SOAP) and is widely used to exchange data between integrated business applications. Because XML is self-describing, it is also a frequent choice for storing data in a format that is robust to application changes. XML data is typically analyzed by evaluating XQuery statements. As the volume of the data stored in XML increases, so does the need for parallel XQuery processors.

The following example shows an XML file describing employees and departments. The XQuery on the right-hand side finds departments with above-average employee count and computes their average salaries.

```
<company>
 <departments>
  <dep id="D1" name="HR"
           active="true"/>
  [...]
 </departments>
 <employees>
  <emp id="E1" name="Kathy"
   salary="2000" dep="D1"/>
  [...]
 </employees>
</company>
```

```
let $avgcnt := avg(
for $d in //departments/dep
let $e := //employees/emp[@dep=$d//@id]
return count($e) )
for $d in //departments/dep
let $e :=//employees/emp[@dep=$d//@id]
where count($e)>$avgcnt
  and data($d//@active)="true"
return
<department>{
 <name>{data($d//@name)}</name>,
 <avgSal>{avg($e//@salary)}</avgSal>
}</department>
```

a) XML data excerpt | b) XQuery example

[1]Omitting Match's output contract prevents reusing its partitioning. If the Match is parallelized through the partitioning strategy, the execution is similar to that of the original MapReduce implementation with two jobs.

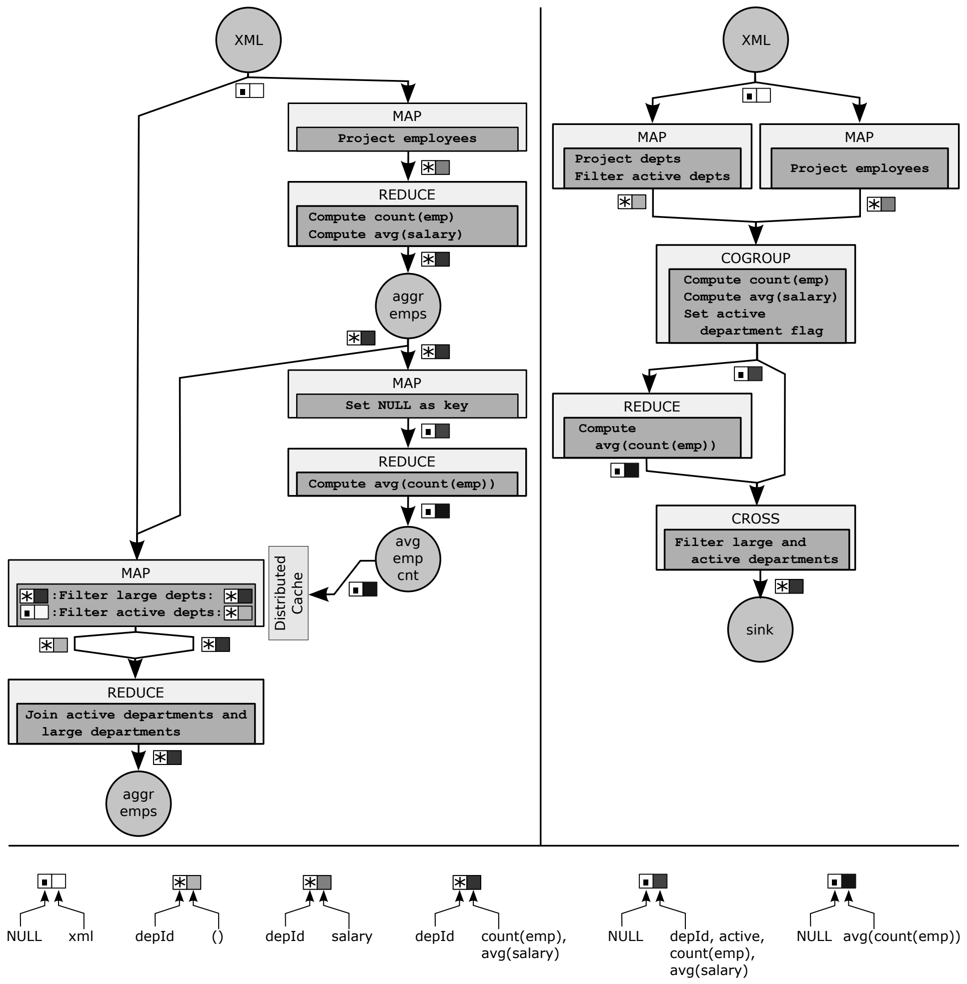


Figure 5: XQuery implementation for MapReduce (left) and PACT (right).

MapReduce Implementing the above XQuery requires in total three MapReduce jobs, as depicted on the left-hand side of Figure 5. The individual jobs are described in the following:

- The first MapReduce job (top) projects employee tuples from the XML file and aggregates the number of employees and their average salary per department. While the mapper performs the projection, the reducer consumes the employee tuples per department and takes care of the aggregations.
- The second job (middle) computes the average number of employees per department. The mapper consumes the result of the first job. Since a global aggregate needs to be computed, a NULL-key is assigned. The reducer performs the global aggregation. The output is a single number that is written to the distributed filesystem.

- The last job combines the departments with the results of the first two jobs. As an initialization step, the result from the second job is added to an auxiliary structure called *Distributed Cache*. Values in the distributed cache are copied to each node so that they are locally accessible to each instance of the mapper. In the actual job, the mapper takes both the original XML file and the result from the first job as input. For records from the original XML file, it selects the active departments. For records from the result of the first job, it compares the employee count against the value from the distributed cache and selects only those with a larger count. It uses the same method of augmenting the key/value pairs with a lineage tag, as described in Section 3.1. The reducer then performs the join in the same fashion as the first reducer for the query given in Section 3.1.

PACT For the PACT implementation of the XQuery, we can make use of the more flexible set of operators (right-hand side of Figure 5). We first use two Map contracts to project employee and active department tuples from the XML file. The CoGroup contract groups employees by their department and compute the number of employees as well as the average salary in the department. Furthermore, an activity flag is set to true if the department is active. Subsequently a Reduce contract is used to compute the average number of employees across all departments. The resulting average number is joined with all tuples via a final Cross contract that filters for those departments which are active and have more than average employees.

3.3 K-Means Clustering Iteration

Task K-Means is a widely used data mining procedure to partition data into k groups, or clusters, of similar data points. The algorithm works by iteratively adjusting a set of k random initial cluster centers. In each iteration data points are assigned to their nearest[2] cluster center. The cluster centers are then recomputed as the centroid of all assigned data points. This procedure is continued until a convergence property is met.

The following pseudo code sketches the K-Means algorithm:

```
initialize k random initial centers
WHILE NOT converged
  FOR EACH point
    assign point to most similar center
  FOR EACH center
    center = centroid of assigned points
END
```

For the parallel implementations we will look at how to implement one iteration of K-Means. This single iteration can then be started as many times as needed from a control program.

[2]According to a specified distance measure, for example the Euclidean distance.

One iteration includes the following two steps:

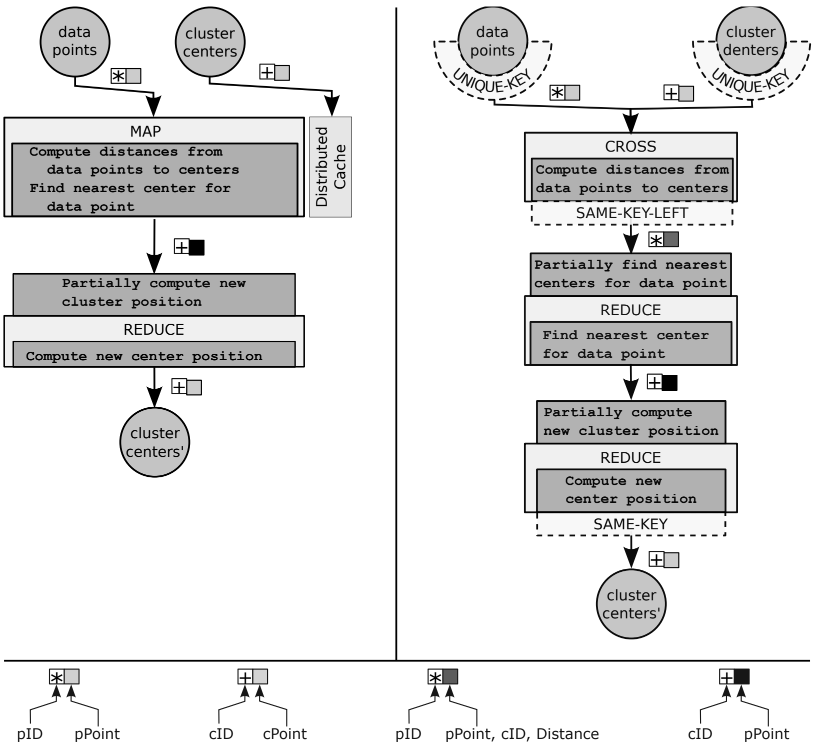
1. Assigning each data point to its nearest center.
2. Recomputing the centers from the assigned data points.

We will assume that data is stored in two files within a distributed file system - the first containing the data points in the form of point-id/location pairs (pID/pPoint), the second one containing the cluster centers in the form of cluster-id/location pairs (cID/cPoint). For the first iteration, the cluster centers will be initialized randomly. For any subsequent one, the result file written by the previous iteration will be used. The actual point data is not of interest for us and can be of arbitrary format, as long as a distance measure and a way of computing the centroid is specified.

MapReduce The MapReduce implementation of a K-Means iteration is illustrated on the left-hand side of Figure 6. As a preparation, we need to give the mapper access to all cluster centers in order to compute the distance between the data points and the current cluster centers. Similar as in the XQuery implementation (c. f. Section 3.2), we use the distributed cache as an auxiliary structure. By adding the file containing the cluster centers to the distributed cache, the centers are effectively broadcasted to all nodes. That way the programmer "hardcodes" the join strategy between cluster centers and data points into the MapReduce program.

The Map function is invoked for each data point and computes the pairwise distance to each cluster center. As a result, it emits for each point a (cID/pPoint) pair where cID is the ID of nearest cluster center and pPoint is the location of the point. The Reduce function processes all pairs of the same cluster center and computes its new location as the centroid from all assigned points. If the distance metric permits it, a Combine function can be introduced to pre-aggregate for each center the centroid on the local machine. Since only the pre-aggregated data is then transferred between nodes, the total volume of data transfer can be significantly reduced.

PACT The PACT implementation of K-Means is depicted on the right-hand side of Figure 6. Instead of having to rely on a distributed cache, the PACTs allows us to directly express the "join" between clusters and points via the Cross contract. The user function attached to the Cross contract computes the pairwise distances between data points and clusters, emitting (pID/pPoint, CID, distance) tuples. The PACT compiler can choose between broadcasting one of the inputs or using a symmetric-fragment-and-replicate strategy to build the distributed Cartesian product of centers and data points as required for the Cross contract. Here, the common choice will be broadcasting the cluster centers, since they form a very small data set compared to the set of data points.
After the Cross function, the program uses a Reduce contract to find the minimum cluster distance for each data point and emit a (cID/pPoint) tuple for the nearest cluster. The final step is similar to the MapReduce implementation: A Reduce contract computes the new center locations, with an optional Combine function pre-aggregating the cluster centroids before the partitioning.



data points
cluster centers
MAP
Compute distances from data points to centers
Find nearest center for data point
Distributed Cache
Partially compute new cluster position
REDUCE
Compute new center position
cluster centers'
data points
UNIQUE-KEY
cluster denters
UNIQUE-KEY
CROSS
Compute distances from data points to centers
SAME-KEY-LEFT
Partially find nearest centers for data point
REDUCE
Find nearest center for data point
Partially compute new cluster position
REDUCE
Compute new center position
SAME-KEY
cluster centers'
pID
pPoint
cID
cPoint
pID
pPoint, cID, Distance
cID
pPoint

Figure 6: K-Means Iteration for MapReduce (left) and PACT (right)

The correct use of the output contracts speeds up this program significantly. Note that in Figure 6, the "data points" source is annotated with a *UniqueKey* output contract, declaring that each key/value pair produced by this source has a unique key. An implicit disjoint partitioning exists across globally unique keys. The function implementing the Cross contract declares via its output contract that it preserves the key of its left input - the data points. If the center points are broadcasted[3] and the data points remain on their original node, the partitioning still exists after the cross function. Even more, all tuples for the same data point occur contiguously in the result of the Cross function. The Reduce contract needs hence neither partition nor sort the data - it is already in the correct format.

This example illustrates nicely the concept of declaring the requirements for parallelization, rather than explicitly specifying how to do it. In MapReduce, the usage of a Reduce function always implies a partitioning and sort step. For PACT, the Reduce contract merely describes that the key/value pairs need to be processed group-wise by distinct key. The compiler can infer for this program that the data is already in the required form. It directly passes the data from the output of the Cross function to the Reduce function without any intermediate processing by the system.

[3]This is the common strategy chosen by the optimizer

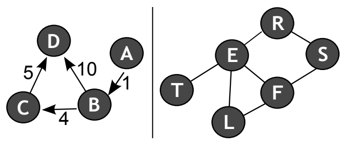


Figure 7: Example graphs for Pairwise Shortest Paths (left) and Triangle Enumeration (right)

3.4 Pairwise Shortest Paths

Task In graph theory, the pairwise shortest path problem is the determination of paths between every pair of vertices such that the sum of the lengths of its constituent edges is minimized. Consider a directed graph $G = (V, E, s, t, l)$ where V and E are the sets of vertices and edges, respectively. Associated with each edge $e \in E$ is a source vertex $v_s = s(e)$, a target vertex $v_t = t(e)$ and a length $l_e = l(e)$. The Floyd-Warshall algorithm (also commonly known as Floyd's algorithm) is an efficient algorithm for simultaneously finding the pairwise shortest paths between vertices in such a directed and weighted graph.

```
for k = 1:n
  for i = 1:n
    for j = 1:n
      D(i,j)=min(D(i,j),D(i,k)+D(k,j));
```

The algorithm takes the adjacency matrix D of G and compares all possible paths through the graph between each pair of vertices. It incrementally improves the candidates for the shortest path between two vertices, until the optimal is found. A parallel variant can be achieved by iteratively performing the following 4 steps until a termination criterion (number of iterations or path updates) is satisfied:

1. Generate two sets of key/value pairs:
   $P_S = \{(p.source, p) | p \in I_k\}$, $P_T = \{(p.target, p) | p \in I_k)\}$
2. Perform an equi-join on the two sets of pairs:
   $P_J = P_T \bowtie_{end=start} P_S$
3. Union the joined set of paths with the intermediate result of the previous iteration:
   $P_U = P_J \bigcup I_k$
4. For all pairwise distances keep the paths with minimal length:
   $I_{k+1} = \{\underset{length}{\min_{i,j}}(\{(a, b) \in P_U | a = i \wedge b = j\})\}$

Here I_k is the set of the shortest paths in the k-th iteration.

MapReduce The proposed MapReduce variant of Floyd-Warshall algorithm consists of a driver program which initiates iterations until a termination criterion is satisfied and two successive MapReduce jobs which alternately join the intermediate shortest paths and determine the paths of minimal length. The MapReduce jobs are shown on the left-hand

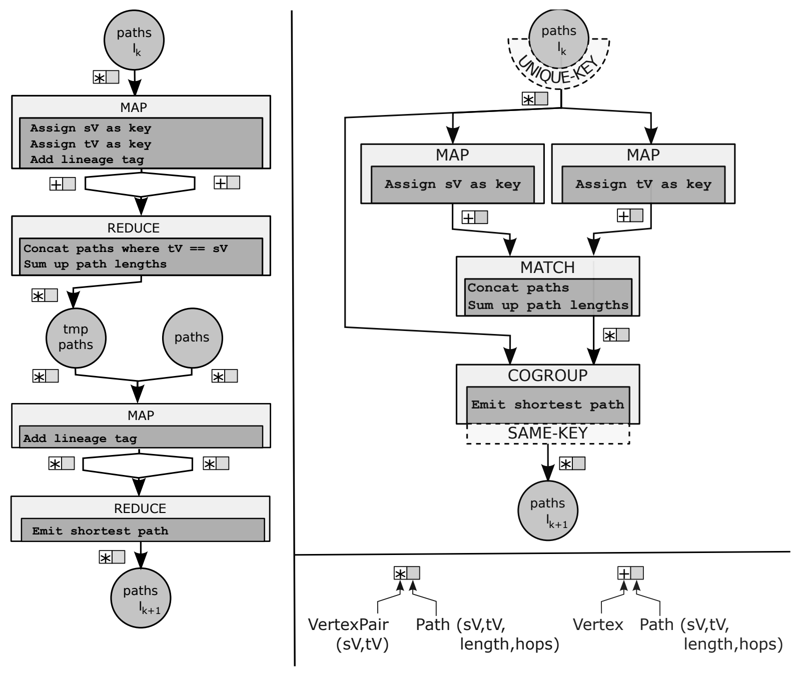


Figure 8: Implementation of the Shortest Paths Algorithm with MapReduce (left) and PACT (right)

side in Figure 8.

We assume that the input data set I_k is structured the following way: Source and target vertex of a path build a composed key, whereas the hops and length of the path from the source to the target vertex are stored in the value. For the initial iteration $(k = 0)$, these paths are the edges from the original graph.

The first Map task processes the inputs set I_k and generates the sets P_S and P_T by emitting each path twice: The paths emitted into P_S use the source vertex of the path as key, the paths emitted into P_T use the target vertex of the path as key. The value is in both cases a description of the path containing the length and the hops. Since the mapper has only one output, a lineage tag is added to each path, describing the set it belongs to (c. f. Section 3.1). The subsequent reducer collects all paths sharing the same key and separates them into the subsets of P_T and P_S. It concatenates the paths of each element in P_T with each element in P_S, thereby effectively joining the paths and creating new candidates for the path that starts at the beginning of the path described by the element from P_T and ending at the end of the path described by the element from P_S. The reducer emits those paths in the same format as the input data, using the source and target vertex together as the key.

The mapper of the second job is basically an identity function. It takes the union of the original input I_k and the result of the first reducer as its inputs and re-emits it. The subsequent reducer receives all paths that start and end at the same vertices in one group. It keeps only the paths with minimal length and emits the set I_{k+1} as result of the k-th iteration.

PACT The right-hand side of Figure 8 shows the PACT variant of the same algorithm. The first mapper of the MapReduce job is substituted by two individual map tasks: the first one emits the source vertex of the path as key, the second one the target vertex. The Match performs the same task as reducer in the MapReduce variant - the join of the sets P_S and P_T, although in a more natural fashion. Finally, the CoGroup contract guarantees that all pairs (with the same key) from two different inputs are supplied to the minimum aggregation. It does so by grouping the output of the Match by individually, selecting the shortest of the new paths for each pair of vertices. Lastly, it compares the length of that path with the original shortest path's length and returning the shorter.

3.5 Edge-Triangle Enumeration

Task Identifying densely-connected subgraphs or *trusses* [Coh08] within a large graph is a common task in many use-cases such as social network analysis. A typical preprocessing step is to enumerate all triangles (3-edge cycles) in the graph. For simplicity, we consider only undirected graphs, although the algorithm can be extended to handle more complex graph structures (like multigraphs) with the help of a simplifying preprocessing step. The algorithm requires a total order over the vertices to be defined, for example a lexicographical ordering of the vertex IDs. The graph is stored in the distributed filesystem as a sequence of edges (pairs of vertices). When generating the key/value pairs from the file, each edge will be its own pair. Inside the pair, both the key and value will consist of both the edge's vertices, ordered by the defined ordering.

MapReduce The MapReduce approach to solve the edge-triangle enumeration problem was proposed by Cohen [Coh09]. It requires the graph to be represented as a list of edges, augmented with the degrees of the vertices they connect. The implementation is depicted on the left-hand side of Figure 9 and comprises two successive MapReduce jobs that enumerate and process the so-called *open triads* (pairs of connected edges) of the graph. The first Map task sets for each edge the key to its lower degree vertex. The subsequent reducer works on groups of edges sharing a common lower-degree vertex and outputs each possible subset consisting of two edges, using the vertex pair defined by the ordering of the two corresponding higher-degree vertices as the key. The mapper of the second job takes two inputs – the original augmented edge list and the open triads from the preceding reducer. It sets the edge's vertices (in order) as the key for the original edges and leaves the open triads unchanged. The technique of adding a lineage tag to each record is used, allowing the second reducer to separate its inputs again into sets of open triads and edges. Hence, it works on groups consisting of zero or more open triads and at most one single edge which completes the triads forming a closed 3-cycle.

PACT The edge-triangle enumeration algorithm can be expressed as a PACT program as shown on the right-hand side of Figure 9. The first MapReduce job, enumerating all open triads, can be reused without any further change. The second MapReduce job is replaced

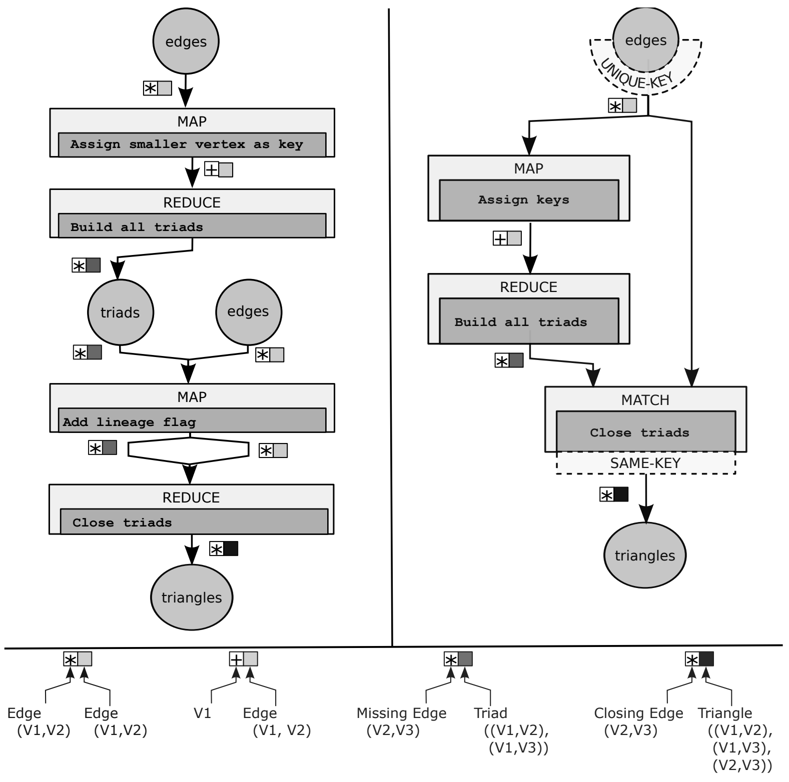


Figure 9: Enumerating triangles with MapReduce (left) and PACT (right)

by a Match contract with two inputs – the Reduce contract's result and the original edge list. Open triads and closing edges are matched by their key. The Match function closes the triad and outputs the three edges as a closed triangle.

4 Related Work

In recent years a variety of approaches for web-scale data analysis have been proposed. All of those efforts base on large sets of shared-nothing servers and a massively-parallel job execution. However, their programming abstractions and interfaces differ significantly.

The MapReduce programming model and execution framework [DG04] are among the first approaches for data processing on the scale of several thousand machines. The idea of separating concerns about parallelization and fault tolerance from the sequential user code made the programming model popular. As a result, MapReduce and its open source implementation Hadoop [Had] have evolved as a popular parallelization platform for both industrial [TSJ+09, ORS+08, BERS] and academic research [YDHP07, VCL10].

Due to the need for ad-hoc analysis of massive data sets and the popularity of Hadoop, the research and open-source communities have developed a number of higher-level languages and programming libraries for Hadoop. Among those approaches are Hive [TSJ$^+$09], Pig [ORS$^+$08], JAQL [BERS], and Cascading [Cas]. Those projects have a similar goals in common, such as to ease the development of data parallel programs and to enable the reuse of code. Data processing tasks written in any of these languages are compiled into one or multiple MapReduce jobs which are executed on Hadoop. However, all approaches are geared to different use-cases and have considerably varying feature sets.
The Hive system focuses on data warehousing scenarios and is based on a data model which is strongly influenced by the relational data model. Its language for ad-hoc queries, HiveQL, borrows heavily from SQL. Hive supports a subset of the classical relational operations, such as select, project, join, aggregate, and union all. Pig's data and processing model is less tightly coupled to the relational domain. The query language Pig Latin is rather designed to fit in a sweet spot between the declarative style of SQL, and the low-level, procedural style of MapReduce [ORS$^+$08]. JAQL is a query language for the JSON data model. Its syntax resembles UNIX pipes. Unlike HIVE or Pig, JAQL's primary goal is the analysis of large-scale semi-structured data.
All of the three languages apply several optimizations to a given query that the PACT compiler applies as well. The key difference is, that they deduce their degrees of freedom from the algebraic specification of their data model and its operators. In contrast, PACT's compiler deduces them from the input and output contracts, thereby maintaining schema-freeness, which is one of the main distinction between MapReduce systems and relational databases. Another important difference is that HIVE, JAQL, and Pig realize their decisions within the user code, which is nontransparent to the execution platform. The PACT compiler's decisions are known to the execution framework and hence could be adapted at runtime. All aforementioned languages could be changed to compile to PACT programs instead of MapReduce jobs, automatically benefiting from the PACT compiler optimizations. Therefore, those works are orthogonal to our.
Cascading is based on the idea of pipes and filters. It provides primitives to express for example split, join, or grouping operations as part of the programming model. Cascading's abstraction is close to the PACT programming model. However, like the declarative languages discussed above, Cascading translates its programs by directly mapping them into a sequence of MapReduce jobs. It performs only simple rewrite optimizations such as chaining map operations. Given the more flexible execution engine Nephele, the PACT compiler considers several alternative plans with different execution strategies (such as broadcasting vs. repartitioning) for a given program.

Despite the success of the MapReduce programming model, its ability to efficiently support complex data processing tasks, e.g. join-like operations, has been a major concern [YDHP07, PPR$^+$09, BEH$^+$10]. As a remedy, Yang et al. [YDHP07] proposed to extend the classic Map and Reduce cycle by a third, so-called Merge phase. The additional Merge phase can process data from two separate input sets and therefore enables a more natural and efficient implementation of join-like operators. Several strategies for efficient domain-specific join operations based on the unmodified version of MapReduce exist, e.g. set similarity joins [VCL10]. Other work focuses on improving Hadoop's support for iterative tasks [BHBE10].

SCOPE [CJL+08] and DryadLINQ [YIF+08] are declarative query languages and frameworks, designed to analyze large data sets. Both approaches differ from the already discussed approaches, as they do not build upon MapReduce. Instead, queries are compiled to directed acyclic graphs (DAGs) and executed by the Dryad system [IBY+07]. In that sense, SCOPE and DryadLINQ are similar to the PACT programming model, which is compiled to a DAG-structured Nephele schedule. However, SCOPE and DryadLINQ are higher-level languages and not offer an abstraction that enables dynamic parallelization. In contrast to MapReduce-based query languages or the Nephele/PACT system, both languages omit an explicit, generalized parallelization model. Instead, they consider parallelization on a language-specific level.

5 Conclusions

We presented a comparison of several analytical tasks in their MapReduce and PACT implementations. The PACT programming model is a generalization of MapReduce, providing additional second-order functions, and introducing output contracts. While the performance benefits of the PACT programming model were shown in previous work [BEH+10], this paper focused on the perspective of the programmer.

We have shown that the extended set of functions suits many typical operations very well. The following points clearly show PACTs advantages over MapReduce: 1) The PACT programming model encourages a more modular programming style. Although often more user functions need to be implemented, these have much easier functionality. Hence, interweaving of functionality which is common for MapReduce can be avoided. 2) Data analysis tasks can be expressed as straight-forward data flows. That becomes in particular obvious, if multiple inputs are required. 3) PACT frequently eradicates the need for auxiliary structures, such as the distributed cache which "brake" the parallel programming model. 4) Data organization operations such as building a Cartesian product or combining pairs with equal keys are done by the runtime system. In MapReduce such functionality must be provided by the developer of the user code. 5) Finally, PACT's contracts specify data parallelization in a declarative way which leaves several degrees of freedom to the system. These degrees of freedom are an important prerequisite for automatic optimization - both a-priori and during runtime.

References

[ABE+10] Alexander Alexandrov, Dominic Battré, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. Massively Parallel Data Analysis with PACTs on Nephele. *PVLDB*, 3(2):1625–1628, 2010.

[BEH+10] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In *SoCC '10: Proceedings of the ACM Symposium on Cloud Computing 2010*, pages 119–130, New York, NY, USA, 2010. ACM.

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

[BHBE10] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *PVLDB*, 3(1):285–296, 2010.

[Cas] Cascading. URL: http://www.cascading.org/.

[CJL+08] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *PVLDB*, 1(2):1265–1276, 2008.

[Coh08] Jonathan Cohen. Trusses: Cohesive Subgraphs for Social Network Analysis. http://www2.computer.org/cms/Computer.org/dl/mags/cs/2009/04/extras/msp2009040029s1.pdf, 2008.

[Coh09] Jonathan Cohen. Graph Twiddling in a MapReduce World. *Computing in Science and Engineering*, 11:29–41, 2009.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[DQRJ+10] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah. *PVLDB*, 3(1):518–529, 2010.

[Had] Apache Hadoop. URL: http://hadoop.apache.org.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, pages 59–72, 2007.

[Mah] Apache Mahout. URL: http://lucene.apache.org/mahout/.

[ORS+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD Conference*, pages 1099–1110, 2008.

[PPR+09] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *SIGMOD Conference*, pages 165–178, 2009.

[TSJ+09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[VCL10] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using MapReduce. In *SIGMOD '10: Proceedings of the 2010 international conference on Management of data*, pages 495–506, New York, NY, USA, 2010. ACM.

[YDHP07] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and Douglas Stott Parker. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In *SIGMOD Conference*, pages 1029–1040, 2007.

[YIF+08] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI*, pages 1–14, 2008.