Oivos: Simple and Efficient Distributed Data Processing[*]

Steffen Viken Valvåg   Dag Johansen
Department of Computer Science
University of Tromsø, Norway
{steffenv, dag}@cs.uit.no

Abstract

The complexity of implementing large scale distributed computations has motivated new programming models. Google's MapReduce model has gained widespread use and aims to hide the complex details of data partitioning and distribution, scheduling, synchronization, and fault tolerance. However, our experiences from the enterprise search business indicate that many real-life applications must be implemented as a collection of related MapReduce programs. Since the execution of these programs must be monitored and coordinated externally, several issues concerning scheduling, synchronization, and fault tolerance resurface. To address these limitations, we introduce Oivos; a high-level declarative programming model and its underlying runtime. We show how Oivos programs may specify computations that span multiple heterogeneous and interdependent data sets, how the programs are compiled and optimized, and how our run-time orchestrates and monitors their distributed execution. Our experimental evaluation reveals that Oivos programs do less I/O *and execute significantly faster than the equivalent sequences of MapReduce passes.*

1 Introduction

Search technologies are often taken for granted. For the majority of Internet users, search services have always been ubiquitous, and frequently in the critical path of their daily work. A search service usually has a deceptively simple interface: a single input box where a query is entered, and an ordered result page that is displayed without any noticeable processing delay. From this it seems natural to conclude that the underlying functionality must be simple if not trivial. Few users realize the paradox in being able to search through billions of web documents in less time than it takes to download any single one of them.

In reality, search engines constitute some of the most complex distributed systems imaginable. Thousands if not millions of users are served in parallel, accessing huge data sets distributed across thousands of machines, and the service remains available despite the inevitable failures of individual software and hardware components. Mastering this complexity is possible through the contributions of hundreds to thousands of skilled engineers and system administrators. A bird's eye view of a search engine reveals a complex set of distributed computations. While queries are served in real-time by look-ups in massively replicated index structures, a whole portfolio of batch-oriented computations process the vast data volumes downloaded from the web, transforming and correlating data before it ends up in its final, indexed form. Due to the sheer volume of data involved, each of these computations must be distributed across numerous machines.

As a reaction to this complexity, Google introduced the MapReduce [3] programming model. MapReduce programs are implemented in a succinct functional style, and can process or generate huge data sets without concern for the non-functional requirements of data partitioning and distribution, scheduling, synchronization, and fault tolerance. This model has gained widespread use, and several open-source implementations exist in addition to the one developed by Google.[†]

In this paper, we identify several limitations with the MapReduce programming model and present an en-

[*]This work is supported in part by the Research Council of Norway through the National Center for Research-based Innovation program. Dag Johansen is also chief scientist at the enterprise search company Fast Search & Transfer (http://fastsearch.com/)

[†]For example, two available MapReduce implementations are Hadoop (http://lucene.apache.org/hadoop/) and MapReduce on Cell (http://sourceforge.net/projects/mapreduce-cell).

hancement that addresses these limitations. Our approach is based on hands-on experience from the enterprise search domain, where large corporations embed search as an integral part of their enterprise platforms. In this field, there is an increasingly apparent need for powerful, high-level abstractions that facilitate the development and deployment of distributed computations.

The rest of this paper is organized as follows. In Section 2 we describe the MapReduce programming model and its limitations. We then present the Oivos[‡] programming model and its benefits. Section 3 describes the Oivos architecture, and details how high-level Oivos programs are compiled into a low-level representation suitable for its distributed run-time. Section 4 evaluates the performance of the Oivos run-time, and makes a quantitative comparison of the Oivos and MapReduce programming models. In Section 5 we discuss Oivos in a broader perspective, and its applications in other domains. Section 6 concludes.

2 Programming Model

In this section, we describe the MapReduce programming model and one of its derivatives. Then we describe the Oivos programming model, and how it addresses limitations experienced with MapReduce.

2.1 MapReduce

A MapReduce program processes a set of input records, producing a set of output records. The programmer implements two functions: a *mapper* and a *reducer*. The mapper is applied (potentially in parallel) to all input records, producing a set of intermediate key/value pairs for each input record. For each unique intermediate key, all values associated with that key are subsequently passed to the reducer, which produces the final set of output records.

A MapReduce program thus transforms one homogeneous set of records into another homogeneous set of records, using a number of map- and reduce tasks. While this simple model suffices for several useful applications, it is cumbersome for relational data processing, which often operates on multiple heterogeneous data sets. This weakness was recognized by researchers at Yahoo and UCLA, who proposed the Map-Reduce-Merge model [10] that extends MapReduce to allow two input sets of records, adding a merging phase after the map- and reduce phases. This way, simple joins can be implemented as a single Map-Reduce-Merge pass. However, most practical applications still require a number of MapReduce or Map-Reduce-Merge passes to perform the desired computation. For example, Yang et al. [10] give a join tree for Query 2 from the TPC-H database benchmark that uses 13 Map-Reduce-Merge passes. Google reports that their web indexing code uses a sequence of 5 to 10 MapReduce passes [3], which discounts the many MapReduce passes reportedly used for related pre-processing of web documents.

When a large distributed computation is broken up into a number of separate MapReduce passes, several issues concerning scheduling, synchronization, and fault tolerance resurface. Some process external to the MapReduce implementation must monitor the status and progress of passes, determining if and when to re-execute a failed pass or start the next one. The programmer must also determine a valid execution order for the MapReduce passes, considering that some of the data might be out of date and need to be regenerated. In the event of potential parallelism, arranging for multiple passes to execute concurrently is another task left to the programmer. Even if an optimal scheduling of passes is achieved, a typical MapReduce implementation will introduce a barrier synchronization point at the end of each pass, requiring every reduce task in one pass to complete before any of the map tasks in the next pass can start. This restriction reduces the potential parallelism in multi-pass MapReduce computations.

2.2 Oivos

As described in the previous section, the limitations of MapReduce force the programmer to reconsider several non-functional requirements when implementing complex real-world computations.

Oivos addresses this concern by raising the abstraction level. A single Oivos program may specify what amounts to numerous MapReduce passes, involving multiple heterogeneous record sets. There is no need to implement a computation as a collection of small programs whose execution must be coordinated externally; a single program specifies everything, and once it is passed to the run-time for execution it will run to completion.

Additionally, Oivos offers very useful semantics similar to those of a *make* program; in the event that a previous run was aborted, or that a subset of files have been modified since the previous run, Oivos will automatically determine which tasks must be re-executed by examining logical timestamps maintained by an underlying distributed file system.

[‡]Derived from the Sami word *oivoš*, denoting the source of a river.

The primary abstraction in the Oivos programming model is the *table*, which represents a homogeneous set of records. Tables may either be declared as input tables, or they may be derived by applying operators to other tables. The table operators are generally parametrized by user-specified functions and may thus be viewed as higher-order functions that transform tables into new tables. Oivos programs are declarative, and do not specify an order of execution; they merely declare all tables, some of which are derived from others. A program is executed by specifying the set of desired output tables; the Oivos run-time will automatically determine which tasks to execute in order to produce those tables.

2.3 Examples

As an example problem domain, consider a web graph manifested by a huge table of link records, each of which contains a source and target URL. Such a data set would typically be produced by a web crawler that logs all hyperlinks encountered in the documents it downloads. Several practical applications rely on analysis of web graphs to improve or add functionality to a web search engine. Most notably, web graph analysis is often integral to the ranking of search results, as evidenced for example by Google's PageRank algorithm [1].

One instance of web graph analysis is to compute the number of in-links for each site in a web graph. Figure 1 shows a simple Oivos program to perform this computation. The program applies the `map` operator to map each of the input `Link` records to a `Site` record that contains the host name component of the link's target URL, and an in-link count of 1. All `Site` records with identical host names are then combined by summing their respective in-link counts, using the `combine` operator. The final line tells the run-time to produce the `sites` table, which will contain one `Site` record per unique host name.

Readers that are familiar with MapReduce will recognize that our first example could be implemented by a single MapReduce program. To better illustrate the benefits of the Oivos programming model, we now give an example implementing a slightly more elaborate analysis, in which the desired output is a site table that contains the number of out-links as well as in-links for each site. The analysis can be run on a recurring basis, each time incorporating the links that have been discovered since the previous run. This fits the real-world scenario of a web crawler that continuously discovers new links; periodically, a batch of links is processed and incorporated into the existing site table. A straightforward way to compute such a site table is using four steps:

```
record Link:
    String fromURL
    String toURL

record Site:
    String host
    Integer inLinks

function Mapper(Link link):
    emit Site(parseHost(link.toURL), 1)

function Combiner(Site a, Site b):
    emit Site(a.host, a.inLinks + b.inLinks)

Table<Link> links = input(``/mylinks'')
Table<Site> tmpSites = links.map(Mapper)
Table<Site> sites = tmpSites.combine(Combiner)
sites.Produce()
```

Figure 1. Computing the number of in-links for each site in a web graph.

1. Computing the number of in-links per site.
2. Computing the number of out-links per site.
3. Merging the output of the first two steps to produce a single site table.
4. Merging the new site table with the previous version of the site table to produce the desired output.

Figure 2 shows an Oivos program that implements the above. Each of the above steps corresponds to one of the four code lines indicated by a comment in the example. A table containing the number of in-links per site is declared by applying the `map` operator followed by the `combine` operator to the table of links, as in the first example. A table containing the number of out-links per site is declared similarly; these two tables are merged to produce an intermediate site table that contains all the aggregated link counts; the intermediate site table is merged with the previous version of the site table to produce the output site table. Note that this program does not specify an order of execution; it merely declares the tables involved in the computation and how they are derived from each other. When the final line invokes `Produce`, the Oivos run-time is requested to produce the site table, and will infer what needs to be done (and in what order) based on the data dependencies. In this particular case, it will recognize that steps 1 and 2 can be performed in parallel, a fact that might go unnoticed by a casual programmer. A MapReduce implementation of this example would require a separate MapReduce program for each of the four steps; discovering that steps 1 and 2 could run in parallel, and arranging for them to do so, would be up to the programmer.

For brevity, we presented these examples using pseudo code. In reality, Oivos programs are imple-

```
record LinkCount:
    String host
    Integer count

record Site:
    String host
    Integer inLinks
    Integer outLinks

function FromMapper(Link link):
    emit LinkCount(parseHost(link.fromURL), 1)

function ToMapper(Link link):
    emit LinkCount(parseHost(link.toURL), 1)

function Combiner(LinkCount a, LinkCount b):
    emit LinkCount(a.host, a.count + b.count)

function Merger(LinkCount in, LinkCount out):
    emit Site(in.host, in.count, out.count)

function SiteMerger(Site old, Site new):
    emit Site(old.host,
              old.inLinks + new.inLinks,
              old.outLinks + new.outLinks)

Table<Site> oldSites = input(``/sites'')
Table<Link> links = input(``/mylinks'')
Table<LinkCount> inLinks, outLinks
Table<Site> newSites, sites

# Steps 1, 2, 3 and 4:
inLinks = links.map(FromMapper).combine(Combiner)
outLinks = links.map(ToMapper).combine(Combiner)
newSites = inLinks.merge(outLinks, Merger)
sites = oldSites.merge(newSites, SiteMerger)

sites.Produce()
```

Figure 2. Computing the number of in- and out-links for each site in a web graph.

mented in Java, using a library that offers the table abstraction. Tables are represented by objects created by the library. To apply an operator to a table a method is invoked on its table object; the return value is another table object. Record types are implemented as regular Java classes; functions that operate on records, such as the mappers, combiners and mergers used in the preceding examples, are regular Java methods that accept instances of the record classes. Java's generics feature is used throughout to avoid any need for casting, and to ensure type safety in the application of table operators. Each table also has an associated key type, which must implement Java's `Comparable` interface; this defines a total order on record keys. In practical terms, the key of a table may be a record field, a combination of record fields, or in general any value that may be derived deterministically from a record. This flexibility is achieved by specifying the key in terms of a user-defined *key function* that accepts a record and returns its key.

There are many advantages to specifying Oivos programs through a programmatic interface, as opposed to inventing a new domain-specific language. Since there is no limit to what a user-specified function such as a mapper could potentially do, a domain-specific table manipulation language would in reality have to include most if not all facilities of a general-purpose programming language. Our approach allows for seamless integration of Oivos programs with an existing code base. Functionality like MD5 check-summing, URL parsing, date formatting, etc. can be implemented simply by invoking the standard Java libraries. Another useful property is that table objects fully encapsulate the specification of how to produce a table; as such, they can be used for lazy evaluation of tables. If an existing component A implements the logic required to produce a table that another component B needs, A can simply construct an appropriate table object that specifies how to produce the table, and pass the table object to B. If or when B requires the table to be produced, it simply invokes the `Produce` method on the table object.

2.4 Table Operators

In summary, Oivos programs declare tables that may be transformed into other tables using table operators. We now present the full set of table operators in Oivos, and describe their exact semantics.

The **Map** operator applies a user-specified mapper function to each record in a table. The mapper function may emit 0, 1, or multiple output records per input record. The result is a new table containing all emitted records. The record and key types of the output table may differ from those of the input table.

The **Sort** operator sorts the records of a table according to a user-specified key. This operator is applied implicitly by other operators that require sorted input, and may be applied explicitly as well. The result is a new table with the same record type as the input table; the key types of the input and output tables may differ.

The **Reduce** operator applies a user-specified reducer function once per unique key in a table; the reducer function accepts a key and an iterator that can be used to iterate over all records with that specific key. Like mapper functions, reducer functions may emit a varying number of output records; the result is a new table whose record and key types may differ from the input table.

The **Combine** operator is used to combine all records with equal keys into a single record. The user specifies a combiner function that accepts two records and returns one; it is repeatedly applied to pairs of records with equal keys until a single record remains per unique key. The combiner function must be associative and commutative, such that the resulting record will be identical regardless of the order in which the function is applied to record pairs. Additionally it may not change the record keys. The result of the combine operator is a new table that has the same record and key types as the input table, with exactly one record per unique key. This operator is similar to the family of higher-order *fold* functions known from functional languages, except that it does not imply a particular order of evaluation.

The **Merge** operator is a binary operator that merges two sorted input tables with the same key type into one new table[§], applying a user-specified merger function for each unique key. The merger function accepts two records and may emit 0 or more output records. In the case of keys that appear in both input tables, the arguments to the merger function are one record from each of the input tables. For keys that only appear in one of the input tables, a `null` record is passed as the "missing" argument. The merger function may thus implement outer or inner joins as desired. The result is a new table, whose record and key types may differ from those of the input tables.

The table operators are inherently well-suited for parallel evaluation. This is because the user-specified functions are either applied once per input record (e.g., mapper functions) or once per unique key (e.g., reducer or combiner functions). In the former case, the function has no dependencies on other records and may thus be evaluated in parallel for all records. Similarly, reducer and combiner functions may be evaluated in parallel for all unique keys. In practice, the actual degree of parallelism to use is determined by the run-time as described in Section 3.2.

Note that using the combine operator when possible is generally preferable to using the reduce operator; this is because reducer functions require a collection of all records with equal keys in order to evaluate. In contrast, a combiner function only requires a pair of records with equal keys and may thus be evaluated earlier (using the principle of upstream evaluation [2]). In short, the combine operator is inherently more parallelizable than the reduce operator.

3 Architecture

So far we have described how our high-level programming model allows the specification of complex distributed computations. Oivos programs need not relate to the details of how data and computations are distributed – this is handled automatically by the run-time. We now explain the architecture of Oivos by introducing its individual components, and describe how these components co-operate to ensure the efficient and fault-tolerant execution of Oivos programs.

[§]A series of binary merges can implement a multi-way merge.

3.1 Components

The internals of Oivos can be broken up into the following three components:

Compiler. The compiler implements the logic for compiling a high-level Oivos program into a set of tasks that process individual files. Each task depends on a set of input files that must be present before the task can execute and produce a set of output files. The low-level representation of an Oivos program is thus a precedence graph of tasks and files.

Task Scheduler. The task scheduler is a distributed system that provides an interface similar to a *make* program; given a precedence graph such as the one produced by the compiler, and a set of desired output files, it will execute any and all tasks needed to ensure that the output files are up to date. Files are considered up to date when they are more recent than all of their ancestors in the precedence graph. The task scheduler may be servicing requests for multiple disjoint task sets concurrently, meaning that multiple unrelated Oivos programs may be running in parallel. While the only abstractions used in Oivos programs are tables and table operators, the task scheduler knows nothing about these concepts and only deals with files and tasks. This separation of concerns is sound from an engineering point of view, as it allows for independent and incremental improvements to either the compiler or the task scheduler. It also implies that the task scheduler can be used for precedence graphs that are manually constructed or do not originate from the Oivos compiler, a feature we have found to be useful on occasion.

File System. The Oivos file system is a distributed file system tailored for the limited requirements of Oivos tasks. In particular, files may only be written sequentially, and are immutable after their initial creation. Furthermore, files may be concatenated in constant time; data blocks are reference-counted and may appear in multiple files, so concatenation is a pure meta-data operation. Files may be replicated to varying degrees; Oivos programs may specify different replication degrees for different tables. Since intermediate tables can be reproduced based on their input tables, the most common practice is to only replicate the initial input tables of a computation. The file system also maintains logical timestamps for all files, used by the task scheduler to determine which files are up to date.

We collectively refer to the task scheduler and file system as the Oivos run-time. The compiler is a part of the Java library used to specify Oivos programs; it thus executes at the client-side of the system.

We now give an overview of how the various components in Oivos relate to each other by walking through the actual sequence of events as an Oivos program is compiled and executed. As noted, the high-level table language is implemented by library code that runs in the client process. The library allows the client program to programmatically declare tables, which are represented by table objects. Methods corresponding to the various table operators may then be invoked on the table objects, resulting in new table objects. None of this implies the initiation of any form of I/O or distributed computation; it is merely a specification of how the various tables are to be produced. When the client program wants to produce a specific table, it invokes the `Produce` method on the corresponding table object, which triggers the following sequence of events.

1. The library compiles the set of tasks required to produce the table.

2. The resulting set of tasks is passed to the task scheduler. The scheduler ensures that the relevant tasks are executed in the correct causal order, i.e. according to the precedence graph formed by the tasks and their input file dependencies. The actual execution of tasks is delegated to a set of worker nodes managed by the scheduler.

3. The worker nodes execute the various tasks, processing input files and producing output files that all live in the shared file system. Each worker reports back to the task scheduler once a task is completed, at which point the task scheduler typically hands the worker another task to execute.

4. Once all tasks have completed, the task scheduler informs the client program that the computation is complete, and the client program resumes by returning from the originating `Produce` invocation. At this point, all the files that represent the table exist in the shared file system. The table may subsequently be used as input to other Oivos programs, exported to some other component for further processing, or both. A typical use case is to convert the output table into a look-up structure to be served by an on-line service, while also using it as an input table in future off-line processing runs.

Oivos relies on re-execution as the primary strategy for fault tolerance. If a worker fails while executing a task, the task scheduler will simply reschedule the task on another worker. If the task scheduler fails, a new one will be started to replace it. This has the effect of aborting all currently executing Oivos programs. Their execution is automatically resumed from the client-side by invoking the `Produce` method again; the new task scheduler will examine the file system state and pick up exactly where the old task scheduler left off.

Like the Google File System [4], the Oivos file system uses a single master process to hold all meta-data in memory. In the event of a failure, a new master can quickly be brought on-line by replaying a meta-data log that is replicated on multiple machines. File data is stored on a set of data nodes; their failure will result in transient or permanent data loss. The risk of losing critical data such as input tables may be reduced through replication. Non-input tables will be reproduced from their inputs even if all replicas are lost.

3.2 Compilation

We now outline how the high-level table language is compiled into a set of tasks that may be executed by the Oivos run-time. Each table is represented as a set of files in the file system. The records of a table are partitioned among the files by hashing the record keys. Each record is thus stored in one (and only one) of the files, and records with equal keys will always be stored in the same file. The number of files used to represent a table determines the maximum degree of parallelism when applying operators to the table; we refer to this factor as the *split factor* and it may be configured manually or automatically decided by the compiler.

A unary table operation (meaning the application of an operator to a single table) is generally compiled into one task per input file. If the output table has the same key type as the input table, as well as the same split factor, we refer to the tables as *compatibly partitioned*. If the operation also preserves record keys, each task can directly output one of the files in the output table. Figure 3 shows the precedence graph for this simple case, where the round nodes labeled “op” represent tasks, and the square nodes represent files. The tasks simply read their input files sequentially, applying a user-defined function (such as a mapper) to each input record, appending the emitted records to the output file. As indicated by the precedence graph, all of the tasks can execute in parallel.

When tables are incompatibly partitioned, or the operation might modify record keys, a more complex compilation procedure which we refer to as *re-splitting* is required, since the records from each input file will be scattered among the output files according to the new

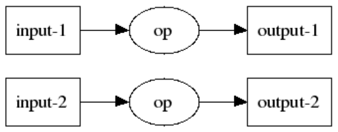

Figure 3. A precedence graph for a unary table operation where the input and output tables both have 2 files and record keys are preserved.

hash values of their keys. The need for re-splitting is detected at compile time, and handled automatically as follows. As before, a single task processes each input file. However, the task outputs a set of N intermediate files, where N is the split factor of the output table. As records are emitted they are appended to one of the intermediate files; which file is determined by hashing the record keys. If the input table has a split factor of M, there will thus be M tasks that each produce N intermediate files, for a total of $M \times N$ intermediate files. The intermediate files are subsequently combined to form the set of N files that represent the output table. This is done by concatenating the first intermediate file from each task to form the first output file, concatenating the second intermediate file from each task to form the second output file, and so on. As noted in Section 3.1, the Oivos file system system supports constant-time concatenation of files, so the tasks that concatenate files do not add any significant overhead. Figure 4 shows the resulting precedence graph when the input table has 3 files and the output table has 2 files.

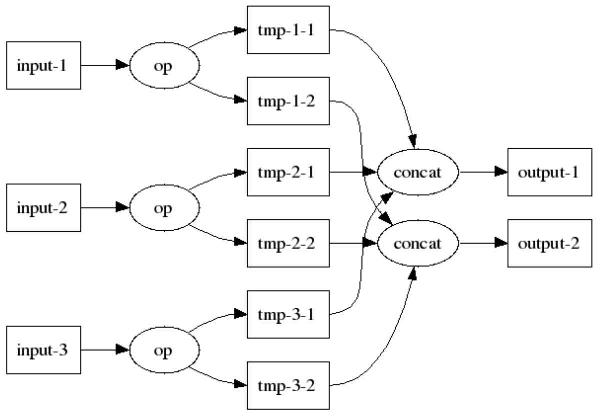

Figure 4. A precedence graph for a unary table operation where the input table has 3 files and the output table has 2 files.

Although re-splitting is performed without programmer intervention, it is an elaborate procedure that incurs the I/O overhead of reading and writing the entire table once more. As such, it should be avoided to the extent possible. One way to minimize re-splitting is to use the same split factor for all tables involved; hence, tables inherit the split factor of their input tables unless another split factor is specified explicitly by the programmer.

We now explain how each individual operator is compiled. The `map` operator is compiled into a separate map task for each file in the input table. Each map task reads records sequentially from a file, applies the user-specified mapper function to each input record, and outputs all records emitted by the mapper.

The `sort` operator requires the input table to be partitioned according to the key by which the records should be sorted. If this is not already the case, the table is re-split as described above, using an identity mapper function that outputs all input records unchanged (but with the correct key). Once the table is correctly partitioned, each input file is sorted independently by adding a separate sort task per file. The sorted representation of a table is thus the same as for an unsorted table, except that each file is internally sorted according to the record keys. Sort tasks will use an external sort if the file to be sorted is too large to fit in memory; however, the split factor is often sufficiently high to avoid this.

Sometimes a table is already in its sorted representation, even if this cannot be inferred by the compiler. Consider the common case of a mapper function that outputs records whose keys have the same sort order as those of the input records (e.g., stemming of string keys); if the input table happens to be sorted, the same will then apply for the output table. However, a mapper function is nothing but an opaque piece of Java code to the compiler, and it cannot in general infer such properties. To address this problem, records are examined at run-time whenever they are appended to a file, checking if the records happen to be written in sorted order. If so, the file is flagged as sorted, and if a subsequent sort task attempts to sort the file it will recognize the flag and simply copy the file unchanged instead of sorting it again. Since copying is merely a special case of concatenation, it is also an inexpensive constant-time operation in our file system, so this simple optimization can greatly reduce I/O load.

The `reduce` operator requires the input table to be sorted; a sort operation is implicitly added by the compiler if the input table is not already sorted. The compiler then adds a separate reduce task per input file. Each reduce task can read its input sequentially, since equal-keyed records will be grouped next to each other in a sorted file. The user-specified reducer function is applied once per unique key encountered; the iterator passed to the reducer will keep iterating over the input records until the next key is encountered. The reduce task outputs all records that are emitted by the reducer.

Merging two tables using the `merge` operator is accomplished by first ensuring they are compatibly partitioned; if they are not, one of the input tables will be re-split. Furthermore, both input tables must be sorted; implicit sort operations will be added if this is not already the case. The tables are then merged by adding a separate merge task for each pair of input files: One task to process the first file of each input table, another to process the second file of each input table, and so on. Since the input tables are compatibly partitioned, equal keys are guaranteed to occur in the same file in both input tables. The merge tasks employ a standard merge algorithm to merge their input records into a sorted sequence; the user-specified merger function is applied once for each unique key, as described in Section 2.4, and all emitted records are outputted. If the merger function does not change record keys, the output will thus be in the sorted representation described above.

The `combine` operator is essentially a special case of the `reduce` operator. However, a combiner function may not change the keys of the input records (a restriction enforced at run-time), so there is never a need to re-split the table. The compiler thus adds a single reduce task for each file in the input table, using the precedence graph in Figure 3. The reducer applied simply iterates over all equal-keyed records, applying the combiner function to pairs of records until a single record remains, which it appends to the output.

4 Performance Evaluation

In this section we evaluate the performance of Oivos by subjecting it to real workloads. Since Oivos programs are generally I/O intensive, an important metric to consider is the amount of I/O performed. Furthermore, we examine how throughput scales as a function of the number of machines employed and the size of the workload.

4.1 Experimental Setup

The experiments presented here are run on a cluster of commodity workstation computers, managed by the open-source Rocks Linux distribution for clusters. There are 8 physical machines, equipped with two quad-core Intel Xenon processors and 8 GB of RAM. Each physical machine runs two Xen virtual machines, for a total of 16 computing nodes.

The workload employed for these experiments is a web graph analysis similar in nature to the one presented in Section 2. It is extended to include HTML anchor texts for each link, and assigns weights to links based on a set of heuristics, some of which involve the IP addresses of the source and target hosts. An additional input table is therefore a log of DNS look-ups. Furthermore, the analysis recognizes that multiple URLs may be equivalent from a logical point of view, meaning they serve the same content; such equivalent URLs may arise for instance from HTTP redirections. The input to the analysis is thus a table of equivalent URLs, a table of links, and a table of DNS look-ups, all of which typically originate from a web crawler. The output table lists an aggregated link score for each set of equivalent URLs, indicative of its authority in the web graph, as well as a set of popular anchor texts used to link to the URLs. This output can be used to influence the ranking of search results in a web search service. The analysis is implemented in its entirety as a single Oivos program that involves on the order of 20 tables.

For comparative purposes, we have developed an alternative implementation of our web graph analysis that stays within the confines of the MapReduce model. Since the object is to compare the *programming models*, as opposed to their implementations, we implement the MapReduce version of the analysis as another Oivos program that only applies an alternating sequence of `map` and `reduce` operators, such that each pair of `map` and `reduce` operations corresponds to a single MapReduce pass. If there is nothing useful for a given `map` or `reduce` operation to do, we parametrize the operators using identity functions that output all input records unchanged, as is required by the MapReduce model. Since this restriction turns out to be sufficient to demonstrate significant performance improvements using our model, we do not enforce barrier synchronization between each MapReduce pass, nor do we prevent unrelated MapReduce passes from executing in parallel. Enforcing these inherent restrictions in the MapReduce model would further disadvantage MapReduce when compared to Oivos.

While there are certainly several equivalent ways of implementing a non-trivial computation like our web graph analysis using MapReduce, a careful review of our MapReduce version has not revealed any obvious way to reduce the number of required passes. To the extent that our Oivos version outperforms the MapReduce version, it is due to the less restrictive programming model, as opposed to ingenious programming.

4.2 Results

Our first experiment measures I/O usage by recording the number of bytes read from or written to the file system when executing our workloads on a sample data set of 10 million links. The I/O usage is unaffected by the split factor and the number of computing nodes used, since a higher split factor simply means that the same volume of data is partitioned into more files. The following table summarizes our findings; in total, MapReduce does 48% more I/O than Oivos. This is due to the superfluous identity `map` and `reduce` operations of the MapReduce workload.

	Reads	Writes	I/O	%
Oivos	1.83 GB	1.39 GB	3.21 GB	100
MapReduce	2.54 GB	2.22 GB	4.76 GB	148

Our second set of experiments examines the relative throughputs of our Oivos and MapReduce workloads, for the same input as before. We define throughput as the inverse of execution time, i.e. $\frac{1}{time}$. We report throughput measurements on a relative scale from 0 to 1, where 1 corresponds to the highest observed throughput.

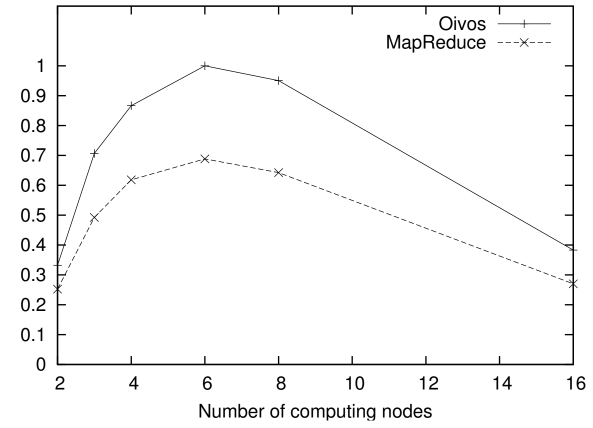

Figure 5. Relative throughput for our Oivos and MapReduce workloads.

One might expect throughput to scale linearly with the number of computing nodes in a system – in reality, this will only be the case for embarrassingly parallel computations, and only if the effective aggregate I/O bandwidth of the system scales linearly with the number of computing nodes. In the precedence graphs of our compiled Oivos programs, there are multiple barrier synchronization points. For example, consider the precedence graph for re-splitting in Figure 4; all of the "op" tasks must complete before any of the "concat" tasks can proceed. At these junctions in the precedence graph, some computing nodes must inevitably remain idle for a period of time. This underlines the importance of avoiding artificial barrier synchronization points such as those introduced by the MapReduce model. In practice, any given workload will have a "sweet spot" with regard to the number of computing nodes employed. For instance, the throughput of our web graph analysis peaks at 4 nodes when executed on 5 million input links. Figure 5 shows the relative throughputs for 10 million input links: throughput peaks at 6 nodes, and eventually degrades below that achieved using only 3 nodes. MapReduce throughput peaks at 69% of the Oivos throughput; it other words, Oivos achieves a speedup of approximately 1.45 over MapReduce.

5 Discussion

Several new programming models [3, 5, 7, 8, 10] have been introduced recently, all aiming to facilitate the development of data-intensive distributed computations. Their common goal is to reduce or hide complexity while maintaining good performance. At the core of these models (although terminology may differ slightly) we commonly find a set of idempotent tasks arranged in a directed acyclic precedence graph. In the case of MapReduce, the topology of the precedence graph is fixed, and the programmer simply modifies the behavior of certain tasks by plugging in user-defined *mapper* and *reducer* functions. As noted, this has the drawback of requiring more complex precedence graphs to be expressed in terms of multiple sub-graphs, each corresponding to one MapReduce pass.

In contrast, the Dryad [5] system from Microsoft Research allows the construction of arbitrary precedence graphs by embedding a graph manipulation language into C++ using extensive operator overloading. While the inflexible precedence graphs imposed by MapReduce are indeed a problem, the approach of explicitly constructing an appropriate precedence graph can be difficult for a novice programmer. By virtue of their declarative style, Oivos programs only relate to the data dependencies of a computation, which naturally give rise to the underlying precedence graph through a fully automated compilation process.

Another way to improve flexibility and expressiveness is to build higher-level abstractions in layers on top of MapReduce or similar infrastructures. For example, DryadLINQ [6] compiles C# code into Dryad programs, Sawzall [8] programs rely on MapReduce for execution, and the high-level Pig Latin [7] language is compiled into a collection of Hadoop MapReduce jobs. In general, such layering can restrict the possible avenues of optimization. As noted in Section 2.1, there are drawbacks to subdividing a computation into discrete MapReduce jobs, whether done manually or by a compiler.

Oivos improves upon the original MapReduce programming model in two ways. First, our declarative programming model is more expressive and allows the specification of computations that span multiple related, heterogeneous data sets. Not only does this reduce complexity for developers – it also allows the elimination of many barrier synchronization points and significantly reduces the I/O load in typical workloads.

Second, we provide functionality similar to that of a *make* program. Given some desired output, our run-time will automatically infer what to do by examining the state of the file system. If a previous processing run was aborted for some reason, the run-time will ensure that the next run picks up exactly where the previous one left off. This functionality also paves the way for other useful structuring techniques such as lazy evaluation of tables.

While our positive experiences with Oivos stem from the problem domains found in enterprise search platforms, we expect to find useful applications of our programming model in other domains as well. For example, MapReduce was recently evaluated as a programming model for multi-core and multi-processor systems [9], and found suitable for application domains such as scientific computing, artificial intelligence, and image processing, in addition to enterprise computing. Since any MapReduce computation can be expressed in Oivos, our improvements do not sacrifice any generality.

6 Concluding Remarks

An emerging trend is that data volumes are growing at a much higher rate than processing power. This is a scaling problem in particular for providers of search services, and large clusters of commodity computers are a widely adopted and relatively cheap way to scale. It is by no means trivial to program applications for such distributed environments. A simple and inexpensive approach to scaling hardware can thus lead to a proliferation of highly complex distributed software.

To cope with the complexity hidden under the hood of enterprise search services, we seek powerful high-level abstractions. With Oivos, complex distributed data processing can be expressed without concern for non-functional requirements such as data distribution and synchronization. The declarative nature of Oivos programs also obviates the need for explicit control flow. Our run-time can automatically deploy, execute and monitor computations and only requires high-level guidance; given a simple specification of the desired output it will infer how to produce it based on the current system state. In conclusion, Oivos greatly reduces the programming efforts required for high performance distributed data processing.

Acknowledgements

We would like to thank our colleagues in the iAD project, in particular Johannes Gehrke and Åge Kvalnes, for helpful comments and discussions on earlier drafts of this paper. We also salute the FDM team at Fast Search & Transfer for the excellent and innovative work that was our main inspiration.

References

[1] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[2] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *PODC*, pages 219–227, 2000.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP*, pages 29–43, 2003.

[5] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72. ACM, 2007.

[6] Microsoft Research. DryadLINQ software. http://research.microsoft.com/research/sv/dryadlinq.

[7] 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.

[8] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[9] C. Ranger, R. Raghuraman, A. Penmetsa, G. R. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *HPCA*, pages 13–24. IEEE Computer Society, 2007.

[10] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, Jr. Map-Reduce-Merge: Simplified relational data processing on large clusters. In *SIGMOD*, pages 1029–1040, 2007.