Optimus: A Dynamic Rewriting Framework for Data-Parallel Execution Plans

Qifa Ke
Microsoft Research Silicon Valley
qke@microsoft.com

Michael Isard
Microsoft Research Silicon Valley
misard@microsoft.com

Yuan Yu
Microsoft Research Silicon Valley
yuanbyu@microsoft.com

Abstract

In distributed data-parallel computing, a user program is compiled into an execution plan graph (EPG), typically a directed acyclic graph. This EPG is the core data structure used by modern distributed execution engines for task distribution, job management, and fault tolerance. Once submitted for execution, the EPG remains largely unchanged at runtime except for some limited modifications. This makes it difficult to employ dynamic optimization techniques that could substantially improve the distributed execution based on runtime information.

This paper presents Optimus, a framework for dynamically rewriting an EPG at runtime. Optimus extends dynamic rewrite mechanisms present in systems such as Dryad and CIEL by integrating rewrite policy with a high-level data-parallel language, in this case DryadLINQ. This integration enables optimizations that require knowledge of the semantics of the computation, such as language customizations for domain-specific computations including matrix algebra. We describe the design and implementation of Optimus, outline its interfaces, and detail a number of rewriting techniques that address problems arising in distributed execution including data skew, dynamic data re-partitioning, handling unbounded iterative computations, and protecting important intermediate data for fault tolerance. We evaluate Optimus with real applications and data and show significant performance gains compared to manual optimization or customized systems. We demonstrate the versatility of dynamic EPG rewriting for data-parallel computing, and argue that it is an essential feature of any general-purpose distributed dataflow execution engine.

Eurosys'13 April 15-17, 2013, Prague, Czech Republic

1. Introduction

Recent advances in high-level programming language support [9, 33, 37, 40, 41] over distributed execution engines [13, 19, 22] have greatly eased the development of large-scale, distributed data-intensive applications. In these systems, a high-level data-parallel program is compiled into an execution plan graph (EPG), representing both the computation and the data flow in a directed acyclic graph (DAG). This EPG is used by a distributed execution engine for task distribution, job management, and fault tolerance.

One major problem with current systems is that the EPG typically remains unchanged during job execution, making it difficult to employ dynamic optimization techniques such as detecting and handling data skews, data dependent re-partitioning, and runtime query plan optimization. These optimizations could substantially improve performance, and in some cases are necessary for job completion, but cannot be done as compile-time optimizations. Even for programs constructed from a small set of predefined operators with known semantics at compilation time, it is often difficult or impossible to obtain in advance the statistics of the input data to each stage of an EPG. This difficulty is compounded by the presence of user-defined functions (UDFs), which are an essential part of many modern "Big Data" computations. As a result, execution plans based on compile-time information are often inefficient [2, 14]. In addition, some computations cannot even be represented by a static DAG. One example is iterative computation where the stopping condition is needed to construct the EPG, but is available only at runtime. In such a case the EPG has to be dynamically constructed.

This paper has two main contributions. First, we built Optimus, a framework for dynamically rewriting an EPG at runtime. Optimus extends dynamic rewrite mechanisms present in systems such as Dryad [22] and CIEL [31] by integrating rewrite policies with a high-level data-parallel language and runtime, in this case DryadLINQ [40]. The high-level rewriting API is simple and flexible, allowing programmers to build specialized domain-specific computations that are difficult, if not impossible, to achieve with the current systems.

Second, we demonstrated the effectiveness of Optimus by building a number of dynamic rewriters, which successfully addressed several hard problems arising in the distributed execution of data-parallel programs:

- We implemented fine-grained dynamic partitioning to handle data and computation skew, including a new co-range partitioning scheme.
- We used the example of matrix computation to show that Optimus allows domain-specific computations to be easily plugged into a general-purpose computational framework. This specialization is much simpler than building a custom matrix algebra system such as MadLINQ [36].
- We demonstrated how to handle iterative computation by dynamically extending an EPG with new iterations until a stopping condition is met.
- We implemented a reliability enhancer that selectively replicates intermediate data on expensive critical paths in an EPG, and hence improves recovery time in the event of failures while minimizing unnecessary data transfers.

The most popular distributed data-parallel languages have been layered over Hadoop [19], an open-source implementation of MapReduce [13], and have consequently been constrained to compile their EPGs into a sequence of MapReduce steps. The constraint of building an EPG using only Map and Reduce operations has hindered the use of some performance optimizations, so Hadoop has recently been rewritten with an updated cluster runtime [20] that will give language frameworks more freedom in their choice of execution graphs. We expect a number of new execution engines to be developed to take advantage of this new Hadoop design, as well as other open-source multi-framework platforms such as Mesos [21], so it is an opportune time to consider what features are most useful to include in a distributed execution engine. We demonstrate in this paper that language-integrated dynamic EPG modification is a versatile addition to an execution framework that enables many optimizations and specializations.

2. Optimus

The goal of Optimus is to enable application-specific runtime rewrites of a distributed execution plan, based on statistics of the data computed during the program execution. In this section we describe the system architecture and some details of the main components.

2.1 Background: Dryad and DryadLINQ

We chose to build Optimus by extending Dryad (the distributed execution engine) and DryadLINQ (the query compiler), which we briefly review in this section. Dryad is a general-purpose execution engine for data-parallel dataflow computations. A Dryad job runs as a set of processes in a shared-nothing compute cluster. The job is coordinated by a root process called the job manager (JM) which maintains

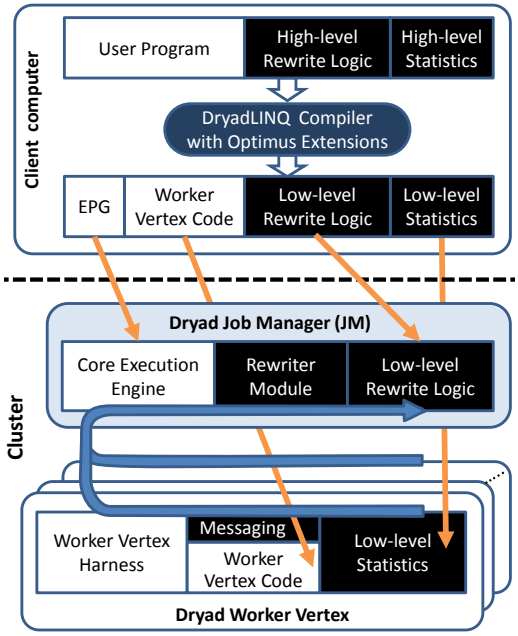


Figure 1. The Optimus system architecture.

the job's EPG and makes requests to the cluster resource manager to schedule worker processes. Each worker process executes a fragment of the distributed computation and communicates progress statistics back to the JM.

DryadLINQ is implemented as a layer on top of Dryad and provides a high-level abstraction of both computation and data. More specifically, DryadLINQ compiles a data-parallel user program, expressed as a LINQ query, into an EPG and a set of worker code fragments containing the application-specific code. The EPG and code fragments are submitted to the cluster for execution.

2.2 Optimus system architecture

While the EPG is the core data structure in Dryad, the semantics of the worker code in each graph node, and the data model in each graph edge, are transparent to Dryad. This design choice decouples the control plane and data plane, and makes Dryad a general execution engine that is independent of the languages and data models of upper layers. However, such a design leads to a semantic gap for dynamic rewriting—Dryad owns and can modify the EPG, but it is unaware of the semantics of code and data that are required for many useful runtime transformations.

Optimus bridges the semantic gap between Dryad's rewrite logic and a data-parallel program. When an Optimus-enabled data-parallel query is being executed in the cluster, Optimus collects data statistics at various vertices, creates a graph rewriting message, and sends this message to a graph rewriter in the control plane. Upon receiving the message, the graph rewriter rewrites the EPG based on the statistics contained in the message.

Optimus introduces several interacting components shown as black boxes in Figure 1:

- A graph rewriting module within the Dryad JM exposes primitives to query and modify the EPG.
- A messaging module within Dryad's worker vertex communicates application-specific statistics from the vertex to the JM, bridging the data and control planes.
- The DryadLINQ compiler is extended to allow applications to specify operator statistics that should be collected, and graph rewriting logic dependent on those statistics.

EPG notation. For the rest of the paper, an EPG may be presented in either expanded or unexpanded form. In an unexpanded EPG, each node is an operator to be applied to a dataset of N partitions. An edge connecting two nodes is either point-wise (1-to-1 connection) or cross-product (all-to-all connection). A node can be expanded into N identical vertices, applying the operator to each partition independently. The expanded EPG is the full expansion of every node in its unexpanded EPG.

2.2.1 Estimating and collecting data statistics

The major advantage of rewriting a graph at runtime is that there is accurate information available about the state of the execution. Graph rewriting often relies on collecting and estimating data statistics to optimize an EPG. An extra pass over a large dataset just to collect statistics could incur substantial overhead. DryadLINQ streams intermediate data through worker vertices, so we piggy-back statistics collection into the existing execution by pipelining application-specific statistics estimators into the processes containing the original vertices, eliminating any additional I/O overhead.

The collected statistics are aggregated and sent to the graph rewriter to perform data-dependent graph rewriting. In order to make the graph rewriter as lightweight as possible, and minimize the danger of overwhelming the JM with large amounts of state, Optimus performs as much computation as possible in the data plane. Statistics are aggregated outside of the graph rewriter by vertices inserted in the EPG. The output of this aggregation is a succinct message (often just a few bytes) that is then sent to the JM.

Optimus provides a system module (shown as "Low-level Statistics" in Figure 1) that implements collectors for a number of standard statistics using streaming-based algorithms, including data down-sampling, approximate histogram from downsampled data [11], count (cardinality), number of distinct keys [7], and frequent items [1, 12]. Algorithms proposed for centralized or streaming scenarios are modified for a distributed setting by collecting statistics at each vertex and then merging them into a single summary [1, 7, 12]. Optimus also allows users to define new statistics collectors to extend the standard module (see Section 2.4). User-defined statistics are shown as "High-level Statistics" in Figure 1.

2.2.2 Messaging module

The messaging module, shown as "Messaging" in Figure 1, is used to send information from vertices to the JM. Optimus uses two types of messages—vertex status messages and graph rewriting messages. Vertex status messages are part of the Dryad framework: periodically while each worker is running, and when it completes, the process sends a status message to the JM indicating the size of data read and written on each dataflow edge, execution timings, etc. These status updates are independent of the semantics of the vertex code or data model. Optimus provides an additional high-level messaging API that can be used by DryadLINQ and user applications to send graph rewriting messages in opaque data blobs from vertices to the JM. Such rewriting messages will only be interpreted and consumed by graph rewriters; Dryad simply ignores them.

2.2.3 Graph rewriting module

We implement a graph rewriting module (shown as "Rewriter Module" in Figure 1) to provide a set of simple primitives to query and modify the EPG. A dynamic graph rewriter uses these primitives to modify the EPG based on the rewriting messages it receives from the data plane.

The EPG data structure is shared by several modules in Dryad, including job management, task distribution, and fault tolerance. The graph rewriter and the other JM modules coordinate access to the EPG using a lock. Every time the Dryad JM receives a message from a worker process (e.g., a progress report), it locks the EPG and invokes a callback registered by Optimus.[1] Inside this callback, the Optimus rewriter has a comprehensive view of the job state, including statistics collection and rewriting messages, and is at liberty to make arbitrary changes to the EPG using the graph rewriting primitives. Together with application logic and operator semantics, the Optimus rewriter then decides whether and how to rewrite the EPG. Since status messages are sent while a vertex process is running, graph rewriters can detect problems such as data skew in a running vertex without waiting for the process to complete.

A vertex is in one of three distinct states (WAITING, RUNNING, or COMPLETED) when it is being modified by Optimus. For each vertex state, the condition determining whether the vertex can be involved in a rewrite is as follows:

- `WAITING`. The vertex has not produced its output, and is not running; it is waiting for inputs to be ready, or to be scheduled. All graph rewriting primitives can be applied to it.
- `RUNNING`. Optimus can kill the running vertex and hence put it into the `WAITING` state, discarding its partial results.

[1] The Optimus graph rewriter works on the EPG concurrently with other modules in the Dryad core—unaffected vertices of the current job proceed as normal while the lock is held by the rewriter.

- `COMPLETED`. Optimus can redirect the I/O of a completed vertex by disconnecting and reconnecting its I/O edges. Changing the input edges allows the vertex to use different inputs if it is re-executed due to a cluster fault (See Section 3.6).

2.2.4 DryadLINQ extensions

Optimus extends the DryadLINQ compiler to take high-level user implementations of statistics-gathering methods and emit code that, when executed on the cluster, calls into the Optimus APIs in the Dryad layer. These APIs then communicate statistics messages to the Dryad JM, where more emitted code rewrites the graph based on high-level user functions operating on the incoming messages. This emitted code is labeled "Low-level Rewrite Logic" and "Low-level Statistics" in Figure 1.

2.3 System-level graph rewriting

Many of the Optimus graph rewriters described in this paper are "system-level" transformations, implemented as extensions to the DryadLINQ compiler to provide performance optimizations to user code, while being transparent to users. These rewriters have the ability to statically insert additional statistics-collection vertices in the EPG, and make modifications to the EPG at runtime based on those statistics. The interface used for EPG modification is general, allowing arbitrary insertion and deletion of nodes and edges, and there is no checking that the resulting EPG is valid (e.g. cycle-free), so the extension writer must exercise care to ensure correctness. Optimus maintains the DAG property and operator semantics for the pre-defined operators in DryadLINQ.

By integrating with DryadLINQ, Optimus is able to implement a set of graph rewriters to perform dynamic optimizations for predefined LINQ operators and patterns such as Join and MapReduce, perform dynamic data partitioning, handle data/computation skews at runtime, and enable computations such as iteration that require data-dependent control flow (details of these rewriters are in Section 3).

2.4 High-level rewrite logic

In addition to the system-level rewriters which are hard-coded into the DryadLINQ compiler, and are thus transparent to application programmers, Optimus exposes a dynamic rewrite API to application-level programmers. This is most useful for user-defined operators and computation patterns where the program's semantics are known only to the user. In this case the user must have control over what statistics to collect and how the computation should be optimized accordingly.

To make the API simple to use and to avoid any potential abuses such as constructing illegal graphs, we have limited the interface to support only substitutions of LINQ subexpressions that have identical input and output dataset types. The user supplies several subexpressions, each of which implements the same sub-computation. Optimus compiles each subquery into a form compatible with the EPG, and inserts the optimal alternative for execution at runtime based on user-defined statistics and policies. By restricting users to the high-level API, we ensure that the resulting EPG is valid. We have found the interface expressive enough for complex applications such as specializing matrix computations.

Optimus defines a base type `GraphRewriter`, from which all user-defined graph rewriters must derive. `GraphRewriter` contains two APIs, one implemented by Optimus and the other for users to implement, as shown below (with simplified signatures):

```
public class GraphRewriter {
    // API for users to implement
    virtual ProcessMessage(message)=0;

    // API implemented by Optimus
    RegisterAlternatives(stats, default,
                         alternatives[]);
    Substitute(default_subquery, replacement);
};
```

The high-level dynamic rewriters are implemented as part of the user application logic. Optimus registers these rewriters into the dryad runtime. The advantage of doing so is:

- The runtime doesn't have to communicate with the user program for every rewriting decision, keeping overheads low.
- The (client-side) user program does not have to keep running during the entire cluster execution of the job.

We use matrix multiplication as an example to explain the above APIs:

```
public static MatrixMultiply(A, B) {
    var m1 = MatrixMultiply1(A, B); // algorithm 1
    var m2 = MatrixMultiply2(A, B); // algorithm 2

    var s1 = A.CollectStats();
    var s2 = B.CollectStats();
    var s  = CombineStats(s1, s2);

    var rewriter = new MatrixRewriter();
    return rewriter.RegisterAlternatives(s, m1,m2);
}
```

In the above code, the user supplies two alternative matrix multiplication algorithms `m1` and `m2`. The user also implements a `CollectStats` method, an arbitrary LINQ expression which is run on each input matrix and collects some desired statistics. The statistics for the two matrices are combined using another user-supplied method `CombineStats`, another LINQ expression but one that is constrained to return a single object rather than a collection; the object is typically the result of an aggregation.

`MatrixRewriter` is a user-defined rewriter derived from `GraphRewriter`. The call to `RegisterAlternatives` supplies Optimus with: the subquery `s` which computes the statistics;

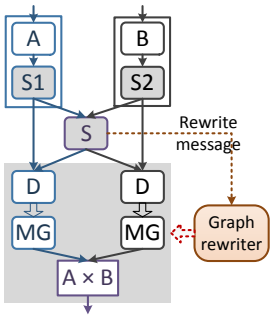


Figure 2. Unexpanded EPG for matrix multiplication. The edge between D and MG is a cross-product edge. The subgraph with a shaded background is rewritten at runtime.

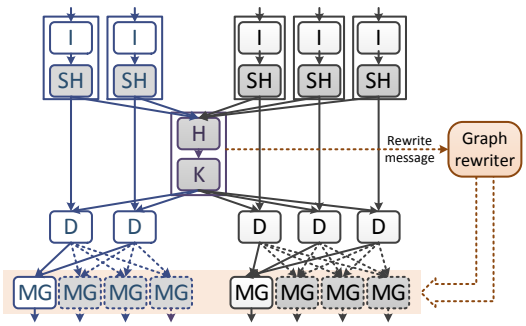


Figure 3. Runtime EPG for co-partitioning two datasets.

a default subquery `m1`; and a list of alternative subqueries, in this case the singleton `m2`. The Optimus-extended query compiler then: (1) constructs an EPG using the default algorithm `m1`; (2) records the alternative matrix multiplication algorithm; (3) adds appropriate statistics collecting/combining nodes (`S1`, `S2`, and `S`) to the EPG; and (4) appends code to the vertex where `S` will run, to send the serialized output of `S` as a graph rewriter message to the JM. At runtime, the MatrixRewriter's `ProcessMessage` method will be called with this serialized output, before `m1` is executed, and applies the user-defined policy to decide between the alternative implementations. If, given the runtime statistics, `m2` is determined to be a better implementation than `m1`, `ProcessMessage` will call the `Substitute` method to insert `m2` into the running EPG.

The resulting EPG with graph rewriter is shown in Figure 2. The subgraph of the default algorithm `m1` is shown in the greyed box; details of `A×B` and the alternative multiplication implementations are given in Section 3.5. When multiplying three or more matrices, the aggregated statistics from all matrices can be used to inform complex rewrites including multiplication order, i.e., `(A×B)×C` or `A×(B×C)`.

3. Graph Rewriters

This section presents several sample dynamic graph rewriters we have implemented to illustrate the range of transformations that can be supported by a runtime graph modification system such as Optimus.

3.1 Dynamic data partitioning

Data partitioning subdivides a large dataset into multiple parts that can be independently processed in parallel. For many operators there is a requirement that the partitioned data be grouped by key, so records with a common key are assigned to the same data partition. Examples include the data shuffling stage in a MapReduce program, and input pre-partitioning for Join.

There are two popular data partitioning methods: hash partitioning and range partitioning. It can be difficult to accurately set the partitioning parameters (number of partitions and/or range keys) when data statistics are not available, for example when partitioning is performed after several stages of processing that include user-defined functions [25, 42]. Matters become even more complicated when it is necessary to use a common parameter set to partition multiple data sets for multi-source operators such as Join. Sub-optimal parameters may lead to partitions being too fine or too coarse grained, as well as to data and computation skew where the data sizes or computational costs are not balanced among partitions of the resulting dataset(s).

The Optimus framework supports the determination of partitioning parameters, and the construction of the corresponding EPG, at runtime. In the following we introduce co-partitioning for two or more datasets. Partitioning one dataset is a special case. Figure 3 shows the runtime EPG for co-partitioning two data sets. (Extending the method to co-partition more than two data sets is straightforward.) We use co-range partitioning as an example to illustrate how Optimus can dynamically compute a set of range keys for partitioning two data sets. Such co-range partitioned data sets are subsequently used by multi-source operators, including Join, Union, and other set operators.

In Figure 3, there are two input datasets, the first one comprised of two partitions and the second one three partitions. The SH node down-samples the data set and computes $h(k)$, the histogram of the down-sampled data [11]. Here $h(k)$ is an approximation to the frequency of data records with key k. These histograms are merged into a final histogram by node H. The K node computes the number of partitions N and the range keys that will balance the final partitions. In this example, K estimates that the sets should be split into four partitions. It sends the number of partitions N to the graph rewriter module in the JM, which splits the MG node into four nodes for both data sets. The range keys from K are broadcast, via the data plane, to every distributor node D, which distributes each record to its target node MG according to the range keys. Each dataset is partitioned using the same range keys, so for example any key k assigned to the first part of the left-hand output will also be assigned

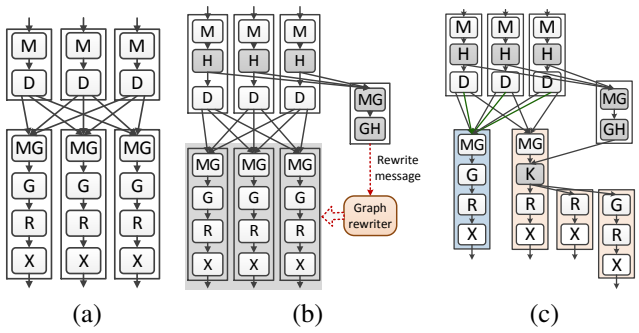


Figure 4. Expanded EPG of (a) Original MapReduce plan, (b) MapReduce with graph rewriter, and (c) Runtime EPG with popular keys isolated and small partitions combined.

to the first part of the right-hand output—a requirement for many multi-source operators.

Estimating range keys. Let $h_1(k)$ and $h_2(k)$ be the histograms of down-sampled data from two data sets. We can compute a composed histogram $h(k)$ as:

$$h(k) = h_1(k) \oplus h_2(k),$$

where $\oplus$ is a composition function. The choice of composition depends on the operation that will consume the co-partitioned data sets. Addition is a suitable choice if the intention is to balance the sizes of the partitions. For operators like Join, a product may be more appropriate. The final histogram $h(k)$ should estimate the overall cost for the consumer to process the data with key k. To produce range keys for N partitions, we divide the ranges of $h(k)$ equally into N parts. If the key type is not ordered we use a hash function to compute a 64-bit integer hash value of each key, and use the histogram to estimate ranges of hash values that will balance the downstream computation.

3.2 MapReduce

The compiler extension to optimize MapReduce patterns is shown in Figure 4. The original EPG contains Map vertices (M), Distribute vertices (D) to repartition the outputs of the Mappers to the correct downstream computer, Merge vertices (MG) to merge the downstream inputs, Group vertices (G) to group together inputs with the same key prior to reduction, Reduce vertices (R), and tail vertices (X) to pipeline local processing on the reducer outputs prior to the next stage of computation. Rectangular boxes indicate vertices that are pipelined into a single process.

After the Map stage of MapReduce, the data must be repartitioned by key before performing the reduction. A crucial parameter at this point is the number of partitions in the reduce stage, which decides the number of reducers and thus the degree of parallelism, and is usually statically set to a large number so that scheduler can balance the workload [13, 19]. However, when the volume of data output by mappers is small, it is better to use a smaller number of partitions in the reduce stage, lowering scheduling overhead and disk I/O cost [13, 34]). For non-decomposable reduce functions [39], the entire reduction must be performed after records have been re-partitioned. This can lead to severe data and computation skew among partitions when some keys are very common.

For the above reasons it is best to determine the re-partitioning parameters at runtime, when statistics are available. The Optimus MapReduce extension, shown in Figure 4(b), causes DryadLINQ to insert additional Statistics vertices (H) in a streaming pipeline between the Map and Distribute vertices, and an additional process containing a Merge vertex and a Global Statistics vertex (GH) to aggregate and summarize the global statistics. This summary is sent to the JM, where a graph rewriter modified the part of the EPG corresponding to the reduce phase in accordance with the collected statistics.

In DryadLINQ's MapReduce implementation a reducer does not start until all of its inputs have been produced. The graph rewrites occur after D has run, but before the Reduce phase (MG) has started. At this point, the total input size to every reducer is known. This information is used by Optimus to change the degree of parallelism of the reduce phase—reducers with small inputs are combined by rewiring the edges between D's and MG's. Figure 4(c) gives an example. The first two output partitions of D are small and hence are combined and processed by the first reducer.

Popular keys can lead to large groups, and thus severe data and computation skew among reducers. Figure 4(c) shows our solution to this problem. Optimus finds these popular keys without much overhead at runtime by pipelining a streaming algorithm to detect frequent items [1, 12] into the H node in the Map process, before data is shuffled. The GH node merges the per-partition detections to find the most common keys across the whole input data set. Optimus knows the mapping from keys to reducers, and can therefore ensure that no reducer partition contains too many common keys. In the case of a very popular key we assign it to a "private" partition containing no other keys. Isolating these large groups in their own vertices has two important benefits. First, when a partition contains only one key, the standard MapReduce grouping stage (`GroupBy`) can be skipped for that partition. Second, by breaking those otherwise extremely large partitions into smaller ones running on multiple machines, we reduce the data skew among vertices and increase the parallelism of the computation.

In the example shown in Figure 4(c), two popular keys are detected in the input to the second reducer. The GH node sends the popular keys to the key isolation node K of their corresponding partitions. Node K divides that partition into three sub-partitions, the first two each containing a single key and the third containing all the remaining records. Note that the GroupBy node (G) is eliminated for the two popular-key partitions.

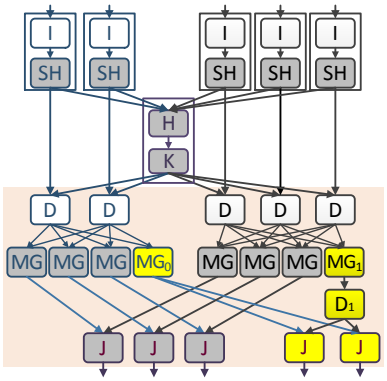


Figure 5. Hybrid Join with two runtime rewritings: (1) Two input data sets are co-range partitioned to prepare data for Join; (2) A skew co-partition MG_0 and MG_1, detected after co-partitioning, is split into multiple partitions. A broadcast Join is applied to this skew partition, while the other three Join vertices proceed using the original partition-wise Joins.

3.3 Join

Join combines records from two different datasets based on some common information. It is widely used and extensively studied in the database community [30]. A variety of algorithms exist for parallelizing Join:

- Partition-wise Join: the two input datasets are re-partitioned using the co-partition scheme discussed in Section 3.1. The Join is then broken up into smaller Joins to be executed in parallel, each of which joins a pair of corresponding partitions from the two input datasets.
- Broadcast Join: the smaller input dataset I is broadcast to every partition p_i of the larger dataset, and Joins are performed in parallel for every pair of (p_i, I).

The most suitable choice of Join depends on the statistics of the input data. Partition-wise Join is typically used when both datasets are large. If one of the input datasets is small a broadcast Join is preferred since broadcasting a small dataset and leaving a large set in place is more efficient than re-partitioning the large input set. As for other DryadLINQ operators, estimating the sizes of the two input datasets for Join is non-trivial at compile time. At runtime however the statistics of the input data are readily available. Optimus therefore dynamically selects an appropriate algorithm and execution plan for Join based on runtime data sizes.

Hybrid join. Since Join generates output for a key proportional to the product of the number of occurrences of the key in its two inputs, Join is particularly susceptible to skews in cases where a single key occurs with even moderate frequency in both inputs (c.f. [15, 27, 32]). Detecting all such cases is difficult using lightweight sampling methods [10], so to handle data and computation skews we adopt a *hybrid Join* algorithm. The two input datasets are first co-range partitioned by Optimus to prepare for partition-wise Join. Co-range partitioning approximately balances the data, however as observed above in some cases skew can be still present.

If a pair of corresponding partitions W_1 and W_2 (one from each input dataset) for a Join vertex is determined to contain skew, the Optimus rewriter is invoked to split this pair of partitions into multiple subparts. At this point, splitting W_1 and W_2 into finer co-partitions via range or hash partitioning may still lead to skewed work if they contain popular keys. To avoid this problem, we use round-robin to divide the larger partition, say W_2, into multiple parts, each of which in general contains a subset of the instances of each popular key, and then broadcast W_1 and perform broadcast Join on the skewed partitions, while other Join vertices proceed as usual using partition-wise Joins. There is a tradeoff between increasing network traffic by broadcasting a partition and balancing computation, however the broadcast is localized to only the sub-partitions that contain popular keys.

Figure 5 illustrates a hybrid Join using dynamic graph rewriting. The graph rewriter setup is similar to Figure 3 and is omitted here. Two runtime graph rewritings happen here. The first is a dynamic co-range partitioning to prepare the data for Join. The second rewriting happens after the data distribution phase D, but before Join vertices are started. In this example, Optimus detects data skew in the fourth co-partition MG_0 and MG_1, and it inserts Node D_1 to split (using round-robin) MG_1 into two sub-partitions, and replace the fourth Join vertex with a broadcast Join consisting of two Join vertices.

3.4 Iterative computation

Many data analysis applications require iteration. Examples include graph computations such as PageRank and connected components, and machine learning algorithms such as k-means clustering. In these examples the application specifies a stopping condition which the system can only check at runtime. A common solution is to choose a fixed number of iterations at plan compilation time and construct an EPG for this sub-job [40]. When the sub-job finishes, the client computes and checks the stopping condition, and submits another job if not. The general Optimus mechanism enables iterated execution of an arbitrary sub-computation within a *single* job, which simplifies job monitoring and fault-tolerance, and reduces job submission overhead. The stopping condition is computed by a vertex and sent to the graph rewriter, and if the condition is not met the rewriter dynamically extends the EPG with another copy of the loop body subgraph, and performs another step of iteration.

Optimus inherits from Dryad the DAG property for the EPG. A DAG-based design is particularly suitable for data-intensive applications and is adopted by many data-parallel systems. Its advantages include simple job scheduling/management, and a simple yet effective fault-tolerance model (an important factor in distributed data-parallel computing). We added a `While` operator to DryadLINQ to conveniently express iterative computations:

```
public While(source, initial, step, condition);
```

Here `step` is the expression that performs one iterative step, and `condition` is the expression, computing a Boolean result, that evaluates the stopping condition. For k-means, for example, `step` computes new cluster centers from the input and the previous centers, and `condition` takes the centers from current step and previous step to determine the stopping condition.

Figure 6 shows the EPG for k-means. The highlighted region is the loop body subgraph, which has three external input edges and two external output edges. To perform a new iteration, a new subgraph is constructed at runtime, where the output from previous iteration becomes the input of current node `A` and `C`. The input from node `In` is reused since the input data does not change between the iterations, and Dryad's ability to schedule tasks based on data affinity avoids transferring the invariant input data across the network at each iteration.

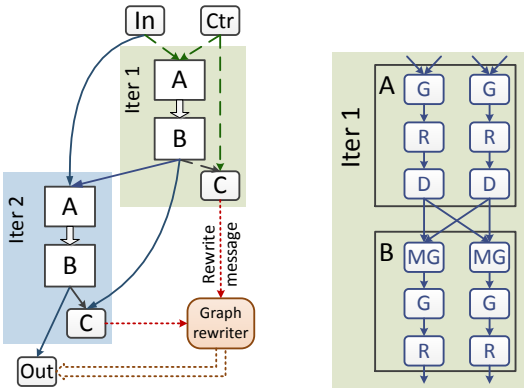


Figure 6. Iterative computation. (a): Unexpanded EPG for k-means algorithm (`PageRank` has a similar EPG), showing two iterations. Node `C` checks for termination. (b): Detailed expanded subgraph for pipelined nodes `A` and `B` in (a). `A` assigns each data point to its nearest center and recomputes centers locally for each partition. `B` merges and groups all centers to compute new centers.

3.5 Matrix multiplication

Due to the specialized nature of distributed matrix computation optimizations, and the wide applicability of linear algebra, customized systems such as MadLINQ [36] have been built dedicated to distributed matrix computation. In this subsection, using matrix multiplication as an example, we show that with dynamic graph rewriting it is possible to build an efficient matrix computation engine on top of an existing general distributed execution engine such as Dryad, and thereby integrate matrix computation with general-purpose DryadLINQ computations within an application.

In addition to choosing between execution plans, opportunities for special-purpose matrix optimization arise because the appropriate data representation can be chosen at runtime. The data models for sparse and dense matrices are different and for efficiency it is important to select the correct data model to match the density of each matrix in the computation. The appropriate implementation for parallel matrix multiplication depends on the matrix dimensions, input and output representation (data model), and sparsity (data statistics). Input matrices can be intermediate results from previous stages, when matrix computation is part of a general computation, or when there is a chain of matrix operations. This indicates an opportunity to choose the data model and the implementation dynamically at run time.

Suppose we want to compute the product of two large matrices $\mathtt{P} = \mathtt{U} \times \mathtt{V}$. We can use co-range partitioning to divide both matrices into smaller sub-matrices to achieve data-parallelism. The most important criteria for the partitioning are balance between computations, and the requirement that the partitions be small enough that the output of the sub-computation will fit in the memory of a single computer. Suppose based on data statistics the co-partitioner decides to divide each input matrix into 4 partitions. The following are the three most common partitioning schemes:

$$\begin{bmatrix} \mathtt{A} & \mathtt{B} \\ \mathtt{C} & \mathtt{D} \end{bmatrix} \begin{bmatrix} \mathtt{E} & \mathtt{F} \\ \mathtt{G} & \mathtt{H} \end{bmatrix} = \begin{bmatrix} \mathtt{AE}+\mathtt{BG} & \mathtt{AF}+\mathtt{BH} \\ \mathtt{CE}+\mathtt{DF} & \mathtt{CG}+\mathtt{DH} \end{bmatrix} \quad \text{(a)}$$

$$\begin{bmatrix} \mathtt{A} & \mathtt{B} & \mathtt{C} & \mathtt{D} \end{bmatrix} \begin{bmatrix} \mathtt{E} \\ \mathtt{F} \\ \mathtt{G} \\ \mathtt{H} \end{bmatrix} = \begin{bmatrix} \mathtt{AE}+\mathtt{BF}+\mathtt{CG}+\mathtt{DH} \end{bmatrix} \quad \text{(b)}$$

$$\begin{bmatrix} \mathtt{A} \\ \mathtt{B} \\ \mathtt{C} \\ \mathtt{D} \end{bmatrix} \begin{bmatrix} \mathtt{E} & \mathtt{F} & \mathtt{G} & \mathtt{H} \end{bmatrix} = \begin{bmatrix} \mathtt{AE} & \mathtt{AF} & \mathtt{AG} & \mathtt{AH} \\ \mathtt{BE} & \mathtt{BF} & \mathtt{BG} & \mathtt{BH} \\ \mathtt{CE} & \mathtt{CF} & \mathtt{CG} & \mathtt{CH} \\ \mathtt{DE} & \mathtt{DF} & \mathtt{DG} & \mathtt{DH} \end{bmatrix} \quad \text{(c)}$$

Each of these schemes corresponds to a different distributed execution plan: Figures 7(a), (b), and (c) show the EPGs for Equations a, b, and c, respectively.

The different execution plans have different trade-offs between network traffic, intermediate result sizes, and the number of stages/vertices. Case (b) generates the least traffic, but the matrix dimensions of the intermediate result at each vertex are as large as those of the final output matrix. (c) has the most traffic, but the matrix computed at each vertex is only one sixteenth the size of the final output matrix. (a) strikes a balance between (b) and (c), with half of the traffic of (c), and a matrix size at each output one quarter that of (b). Case (d) is a special case of (c). When the data size of `V` is small enough (e.g. `V` is a sparse matrix) we can broadcast the whole matrix to each partition in `U` without subdividing `V`.

Case (b) is suitable when the data size of the output matrix `P` is small, either because the dimension of `P` is small or because `P` is sufficiently sparse. (c) is good when `P` is dense and the inputs are sparse, so that network traffic is not overwhelming. When both inputs and outputs are dense and

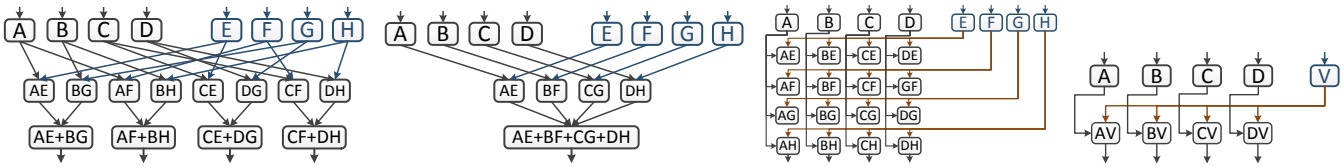


Figure 7. Matrix multiplication: expanded subgraph for different matrix partitioning schemes. Each can substitute node `A×B` in Figure 2.

large, (a) should be used. If one of the inputs is small (low-dimensional or very sparse) then (d) is the most efficient. Matrix data sizes are often best inferred at runtime, at which point Optimus can generate the appropriate execution plan.

A matrix can be represented by a stream of triplets $\{i, j, \mathtt{v}\}$. Here `v` could be a single value (for a sparse matrix) or a rectangular sub-matrix tile. In real applications large matrices are often sparse, and the sparse representation is a good choice. If the result of the multiplication of two sparse matrices is a dense matrix we should switch to use the dense data representation. In other words, we *dynamically* change `v` from a single value to a sub-matrix.

We have built a full matrix algebra package using Optimus that allows users to specify matrices using a high-level Matrix type, and dynamically infers the appropriate execution plan and data model for each operation.

3.6 Reliability enhancer

In data-parallel compute frameworks such as Dryad the intermediate data of a job are not replicated, since replication would generate unnecessary I/O and the results can be recomputed on failure by backtracking in the EPG. In some cases, however, it is desirable to protect a subset of important intermediate data by replication, for example the results from compute-intensive vertices that would take a long time to regenerate, and those that would cause a chain of vertices to re-run if lost, as explained below.

In typical data-intensive compute frameworks each cluster node provides both computing cycles and storage, and the cluster scheduler attempts to schedule a worker process close to its input data. Such data locality is a key advantage in MapReduce and Dryad, as it significantly reduces network traffic. However, such a scheduling strategy can lead a chain of vertices in the EPG to be scheduled on a single computer, as each downstream vertex is scheduled on the computer that generated its input. We denote such a chain of vertices a *critical chain*. The existence of long critical chains makes fault tolerance less effective—if the computer they all ran on fails, the whole chain of vertices has to be re-executed to reproduce the result [22, 26, 41], leading to a large delay. In a long-running iterative computation, it is common to encounter critical chains.

While one could change the scheduler to avoid critical chains, cluster schedulers are typically ignorant of the semantics and EPGs of the jobs that are running on the cluster, in order to simplify their design and preserve the generality of the cluster computing framework. We therefore implemented an Optimus extension to detect and mitigate long critical chains. This can be done at runtime by periodically walking the EPG graph of completed vertices (efficiently, using dynamic programming): since the running time of each completed vertex is known it is possible to accurately estimate the amount of work that would be required to regenerate the lost data if any given computer were to fail. If any computer is found to host a critical chain, the outputs at the tail of that chain are copied to another computer by extending the EPG as shown in Figure 8. Here the output of vertex `A` is to be protected, and the necessary replication subgraph is inserted, including a copy vertex `C` and an "Or" vertex `O` which is able to choose any one of its available inputs in the case of re-run due to failure.

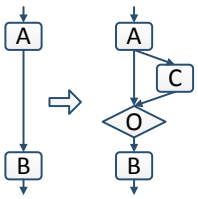


Figure 8. Reliability enhancer: replication subgraph to protect vertex `A`'s output.

Critical chains are not generated by MapReduce jobs because *all* data are replicated in the file system after each Reduce step. Replicating only the output of critical chains can lead to a substantial reduction in disk and network I/O, especially in the common case where workers in a complex multi-stage job are each fairly quick to re-execute, so critical chains may grow to a substantial depth before needing to be protected.

In Mantri [3] the JM replicated the outputs of tasks deemed important based on a cost-based analysis. The reliability enhancer in Optimus detects and breaks critical chains by inserting replication subgraphs, which avoids copying all outputs of a stage or task and enables optimizations such as choosing the best place along the chain to insert the replication subgraph. The replicating tasks are delegated to copy vertices, which run in parallel in the cluster and minimize impact on the JM.

4. Evaluation

We evaluate Optimus using several real-world applications and datasets. The results show that by rewriting the EPG at runtime, Optimus substantially improves the performance of these applications over their existing implementations. It also simplifies the implementations since many of them contain manual optimizations that are either unnecessary or automated by Optimus.

Unless otherwise noted, the evaluations were performed on a cluster of 61 computers running Microsoft Windows Server 2008 64-bit operating system and connected by 1Gb Ethernet switches. Each computer had two Quad-core AMD Opteron 2373 EE CPU with a clock speed of 2.1 GHz, 32 GB DDR2 RAM, and four 1.0 TB SATA disks. The running times for all the measurements reported in this section were averaged over 10 runs, and the standard deviations were less than 3% of the average.

4.1 MapReduce for online-service security

In this experiment, we evaluate the effectiveness of Optimus using a typical MapReduce application that has severe data and computation skew. The application computes IP properties using anonymized user-login events from Hotmail [38]. The input data size is 278 GB compressed, stored in 1000 partitions. It contains 26 billion login events from 314 million IP addresses. The task is to group these login events by IP address, then apply a function to each group to compute certain properties for the IP addresses in the group.

The data is heavily skewed in IP group sizes. On average each IP group has about 83 login events, but the top 10 IP groups contain 3.75 billion events. As a result, there were six vertices in the reduce stage that took more than 8 hours to complete while the rest of the vertices completed in less than 60 minutes. Using the techniques described in Section 3.2, Optimus identified 12 large groups, processed each of them in a separate vertex, and completed the job in less than 150 minutes.

Graph rewriting overheads. We measured the overheads introduced by Optimus in all the experiments. This experiment incurred by far the largest overhead so we report it here. The overhead of Optimus mainly consists of runtime statistics collection and communication between the vertices and the graph rewriter. The statistics collection was pipelined in-process with data processing logic and processed 2-3 million records per second, including the detection of high-frequency keys. So its impact on the overall job performance was negligible. Merging and aggregating the statistics (160 KB compressed in total) from 1000 partitions and sending the resulting rewriting message (120 bytes compressed) to the graph rewriter took less than 20 seconds, which was also negligible compared to the total job running time (150 minutes). The total time spent on rewriting the graph (constructing the new subgraph and replacing the reduce pipelines for large IP groups) was less than a second.

4.2 Product-offer matching using Join

A key computation in a commerce search engine is to match unstructured offers from thousands of merchants to a predefined structured product catalog containing millions of product items [24]. This product-offer matching can be easily expressed by the `GroupJoin` operator:

```
offers.GroupJoin(products,
                 o=>o.category,
                 p=>p.category,
                 (o, c) => Match(o, c));
```

Here `GroupJoin` combines records of same category from `offers` and `products`, and groups the joined results based on product category. `Match(o,c)` is a matching function that takes an offer `o` and the grouped products of the matching category `c` and returns the matching products. This computation is challenging when some categories contain a large number of products, due to data and computational skew.

We use this computation to evaluate the hybrid join technique presented in Section 3.3. We compared Optimus with three existing implementations, using real datasets from a popular search engine.

- Baseline: standard partition-wise `GroupJoin`.
- Broadcast-Join: the offers are round-robin partitioned and the products are broadcast to each offer partition to perform `GroupJoin`.
- CoGroup [16, 33]: the product and offer datasets are merged into one stream of co-groups of common categories, and matching is performed within each co-group. A co-group `(O,C)` with computational cost higher than a certain threshold is further split into k sub-groups $(O_1,C),\ldots,(O_k,C)$ by splitting `O` and duplicating `C`. The final co-group stream is round-robin partitioned into N parts for a more balanced computation.

Figure 9(a) shows the job completion times for a dataset with 4 million offers and 5 million products. The Baseline approach performed worst as it could not handle skew properly—its job running time was dominated by a few partitions containing popular categories. When the data were hash-partitioned into 60 parts, CoGroup and Broadcast-Join have similar performance—both handled the skew well and were significantly better than Baseline, and Broadcast-Join was about 10% better than CoGroup. Optimus further reduced the running time by about 30%.

Increasing the number of partitions only slightly improved the performance for Baseline and CoGroup. For Baseline the job completion times were dominated by a few popular categories, which could not be split by further partitioning on the hash of category keys. CoGroup achieved its best performance using 180 partitions. Optimus adaptively split a skewed data partition and used the broadcast join for those split partitions, thus it was less sensitive to the number of input data partitions. Broadcast-Join performed worse

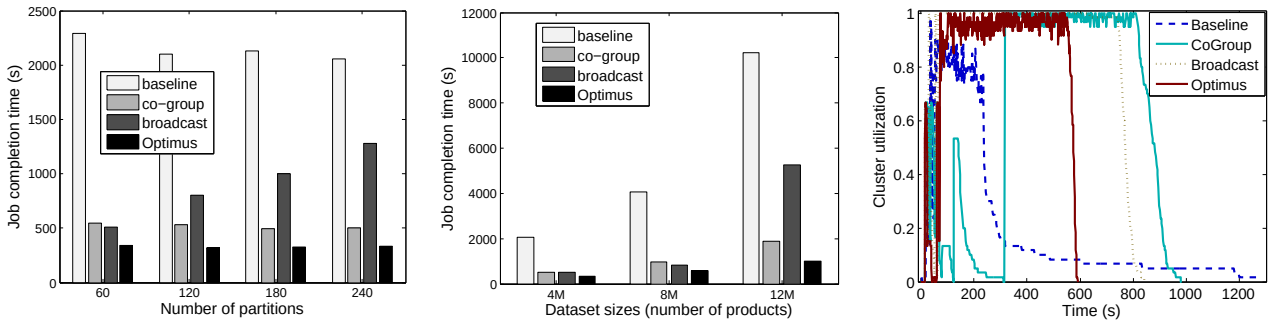


Figure 9. Performance comparisons. (a): Job completion time for a dataset of 4 million offers and 5 million products; (b) Job completion times for different dataset sizes; (c) Cluster utilizations for the 8M dataset in (b).

when the number of partitions was larger than the number of computers. The reason is that the whole product dataset was broadcast to every vertex in the GroupJoin stage. Increasing the number of vertices led to a larger network I/O overhead.

To evaluate scalability we measured performance using different dataset sizes. Fig. 9(b) shows the total running time for each approach. The 8M dataset for example means that it contains 8 million offers and 8 million products. Optimus clearly has the best scaling performance.

To further understand the performance differences, we analyzed both cluster and CPU utilization. The cluster utilization is the fraction of computers utilized by the job over time. Figure 9(c) shows the cluster utilization for the 8M dataset. Baseline had poor cluster utilization due to data and computation skew. CoGroup required several rounds of global data shuffling, and the final partitioning stage, to distribute the co-groups, introduced another global barrier (at $t \approx 300$), waiting for a small number of machines to distribute sub-groups of popular categories. In contrast, except for data co-partitioning required by `GroupJoin` (at $t \approx 50$), all other data splitting in Optimus was localized to vertices containing popular categories and did not introduce any new global barriers.

Broadcast-Join had the highest cluster utilization in the final matching stage. However, as Table 1 shows, its aggregated CPU utilization for those vertices in the matching stage was only 55%, due to threads blocked on network I/O during broadcast. Optimus's CPU utilization was about 10% better than CoGroup. The difference comes from the fact that the work items assigned to the threads in a vertex process were different. In CoGroup a work item was a co-group while in Optimus it was a single offer item, which was smaller (in terms of size and variance) than a co-group, leading to better thread-level parallelism within a vertex process.

4.3 Matrix multiplication

We now evaluate the performance of our matrix computation engine described in Section 3.5. The application is collaborative filtering for movie recommendation. In order to compare with MadLINQ [36], we use the same baseline algorithm as MadLINQ running on a comparable 48-computer cluster[2]. The input is a dataset from the Netflix challenge [6], also the same one used by MadLINQ. It is a $20K \times 500K$ matrix `R` with a sparsity of 1.19%, where a non-zero element `R(i, j)` represents the rating for movie i by user j.

Baseline	CoGroup	Broadcast	Optimus
0.82	0.72	0.55	0.81

Table 1. Aggregated CPU utilization of vertices in matching stage.

The predicted ratings on all movies from all users can be computed by $\mathtt{C} = \mathtt{R} \times \mathtt{R}^\top \times \mathtt{R}$, which has two multiplications: $\mathtt{A} = \mathtt{R} \times \mathtt{R}^\top$ and $\mathtt{C} = \mathtt{A} \times \mathtt{R}$. Optimus automatically chose the plan (d) in Figure 7 for both multiplications because `R` was sparse and small in size. Optimus also automatically switched the data model to use a dense representation for `A`, as `A` is a dense matrix. We denote this execution plan as S-D-D (Sparse-Dense-Dense for the data models of the matrices `R`, `A`, and `C`). For comparison purpose, we also include the results of plan (d) using only the sparse data model (denoted by S-S-S).

Table 2 shows the job completion times using Mahout [28] on Hadoop, MadLINQ, S-S-S, and S-D-D. The times shown for Mahout and MadLINQ are from the MadLINQ paper. Optimus substantially outperformed both MadLINQ and Mahout. The poor performance of Mahout is because its (static) execution plan generates a full-size dense intermediate matrix at each vertex and is not able to adapt to the change of matrix densities. As a result, one stage of the computation generates very large intermediate data (about 20 TB) that could be avoided given a better EPG. Of the Optimus variants, the automatically-chosen S-D-D performed better than S-S-S, mostly because the dense data model in S-D-D results in significantly less intermediate data.

[2] Each computer in this cluster has two quad-core CPUs with a clock speed of 2.1GHZ, 16GB RAM, and four 1TB disks.

	Mahout	MadLINQ	Optimus	
			S-S-S	S-D-D
$A = R \times R^\top$	630	347	255	254
$C = A \times R$	780m	570	254	170

Table 2. Job completion time (in seconds, except for 780m in minutes) for the two matrix multiplications in $C = R \times R^\top \times R$.

Runtime rewriting policies can capture sophisticated domain-specific knowledge. Optimus, by providing a programmable interface for runtime rewriting, lets us extend a general distributed execution platform with domain-specific execution plans, and expose them as high-level user libraries. We have validated this approach by integrating a high performance matrix library with DryadLINQ, purely using the API exposed by Optimus.

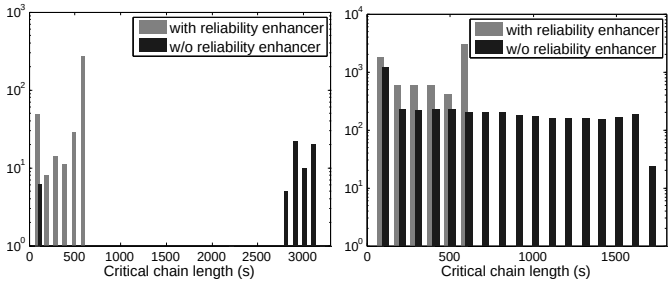
4.4 Reliability Enhancer

This experiment uses the iterative PageRank computation (Section 3.4) to evaluate the effectiveness of the reliability enhancer to detect and mitigate critical chains. The input dataset is a collection of about 120M web pages that was pre-partitioned into 60 partitions based on the page URLs. As expected, due to data locality many long critical chains were formed during the execution. This means that any computer failure would very likely cause the whole computation to be back-tracked to the very beginning and then re-executed from there.

Our reliability enhancer identified all the critical chains in this experiment, and split each chain by inserting the data-protection subgraph when the chain length (in terms of accumulated running time) was greater than a threshold θ, set here to 10 minutes. Figure 10(a) shows the histograms of the length (in seconds) of all critical chains for two runs, one with the reliability enhancer and the other without. The reliability enhancer effectively broke all critical chains into chains less than 10 minutes.

The formation of long critical chains could be affected by the number of partitions used by the computation. Figure 10(b) shows the critical chains when the input was re-partitioned into 120 partitions. A large number of long critical chains were still formed during the execution, but the reliability enhancer was again able to detect and break them to be shorter than 10 minutes.

The reliability enhancer incurred little overhead—in the example in Figure 10(a), 336 additional copy vertices were created, copying about 4.1 GB of data in total. On average each copy vertex took 1.75 seconds to copy about 12 MB bytes of data. Since the replication subgraph can be inserted anywhere on the critical chain, as a heuristic we back-track 20% (in seconds) of the chain to choose the vertex with smallest output data size to copy. Multiple copy vertices were usually started in batch and run in parallel. The impact on job completion time is negligible: the reliability enhancer



(a) (b)

Figure 10. Histograms of critical chains with and without reliability enhancer from running PageRank on web pages in (a) 60 partitions and (b) 120 partitions.

incurred on average around 1% overhead, adding 66 seconds to the total running time of 101.4 minutes.

We simulated a computer failure by removing one computer from the cluster in the middle of the computation. Only the vertex chain between the "failure" point to the last protection point was re-executed. This significantly improves the recovery time in the event of machine failures.

5. Related Work

Dryad [22] provides basic graph rewriting mechanisms for runtime optimization, including local data aggregation and dynamic data partitioning, but it has several limitations. First, it is not extensible: the user has to modify the Dryad core to implement a rewriter. Second, Dryad doesn't know the semantics of the worker code and data model. The optimizations we report here (with the exception of the reliability enhancer) require semantic information, such as key statistics, cardinality, or loop convergence, that can only be inferred by integrating the language layer with the graph rewriter and thus cannot be implemented on top of an unmodified Dryad system. Optimus re-architected Dryad and DryadLINQ with a design that addressed both problems. In particular, the user-level rewriting API allows runtime optimization for user-defined operators. Most of the optimizations enabled by Optimus are either new by themselves or new in the context of data-parallel computing.

CIEL [31] takes a different approach to Dryad to represent the EPG: instead of holding it in a central job manager, CIEL programs dynamically build the computation graph using a custom scripting language that executes at each worker and optionally expands its output graph based on its local computation. CIEL is better suited to techniques such as recursion and functional programming where it is natural to dynamically subdivide a computation as it proceeds. We believe Optimus enables a more natural way for programmers to express declarative computations, such as expression trees of LINQ operators, where the modifications to the graph are "global" rewrites of the expression. Furthermore, Optimus enables optimizations not possible in the cur-

rent CIEL system: asynchronous changes to the graph (e.g., canceling an existing task and replacing it with a subgraph), and non-local graph changes (e.g., the reliability enhancer of Section 3.6). On the other hand an Optimus-like component could be added to CIEL to support dynamic optimizations like those reported in this paper; the concepts of Optimus are general and not limited only to Dryad/DryadLINQ.

FlumeJava [9] transforms its high-level parallel operators into MapReduces. The key optimizations described in the FlumeJava paper are all static, compile-time optimizations rather than the dynamic optimizations provided by Optimus.

HaLoop [8] specializes Hadoop for iterative MapReduce computations. Piccolo [35] and Spark [41] use persistent in-memory datasets to efficiently execute iterative steps. These systems do not support general dynamic graph rewriting.

Adaptive query optimization for a single database server has been studied in the database community [4, 5, 14, 17, 23, 29]. Compared to Eddies [4] that does finer-grained (record-level) optimization in a single server environment, Optimus does coarser-grained (vertex-level) optimization in a distributed data-parallel compute cluster. Combining the two is an interesting direction for future work. In the data-parallel setting, Nectar [18] caches the results of subqueries to avoid recurring computations; and RoPE [2] collects statistics of previously-executed queries to generate better execution plans for new jobs using the same query. Optimus complements these previous approaches by providing a general framework to rewrite arbitrary subgraphs in the EPG (including already-executed subgraphs) to optimize previously-unseen queries at runtime. It also supplies system-level and user-level interfaces to specify new runtime optimization policies. RoPE can make limited changes to the un-executed parts of the plan, such as altering the degree of parallelism of non-reduce stages, but cannot switch between alternative plans at runtime using observed statistics. It also composes statistics using the control plane, which could overwhelm the JM in the case of a complex composition, whereas Optimus exposes statistics collection as an extension of the data-plane and only sends a small number of succinct messages to the JM. Optimus provides opportunities to translate query optimization techniques designed for single database servers to a distributed setting. For example, A-Greedy (c.f. [14]) can be adapted to re-order operators inside a pipelined vertex. However, re-ordering multi-way Join is tricky, as it may require a dataset to be partitioned in multiple ways.

Sampling approaches [25, 33] downsample the inputs to a job, and first run the job on those samples to gather statistics to optimize the EPG before running on the full data. Besides adding the overhead of running an additional job, this requires an additional pass over the entire input dataset. Moreover, sampling does not work well to estimate statistics for operators at later stages of large EPGs, since the input-stage sampling is no longer representative once it has passed through a complex dataflow. Finally, some of the rewrites described here can not be performed by sampling approaches; these include the reliability enhancer, iterative computation, and the second rewrite to handle task-level skews in the Join example.

6. Conclusion

Optimus provides a flexible framework to modify a distributed execution plan at runtime. The main improvement over previous mechanisms is the ability to integrate statistics collection over application data, programmed in a high-level language, with a rich application-defined rewrite logic. We demonstrated the wide utility of dynamic EPG rewriting by evaluating a rich set of applications and showing in each case a substantial performance benefit compared to a statically generated plan. We believe Optimus is a versatile addition to a data-parallel execution framework to enable a variety of runtime optimizations and specializations that are hard or impossible to achieve in existing systems.

Acknowledgements

We would like to thank Chandu Thekkath, Derek Murray, and Mihai Budiu, as well as the reviewers of our paper, for their many helpful comments and constructive feedback.

References

[1] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi. Mergeable summaries. In *PODS*, 2012.

[2] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing data parallel computing. In *NSDI '12*, April 2012.

[3] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in mapreduce clusters using mantri. In *OSDI'10*.

[4] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *ACM SIGMOD*, 2000.

[5] S. Babu, P. Bizarro, and D. DeWitt. Proactive re-optimization with Rio. In *ACM SIGMOD*, 2005.

[6] J. Bennett and S. Lanning. The Netflix prize. In *ACM SIGKDD 2007*.

[7] K. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On synopses for distinct-value estimation under multiset operations. In *SIGMOD 2007*.

[8] Y. Bu, B. Howe, M. Balazinska, and M. Ernst. Haloop: Efficient iterative data processing on large clusters. In *VLDB*, 2010.

[9] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI'10*.

[10] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *SIGMOD 1999*.

[11] S. Chaudhuri, R. Motwani, and V. R. Narasayya. Random sampling for histogram construction: How much is enough? In *SIGMOD 1998*.

[12] G. Cormode and M. Hadjieleftheriou. Finding frequent items in data streams. In *VLDB 2008*.

[13] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[14] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[15] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *VLDB 1992*.

[16] E. Gonina, A. Kannan, J. Shafer, and M. Budiu. Parallelizing large-scale data processing applications with data skew: a case study in product-offer matching. In *Intl. Workshop on MapReduce and its Applications (MAPREDUCE)*, 2011.

[17] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[18] P. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic Management of Data and Computation in Datacenters. In *OSDI'10*.

[19] The Hadoop project. `http://hadoop.apache.org/`.

[20] Hadoop NextGen MapReduce (YARN). `http://hadoop.apache.org/common/docs/r0.23.0/hadoop-yarn/hadoop-yarn-site/YARN.html`.

[21] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI '11*, March 2011.

[22] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys 2007*.

[23] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *ACM SIGMOD*, 1998.

[24] A. Kannan, I. E. Givoni, R. Agrawal, and A. Fuxman. Matching unstructured product offers to structured product specifications. In *ACM SIGKDD 2011*.

[25] Q. Ke, V. Prabhakaran, Y. Xie, Y. Yu, J. Wu, and J. Yang. Optimizing data partitioning for data-parallel computing. In *HotOS '11*, 2011.

[26] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. On availability of intermediate data in cloud computations. In *HotOS '09*, 2009.

[27] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. SkewTune: Mitigating skew in MapReduce applications. In *ACM SIGMOD*, 2012.

[28] Mahout project. `http://mahout.apache.org/`.

[29] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *ACM SIGMOD*, pages 659–670, 2004.

[30] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Comput. Surv.*, 24(1):63–113, 1992.

[31] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: a universal execution engine for distributed data-flow computing. In *NSDI 2011*.

[32] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *ACM SIGMOD*, pages 949–960, 2011.

[33] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD 2008*.

[34] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *ACM SIGMOD*, 2009.

[35] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, 2010.

[36] Z. Qian, X. Chen, N. Kang, M. Chen, Y. Yu, T. Moscibroda, and Z. Zhang. MadLINQ: Large-scale distributed matrix computation for the cloud. In *EuroSys 2012*.

[37] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a Map-Reduce framework. *Proc. VLDB Endow.*, 2(2), 2009.

[38] Y. Xie, F. Yu, and M. Abadi. De-anonymizing the internet using unreliable IDs. In *SIGCOMM 2009*, August 2009.

[39] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP*, 2009.

[40] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[41] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI 2012*.

[42] J. Zhang, R. C. Hucheng Zhou, X. Fan, Z. Guo, H. Lin, J. Y.Li, W. Lin, J. Zhou, and L. Zhou. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *NSDI*, 2012.