In the previous chapters, we have introduced the DISPEL (data-intensive systems process engineering language) language and presented an environment for developing DISPEL scripts and components. This chapter describes the processing of DISPEL requests. We call the overall process DISPEL enactment. We first present an overview of the four stages of the DISPEL enactment process and further describe each stage in the subsequent sections. This chapter is targeted at the data-intensive engineers who work on the implementation of the data-intensive platforms.

12.1 OVERVIEW OF DISPEL ENACTMENT

Recall from Chapter 9 that there are four stages in the enactment process of data-intensive computations. Any implementation of a data-intensive platform will probably implement these in some form of the following:
1. **DISPEL Language Processing** includes parsing and validating a DISPEL scripts and creating its corresponding dataflow graph—usually annotated with the information deduced during this phase.

2. **Optimization** includes selection of PEs (processing elements), transformation of the dataflow graph, substitution of PEs, identification of available resources, and the mapping of PEs to resources.

3. **Deployment** includes compiling the graphical representation into platform-specific executable graphs and setting up resource and dataflow connections.

4. **Execution and Control** includes instrumentation and performance measurement, failure management, results delivery, termination, and clean up.

These phases are shown in Figure 12.1. In this chapter, we outline these processes in detail. Section 12.2 deals with DISPEL language processing. Section 12.3 outlines the optimization process, Section 12.4 addresses deployment, and Section 12.5 outlines the execution and control phase.

This enactment framework provides a high level abstraction of data-intensive applications. This is achieved through a *separation of concerns*, where the software details are abstracted at various levels, for example, the application level, algorithmic level, and execution level. This framework uses a data-streaming execution model, abstracted using the DISPEL language for the higher-level expression of dataflow graphs. Using this abstraction, the framework validates and optimizes the execution of data-intensive applications. Every component in the framework uses this language to communicate data-related processes and their interrelationships. For instance, the enactment gateway, which is the interface for submitting data-intensive jobs, communicates with the registry to retrieve reusable components that the registry supplies in the form of valid DISPEL scripts and implementations of PEs.

Recall that DISPEL is an abbreviation for the *Data-Intensive Systems Process Engineering Language*. It is a scripting language that is processed by a DISPEL parser to generate dataflow graphs in the form of executable workflows. The primary function of DISPEL is to express how a data-intensive application uses PEs (for instance, that provide noise filtering algorithms) and how these elements communicate with each other. In other words, DISPEL is a language for expressing a directed graph, where PEs represent the computational nodes and the flow of data between PEs is represented by connections. Thus, DISPEL provides an abstraction technique for a data-streaming execution model. At the lower level, DISPEL also handles validation and provides the required model for carrying out workflow optimizations.

In this chapter, we use a real-world workflow from a life sciences use case, EURExpress, which is described in Part IV, Chapter 16 to describe the DISPEL enactment stages. This workflow retrieves data samples from a database splitting them into two sets: one set for training a classifier and the other for evaluating the trained classifier—following the *k*-fold cross-validation pattern described in Section 10.8.2. In the implementation, we have split this workflow into three DISPEL requests: data preparation, feature selection, and *k*-fold cross validation.
12.2 DISPEL LANGUAGE PROCESSING

12.2.1 Compilation

When a request is received by the gateway (Section 12.4.1), the DISPEL script is parsed and its syntax is validated, followed by type checks at the language level, for example, ensuring that values are assigned correctly to variables, and function parameters and return types are assigned and used properly.

Compilation is an iterative process as reusable components, such as functions, types, and composite PEs (e.g., those containing PEs), referenced in the DISPEL request must be retrieved from the registry. These imported entities are represented
DISPEL ENACTMENT

by their metadata descriptors (available from the registry) and DISPEL code and must be compiled before they can be used. Therefore, for each imported component, a compilation subprocess is triggered, the results of which are imported into the ongoing compilation process. If a component does not exist in the registry or the compilation of its implementation DISPEL script fails, a compilation error is raised and DISPEL processing terminates sending back results.

An important example of this iterative process results from the invocation of functions to create components of a workflow. For example, when the \texttt{makeSieveOfEratosthenes} function is invoked to create the \texttt{PrimeGenerator}, see Section 10.7, an array of \texttt{PrimeFilter} PEs is also created and must also be processed by the DISPEL compilation environment.

Note that primitive PEs, those not implemented directly in DISPEL, must have executable implementations available for deployment later in the enactment process. The data-intensive platform may import these implementations in a variety of ways. Some may be core components of the platform such as the \texttt{Results} PE. Others may be implemented in other languages such as Java that can be dynamically loaded at deployment time. These components may be precompiled, compiled on the fly as part of this process, or passed to the platform as a script for runtime interpretation, in the case of Perl for example. In the registry entry for a component, one property is often a URL indicating the location of the component in a repository. This helps to promote the availability and sharing of components.

12.2.2 Graph Generation

The result of a successful compilation process is a fully expanded graph of the data-intensive computation represented by the original DISPEL script. All the nodes of this graph are primitive abstract PEs that will be bound to actual implementations later in the enactment process. The edges of the graph represent the dataflow between PEs. Each connection has a source PE producing data and a list of sink PEs that consume the data. Because a given PE may appear many times in a graph, each node is given a unique identifier that will be used later to identify the actual deployed PE instance in the executed data-intensive computation. This identifier will be used for monitoring and diagnostics.

As the graph is validated, checks are made to ensure that implementations of the components exist that can perform actions as requested and that the descriptions of those components are compatible with the specifications in the request. If a nonexistent component is referenced in the request, for example, when attempting to connect to a PE that has no outputs, an error is raised and the DISPEL processing terminates.

PEs and connections in a DISPEL graph may carry additional information as annotations. An annotation is a property value that can be identified by a key. Annotations are used to convey type information and semantics, as outlined in the language definition, to the later stages of DISPEL processing.

Connections and PE inputs and outputs have structural (\texttt{Stype}) and domain (\texttt{Dtype}) types. The PE descriptor defines the types that an input can consume.
and the types that an output produces. These types can be further restricted for individual requests by annotations attached to connections in the DISPEL script. When these types are propagated across the graph at a later point (Section 12.3), this information allows type verification for compatibility by testing structural subtyping for S\text{type} and by querying the type hierarchy in the registry and possibly resolving type clashes by inserting type converters, as described in Chapter 10.

The resulting graph carries all of the structural and domain type information and semantics that are provided by registered metadata as well as request-specific semantics. PE descriptors carry modifiers on their data connections that specify termination behavior, data consumption rate, type transient behavior, and others, as outlined in the language definition (Chapter 10). For example, modifiers may indicate the rate at which data are consumed or emitted by a PE, or the order in which data are read from the set of input streams. Other semantics, such as type assertions and non-default termination behavior of a PE, may also be defined using modifiers and communicated to the later phases of enactment via these descriptors.

During the compilation, data inputs marked as a locator are identified and their values are added to the annotations if they are included as literals in the request. These locator values are important parameters used by the optimization stage that prepares the DISPEL graph for deployment.

At the end of the compilation process, reusable components are registered with the registry directed by the register command. Submitted dataflow graphs are passed to the next stage, optimization (Section 12.3).

12.2.3 Registration

DISPEL entities must be registered before they can be reused. Using the register command (as described in Section 10.1), a component can be published in the registry.

When registering an entity, the DISPEL processor creates a metadata descriptor and places it in the registry. The DISPEL script representing the implementation of the component is stored for use when deploying a workflow.

The registration of a new reusable component may fail, for example, if another component with the same name already exists in the registry. The DISPEL processor attempts to register as many entities as possible and provides warnings for failed attempts.

12.3 DISPEL OPTIMIZATION

The output of the DISPEL language processing stage is a fully expanded and annotated DISPEL graph. The next step is to organize the distributed computations on the execution engines—computing resources used to deploy and execute the DISPEL graph. In general, a typical scientific workflow comprises complex computation steps on heterogeneous data and is enacted on distributed computing platforms. The performance of enacting such a workflow relies on various
factors, for example, the selection of scattered data sources in the workflow may trigger a high communication cost of moving data between execution engines. Workflows need to be mapped onto the appropriate execution engines, in terms of workloads, computing paradigms, and resource availability, in order to achieve maximum efficiency.
Thus, it is a challenge to find a way of organizing the distributed computation that will deliver the same results within the application-dependent criteria at the least cost. Various cost functions may apply, for example, time to initial output, time to completed output on all delivery streams, or amount of energy used. This challenge is known as *multigoal optimization* (Section 12.3.3), where a set of *goals* is defined and is achieved through a series of *optimization approaches* within some predetermined *constraints*.

Let us say that we want to reduce the overall *make*span—time difference between the start and end of the enactment—of enacting the graph shown in Figure 12.2. We can identify the most appropriate PE implementations for the enactment based on the enactment requests, data, and available execution engines. By identifying these factors and designing a structured system to capture relevant performance data, useful information can be extracted to further improve the enactment. Section 12.3.1 describes the rationale and implementation of the performance database (PDB) to capture such data.

Assume that the performance data collected from previous enactments show that it takes longer to process the training subgraph than it does to process the testing subgraph. Given enough computational resources, we can take another optimization approach: to split the training subgraph into parallel streams and enact them on multiple execution engines. Section 12.3.2 describes this parallelization as well as other graph transformations. The optimization process produces concrete workflows that are ready for deployment on the execution engines, as discussed in Section 12.4.

To perform all of the processes above, the optimizer must first obtain information about:

1. the descriptions of components provided by the domain and data analysis experts (who designed and implemented them);
2. the performance of the component instances gleaned from observing previous enactments, for example, processing time per unit of data (unit cost), memory footprint, and so forth;
3. the descriptions of the data-intensive platforms, both hardware properties (e.g., number of computing nodes, bandwidth between their connections, etc.) and software properties (e.g., database engines, execution engines, etc.).

This requires a good understanding of where and how to collect such data, and how to transform these data into useful information for the later stages of optimization and deployment.

### 12.3.1 Performance Database

The PDB, a component of the optimization phase of compilation, is designed to gather information at the level of PE classes so that we can determine how each class and data stream behaves. For instance, information collected from a previous enactment can indicate whether co-locating certain PEs within the same execution
engine will result in poor performance because these PEs are competing for the same resources. The use of performance data collected from previous enactments is worthwhile for two reasons. Firstly, domain experts tend to repeat similar enactment requests to iterate their understanding or to process multiple similar data samples. Secondly, there are many fundamental PEs that are used across domains and consequently appear in many enactment requests, for example, SQLQuery and Split used in the DISPEL graph shown in Figure 12.2 are commonly found in any workflow that requires access to a database and to split the data stream into training and testing datasets during the data integration process.

We introduce a four-stage life cycle for performance data: collect, organize, use, and discard, as shown in Figure 12.3 to describe how the performance data are collected, transformed into information, used for optimization, and then discarded.

12.3.1.1 Data Sources and Collecting Mechanism We define a data-intensive virtual machine (DIVM) as an abstraction for the computational environment (i.e., the layers of software and hardware) in which a processing element instance (PEI) runs during enactment. Understanding a DIVM’s configuration is important because it allows us to discover whether two PEIs are sharing a common platform element and whether that in turn influences their enactment performance. This information is extracted from system log files.

As a typical operational environment is complex, deriving relevant information, for example, whether two PEIs are competing for the same CPU, is difficult. The DIVM abstraction is used to reduce the complexity by suppressing detail, so that queries against the PDB can discriminate such conflict criteria. This abstraction also has to reflect locality so that relative costs of inter-PEI communication can be estimated using the PDB.
The overall performance of an enactment is affected by the behavior of the PEIs, including how they are connected and on which DIVM they are deployed. In general, the tasks performed by PEIs include computation, communication, and I/O operations. These are performed at different rates depending on the assignment of PEIs to DIVM instances—remember the distributed data-intensive platform is normally made up of heterogeneous components.

The Measurement framework captures these enactment-performance data using a specific type of PE, named an observer. An observer receives data from input streams from a previous PE, applies a time stamp, and outputs the data to the following PE without altering the data. By placing observers on the data streams, detailed enactment information can be captured and used for making appropriate optimization decisions. In theory, we can have three types of observers, each with minimum impact on performance and a capability to capture performance data from a different perspective:

- **Type observer** is used to capture type information of the dataflow on any given data stream. Together with the semantic information of the workflows, the type information may be useful in estimating the data transfer cost and determining the ability to split a data stream and process in parallel. The type information should be collected before execution.

- **Rate observer** measures the data processing rate of data streams. When used in pairs, rate observers can capture the processing time per unit of data of a PE. As shown in Figure 12.4, a rate observer is placed before FeatureGeneration to capture its input data rate during the enactment. Together with the output data rate measured by another rate observer, we can infer the processing rate of FeatureGeneration.

- **Buffer observer** is used to observe the buffer implementation of data streams. Buffers are used when there are variations in the data processing rate of any connected PEs. In Figure 12.4, buffer observers on the two input streams of TupleBuild determine the rates at which data arrive on each stream and from which we can infer the critical path of the workflow.

For the implementation of a data-intensive platform, the type observer is applied during the DISPEL language processing stage. When the DISPEL language
processor walks the generated graph verifying that source assertions and destination requirements about the structure types of values in the data stream are compatible, the input and output structural type of every PEI in the request will be recorded. Both rate observer and buffer observer are implemented during the execution stage, observing the pipe buffer events generated by the data stream. During the enactment, the data producer of a data stream writes the data into the buffer while the consumer reads data from it. Both operations will trigger different events. Another two interesting events to record are blocking from read and write, for example if a fixed-size buffer is empty or full respectively.

The results collected from observers are sent to a gatherer that will insert these data into the PDB after the enactment is finished. To further reduce the overhead incurred during enactment, each gateway should have gatherers that run on separate execution engines to process the collected data on the fly and insert the derived and hence much-compressed performance data into the PDB. However, the platform should provide an option to keep all of the recorded events when data-intensive engineers are trying to trace the events that occurred for diagnostic purpose.

12.3.1.2 Organizing Performance Data and Transforming into Information

The tables in the PDB are divided into three categories according to how their data are collected (Fig. 12.5). The first type of table stores data harvested from log files, for example, DIVMInstance and DIVMInstallation. The second type of table stores data collected from the measurement framework, for example, DataStream. These two types of data are considered raw data. The final type of table stores derived data, for example, PerfOfInstance, which are preprocessed by gatherer on all of the events recorded. These data are used in the calculation of the unit cost for any given PE on the DIVM where enactment occurred.

The performance data gathered in the PDB are used in stages, as shown in Figures 12.1 and 12.3. For each stage, we formulate different sets of queries to access the PDB. The PDB data allow us to understand and validate hypotheses
about PEs and their enactment behavior, such as the type of dataflow in the data stream\(^1\) and the processing rate of PEs on different execution engines, through queries such as the following:

\[
\begin{align*}
\text{SELECT} & \text{ AVG(PerfOfPEInstance.time per unit), MIN(PerfOfPEInstance.time per unit),} \\
& \text{ MAX(PerfOfPEInstance.time per unit), COUNT(DIVMInstance.instance id),} \\
& \text{ DIVMInstance.instance id} \\
\text{FROM} & \text{ PerfOfPEInstance, PEInstance, DIVMInstance, PEInstallation} \\
& \text{WHERE PerfOfPEInstance.instance id = PEInstance.instance id} \\
& \text{AND PEInstallation.instance id = PEInstance.instance id} \\
& \text{AND PEInstallation.install on = DIVMInstance.instance id} \\
& \text{AND PEInstance.class id = 'PEa'} \\
& \text{AND (DIVMInstance.instance id = 'DIVM1' OR DIVMInstance.instance id = 'DIVM2')} \\
\text{GROUP BY} & \text{ DIVMInstance.instance id}
\end{align*}
\]

The query will retrieve all of the previous execution records of a PEI on all of the DIVMs, filtered by DIVM\(_1\) and DIVM\(_2\), to find the more suitable DIVM on which to enact a PEI. This enables the optimizer to deploy PEs on the execution engines best able to support them in a heterogeneous environment. More queries and demonstration of the PDB use can be found in [1].

The size of the PDB is expected to grow rapidly. To sustain the performance of the PDB, a cleaning process is needed to remove outdated or less important data. The PDB is cleaned in two ways: (i) by removing data associated with deprecated versions of a PE and (ii) by removing data that are obsolete (e.g., pertaining to a discontinued DIVM or after a predefined number of days).

### 12.3.2 Graph Transformation

The graph generated in the DISPEL language processing stage will go through a series of transformations as a result of optimization. This process is conducted repeatedly until a final graph is produced, which is ready for the deployment stage (Fig. 12.2). In this section, we further examine four types of common transformations: subgraph substitution, type conversion, parallelization, and reordering.

During the optimization stage, the optimizer tries to identify candidate implementations for every abstract PE (nodes in the DISPEL graph). An abstract PE can be mapped to a single physically located PE instance, or, if it is a composite PE, to a group of implemented PEs. In the latter case, the PE node will be substituted by a subgraph of PE nodes. This process continues until all concrete PEs are processed. The FisherRatio PE used in the DISPEL example, shown in Figure 12.2, has two implementations: sequential and parallel. The parallel implementation is defined using a composition of three implementable PEs: (i) StandardDeviationCalculation, (ii) StandardDeviationCombine, and (iii)

\(^{1}\)This may not be known a priori because the DISPEL scripts deliberately use Any and rest to suppress details, accommodate change, and support domain-specific formats. Once enactment is being considered, the optimizer needs to “look under the hood.”
FisherRatio\textsubscript{Parallel}. In this case, the FisherRatio PE is expanded into a sub-graph that comprises three implemented PEs.

\textsc{Dispel} introduces a sophisticated type system to validate the connections between the PEs (Chapter 10). Together with the semantic description of the PEs stored in the registry, this architecture provides a powerful capability for \textit{type conversion} in the \textsc{Dispel} request, which inserts a \textit{converter PE} into the graph to convert the data type \cite{2}. For example, the metric for temperature stored in the data source is expressed in \textit{Kelvin}, whereas the metric used in the data mining PEs is expressed in \textit{Celsius}. In such a situation the optimizer finds a \textit{converter PE} to be placed into the data stream to perform a Kelvin-to-Celsius conversion, which alters the \textsc{Dispel} graph.

The optimizer improves the enactment performance by exploring \textit{parallelization} opportunities. To speed up a slow-processing-rate subgraph, the optimizer looks for parallel-executable PEs and splits the data streams. This parallelization is categorized as data parallelism, where a data stream is split into multiple streams for parallel execution. As illustrated in Figure 12.2, a \textit{Split} PE is added after the \textit{SQLQuery} PE to split the data for parallel execution.

\textit{Reordering} is about transposing the PEs' order based on a quality of service metric. It follows the approach in database optimization, where the reordering is guided by a set of rules based on relational algebra. For instance, placing a filtering operator before a projection operator may reduce the computation cost of performing these operators in reverse since the filtering operator reduces the amount of data to be processed by the projection operator. In order to support reordering PEs, the optimizer needs to obtain semantic information from the registry that shows whether these PEs are transposable.

### 12.3.3 Multiple Goal Optimization

The optimization challenge is to find a way of organizing the distributed computation that will deliver the same results within the application-dependent criteria at the least cost. In optimization, we define the \textit{goal} and the \textit{constraints}, then we choose the appropriate \textit{strategies}. The goal is the main focus of the optimization at which improvement is aimed. The goal is achieved through a set of approaches. However, there are certain restrictions that limit the approaches, which are referred to as \textit{constraints}. There are two common optimization goals: \textit{time-based criteria} and \textit{resource efficiency}. The time-based optimization is focused on the application perspective, especially on minimizing the response time or execution time. In contrast, the resource efficiency type of optimization looks at the resource perspective (computation, storage, network traffic, or energy). For instance, optimizing the system throughput seeks to get the most out of the data-intensive platform, reducing disk storage to accommodate more applications, or minimizing energy consumption.

To achieve these goals, the optimizer needs to understand appropriate and available optimization strategies, such as
• ameliorating performance bottlenecks by executing in parallel;
• distributing enactment to balance workload (load balancing among execution engines);
• minimizing data movement by co-locating large dataflows within the same region of a platform;
• clustering small tasks to reduce deployment and data movement costs;
• performing logical transformation (e.g., reordering PEs);
• dynamically selecting services (or PEs) based on availability and performance.

Some strategies may not be applicable to all computing models. For instance, if the goal is to increase resource efficiency and the limitation of storage space is a constraint, the common approaches taken in other workflow systems is to either remove data files at runtime (when they are no longer needed) [3] or to restructure the workflow (reduce the “width” and increase the “depth” of the workflow). However, these strategies are less applicable in the data-streaming model because all of the PE instances are very likely to be loaded before the enactment starts and overlap during the execution. The model already eliminates many intermediate files.

All of the approaches described earlier are performed before execution (during the mapping stage). The optimizer makes all of the decisions before the graph(s) are deployed. From the deployment stage onward, the optimizer does not interfere in the execution. The design of the data-intensive architecture is intended to enable the enactment of DISPEL on different data-intensive platforms and not every platform allows the alteration of PE instances during the execution. However, given the ability to control the state of PE instances and the data stream on the execution engines, dynamic optimization that looks at load balancing during runtime is achievable, but it will be platform dependent.

The optimizer needs to understand the list of constraints that it has to satisfy, such as accessibility restrictions. In principle, the PE implementation codes are stored in the repository and are loaded to the execution engines during the deployment stage. However, there are PEs that are mapped to proprietary implementations, which are only accessible at a specific gateway. Similarly, some confidential data sources are only available via specific gateways. Another reason that restricts the choice of mapping candidates is the processing capability. Some PE implementations are platform dependent and require a dedicated execution engine for their deployment.

12.3.3.1 Three-Stage Optimization Algorithm for Data-Streaming Model
Finding the scheduling candidates of PEIs on DIVMs involves exhaustive computation, especially for large workflows that comprise many PEIs. Some of the PEIs have a relatively small time to process a unit of data (i.e., unit cost), and the assignment of these PEIs onto any of the DIVMs may not impose a significant workload. Most of the PEIs have a relatively small unit cost and a few have much larger unit costs. Optimization, therefore, focuses on assigning these
expensive PEIs to DIVMs. They first have to be identified by defining a performance threshold that divides the PEIs into two categories based on their unit cost. The performance threshold needs to be determined from experiments and may be domain specific. We apply job-shop scheduling on these heavy PEIs with larger unit cost and use a bead-sliding algorithm (Section 12.4.1) to assign the remaining light PEIs. We also consider the locality of the PEIs besides the unit cost criterion. Some PEIs have to be enacted on selected site because of the accessibility or processing capability constraints.

This type of PEI is marked as an anchor and handled separately during the assignment process.

We summarize the assignment algorithm in the following steps:

1. Identify the anchors and assign them accordingly to the DIVMs
2. Apply job-shop scheduling to assign heavy PEIs
3. Apply bead-sliding technique to assign light PEIs.

12.3.3.2 Partitioning the PEIs into Three Subsets

The optimizer scans the set of PEIs and identifies the following two critical subsets to be allocated to DIVMs first:

1. The Anchored PEI. These are PEIs that have the location modifier (Section 10.2.5) and have been annotated with a stream literal expression defining data to which they are anchored.
2. The Heavy PEI. These are the PEIs not in the above set, which have been identified as having high unit processing costs, that is, that are above a defined threshold.

All of the remaining PEIs are categorized as light PEIs.

12.3.3.3 Assigning Anchored PEIs

The anchored PEIs can be assigned only to one of the DIVMs near their source data unless the DIVMs do not exist or are overloaded. In the latter case, they are treated similar to heavy PEIs with a data movement precursor job. The optimizer first attempts to assign these anchored PEIs. For each PEI in the anchored set, the optimizer discovers all of the instances of the required data and, for each of these, discovers all of the DIVMs close to the data and capable of accessing it. It then allocates each anchored PEI to one of these DIVMs, distributing load and reducing data movement.

12.3.3.4 Assigning Heavy PEIs

One of the important characteristics of the data-streaming model is that PEs are connected in a pipeline that allows task executions to overlap. PEs start to process as soon as they have received sufficient data from their predecessors and emit data as soon as their processing on a unit of input has finished. This overlapping behavior of the PEIs’ execution generates

\(^2\)The overlapping behavior is intended to allow multiple PE steps to work on data while they are close to the processors in the execution engines’ memory hierarchy.
a problem when calculating the total execution time of a DISPEL graph. One of the possible solutions is to model the time to produce an element of the final result and think of this as a job-shop scheduling problem. A job-shop scheduling problem is about scheduling a set of jobs $J$ on $M$ machines to achieve the minimum completion time.

We define a job, Job$_{i,j}(k)$, as the processing of the PEI$_i$ to generate a data element on output $j$ and $k$ is an output element counter $\{1, 2, \ldots\}$, for example, $J_{\text{MF.data}(1)}$ is the execution of MedianFilter to produce the first element on output data (Fig. 12.4). We then define a dependency graph between all of these jobs $J_{i,j}$ according to the dataflows along each connection. For instance, the processing of the first output element of TupleBuild, $J_{\text{TB.output}(1)}$, is dependent on the output of $J_{\text{Post.filename}(1)}$ and input Connection mask, which both have dependencies on other jobs.

The jobs will be scheduled to available DIVMs. When the scheduling requires moving data elements across DIVMs, it incurs additional communication time. To model the communication time, we consider the data movement as a separate job, which can only be executed in an abstract DIVM, transport DIVM. We propose a transport DIVM, $T_{i,j,d}$, to move an element of data from DIVM$_i$ to DIVM$_j$. For instance, assume that MedianFilter and FeatureGenerate are mapped to DIVM$_i$ and DIVM$_j$; accordingly, a data movement job $J_{\text{MF.FG.data}(1)}$ is added to $T_{i,j}$ to move the output element of $J_{\text{MF.data}(1)}$ from DIVM$_i$ to DIVM$_j$ for $J_{\text{FG.data}(1)}$. In terms of dependency, $J_{\text{MF.FG.data}(1)}$ must be scheduled after $J_{\text{MF.data}(1)}$ and before $J_{\text{FG.data}(1)}$ without any overlap.

Our optimization goal is to minimize the time to produce a unit of result (a unit of output element of the PEI). In other words, if $T(DIVM_i)$ is the time spent in processing all of the jobs in DIVM$_i$, we want to minimize the max($T(DIVM_i)$), for all $1 \leq i \leq n$. To achieve the optimization goal, our scheduling algorithm should (i) minimize the idle time of DIVMs and (ii) minimize the movement of data across DIVMs (minimizing the jobs on the abstract transfer DIVM).

The following is a list of known constraints:

- All of the jobs from the same PEI must be scheduled on the same DIVM.
- Precedence of data must be obeyed (jobs must be executed in order).
- All PEIs are allowed to overlap their execution, but this is NOT necessary.
- The jobs are not preemptive.
- Each DIVM executes one job at a time (even though each DIVM is scheduled to multiple PE instances during enactment).

12.3.3.5 Assigning Light PEIs We see the DISPEL graph as beads connected with strings that correspond to a sequence of their dataflow interconnections, and the DIVMs as the bowls to store the beads, as illustrated in Figure 12.6. The assignment problem is the task of allocating beads into bowls. For each bead, we have to decide whether it should slide down into the left bowl or to the other direction into the right bowl. For sliding PEIs into DIVMs, the decision
is affected by two criteria: the volume of dataflow of each stream connecting the PEI to its predecessors and successors, and the workload of each of the DIVMs. If there are multiple connections between these beads (i.e., PEI with more than one input/output), there may be several strings pulling them in different directions. In such a situation the string that has the biggest pull will be chosen because it is transferring the most data.

12.4 DISPEL DEPLOYMENT

At this point in the enactment process, the optimizer has created a well-annotated graph of the computation described by the original DISPEL script. If the graph needs to be executed across several gateways (which is assumed to be the normal case), the graph has to be decomposed into separate subgraphs for deployment at the sites specified by the annotations. The information included in the annotations includes the location of the execution engine that will perform that subgraph, the location of all data sources (databases or streaming sources), locations where temporary repositories for intermediate results will be located, and the assurance that the code needed for each PE is runnable at the execution engine identified for the PE. This graph (or rather set of graphs) is mostly platform independent. What remains to be done is to complete the deployment process, a largely platform-specific function. The deployment process involves obtaining and deploying executable code for the PEs, linking the data connections between PEs within an execution engine and across multiple engines, and establishing the monitoring and control infrastructure.

The compiler will have verified that an implementation of each PE is available to each execution engine that runs a given PE. However, there are a number of platform-specific ways in which this code can be deployed. A very simple implementation of the platform might require that the code associated with each PE be precompiled as part of the platform. This might be suitable in specialized scenarios, but, in general, a more flexible code deployment strategy will be expected. A highly flexible platform would maintain a repository of PE implementations (possibly centrally) and, during this deployment phase, distribute the PE code to the various gateways for dynamic loading into the execution engine. To some extent, the deployment strategy for code will depend on the implementation language of the PE itself. Java code, for example, can be linked with the platform itself, dynamically loaded as required, or indeed compiled on demand. Perl or other scripting
languages are likely to be run interpretively in any case and, therefore, will use a more dynamic deployment strategy. A very flexible platform supporting multiple implementation languages will likely use several deployment strategies.

Clearly, one of the primary services provided by the platform is connecting the various PEs with dataflows, as specified by connections in DISPEL. Platforms have a wide variety of mechanisms available to implement this function depending on many design factors and the context in which a given connection is being deployed. The following are a few examples of strategies that could be taken and the contexts in which they might make sense:

1. In the simplest rendering of an execution engine, all PEs might run within the same JVM. In this case, connections could be implemented using something similar to Object Streams. These could be extended to provide various levels of buffering and monitoring.

2. A high performance data processing implementation might use a high performance network passing data along connections using network-specific protocols of libraries carefully tuned to that network, for example, MPI.

3. Similarly, a site that offered a number of different execution engines, some of which might be simple data sources such as detectors, the interconnection technology used to connect PEs representing these sources would need to account for the site’s infrastructure capabilities.

4. When execution engines are located at different sites, the communication between the subgraphs may be over low bandwidth or even public networks and, therefore, data compression and encryption would need to be incorporated in these connections.

5. It is also possible that gateways on different sites might be provided by different implementations, in which case the implementations would need to incorporate a negotiation step into the deployment whereby they agree on a data exchange protocol standard to use.

The last, and in some way optional, aspect provided by the platform implementation as part of deployment is the monitoring infrastructure. The more information gathered by the infrastructure about performance, the more sophisticated the optimizer can become. Even simple implementations will find that a little monitoring information can aid in diagnostics and help spot performance bottlenecks. More complex implementations will also provide real-time visualization of the graph, as it is processed by a collection of execution engines.

12.4.1 The Role of Gateways

In distributed computing contexts, the gateway takes on an additional role beyond that described in the hourglass discussion (Fig. 9.2). When data processing is spread across multiple sites, the gateway provides additional infrastructure.

Firstly, the gateway can accept completely annotated DISPEL subgraphs that have been generated as part of the optimization and distribution process. These
DISPEL descriptions of subgraphs will typically be fully optimized already, but nonetheless, the gateway processes them as usual. Since these partitioned graphs will need to communicate with each other, these subgraphs will contain additional information (in the form of either annotations, additional PEs, or both) to facilitate the interconnections between subgraphs.

Secondly, the original gateway will serve as the user’s point of contact with their running workflow. This gateway will need to collate information such as errors, performance data, and progress indicators from any other gateways to which it sent partitions of the original DISPEL graph. It is also through the initial gateway that the user will receive some of the results from the workflow, although where significant data volumes are concerned, the user may be referred to the gateway actually holding the data in question.

Thirdly, in production environments, the gateway will also provide security functions, such as overall authentication and authorization. This level of security is needed to control access to the site running the gateway. Further mechanisms are needed to manage fine-grained access to the data sources themselves, including record level access databases.

In the following section, we discuss the execution phase of enactment.

12.5 DISPEL EXECUTION AND CONTROL

At this stage in the enactment process, see Figure 12.1, the following have all been completed:

1. The DISPEL script has been compiled and run to generate an abstract graph.
2. The graph has been optimized in the context of available resources at several possible locations.
3. The collection of PEs (rendered in location-specific forms) has been deployed and resources allocated for their execution.

What remains is the execution of the data-intensive application itself. As the process of enactment has progressed as outlined above, the technology involved has become more and more precise. At this point, we are ready to execute the primitive instructions of the data-intensive application. Any implementation following this approach will need mechanisms to handle a number of key aspects of execution and control.

Up to this point, there has been nothing specific stated about how the data are passed from one PE to another or even what processing paradigm is used to implement the PEs themselves. The discussion so far has been limited to the definition of a kind of abstract machine for implementing data-intensive distributed computing (DIDC), which can be built using a number of different technologies. In the ADMIRE project [4], the implementation of the execution phase was based on the OGSA-DAI [5] distributed data integration platform. We will not discuss the
details of this particular implementation here, but rather provide a description of the capabilities that any platform implementing DIDC would need to have. Given that there are many possible ways to render the platform, there are a number of issues that all platforms executing DIDC applications will need to address. These are discussed in the following sections.

12.5.1 Data Flow Control

The single, most fundamental aspect of the DIDC platform arises from the streaming dataflow model implicit in the whole approach. All processing follows a dataflow paradigm. In particular, computation only takes place when the data needed for the computation becomes available. This is in contrast to the traditional von Neumann model where the computing engine fetches the data as it moves through a computation described by a script. Thus, the management of data streams is foremost to the concept of DIDC. The notion of a data stream must be a first class object in any implementation. In fact, the approach taken to addressing data streams embodies the distinguishing characteristic of an implementation. Typical approaches might include event-driven systems where each DISPEL tuple would be treated as an event and processed in a manner similar to complex event processing systems [6]. A primitive implementation might be rendered as a collection of Unix processes connected by pipes. More sophisticated implementations would provide direct networked sockets between processes (with or without threading within the processes). Implementations targeted at widely distributed deployment might opt for Web-services-based protocols.

Once a basic paradigm is identified, the next issue to address is the initiation of the dataflow that drives the computation. The graphical nature of the DISPEL-described graph does provide for the existence of a root node or nodes that provide the initial stream of data that will in turn drive the computation. However, the distributed nature of the application may mean that the actual initiation data source is remote. In these cases, the implementation will need to identify the incoming data stream as the local point of initiation. Once computation begins, it must then be managed. There are two further aspects to managing dataflow, data marshaling (usually with buffers) and dataflow termination.

Data marshaling is needed as part of the platform itself, as even in the most precisely defined applications there will be instances when, because of resource constraints, for example, data will either build up at the output of one or the input of another PE. In Chapter 10, we described a number of language level mechanisms that provide hints to the management of dataflow. The platform should use the hints to improve the marshaling of data in and out of PEs running in the platform. For example, emits indicated the rate of output from a PE relative to its input. The notation round-robin indicates that data is taken from each input in turn. These annotations can help a platform better optimize the management of information flow in the system.

However, there will be times when buffering will be required and the platform must provide the ability to accommodate this. Even with buffering in place, it is
important for platforms to provide other capabilities that could be used to prevent excessive buffering. This may be important when the various stages of a computation have widely varying execution times. In these cases, the platform may be able to notify the upstream PE that it should throttle back on the production of tuples. In a quality implementation, this function should not be visible to the user, but rather provided as a quality of service capability of the platform.

The reverse is also common in data-intensive applications. A given PE may be able to process much more quickly than the PE upstream from it and, therefore, be subject to starvation. While not a resource constrained problem, the potential for efficient use of resources is possible on platforms that acknowledge and exploit the possibility of starvation. For example, see the next section with respect to virtualization.

Closely related to overall termination, the termination on a given data stream needs to be handled by the platform. The hosting environment that controls the execution of PEs and, therefore, also manages the resources they consume needs to be informed when a data stream no longer is expected to deliver data to the connection. The transmission and receipt of these EoS (end of stream) tokens controls one-half of the termination model of DIDC. In the other direction, that is, upstream, termination conditions are also supported. In this case, the PE that has terminated (for whatever reason) will notify upstream PEs that it expects no more data. The propagation of this information and the anticipation of the changes in execution they imply need to be handled by the platform.

12.5.2 Processing Control

While DIDC is focused mostly on the data, it is not without a computational element. The role of the platform in this respect can be best summarized as one of resource management. The deployment phase pairs up computational elements (PEs and Connections) with specific resources associated with each gateway, but it is the platform that controls and manages their execution during processing of the data-driven workflow. The level of support offered by a platform will be one of the distinguishing features of the platform. The simplest platform resource manager will simply adopt the course-grained allocation provided in the deployment phase. This would have consequent issues of resources becoming overloaded or idle depending on other activity in the environment. A more sophisticated platform would support some type of dynamic resource allocation.

The extent to which a platform can be flexible with respect to these variations in load and resource utilization will greatly depend on the level of support provided by the final stage of the enactment process. In particular, a platform that supports dynamic binding of abstract PEs to their implementation will have much greater flexibility with respect to resource allocation. In some cases, this can be as powerful a capability as a further pass at optimization, albeit at a local level only. Platforms could even be permitted to substitute a parallel implementation for particular PEs without recourse to the user or developer. The benefits of using spatial as well as temporal optimization (trading time versus space in either direction) will create the
potential for a market place where quality of service can be part of the decision making process.

The sensitivity of the platform to conditions in the execution environment and the state of execution of all the activities under its control will also provide opportunities for improved efficiency. In many cases, a particular PE will enter a temporary phase of starvation while it waits for additional input on one or more of its input connections. At this time, a platform should take advantage of the increase in resource availability to perform other, possibly unrelated work. Platform implementers are likely to exploit the value of virtualization in addressing this issue. Because of the specific nature of the DIDC environment, this virtualization need not be simply at the virtualized host or operating system level, but virtualization technology within the platform itself could provide greater benefit to the users than more brute force approaches.

In some cases, there will be factors beyond simple optimization that require the platform to respond through resource management. For example, a memory overload due to uncontrolled buffering might require the migration of the overloaded task (or another one sharing the same resource) to ameliorate the situation. Similarly, there may be changes in execution priority or quota limits that require the platform to respond to the situation dynamically. Again, as with virtualization, the unique nature of DIDC means that the implementers of the platform do not need to rely on operating system services to enable their platforms.

The nature of DIDC described here allows for platforms to carry out local optimizations that are highly specialized. One example is the ability to reuse precomputed data. A frequently requested partial result could be identified by the platform and reused on a subsequent occasion, thus saving the recomputation of the result. This capability is supported by the DIDC architecture described here through the registry and repository model. The accurate semantic description of results (partial or otherwise) allows platforms to participate in the same kind of reuse optimization that only the DIDC developers would normally undertake.

The structure and details of information needed in support of the optimization process are discussed in Section 12.3. However, it is the platform that must gather, maintain, and distribute this data to the distributed gateways performing the optimization phase of enactment. While this data are fairly specific, the data will be accompanied by platform (and possibly gateway)-specific accounting information.

In many cases, users will be interacting with the workflow, making decisions to continue, archive, or, in other ways, influence the execution of a workflow. The provision of this interactive element adds significant complexity to the platform’s resource management strategy. Users may interact at specific times, meaning that the platform must “hold” the computation until requested by the user. Likewise, the user may “interrupt” the computation to alter running conditions or parameters. Both situations should be catered for by the platform.

In line with the separation of concerns adopted by the DIDC architecture described here, the platform developers can exploit the abstraction provided by the neck of the hourglass in Figure 9.2 and provide significant creativity to their rendering of the architecture. For example, the more flexible and well-instrumented
platform will be able to include green assessments and carbon accounting in their resource summaries and, therefore, offer users the opportunity to choose a greener implementation of their application.

12.5.3 Termination, Error Management, and Cleanup

While less exciting than green resource management, the ability of the platform to deal with the fundamental issues of termination and failure are critical. Termination of a distributed, graph-based, data-driven computation is complicated to begin with. The possibility of failure makes this a real challenge to implementations. The termination criterion defined by the DIDC architecture can be summarized as follows.

The default termination behavior of a PE occurs when either all the inputs are exhausted or all the receivers of outputs have indicated that they do not want more data. In order to reach this condition across the whole graph, when all of a PE’s inputs have received EoS, the PE completes the use of its current data, sends an EoS on all of its outputs, and then stops. Likewise, when all of a PE’s outputs have received a “no more data” signal, the PE sends a “no more data” signal on all of its inputs and then stops. In this way, termination propagates across a distributed enactment based only on local decisions.

The above describes the default behavior. Termination conditions can be defined on PE descriptors (e.g., a PE terminates when any one of the input streams is exhausted). Thus, platforms will need to extend this default behavior to deal with the following special situations:

1. One PE in the graph may execute an explicit Stop operation.
2. A fault occurs in one of the PEs. The implementation will also need to distribute this fact back to the user.
3. Deadlock is detected, although this will be rare in the presence of throttled buffer management, because of the DAG (directed acyclic graph) nature of the graph.
4. Time-out may also be included in places within the enactment to detect silent failures.
5. The propagation of distributed termination in the presence of network failures.

12.5.4 The Future Role of Standards

As noted above, the rendered platform is, at present, very specific to one implementation. However, as data sources and data-intensive resources are already widely varied in terms of implementation technology, data format, communication protocols, and so forth, integration of all this potential will be nearly impossible in the long term without some form of standardization. There are several places where standards can play the greatest role and others where the benefits will not pay off in the short or medium term.
Access to the data sources themselves is an obvious initial target for use of standards. However, with the exception of database access languages such as SQL, there are no universally agreed standards for remote access to data resources. Some efforts, within closed domains such as Web services, do exist [7]. Implementations will need, for the time being, to provide effective tooling for wrapping and interfacing to these data sources using the data sources that the provider makes available.

In Chapter 10, the outlined dataflow control mechanisms provide another opportunity for standards. If the interaction between execution PEs can be agreed across implementations of DIDC platforms, then the potential for greater collaboration and/or expanded business models become possible. These standards will need to address the issues described above around flow control, compression, encryption, termination, and failure. In particular, standardizing the connection interface between different gateways implementing instances of the DIDC platform would address many of the interoperability issues inherent in this architecture.

Since the rendering on individual PEs has been designed to be independent of any particular execution language or platform, this area is unlikely to be a focal point for standardization. The exception would be the DISPEL language itself.

REFERENCES