Performance database: capturing data for optimizing distributed streaming workflows

BY CHEE SUN LIEW1,2,* , MALCOLM P. ATKINSON1, RADOSLAW OSTROWSKI3, MURRAY COLE1, JANO I. VAN HEMERT1,4 AND LIANGXIU HAN5

1 School of Informatics, University of Edinburgh, 10 Crichton Street, Edinburgh EH8 9AB, UK
2 Faculty of Computer Science and Information Technology, University of Malaya, 50603 Kuala Lumpur, Malaysia
3 EPCC, University of Edinburgh, JCMB, The Kings Buildings, Mayfield Road, Edinburgh EH9 3JZ, UK
4 Optos, Queensferry House, Carnegie Campus, Enterprise Way, Dunfermline KY11 8GR, UK
5 School of Computing, Mathematics and Digital Technology, Manchester Metropolitan University, Chester Street, Manchester M1 5GD, UK

The performance database (PDB) stores performance-related data gathered during workflow enactment. We argue that, by carefully understanding and manipulating these data, we can improve efficiency when enacting workflows. This paper describes the rationale behind the PDB, and proposes a systematic way to implement it. The prototype is built as part of the Advanced Data Mining and Integration Research for Europe project. We use workflows from real-world experiments to demonstrate the usage of PDB.

Keywords: performance data; streaming workflows; measurement framework

1. Introduction

It is evident that data-intensive research is transforming the research landscape, as recognized in The Fourth Paradigm [1]. We are facing the challenge of handling the deluge of data generated by sensors and modern instruments that are widely used in all domains. Owing to the scale, complexity and heterogeneity of data gathered in scientific experiments, it is difficult to extract useful information and knowledge through exhaustive and unstructured computations. To survive the data bonanza, we need to improve our apparatus for the exploration and exploitation of the growing wealth of data.

In a previous paper [2], we have proposed optimization based on information on performance gathered in previous runs. This depends on an assumption that scientific users repeat similar requests over similar data as they iterate their understanding or process various samples in the exploration of variants and experimentation settings [3]. The performance database (PDB) is designed to

*Author for correspondence (c.s.liew@sms.ed.ac.uk).

One contribution of 12 to a Theme Issue ‘e-Science: novel research, new science and enduring impact’.
gather information at the level of processing element (PE) class instances, so that we can determine how each class and data stream behaves. PEs are software components that handle the distributed computation by processing and emitting streams of values, and are connected via data streams to form a workflow. For instance, information collected from a previous enactment can tell whether co-locating certain PEs within the same execution engine will result in poor performance because PEs are competing for resources. To be able to predict the enactment performance in this scenario, we must first obtain information about (i) the performance of the class instances, e.g. processing time per unit of data (unit cost), (ii) the hardware configuration for the enactment platforms, (iii) the software stack running on the enactment platforms, and (iv) the workload of the enactment platforms during the enactment. This requires a good understanding of where and how to collect such data, and how to transform the data into useful information in the later stages.

This paper describes the rationale behind the PDB, and proposes a systematic way to implement it. In §2, we introduce a four-stage performance data life cycle—collect, organize, use and discard—to describe how the performance data are collected and transformed into information. We propose a schema for the PDB and follow Gray’s Laws on approaching a large-scale scientific data engineering challenge [4], and identify important questions that our PDB should answer. Section 3 describes the PDB prototype. We have implemented a measurement framework (MF) using open grid services architecture data access and integration (OGSA-DAI) [5] and conducted experiments with a real-world life sciences use case from the Advanced Data Mining and Integration Research for Europe (ADMIRE) project [6] to collect data into the PDB. Section 4 shows how we use these data to test our hypothesis about optimization. The related work is discussed in §5. We conclude and discuss further research in §6.

2. Performance data life cycle

A typical scientific workflow comprises complex computation steps on heterogeneous data, and enacts on distributed computing platforms. The performance of enacting such a workflow relies on various factors, e.g. the selection of scattered data sources in the workflow may trigger a high communication cost of moving data to the computing platforms. However, workflows need to be mapped onto the appropriate platforms, in terms of workloads, computing paradigms and resource availability, in order to achieve maximum efficiency. By identifying these factors and designing a structured system to capture relevant performance data, useful information can be extracted to further improve the enactment. In the following sections, we describe in detail how this information is captured and stored in the PDB, and how it can be used during workflow optimization in the four-stage life cycle: collect, organize, use and discard.

(a) Collect stage: data sources and collecting mechanism

(i) Data sources

Figure 1a shows an overall picture of the data sources for a PDB. The data-intensive virtual machine (DIVM) configuration and PE deployment and connections data are harvested from system log files as they record the way the
software was configured. The hardware configuration is currently obtained by hand but should be based on common information model data. The timings of enactments are collected by an MF. We define a DIVM as an abstraction for the computational environment in which a PE instance (PEI) runs during enactment. Figure 1b shows a DIVM$^k$ which is an abstraction of the layers of software and hardware. As a typical operational environment is complex, deriving relevant information, e.g. 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 on 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 PEIs are executed in execution engines (EE), which are the highest layer of DIVMs. The mapping of a particular EE, e.g. EE$^k$, onto a DIVM, e.g. DIVM$^m$, is captured in the PDB from the log analysis described above. In order to optimize enactment a system chooses where to deploy PEIs, i.e. on which EE they should run. To interpret their behaviour it is necessary to record their identity and their host EE. This information is obtained by monitoring deployment during the preparation for enactment execution.

(ii) Collecting mechanisms

We use system logs and the MF to collect three types of data: configuration of DIVMs, deployment and interconnection of PEIs, and timing of the actual enactments. System logs keep track of the activities involved in managing DIVM lifetimes. The data gathered from the log files allow reconstruction of the software and hardware stack at a given time. Logs are also generated to track which PEIs are running on each DIVM at a given time.

PEI$^j$ is terminated on EE$^3$ at $t_{21}$

...  

PEI$^j$ is installed on EE$^3$ at $t_{15}$  

...  

EE$^3$ is installed on JVM$^x$ at $t_4$  

JVM$^x$ is installed on Linux$^y$ at $t_3$  

...
Assume that the log above is generated by the system, and X is installed on Y at \( t_z \) indicates that a software component X is installed or instantiated on another software or hardware component Y at time \( t_z \). From the log, the software and hardware stack for the DIVM can be reconstructed, as illustrated in figure 1b.

The MF captures the enactment performance using a specific type of PE, named an observer. As illustrated in figure 2a, an observer receives data from input streams from a previous PE, performs a time-stamping, and outputs the data to the following PE without altering the content of the data. By placing observers on the data streams, detailed enactment information can be captured and used for making appropriate optimization decisions. We have designed three types of observer, each with minimum impact on performance and a capability to capture performance data from a different perspective, as illustrated in figure 2b:

- **Type observer** is used to capture type information of the data flow 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 prior to enactment. In due course this will be superseded by capturing the equivalent data from the language parser.

- **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 PEI. As shown in figure 2b, a rate observer is placed before PE\(_k\) to capture its input data rate during the enactment. Together with the output data rate measured by another rate observer, we can know the processing rate of PE\(_k\).

- **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 2b, buffer observers on the two input streams of PE\(_l\) determine the rates at which data arrive on each stream, from which we can infer the critical path of the workflow.

The results collected from observers are sent to a gatherer PE, which will insert these data into the PDB after the enactment is finished. This reduces the overhead incurred during enactment.
(b) Organize stage: organizing performance data in the performance database

The tables in the PDB are divided into three categories according to how their data are collected (figure 3). The first type of table stores data harnessed from log files, e.g. Installation, WorkflowSubmit, EnactmentStart, PEmvInstance and DIVMvInstance. The second type of table stores data collected from the MF, e.g. DataStream, Events. These two types of data are considered raw data. The final type of table stores data that are derived by analysing the raw data from the above tables, e.g. PerfOfPEInstance. For instance, the table Events stores the events that occur on each data stream. By processing these data, we can calculate the unit cost for any given PE on the enacted DIVM.

(c) Use stage: transforming performance database data into information

Figure 4 illustrates the stages of PDB use. 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 behaviour, such as the type of data flow in the data stream and the processing rate of PEs on different platforms. Following Gray [4], we identify the most prevalent 20 questions covering the three stages. An example is shown below, and two more appear in appendix A.

Question. What is the characteristic performance of instances of class PEa on DIVM1 compared with its performance on DIVM2?
Usage. To find out the most suitable DIVM to enact a PEI. The query will retrieve all of the previous execution records of a PEI on all of the DIVM, filtered by DIVM\textsubscript{1} and DIVM\textsubscript{2}.

```
SELECT AVG(PerfOfPEInstance.time_per_unit), MIN(PerfOfPEInstance.time_per_unit),
       MAX(PerfOfPEInstance.time_per_unit), COUNT(DIVMInstance.instance_id),
       DIVMInstance.instance_id
FROM PerfOfPEInstance, PEInstance, DIVMInstance, PEInstallation
WHERE PerfOfPEInstance.instance_id = PEInstance.instance_id
  AND PEInstallation.instance_id = PEInstance.instance_id
  AND PEInstallation.install_on = DIVMInstance.instance_id
  AND PEInstance.class_id = 'PEa'
  AND (DIVMInstance.instance_id = 'DIVM1' OR DIVMInstance.instance_id = 'DIVM2')
GROUP BY DIVMInstance.instance_id
```

This question focuses on the enactment behaviour of the PEIs and tries to validate the effect of DIVM selection on the enactment performance. The last question in appendix A attempts to understand the cohabiting behaviour of PEIs as that should affect the mapping decisions. The information gained from the study of enactment behaviour of the PEs is crucial for constructing a cost model to analyse the performance of previous enactments and to predict the performance of future enactments. For instance, data from previous enactments show that mapping PEI\textsubscript{j} and PEI\textsubscript{k} onto the same DIVM\textsubscript{1} will result in low performance owing to central processing unit (CPU) cycle competition among PEIs. However, the data also indicate that the transfer rate between DIVM\textsubscript{1} and DIVM\textsubscript{2} is low. To find the optimum mapping of PEIs onto DIVMs, we must perform a cost analysis over the potential mappings. The PDB can be used to estimate the parameters used in the cost model for such analyses. Our ultimate goal is to dynamically optimize enactment based on performance data gathered from previous runs.

\textit{(d) Discard stage: cleaning performance database data}

The size of the PDB is expected to grow rapidly. The number of rows inserted into the \textit{Events} table is proportional to the number of input and output data streams, and how frequently a time-stamping is performed. To sustain the
Figure 5. DIVM abstraction in the prototype implementation. (a) DIVM abstraction and (b) example set-up of two ADMIRE sites.

The performance of the PDB, a cleaning process is needed to remove out dated or less important data. The PDB is cleaned in three ways: (i) by removing the raw data associated with derived data (e.g. clean up the Events table after data have been digested and added into PerfOfPEInstance), (ii) by removing data associated with deprecated versions of a class, and (iii) by removing data that are obsolete (e.g. pertaining to a discontinued DIVM or after a predefined number of days).

3. Prototype implementation

The prototype has been implemented in the ADMIRE project (http://www.admire-project.eu/) — a European-funded project that is pioneering architecture and models that deliver a coherent, extensible and flexible framework to facilitate exploration and exploitation of data.

Figure 5a shows the DIVM abstraction in the prototype implementation. We have reduced the software and hardware stacks for the prototype to four layers. The only hardware layer in the current abstraction is the processor. We assume that the layers below the processor, e.g. network, are kept in separate systems. As for the prototype, we consider that each processor is running a single operating system on a single virtual machine, thus the virtual machine layer is immediately on top of the processor layer. The virtual machine encapsulates a packaging of hypervisor, operating system and Java virtual machine and in future the model may need to characterize data-storage architecture and communication architecture. Each virtual machine can run one or more services, e.g. OGSA-DAI engine (ODE), ADMIRE gateway or data source. Figure 5b illustrates an example set-up of two ADMIRE sites. Site$_a$ consists of DIVM, which serves as a gateway and hosts a data source. The gateway is connected to another gateway running on DIVM$_g$ at a remote site, site$_g$. Site$_b$ comprises a gateway (DIVM$_b$), a data source (DIVM$_d$) and $n$ ODEs ([DIVM$_{k1}$, DIVM$_{k2}$, ..., DIVM$_{kn}$]). The Registry hosted in Site$_b$ keeps all the semantic information about the PEs and data sources that are used for type validation and optimization [6,7]. The type information includes the structure of the value streaming through a connection. Each sites has a PDB that keeps the performance data collected from the local enactments.
Performance database

(a) Collecting data from configuration logs

There are three different types of services in the example shown in figure 5b, and the configuration logs are collected as follows:

— When an ADMIRE gateway is started, the gateway start-up script will detect on which processor (machine) it is hosted, e.g. DIVM_g.
— When an ODE is started, the ODE will determine which processor (machine) it is running on, e.g. DIVM_k.
— To capture when a data source such as a MySQL daemon is launched, we have to add additional commands to the start-up script to capture the data, e.g. DIVM_d.

(b) Collecting data from the measurement framework

In an earlier section, we proposed a logical design of the MF that uses three types of observer to collect performance data: type observer, rate observer and buffer observer. The type observer is applied during the Data-Intensive System Process Engineering Language (DISPEL) processing stage [7]. DISPEL is a language developed in the ADMIRE project which is processed to generate a graph for optimization and enactment. 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 enactment stage. We collect these data by observing the pipe buffer implementation in the data stream. During the enactment, the data producer of a data stream writes (W) the data into the buffer, while the consumer reads (R) data from it. Both operations will trigger different events. Another two interesting events to record are blockings from read and write (RB and WB). The MF collects data for the following aspects of the enactment:

— Requests: a unique identifier for every workflow and a Unix time stamp indicates when the workflow request is received.
— Activities: the PEs and the PEIs used in every workflow. In the prototype, PEs are implemented as OGSA-DAI activities and executed in ODEs, thus, the PEs’ names appear in this log as OGSA-DAI activities. Every PEI is given a unique identifier. PEIs are connected with data streams (OGSA-DAI pipes). The information about the sources and targets of every data stream are stored separately: pipes–sources and pipes–targets, which allow the MF to store precisely when every buffer event occurs in both source and target of any given data stream.
— Activities–events: the states of every PEI i.e. PENDING (PEI is initiated and is waiting for execution), STARTED (PEI has been picked up by one of the threads and execution has started) and COMPLETED (execution has finished).
— Pipes: the sources and the targets of data streams (outputs of PEIs) in every workflow. Each data stream (pipe) is given a unique identifier.
Table 1. Information extracted from the events log.

<table>
<thead>
<tr>
<th>Information extracted</th>
<th>Operations for extracting information</th>
</tr>
</thead>
<tbody>
<tr>
<td>total units of data flow in a data stream</td>
<td>count of W events of the pipe-source or count of R events of the pipe-target</td>
</tr>
<tr>
<td>time spent in reading from/writing to a data stream</td>
<td>the reading time is the average difference between successive R events; writing time is similarly determined. An intervening WB or RB invalidates that particular interval</td>
</tr>
<tr>
<td>average data production time of a source PEI</td>
<td>the interval between first and last W event divided by number of W events minus 1</td>
</tr>
<tr>
<td>data production rate of a source PEI</td>
<td>total units of data [\text{PEI execution time} \times \frac{1}{\text{average data production time}}] or [\frac{\text{PEI execution time}}{\text{average data production time}}]</td>
</tr>
<tr>
<td>average data consumption time of a target PEI</td>
<td>same as average data production time but observing R and RB events</td>
</tr>
<tr>
<td>data consumption rate of a target PEI</td>
<td>same as above</td>
</tr>
<tr>
<td>compare the data production rate of source PEI and data consumption rate of target PEI</td>
<td>observe the number of WB and RB events in the data stream. Faster data production rate of source PEI can be identified when more WB events are observed, and vice versa</td>
</tr>
</tbody>
</table>

— Pipes-events: the Unix time stamp for every buffer event on both pipes-sources and pipes-targets. Four types of events are recorded: W (pipe-source write data), R (pipe-target read data), WB (pipe-source is blocked from writing data) and RB (pipe-target is blocked from reading data).

PEIs and data stream identifiers are unique and never re-used so that instances in successive requests are not conflated in queries against the PDB. From these enactment data, we extract the information shown in table 1. We have automated the instrumentation by embedding the MF in the OGSA-DAI implementation. During the enactment, event listeners are running on separated threads to capture the data stated above. There are two ways to organize these data: record in log files then load the records from these log files to populate the PDB after the enactment, or populate right away to the PDB hosted on a separate DIVM. The file-based logging was used first to decouple the PDB scheme design from the instrumentation design. Now these have stabilized direct updates to PDB have been implemented. Our next architectural step is to automate the analysis process in which the gateway that handled the enactment request will trigger the processing of the performance data after each enactment.

4. Demonstration of performance database usage

The previous sections describe the overall design of the PDB and how the prototype is implemented in the ADMIRE project. We have deployed the measurement probes in ADMIRE gateways and populated the PDB with
performance data collected during the enactment of real-world workflows. The main usage of the PDB is to bridge the gap between data collection and optimization. In this section, we evaluate how performance data for two workflows are used in extracting information for later optimization. The details about the optimization algorithm for streaming workflows are outside the scope of this paper and will be presented in a separate publication.

The first workflow is a data integration and transformation workflow that is frequently found across domains, as shown in figure 6. This workflow is a typical scientific workflow that involves: (i) retrieving data from distributed and heterogeneous data sources (retrieve data from a database using SQLQuery and streams of data from a remote data source using ObtainFromDataSource), (ii) merging two data streams (join two data streams using TupleMergeJoin), (iii) transforming the data streams (transforming the tuple output into XML WebRowSet format using TupleToWebRowSetCharArrays), and (iv) delivering result (write the result into a data source using WriteToDataSource). The data from the log files are extracted and populated into tables: WorkflowSubmit, PEInstance, PEInstallation and DataStream. Graphs in figure 7 are the event trace during the enactment which is extracted from querying these tables (note that the vertical scale is logarithmic).

The first few RB events correspond to the period when the PEIs have started and are trying to read data from their input data streams. Once SQLQuery has executed it starts to write data into its output data stream which is read 2ms later by TupleMergeJoin in input data1. After performing 20 writes, SQLQuery is blocked from writing (the default buffer size is 20 blocks of data). The blocking continues until TupleMergeJoin performs a read from the other input data2, and then SQLQuery performs its 21st W.

The optimization challenge is to identify which branch in the graph (subDAG) is causing the delay. One of the ways to identify this problem is by looking at the blocked events. We executed the query below to count the events that occurred during the enactment to plot the graph shown in figure 8 (note that the vertical scale is logarithmic).
When we traced the events that occur in all of the data streams, we find that \texttt{TupleMergeJoin} received data from two input streams and there is clearly a long wait for the arrival of data from one of the streams. Observe that the two data streams into \texttt{TupleMergeJoin} suffer a large number of delays.
write blocks and that the remaining streams suffer negligible read blocks. We can conclude that TupleMergeJoin has a relatively large unit cost (time to process a unit of data) because of the WB events recorded by its predecessors (SQLQuery and ByteArraysToTuple) and the RB events recorded by its successor (TupleToWebRowSetCharArrays).

Phil. Trans. R. Soc. A (2011)
The second workflow shown in figure 9 is used in automating gene expression annotation from the EURExpress-II project [8]. The current annotation is made manually by domain experts, and is costly and time-consuming (4 TB image data). Optimizing the enactment of this workflow is challenging because finding all of the mapping candidates of 43 PEIs onto DIVMs within multiple constraints (e.g. data source location, load balance among DIVMs) involves exhaustive computation. In general, some of the PEIs have a relatively small unit cost, e.g. ListRemove, Split and TupleSplit, when compared with FeatureGenerationArrayActivity. The assignment of these lightweight PEIs onto any DIVMs may not impose a significant workload. Thus, we need to define a performance threshold which divides the PEIs into two categories based on unit cost and handle the PEIs on either side of the threshold with different mapping approaches. In making the optimization decisions, PDB provides two crucial items of information: the unit cost of PEIs (time to process a unit of data) and the total units of data flow along data streams. We derive the unit cost, $t_{\text{cost}}$, from observing the data reading and writing time for PEIs as follows:

$$t_{\text{cost}} = \begin{cases} t_{w_i} - t_{r_i} & \text{for PEIs with single input and output} \\ \min(t_{w_i}) - \max(t_{r_i}) & \text{for PEIs with multiple inputs and outputs} \\ \max(t_{r_i} - t_{r_{i-1}}) & \text{for PEIs with input(s) only} \\ \max(t_{w_i} - t_{w_{i-1}}) & \text{for PEIs with output(s) only} \end{cases}$$

where $t_{w_i}$ is the time stamp taken when the PEI wrote the $i$th unit of data, $t_{r_i}$ is the time stamp taken when the PEI read the $i$th unit of data, $\max(t_{r_i})$ is the time stamp when the PEI read the last $i$th unit of data from one of the inputs, and $\min(t_{w_i})$ is the time stamp when the PEI wrote the first $i$th unit of data to one of the outputs. We then calculate the average unit cost for PEI$x$ that processes $n$ units of data, $t_{\text{cost}} = (\sum_{i=1}^{n} t_{\text{cost}_i})/n$. For instance, $t_{\text{cost}}$ for ImageToMatrixActivity is determined by the average time spent between reading and writing a unit of data. $t_{\text{cost}_i}$ for ClassificationActivity is calculated by subtracting the largest $t_{r_i}$ among the three inputs (group, image and data) from $t_{w_i}$. We then use these data to find the performance threshold.

The total units of data flow along data streams is important in making mapping decisions. Assigning two PEIs connected with a heavy data stream onto separate DIVMs will incur a large communication cost in moving the data across DIVMs. For instance, FeatureGenerationArrayActivity produces features as a two-dimensional array for every image it receives. The size of the data written by this PEI depends on the resolution of the image and the number of images that need to be processed. During the experiment, FeatureGenerationArrayActivity read 19200 images with size 190 $\times$ 118 pixels and generated 19200 double[190][118] matrices for the succeeding PEIs. Thus, a good optimization choice is to avoid assigning these PEIs on separate DIVMs unless the competition for the CPU or RAM outweighs the transfer cost. We have achieved a linear speed-up in optimizing this workflow, as reported in Liew et al. [2] and Han et al. [9].

5. Related work

Using workflow systems to organize the enactment of scientific experiments is becoming increasingly common and a wide range of workflow systems are
available. Deelman et al. [10] propose a taxonomy of workflow features and review existing workflow systems. Some of the workflow systems, e.g. Condor DAGman [11] and Taverna [12], provide monitoring services to monitor events corresponding to the state transition of processing entities. This information is mainly used for fault tolerance and not for optimization. Our work looks into fine-grained performance monitoring of processing entities including timing information of every single unit of data in the data stream.

Similar to our work, ASKALON [13] monitors the status of execution and provides feedback for optimization. The execution engine allows users to observe various metrics of the execution progress. ASKALON has developed a performance analysis system and a formal overhead analysis model that is useful in estimating task execution times and data transfer in a distributed environment. Instead, we focus on buffering analysis at data-stream level and performance data collected from previous runs to estimate the enactment time.

ParaTrac [14] is a fine-grained profiler that provides an effortless way to extract low-level input/output (I/O) profiles by looking deep into user-level file system and process-tracing techniques. It provides a more accurate study of workflow characteristics and supports fine-grained and realistic scheduling of workflows. However, its file I/O profiling technique is not applicable to streaming workflows. In contrast, our proposed MF looks at the buffer implementations for data streams and keeps track of every buffer event.

DIPAS [15] is a distributed performance analysis service that supports tracing execution, analysing performance overheads and searching for performance problems of Web service-based workflows in the Grid. It can be easily integrated with other Grid workflow middleware services through a loosely coupled Web services resource framework (WSRF)-based interface. Again, the main differentiation is that our work focuses on streaming workflow.

6. Conclusion and future work

The data-streaming model allows overlapping execution of computing elements, which shortens the overall response time of workflow enactment. The data streaming avoids writing intermediate values to disk except when buffers overflow RAM. It therefore can process requests with large-scale data by an efficient implementation of buffering in the main memory where the processing speeds of memory access outperform the disk by a factor of more than $10^5$ [16]. Thus, the streaming model enables users to process large-scale data with modest computing facilities. However, organizing the enactment of streaming workflows is difficult because it needs to allocate the whole workflow at once, which requires a good understanding of each PEI’s enactment behaviour.

The present work aims to solve this problem by introducing a PDB, where information is extracted from performance data gathered in previous runs. We proposed the performance data life cycle and discussed the rationale for the PDB. We presented a systematic way to implement the PDB with a novel approach to organizing performance data for streaming workflows. We presented an initial design for its schema and illustrated its power with example queries. We built a prototype as part of the ADMIRE project and used it to collect

Phil. Trans. R. Soc. A (2011)
performance data for real-world use cases. We showed phenomena in the derived data that are highly relevant to understanding data streaming and to optimizing workflows. We have then discussed how these data can be used in making optimization decisions.

The value of the PDB depends on two criteria: stability and recognition. The former is related to the consistency of the PDB behaviour across application domains, users, scales and execution environments, while the latter is related to how the PDB identifies and supports numerical characterization of relevant performance phenomena. The data to date are based on small-scale workflows from a few domains and enacted on workstations. We will extend the range of experiments to a representative set of applications and scales. We will then refine the queries and the architecture to handle the new issues exposed. We would be pleased to cooperate in the extension of the PDB to serve any workflow systems and optimization. Work is underway to use the PDB to validate models of streaming data performance and to use these models with parameters from the PDB to optionally deploy data-streaming enactments using distributed data on heterogeneous platforms.

The work presented in this paper is supported by the ADMIRE project (funded by EU FP7-ICT-215024) and the e-Science Core Programme Senior Research Fellow programme (funded by the UK EPSRC EP/D079829/1). We also acknowledge our very helpful colleagues in the OGSA-DAI team.

Appendix A. Example questions

1. What is the performance of instances of $\text{PE}_a$ compared with instances of $\text{PE}_b$?
   Usage: To choose between two PEs with equivalent functionality.

   ```sql
   SELECT AVG(PerfOfPEInstance.time_per_unit),
         MIN(PerfOfPEInstance.time_per_unit),
         MAX(PerfOfPEInstance.time_per_unit),
         COUNT(PEInstance.class_id), PEInstance.class_id
   FROM PerfOfPEInstance, PEInstance
   WHERE PerfOfPEInstance.instance_id = PEInstance.instance_id
   AND (PEInstance.class_id = 'PEa' OR PEInstance.class_id = 'PEb')
   GROUP BY PEInstance.class_id
   ```

2. Compare the performance of instances of $\text{PE}_a$ depending on whether or not there is an instance of $\text{PE}_b$ in the same DIVM.
   Usage: To find out whether co-locating two PE instances on the same DIVM will result in loss of performance. This query can guide the decision of whether to split the workflow across multiple DIVMs to speed up the performance.
   If two instances are associated with the same request_id, then we can infer that they coexist at the same time and potentially overlap in their use of a DIVM. This is achieved with the following queries:
   Construct a view, $\text{PEIonDIVM}$ that joins data from related tables.
CREATE VIEW PEIonDIVM AS
SELECT Class.name AS pe, PEInstallation.instance_id AS pei,
     PEInstallation.request_id, DIVMInstance.name AS divm,
     PerfOfPEInstance.time_per_unit
FROM PEInstance, PEInstallation, PerfOfPEInstance, Class, DIVMInstance
WHERE PEInstallation.instance_id = PEInstance.instance_id
    AND DIVMInstance.instance_id = PEInstallation.install_on
    AND PEInstance.class_id = Class.class_id
    AND PerfOfPEInstance.instance_id = PEInstance.instance_id;

Construct a view, Co_located to compute a subset of PEIonDIVM where the request_id and divm are equal.

CREATE VIEW Co_located AS
SELECT DISTINCT a.pe, a.pei, a.time_per_unit, a.request_id,
              a.divm, 'TRUE' AS co_located
FROM PEIonDIVM a
INNER JOIN PEIonDIVM b
ON a.request_id=b.request_id AND a.divm=b.divm
WHERE (a.pe='PEa' AND b.pe='PEb') OR (a.pe='PEb' AND b.pe='PEa');

Construct another view, Not_co_located to compute a subset of PEIonDIVM where the request_id and divm are not equal.

CREATE VIEW Not_co_located AS
SELECT pe, pei, time_per_unit, request_id, divm, 'FALSE' AS co_located
FROM PEIonDIVM
WHERE pe='PEa' AND pei NOT IN (SELECT pei from Co_located)
UNION
SELECT pe, pei, time_per_unit, request_id, divm, 'FALSE' AS co_located
FROM PEIonDIVM
WHERE pe='PEb' AND pei NOT IN (SELECT pei from Co_located);

Finally, compute the performance from the UNION of both views above.

SELECT AVG(u.time_per_unit), MIN(u.time_per_unit), MAX(u.time_per_unit),
       u.pe, u.co_located, COUNT(u.pe)
FROM (SELECT * FROM Co_located
     UNION
     SELECT * FROM Not_co_located) AS u
GROUP BY pe, co_located;
References


