Dataflow Processing and Optimization on Grid and Cloud Infrastructures[*]

M. Tsangaris, G. Kakaletris, H. Kllapi, G. Papanikos, F. Pentaris,
P. Polydoras, E. Sitaridi, V. Stoumpos, Y. Ioannidis
Dept. of Informatics & Telecom, MaD*g*IK Lab, University of Athens, Hellas (Greece)
{mmt,gkakas,herald,g.papanikos,frank,p.polydoras,evas,stoumpos,yannis}@di.uoa.gr
http://madgik.di.uoa.gr/

Abstract

Complex on-demand data retrieval and processing is a characteristic of several applications and combines the notions of querying & search, information filtering & retrieval, data transformation & analysis, and other data manipulations. Such rich tasks are typically represented by data processing graphs, having arbitrary data operators as nodes and their producer-consumer interactions as edges. Optimizing and executing such graphs on top of distributed architectures is critical for the success of the corresponding applications and presents several algorithmic and systemic challenges. This paper describes a system under development that offers such functionality on top of Ad-hoc Clusters, Grids, or Clouds. Operators may be user defined, so their algebraic and other properties as well as those of the data they produce are specified in associated profiles. Optimization is based on these profiles, must satisfy a variety of objectives and constraints, and takes into account the particular characteristics of the underlying architecture, mapping high-level dataflow semantics to flexible runtime structures. The paper highlights the key components of the system and outlines the major directions of its development.

1 Introduction

Imagine you have developed an innovative web-based search service that you would like to offer to the world. Cloud Computing enables you to host this service remotely and deal with scale variability: as your business grows or shrinks, you can acquire or release Cloud resources easily and relatively inexpensively. On the other hand, implementation and maintenance of data services that are scalable and adaptable to such dynamic conditions becomes a challenge. This is especially the case for data services that are compositions of other, possibly third-party services (e.g., Google Search or Yahoo Image Search), where the former become data processing graphs that use the latter as building blocks (nodes) and invoke them during their execution. Running services under various quality-of-service (QoS) constraints that different customers may desire adds further complications. Handcrafting data processing graphs that implement such services correctly, make optimal use of the

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[*]Partially supported by the European Commission under contracts INFRA-2007-212488, FP6-IST-027749, and FP6-IST-004260, for the D4Science, Health-e-Child, and DILIGENT projects, respectively.

resources available, and satisfy all QoS and other constraints is a daunting task. Automatic dataflow optimization and execution are critical for data services to be scalable and adaptable to the Cloud environment.

This is in analogy to query optimization and execution in traditional databases but with the following differences: component services may represent arbitrary operations on data with unknown semantics, algebraic properties, and performance characteristics, and are not restricted to come from a well-known fixed set of operators (e.g., those of relational algebra); optimality may be subject to QoS or other constraints and may be based on multiple diverse relevant criteria, e.g., monetary cost of resources, staleness of data, etc., and not just solely on performance; the resources available for the execution of a data processing graph are flexible and reservable on demand and are not fixed a-priori. These differences make dataflow optimization essentially a new challenging problem; they also generate the need for run-time mechanisms that are not usually available.

This paper presents **ADP** (Athena Distributed Processing), a distributed dataflow processing system that attempts to address the above challenges on top of Ad-Hoc Clusters (physical computer nodes connected with a fast local or wide-area network), Grids [5], and Clouds [9], each time adapting itself to the particular characteristics or constraints of the corresponding architecture. These architectures do not represent arbitrary unrelated choices, but can be considered as distinct points in an evolutionary path. While an Ad-Hoc cluster simply provides raw compute power, the Grid additionally provides mechanisms for managing computational, storage, and other resources in a synergistic way. When evolving from Grids to Clouds, additional resources are made available for lease, offering opportunities for more complex systemic scenarios, but also making service scalability, and performance, and composability even more challenging.

The paper begins with the internal representations of ADP queries. It continues with the runtime system of ADP, the stand-alone representations of operator properties, and the key features of its query optimization. It concludes with the implementation status of ADP, a comparison with related work, and some future directions.

2 ADP Query Language Abstractions

User requests to ADP take the form of queries in some high-level declarative or visual language, not described here. Internally, they are represented by equivalent queries in procedural languages at various abstraction levels:

Operator Graphs: These are the queries in **ADFL** (Athena Data Flow Language), the main internal ADP language. Their nodes are data **operators** and their (directed) edges are operator interactions in the form of producing and consuming **dataflows** (or simply **flows**). Operators encapsulate data processing algorithms and may be custom-made by end users. Flows originate from operators, are transformed by operators, and are delivered as results by the root operator of a query. A flow is a *finite sequence* of **records**. ADP treats records as abstract data containers during processing. Their properties (e.g., type name, type compatibility, keys, size) are stored in **record profiles** and play an important role when establishing operator-to-(operator or end-user) flows.

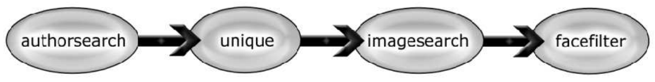

Figure 1: The Query Operator Graph

Example: Suppose a user wants to search the Web for images of authors of IEEE and ACM papers. Figure 1 shows an ADFL query that corresponds to this need. The query is essentially a chain (composition) of operators: First, there is a custom operator, AUTHORSEARCH, communicating with some particular external digital-library service to retrieve author names, filtered to select only authors of IEEE and ACM papers. Then, the standard operator UNIQUE eliminates duplicate author names. Next, the resulting flow of author names is sent to another custom operator, IMAGESEARCH, which uses the names to identify corresponding images in an external database. Finally, a face detection operator, FACEFILTER, is used to identify images that contain only faces and forced to output only the best match (topk=”1”). In text form, the query in Figure 1 is expressed as

FACEFILTER{topk=”1”} ON IMAGESEARCH ON UNIQUE ON AUTHORSEARCH{pub=”IEEE” or ”ACM”} □

Concrete Operator Graphs: These are similar to operator graphs but their nodes are **concrete operators**, i.e., software components that implement operators in a particular way and carry all necessary details for their execution. The UNIQUE operator, for example, has to store all records (or record keys) seen so far; it may be realized as two alternative Java implementations, one based on main memory (fast but limited by memory size) and one based on external storage (slower but limited only by disk size). Choosing among them becomes an optimization decision, based for example, on expected input flow sizes.

Subject to its initialization, as part of its execution, a concrete operator may be contacting external services to retrieve data from them. In that case, it must deal with all physical, security, and semantic issues related to such external communication. This is transparent, however, to the operator that consumes the flow resulting from such external services: all sources of flows appear the same, independent of any external interactions.

Execution Plans: These are similar to concrete operator graphs, but their nodes are concrete operators that have been allocated resources for execution and have all their initialization parameters set.

3 ADP Runtime Environment

Concrete operator graph queries are eventually evaluated by the **ART** (ADP Run Time) subsystem, which has two main software parts: **Containers** are responsible for supervising concrete operators and providing the necessary execution context for them (memory resources, security credentials, communication mechanisms, etc.). **ResultSets** are point-to-point links to transport polymorphic records between concrete operators and implement the query flows. While different manifestations of data are supported, e.g., native objects, byte-streams, or XML content, ResultSets are type agnostic.

Containers are the units of resource allocation between ART and the processing nodes of the underlying distributed infrastructure. Based on the optimizer’s decisions, ART dynamically creates or destroys containers to reflect changes in the system load. Thus, a complex query may be distributed across multiple containers running on different computer systems, each container being responsible for one or more of the query’s concrete operators and ResultSets. Likewise, based on the optimizer’s decisions, ResultSets control the mode of record transportation, ranging from pipelining (totally synchronous producer/consumer operation) to store & forward (full buffering of entire flow before making it available to the consumer).

Containers feature the same set of tools to support ADFL query evaluation but are implemented differently depending on the characteristics of the underlying distributed infrastructure architecture, hiding all lower-level architectural details and the corresponding technology diversity. Containers on Ad-Hoc clusters or Clouds, for example, may simply be just processes, while on Grid, they may be Web Service containers. ResultSets may utilize WebService transport (SOAP) on Grids, or simply TCP sockets on Ad-Hoc Clusters or Clouds. The optimizer may try to minimize Cloud resource lease cost by shutting down some or not using several containers, while this is not an issue in other architectures.

The runtime mechanisms provided for query execution have a major impact on application development in that they liberate implementers from dealing with execution platform or data communication details. Note that there is a particularly good match between Cloud architectures and certain characteristics of ADP: Custom operators within ADFL queries are an easy and attractive way to use ad hoc third-party services (e.g., AUTHORSEARCH or FACEFILTER), which is an important feature of Clouds. More importantly, dynamic acquisition and release of resources (containers and virtual machines) by ADP as a systemic response to load or QoS requirements fits very well with the canonical Cloud business model; the presence of Service Level Agreements (SLAs) that must be met requires such flexible resource allocation, which in turn, calls for sophisticated optimization of the kind ADP is designed to offer. The need for advanced optimization strategies is less marked in other architectures, e.g., in Grid, where simple matching of operations to resources usually suffices.

4 Operator Profiles

Given the ad hoc nature of most ADP operators, no pertinent information about them is hardwired into the system; instead it is all provided by users and stored in **operator profiles**. Typically, for each level of internal language abstraction, there is relevant information in an operator's profile. Accordingly, ADP uses the profile contents recursively to drive the corresponding stages of query optimization and execution. Below, we indicate some fundamental properties that may be found in (or derived from) an operator's profile for each abstraction level, emphasizing primarily those that generate equivalent alternatives that the optimizer must examine when a query with that operator is considered. We avoid describing the precise structure/schema of the profile or the language used to express some of its contents; instead, we use a stylized pseudo-language for easy exposition.

Operator Graphs: At this level, in addition to its signature (input/output flows with specific record profiles), of great importance are algebraic equivalences that operators satisfy. These include typical **algebraic transformations**, e.g., associativity, commutativity, or distributivity, **(de)compositions**, i.e., operators being abstractions of whole operator graphs that involve compositions, aggregations, and other interactions of more specific operators, and **partitions**, i.e., operators being amenable to replication and parallel processing by each replica of part of the original input, in conjunction with some pre- and post-processing operators.

Example: Consider the following information being known about the operators of Figure 1:

- Algebraic transformation - Filtered on multiple publishers, AUTHORSEARCH is equivalent to merging the results of itself filtered on each one of them separately:
  operator AUTHORSEARCH{pub=x or y} is MERGE on AUTHORSEARCH{pub=x} and AUTHORSEARCH{pub=y};
- Operator decomposition - Filtered on IEEE or ACM, AUTHORSEARCH is equivalent to another operator that searches directly the IEEE or ACM Digital Libraries, respectively:
  operator AUTHORSEARCH{pub="IEEE"} is IEEESEARCH;
  operator AUTHORSEARCH{pub="ACM"} is ACMSEARCH;
- Operator partition - FACEFILTER is trivially parallelizable on its input, with operators SPLIT and MERGE performing the necessary flow pre- and post-processing, before and after the parallel execution of an arbitrary (unspecified) number of FACEFILTER instances:
  operator FACEFILTER is splitable with {pre-process = SPLIT ; post-process = MERGE}; □

Concrete Operator Graphs: At this level, capturing an operator's available implementation(s) is the critical information. In general, there may be multiple concrete operators implementing an operator, e.g., a low-memory but expensive version and a high-memory but fast one; a multi-threaded version and a single-threaded one; or two totally different but logically equivalent implementations of the same operator. For example, there may be a standard UNIQUE implementation determining record equality based on the entire record, while an alternative custom implementation may only look at a specific key record attribute. Also, IMAGESEARCH may have just a multi-threaded implementation associated with it, but FACEFILTER may have both a single-threaded and a multi-threaded one. All these concrete operators should be recorded in the corresponding operator's profile.

Execution Plan: At this level, the profile of a concrete operator stores information about its multiple potential instantiations in a container, its initialization parameter values, and any constraints on resources it may use, e.g., number of threads, size of memory, software licenses, input/output rates, communication channels, etc. It also stores information about how the optimizer may evaluate a particular instantiation of the concrete operator. For example, the multi-threaded concrete operators for IMAGESEARCH and FACEFILTER have several additional degrees of freedom at the execution plan level, as they can use multiple local CPUs and cores.

5 Query Optimization

Evaluation Parameters: Evaluation of query execution plans is at the heart of query optimization, regarding both the objective function being optimized and any (QoS or other) constraints being satisfied. Depending on the application, such evaluation may be based on a variety of parameters, e.g., monetary cost of resources or freshness of data, and not just solely as is traditional on performance metrics. Given the ad hoc nature of operators, their profiles store mathematical formulas to describe such parameters and any properties of their inputs and outputs that are deemed relevant, e.g., image resolution for FACEFILTER cpu cost, or image database age for IMAGESEARCH freshness. Consequently, for any parameter that may be important to the operators' evalution, statistics should be either maintained or obtained, for example, on the fly through some sampling. Similarly, the mathematical formulas associated with the evaluation of an operator may be either explictly inserted into its profile by some user or predicted based on some sample or prior executions of the operator. ADP is designed to offer generic functionality for synthesizing appropriate formulas and propagating parameter values through the operators of an execution plan to obtain its final evaluation.

Space of Alternatives: Transformation of an ADFL query to an execution plan that can generate the requested results goes through several stages, corresponding to the levels of internal language abstractions, where every alternative in one level has multiple alternative mappings to the next lower level according to the properties in the profiles of the operators involved. There are several operator graphs that are algebraicly equivalent to the original query, each one mapping to several concrete operator graphs (based on the corresponding mappings of its operators), each one mapping to several execution plans by instantiating containers and ResultSets and assigning the instantiated concrete operators and flows of the concrete operator graph to them.

Example: The algebraic properties in the profiles of AUTHORSEARCH and FACEFILTER (assuming 3-way parallelism for the latter) generate the operator graph indicated in Figure 2 as an alternative to Figure 1. Instantiating an execution plan for that graph requires choosing concrete operators and then: container instantiation - the set of containers available to the query are chosen, through dynamic release or acquisition of containers and (virtual) hosts, or reuse of existing ones; concrete operator instantiation - all concrete operators are initialized (e.g., the number of threads for the multi-threaded implementations of IMAGESEARCH and FACEFILTER is set) and assigned to containers; flow instantiation - connected as inputs and outputs of concrete operators, the endpoints of each flow are instantiated via technology-specific endpoint implementations of the ResultSet, fine-tuned for the flow's and connected operators' needs. Figures 2 and 3 indicate particular alternatives with respect to these choices, at the level of the operator graph and the execution plan, respectively. □

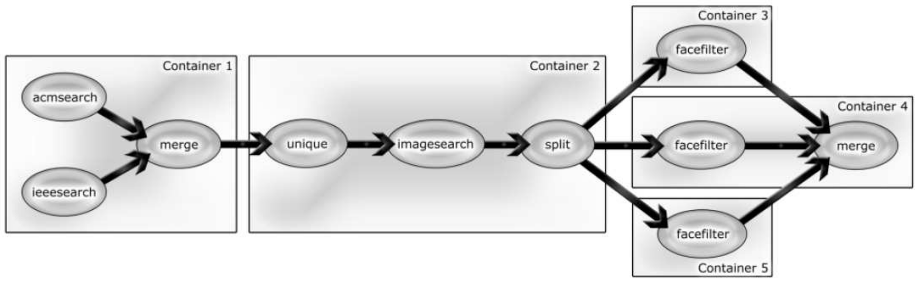

Figure 2: Query Operator Graph after all operator transformations and assignments to containers

Optimization Stages: In principle, optimization could proceed in one giant step, examining all execution plans that could answer the original query and choosing the one that is optimal and satisfies the required constraints. Alternatively, given the size of the alternatives' space, optimization could proceed in multiple smaller steps, each one operating at some level and making assumptions about the levels below. ADP optimization currently proceeds in three distinct steps, corresponding exactly to the three language abstraction levels of ADP.

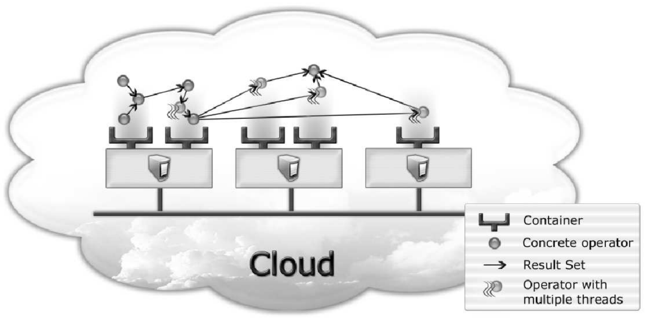

Figure 3: Query Execution Plan with explicit mapping of operators to containers on Cloud (virtual) hosts

The techniques developed for the first two steps are not discussed here due to space limitations.

Execution Plan Instantiation: Assignment of concrete operators to containers is currently modeled and implemented in ADP as a constraint satisfaction problem (CSP) as follows: Consider N containers $L_1, L_2, \ldots, L_N$, each with a current resource capability $CAP(L_i), 1 \leq i \leq N$, and a host $H(L_i)$ where it resides. Let $NETCAP(L_i, L_j)$, $1 \leq i, j \leq N$ be the network capacity (bandwidth) between $H(L_i)$ and $H(L_j)$. Let $P_1, P_2, \ldots, P_M$ be the set of concrete operators present in the respective operator graph and $DEMAND(P_i)$, $1 \leq$ i $\leq M$, be the resources that operator P_i requires. The CSP solved is to assign each concrete operator to a container ($C(P_i) = L$) so that a user-defined cost function (e.g., $\sum_{1 \leq i,j \leq M} NETCOST(P_i, P_j)$, network cost between operators) is optimized subject to the constraints:

- container resource capability is not exhausted: $\sum_{C(P_i)=L} DEMAND(P_i) \leq CAP(L)$,
- internode bandwidth is not exhausted: $\sum_{C(P_i)=L,\ C(P_j)=L'} NETCOST(P_i, P_j) \leq NETCAP(L, L')$.

The above problem can naturally be expanded to model more complex situations, e.g., taking into account operator gravity (preference for operators to be assigned to the same container), or different optimization objectives.

In high-load conditions, we deploy an admission control algorithm to ensure optimal balancing of workload. Before entering the CSP solution process, the optimizer broadcasts information on the concrete operator graph and “asks” containers to declare which operators they can instantiate (if asked to do so). Containers monitor these requests as well as the actual optimizer decisions and use this information to restrict the number and type of concrete operators they are willing to instantiate. Each container makes potentially a different decision, which are all then used to restrict the space of possible CSP solutions. If the existing containers’ decisions do not allow all concreted operators of a query to be instantiated so that an execution plan may be obtained, then the query is automatically resubmitted after a short time expecting greater container availability. If this is not the case, additional containers are requested based on the number and type of concrete operators that are unassigned.

The algorithm used by containers to restrict the amount and type of operators they are willing to instantiate follows the lines of [7]. In critical load situations, it effectivelly diverts resources used by concrete operators that are rarely requested to the ones that are frequently used. This is achieved using the following microeconomics-inspired mechanism: Assuming that C is the number of concrete operators, containers internally hold a private vector $\vec{p} \in \mathbb{R}_+^C$ of virtual concrete operator prices. These prices are never disclosed; they only provide to the admission control algorithm the means to measure the contribution of a concrete operator to the performance of the whole distributed system. As demand for a concrete operator increases/decreases, its respective prices increase/decrease as well. Each container uses its privately held prices to periodically (every t units of time) select a vector $\vec{s} \in \mathbb{N}^C$ of operators to admit. This vector is different for each container, represents the type and number of operators admitted, and is the one that maximizes the virtual price ($\vec{s} \cdot \vec{p}$) of the admitted operators under the resource constraints of each container. That is, each container solves $\max_{\vec{s}} \vec{s} \cdot \vec{p}$ so that $\vec{s}$ is feasible, i.e., the container has enough resources to instantiate all conreate operators in $\vec{s}$ within t units of time.

6 Implementation Status

An initial implementation of ADP is in operation for a year now and is used by data mining and digital library search applications. The ADFL language enables ad-hoc operators to be introduced, without the need to change its parser. Operator profiles contain information such as the java classes implementing operators. ART uses Axis-on-Tomcat Web Service containers, each one running on an Ad-Hoc cluster node or on the Grid. The optimizer performs simple rewriting driven by operator profiles, changes the number of containers dynamically based on current “load”, and decides how to assign concrete operators to containers, as described above. A simulated annealing optimization engine is used to generate the execution plan, based on the concrete operator graph. ResultSets have been implemented, attached to the producing operator, and communicating via TCP sockets or SOAP messages to the consuming operators. Slotted Records model relational database table rows, whereas XML Records provide additional structure. Finally, in addition to the default execution engine, there is a second implementation based on BPEL (Business Process Execution Language), as well as a proof-of-concept “standalone ADP” implementation, which is a single java executable including an ADFL parser, ART, a single container, a library of operator implementations, and a ResultSet implementation.

7 Related Work

ADP provides both a testbed for validating research ideas on distributed data processing and a core platform for supporting several data-intensive distributed infrastructures. It has been influenced by lessons learned from on-going work on data services in the areas of Digital Libraries, e-Health, Earth Sciences, etc., which all need a scalable software layer that can perform compute-intensive data tasks easily, reliably, and with low application complexity. The ADP concepts have been validated in the context of the DILIGENT [12], Health-e-Child [14], and D4Science [13] projects, and form the heart of the gCube system’s information retrieval facilities [15]. Although several core ADP concepts can be found elsewhere as well, integrating them in one system and handling the resulting increased complexity does not appear common. ADP incorporates ideas from databases, streams, distributed processing, and service composition to address the relevant challenges and offer a flexible system that can hide the complexities of its underlying architectural environment.

Typically, some middleware is used to execute user-defined code in distributed environments. In the Grid, OGSA-DAI [2] formalizes access to and exchange of data. Paired up with OGSA-DQP [1], it also addresses query optimisation and scheduling. The Condor / DAGMan / Stork set is a representative technology of the High Performance Computing area. Its capacities for scheduling, monitoring and failure resilience render it a robust and easily scalable mechanism for exploiting vast scientific infrastructures of (mostly) computational resources. Furthermore, Pegasus [4] supports a higher lever of abstraction for both data and operations, and therefore offers true optimization features, as opposed to simple matching of operators to a fixed set of containers.

ADP builds on top of these technologies and introduces ADFL to describe user-defined code in a semantically, technologically and operationally domain agnostic manner. The definition of an Operator, and most importantly the Operator Profile, is by itself a new challenge, since traditional database operations like query re-writing, cost-estimation, completion times, selectivity, co-location capacities/requirements, etc., are not a-priori defined or not known at all. Distributed databases do support custom operations on data via functions or (extended) stored procedures and handle data exchange (i.e., flows) via efficient proprietary mechanisms, but optimization is based on established assumptions of (distributed) relational databases.

The notion of processing multiple dataflows in ADFL is also common in the literature. In its more recent form, Mashups (such as Yahoo! Pipes [16], Google Mashup Editor [10] and Microsoft Popfly [11]) carry out content processing over well known sources (RSS, ATOM, HTTP). The visual languages in these systems serve as a starting point for ADFL, which in addition, deals with alternative representations for Operator Profiles. In (e-)Business integration, workflow languages such as WS-BPEL are used to express complex queries that call for systems that support multiple execution granularities, planning, execution, and monitoring mechanisms, etc. Compared to these systems, ADP is designed to offer a rich query rewriting alternatives in the query optimizer. Finally, SawZall[8] and PigLatin [6] use a higher-lever query language that is executed on MapReduce [3] systems that support massive parallelization and achieve failure resilience. However, the language model of the MapReduce framework is somewhat restricted and restricts opportunities for optimization.

8 Conclusions and On-Going Work

We have given a high-level description of ADP, a distributed dataflow processing system under development, which is designed to run on top of Ad-Hoc Clusters, Grids, and Clouds, in an adaptive manner. It deals with dataflow queries that involve user-defined operators, stores the operators' properties in profiles, and uses those to optimize queries at several levels. Optimization may be based on diverse optimality criteria and constraints but currently focuses on the conventional cpu work parameters.

Work on ADP moves in several directions. These include expressive declarative languages on top of ADFL, mechanisms to deal with operators that preserve state, and fine-grade security. On the optimization side, the focus is on Cloud-related architectures, on refining dynamic resource acquisition and release, and on dealing with complex constraints on these resources. Additionally, the role of execution risk in ADP operators is being ivestigated, in scenarios where different plans are exposed to different execution risk profiles and users have different attitudes towards risk (e.g., risk aversion).

References

[1] M. N. Alpdemir, A. Mukherjee, N. W. Paton, P. Watson, A. A. A. Fernandes, A. Gounaris, and J. Smith, "Service-based Distributed Querying on the Grid," in *ICSOC*, 2003, pp. 467–482.

[2] M. Atkinson et al., "A new Architecture for OGSA-DAI," in *Proceedings of the UK e-Science All Hands Meeting 2005*, September 2005.

[3] J. Dean and S. Ghemawat, "Mapreduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[4] E. Deelman et al., "Pegasus: Mapping Large Scale Workflows to Distributed Resources in Workflows in e-Science," *Springer*, 2006.

[5] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *International J. Supercomputer Applications*, vol. 15, no. 3, 2001.

[6] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: a Not-so-Foreign Language for Data Processing," in *Proc. 2008 ACM SIGMOD Conference on Management of Data*, 2008, pp. 1099–1110.

[7] F. Pentaris and Y. Ioannidis, "Autonomic Query Allocation Based on Microeconomics Principles," in *Proc. 23rd Int'l Conf. on Data Engineering (ICDE)*, 2007, pp. 266–275.

[8] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the Data: Parallel Analysis with Sawzall," *Sci. Program.*, vol. 13, no. 4, pp. 277–298, 2005.

[9] L. Vaguero and et al., "A Break in the Clouds: Towards a Cloud Definition," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, 1 2009.

[10] "Google Mashup Editor," code.google.com/gme/.

[11] "Microsoft Popfly," www.popfly.com.

[12] "Project DILIGENT," 2004, www.diligentproject.org.

[13] "Project D4Science," 2007, www.d4science.eu.

[14] "Project Health-e-Child," www.health-e-child.org.

[15] "The gCube System," www.gcube-system.org.

[16] "Yahoo! Pipes," pipes.yahoo.com/pipes/.