Ersch. in: Search computing - challenges and directions / Stefano Ceri, Marco Brambilla (eds.). - Berlin [u.a.] : Springer, 2010. - S. 225-243. - (Lecture notes in computer science ; 5950). - ISBN 978-3-642-12309-2

Chapter 12: Panta Rhei: Flexible Execution Engine for Search Computing Queries

Daniele Braga, Stefano Ceri, Francesco Corcoglioniti, and Michael Grossniklaus

Dipartimento di Elettronica e Informazione, Politecnico di Milano,
Piazza Leonardo da Vinci 32, 20133 Milano, Italy
{braga, ceri, corcoglioniti, grossniklaus}@polimi.it

Abstract. The efficient execution of data-intensive computations over services is a challenging task: data are retrieved from remote sources and therefore are not available in the query engine until after the execution of these calls, but the system must be inherently efficient thereafter, by guaranteeing that data is immediately cached and processed efficiently, according to the best query plan. In this chapter, we present a flexible execution model for search computing queries, named Panta Rhei. The proposed execution engine paradigm adopts the producer/consumer model and supports both data-driven and event-driven synchronization, and their interplay. Query plans are modeled as directed graphs, whose nodes are processing units and whose edges are either control or data flows. While control flows synchronize service calls and unit execution, data flows transfer data between units that process data flows to produce query results. We present the specification of Panta Rhei by formally defining the units for data production, consumption, manipulation, and caching, as well as the control and data flows. Finally, we discuss how a query plan is expressed in terms of a query execution plan.

1 Introduction

Query execution in Search Computing is a data-intensive process. The computations required for answering a query, although performed upon the data resulting from service calls, are very similar to those performed by database management systems working on physically optimized tables. Therefore, a query execution engine supporting Search Computing must be able to efficiently support dynamic data extraction, storage and caching, as well as efficiently route data flows between special-purpose computational units, whose design has been optimized so as to guarantee the fast production of query results.

Due to the very nature of many of these tasks and their embedding within Web-based contexts, which are subject to continuous change, performances of data-intensive service interactions are very hard to predict. Moreover, the execution engine must be strongly connected to the query user interface, so as to adapt to user requests that dynamically alter the query requirements, either by specializing current requests or by adding new requirements. For these reasons, the design of the query execution engine for Search Computing has required several architectural solutions for supporting dynamic adaptation which are quite original, especially for what concerns the synchronization aspects.

The main operation in Search Computing is the join of search services and, therefore, the execution engine is optimized to support joins, under the constraint that join data operands are not immediately available to the execution engine, but are produced by interacting with services, ranked and separated in chunks. Join processing, as explained in the previous chapter, aims at exploring given compositions of chunks returned by the services. In this setting, optimization consists in minimizing the number of service calls and, at the same time, in efficiently exploring the search space so as to rapidly produce results.

Supporting join executions requires synchronizing pairs of services. To effect this synchronization, we introduce particular units, called clocks, whose effect is to give pulses to services so as to synchronize them according to certain mutual relationships that can be dynamically adapted. In order to respond to variability, synchronization is subject to feedbacks which are generated within the execution environment. The explicit (and user controllable) synchronization and adaptation of join computations through clock units is the most significant (and original) aspect of the execution engine, being used both for pipelined and parallel execution with a uniform style.

Original aspects of the execution engine concerns the explicit management of chunks within the data flow, which is at the basis of the design of both the chunker units (capable of changing the size of chunks along the data flow) and the cache units (which store the results of service calls by chunks). In SeCo joins, a given chunk of a service's results can be involved in many chunk combinations, performed after its initial loading, and cannot be discarded until query processing is completed. Chunk support allows for an intermediary granularity level, which is a good compromise between tuple-level (each tuple flows individually) and table-level (each data collection or table flows as a unit) granularity. We believe that this solution yields to a good trade-off between flexibility, adaptability, and performance.

While clocks and chunks are, therefore, the main ingredients of the flexible execution engine, many other features characterize its design. The system must, of course, support sorting (i.e. ranking of results) which is a critical operation, because it is "blocking" (in order for the sort to be applicable to a given collection, all the items of the collection must be available) and data flow machines must try to minimize blocking operations. In addition, the system should support the early evaluation of selection predicates in order to reduce the size of data flows.

The organization of this chapter is as follows. In Section 2 we present the state-of-art of data-driven execution engines, first by highlighting the issues which arise in interpreted environments (such as ours) and then by focusing on adaptability of computations, the main quality offered by Panta Rhei. Section 3 presents the model, with its nodes representing units and edges representing data and control flows. Then, Section 4 sketches the translation of query plans into query engine execution plans, and Section 5 shows the typical translations of parallel, pipe, and top-k joins into schedules.

2 State of the Art

This section gives an overview of the state of the art of query execution with a focus towards the domain of Search Computing. First, we discuss different query processing

paradigms that serve to position the proposed query execution environment for Search Computing. A distinguishing feature of a query evaluation paradigm is the degree of query plan adaptation that is supported by an execution environment. In the second part of this section, we motivate the need for query plan adaptation at run-time and give an overview of related work on adaptation in other application domains.

2.1 Query Processing Paradigms

An important criterion in the design of a query execution engine for Search Computing is its query processing paradigm. In the past, several types of query execution engines have been proposed in the scope of traditional DBMS, such as interpreted or compiled [19] execution engines. On the one hand, interpreted approaches translate queries into query plans that are optimized and evaluated leveraging a general-purpose set of operators provided by a virtual machine such as the query evaluator of a database management system. Compiled approaches, on the other hand, use code generation to translate each query into a static program that is compiled and executed natively, i.e. directly on the operating system. The main strength of compiled engines is their performance as all meta-information required for evaluating a query is directly hardwired into the program code. The gain in performance comes, however, at the price of flexibility. While compiled engines are fast, it is more difficult to cater for run-time adaptation of query plans as this would require a recompilation of the program while it is executing. Due to the requirements of Search Computing, we have, therefore, chosen to build an execution environment that follows the approach of an interpreted engine, and therefore we focus this state-of-the-art on interpreted query engines.

Interpreted engines can be further classified according to the query evaluation model that they use. Within interpreted engines, query execution plans require both control flow, which dynamically defines how engine modules are synchronized, and the data flows, which dynamically define data exchanges. From the viewpoint of data flows, components are characterized as producers and consumers and a query computation may involve several modules. At its beginning, a query plan involves producer modules, later intermediate components play both roles, and eventually query interfaces present their results to the user who is the "final consumer" of the system. Execution plan components, or "nodes", have four possible behaviors relative to control and data flows, presented by Graefe (see [12], p.149ff) and shown in Fig. 1.

- *Standard iterators*. In most query processing systems, the data flow is demand-driven and controlled by the consumer. In this case, control and data flow point into opposite directions. According to [17], most state of the art approaches for *distributed query processing* use the iterator model [13] in which all operators exhibit an *open()-next()-close()* interface.
- *Data-driven operators*. There are however systems such as real-time or data stream systems where the data flow is paced by the producer as it needs to unload the data as it arrives, e.g. sensor data. In a data-driven operator, control and data flow point into the same direction.

To combine demand-driven and data-driven operators, it is necessary to introduce *flow translation nodes* [12] that mediate between the two types of operators.

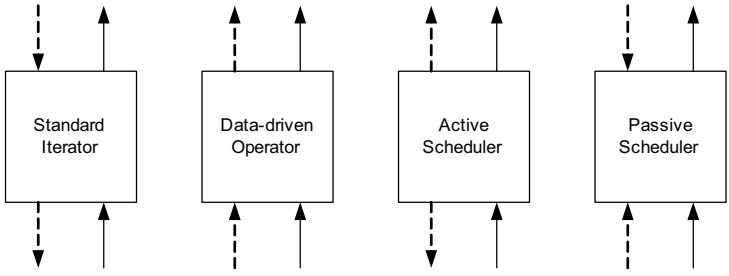

Fig. 1. Nodes of an execution plan with control and data flow [12]; control flows are dashed arrows and data flows are solid arrows

- *Active scheduler*. A data flow translation node that can be used to schedule a demand-driven operator (iterator) as producer and a data-driven operator as consumer. This node actively requests data from a demand-driven producer and a passes it on to the data-driven consumer.
- *Passive scheduler*. A data flow translation node that can be used to schedule a data-driven operator as producer and a demand-driven operator as consumer. As soon as the data-driven producer delivers data, this node accepts and buffers it until the data is requested from the eventually resumed demand-driven consumer.

Another important characteristic of the query processing paradigm is whether the execution is governed *centrally* by a global scheduler that has complete knowledge or in a *distributed* setup where the nodes of the execution engine make local scheduling decisions based on incomplete knowledge. For example, data flow systems [25] and stream processing systems, e.g. Aurora/Borealis [1], have addressed the problem of scheduling data-intensive computations. More recently, scheduling algorithms have been proposed that control the execution of a computation in peer-to-peer networks, such as the economic model [2] or approaches based on reinforced learning [20]. While data flow systems are designed for the execution of fine-grained computations, workflow systems [27] address coarser-grained processes executed over Web services. While these two families of systems are similar in terms of goals, the latter tends to use central scheduling that operates on global information.

2.2 Adaptation

The capability of adapting software systems to internal or external requirements is often referred to as *adaptation*. Adaptation can be effected at design or compile-time of a system as well as at run-time. The need for adaptation is present in many application domains and adaptation can be supported at very different levels of granularity. Therefore, the entire body of research on adaptation is very vast and its complete review out of the scope of this chapter. Instead we will limit the discussion to work that is relevant in the context of query execution in Search Computing and structure them according to the scope of their application, from coarse-grained to fine-grained. We will start with adaptation at the level of the architecture, then discuss the adaptation of applications and conclude by presenting solutions to adapt processes in particular data-driven computations.

On an architectural level, the focus on adapting processes lies on leveraging the resources that are at disposition best. To that end, load-balancing schemes or the dynamic reassignment of resources to computation nodes are techniques that are often employed in the area of distributed computing, such as grid or cloud computing. On a level of finer granularity, we note that the adaptation of applications is a requirement that frequently arises in mobile and ubiquitous computing as well as in Web engineering. Context-awareness is a solution often proposed to adapt applications to limited device resources, environmental factors or multiple output channels. In the area of context-awareness, work has also been done on context-aware data management and querying [14]. Finally, it is also possible to perform adaptation on the level of individual computations that can be both process-driven and data-driven. In the following, we will focus entirely on adaptation of data-driven computations such as query plan adaptation since this is most closely related to the query execution engine presented in this chapter.

Generally, query plan adaptation can be classified according to when it is taking place into *compile-time* and *run-time* adaptation. On a finer level, query plan adaptation can be refined further according to the information that is used as input to effect the optimization. We distinguish the types of input information given below and, in the following, discuss how each one of them can be used for adaptation.

- *Data statistics* such as the cardinality of tables and the selectivity of predicates.
- *Usage statistics* obtained through profiling of query execution or mining of query execution history to get dynamic statistics (self-tuning databases).
- *User control* that determines the adaptation of the query plan.

Clearly, some of these types of information are mostly used for compile-time adaptation, while others only make sense for run-time optimization. For example, data statistics are usually leveraged at compile-time by the optimizer to plan the execution of the query in the best possible way. While usage statistics are typically gathered at run-time, either using "pay-as-you-go" frameworks [5] or in separate mining processes, this information is also applied to the adaptation of the query plan at compile-time. As a consequence, the queries that are profiled or mined do themselves not profit from this information as only later executions of the same or similar queries are adapted accordingly.

Nevertheless, approaches that use data and usage statistics for supporting the dynamic reoptimization [16] and adaptation of query plans exist. Among those approaches are adaptive operators, query scrambling, the interleaving of query planning and execution, and opening up the query optimizer to application input. Adaptive operators, such as e.g. *choose nodes* [6], *XJoin* [23], or *BindJoin* [18], are query plan nodes that defer certain decisions until execution. In the former case, choose nodes select at run-time from a set of query sub-plans that was defined at compile-time. In the latter cases, the join implementations themselves are capable of adapting to delays at run-time. Another more dynamic approach for dealing with unexpected delays is *query scrambling* [24] that modifies the query execution plan on the fly based on heuristics. In approaches that use *interleaving*, e.g. [26] or [8], the optimizer only produces a partial plan for the execution engine and decides how to proceed once that partial plan has been evaluated. Finally, the author of [4] argues that future query optimizers should also benefit from rich usage data and application

input. Adaptive query execution systems for *data integration over the Web* address the problems of absence of statistics and unpredictable data arrival characteristics. Most of these systems combine novel approaches, e.g. incomplete query plans that are completed and (re)optimized incrementally [15] with existing concepts such as the previously discussed interleaving of query planning and execution as well as adaptive operators. Adaptive query processing approaches that leverage information captured through self-monitoring of the query execution have also been proposed for Grid computing [11].

Finally, *user control* as an input for process adaptation has been addressed in systems that allow performance and query execution to be expressed through interactive dashboards [7]. As most work on dashboards has been done by the HCI community, it largely addressed the interface level in terms of visualizing complex and large sets of information in a comprehensive and graspable way. Nevertheless, there are also approaches that focus on the evaluation of queries in the presence of a visual and interactive interface. For example, [22] shows how dynamic query interfaces can be supported in large databases through the use of incremental data structures and algorithms. The approach introduces the notion of an active subset of the database that is enhanced with auxiliary data structures designed to support continuous querying. These auxiliary data structures are directly coupled to the interface and are only reprocessed in the event of user interaction. Results are visualized incrementally by computing and displaying the delta resulting from the user input. In [3], a classification and survey of visual query systems for databases is presented.

3 Panta Rhei Specifications

While classic execution engines operate upon databases which are initially stored within the memory (possibly distributed and replicated), query execution in Search Computing requires the efficient execution of joins between results of service invocations and, hence, the main flows of data production fall outside of the engine's control. The need of combining service invocations with data-intensive operations is the main architectural challenge, approached by a modular decomposition of the process into processing units and by an explicit description not only of the data flow, as it is typical of many run-time architectures, but also of the control flow, through dedicated units and signals. Control flow modeling enables to explicitly tune execution, adapting it to unexpected behaviors of the components.

This concept is illustrated in Fig. 2. In the plan, the input unit, after its activation at query start, sends a control pulse to a search service unit, which executes a call. The call's result is a data flow which is sent to the output unit and, hence, returned to the query interface.

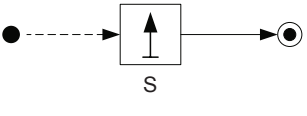

Fig. 2. Simple execution plan

In the following, control flows, data flows, and processing units are described in detail.

3.1 Structure of Execution Plans

Panta Rhei is a dedicated environment for the processing of execution plans. Every execution plan represents the physical evaluation of a query plan, and consists of a directed graph of nodes (units) and edges (data and control flows), where

- The *data flow* is a directed acyclic graph connecting processing units and whose closure defines a precedence relationship between units (the "flow of execution"). The data flow itself consist of chunks of combinations (tuples) which are progressively created by joining pairs of services, therefore in the end the flow of executions produces the result tuples. Search computing results are duplicate-free, and therefore once a tuple is formed along the dataflow, another identical tuple can be removed from the computation.
- The *control flow* includes pulse signals which are propagated "forward" (i.e. along the flow of execution) in order to time and synchronize service calls, and suspend/resume signals which are propagated "backward" in order to re-synchronize execution when anomalies are detected. Therefore, the forward controls determine producer-consumer relationships according to the query plan, and the backward controls optionally conditions those producer-consumer relationships that deviate too much from the optimal plan determined at query optimization time. A control edge may start from a data producer and, in this case, every new chunk of data produced by the unit also produces a new pulse signal.
- The behavior of each node is completely determined by its input and state. Some units accept at most one input pulse, if the pulse is omitted then the unit responds just to data flows. All nodes receive their data input from one predecessor, with the exception of parallel joins and cache units, that can have more than one data flow edges as input, as they implement binary operations (join and union).

Query plans include parallel and/or pipe joins (as presented in Chapter 10) which are translated into nodes of the execution plan. While a pipe join is represented as a sequence of service calls in which the second call implements the join, a parallel join requires an explicit join unit which has two service units as predecessors. The parameter setting of nodes involved in join computation is optimized according to the service interface specifications (particularly, their chunk sizes and service costs). The translation of an optimal query plan into its execution plan is rather straightforward, as the topology of the execution plan can be immediately drawn from the query plan. Instead, the initialization of node parameters dictating the specification of the operations implemented by them is not covered in the book. At the moment, we use simple heuristics to initialize the parameters, but we expect to fine-tune the heuristics after experimentation.

Conceptually (the implementation may be different), each node is mapped to a thread which is activated at query start, waits for input, and produces output. Queries can be suspended and resumed by users according to the liquid query interface controls, described in the next chapter. At query start (or resume), some user-controlled

parameters may be fetched into appropriate "slots" of units to fully specify their behavior. Most of these parameters are defined by the query optimization process. Then, the start node of the execution plan is activated, which triggers the start of its successor nodes. Nodes either act as data producers or consumers, or play both roles. During the execution, data producers can send "EOF" data along one data flow link, with the semantics that there will be no more data along the link. The "EOF" data is propagated by consumers until it reaches the output node, causing query termination and then the output to be produced.

The liquid query interface communicates with the execution engine by various controls, and the effect of controls may suspend or terminate a query execution. Users may also change the content of some of the query "slots" which are exposed to the user interface (through user-friendly formats). Threads are eliminated only when the user "changes" the query. Memory caches, however, might be emptied if the user "repeats" a query with a different input.

3.2 Scheduling Units in Panta Rhei

The semantics of execution plans is rather complex, as it requires introducing a number of ingredients (concerning units and their control) which interact with each other. We have decided to first present all ingredients and then to show their interplay through examples. The types of flows offered by the execution engine have already been introduced above and we recall that control flows are signals carrying no information other than their intrinsic nature (pulse, suspend, resume), while data flows carry chunks of tuple combinations, made up by matching results of the previous service calls in the flows, and emitted by units in chunks, according to the unit's execution semantics. In the following, we therefore focus on an in-depth presentation of the various kinds of units, which are shown in Fig. 3.

Input and Output Units. The *input unit* injects user-provided input into suitable slots of given units, and then starts the execution. Each execution plan must have exactly one input node, which has one or more successor nodes.

The *output unit* is a consumer node collecting query results. Each plan must have exactly one output node. Its execution activates the *liquid* interface showing the query results.

Clock Unit. A *clock unit* plays the role of coordinating service calls to perform pipe and parallel joins – thus, it is neither a producer nor a consumer. Topologically, every clock unit has in its children at least two service calls. Every clock in a plan controls a sequence of joins, where each join in the sequence is either a pipe or a parallel join, and the topology of the execution plan indicates the operands of each join[1].

Every clock is activated by a start pulse signal (a control edge connects the input node to the clock) or by a data-producer unit which produces its first data (in this case, a control edge connects the data producer unit to the clock). Clocks emit *pulse*

[1] Currently, we associate every query with exactly one clock unit controlling all of its joins, but we plan to experiment with more general settings. As clocks can be activated during the execution flow, the semantics of clocks, service, and join units in the context of scheduling plans does not force plans to have a single clock.

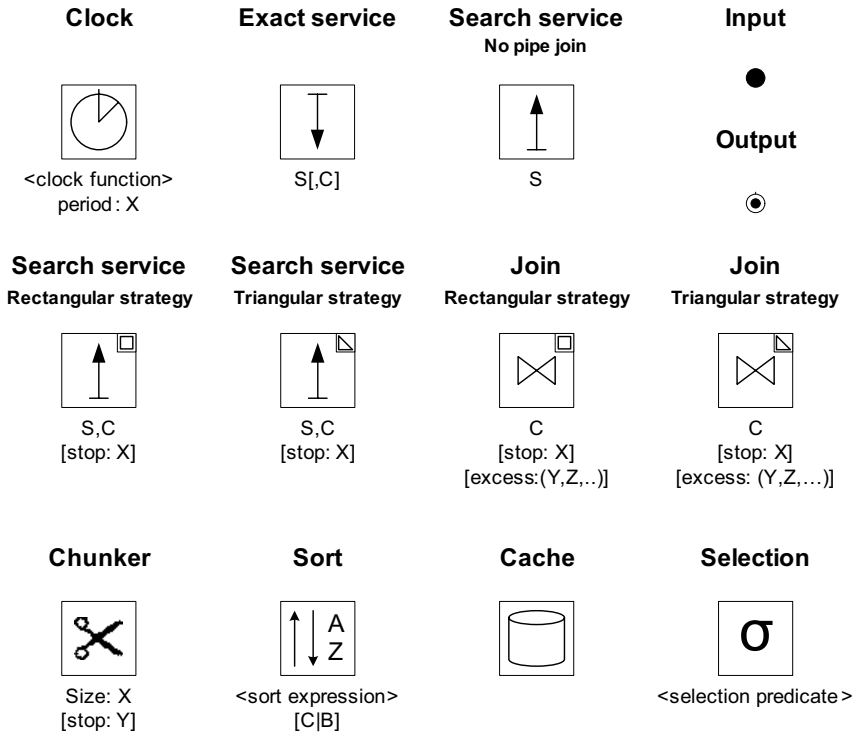

Fig. 3. Nodes of execution plans

signals to two or more service invocation units, ordered from 1 to *n* (the order of controlled units is given by the numbers labeling the pulse edges).

Every clock has a parameter, called *clock function*, defined by a regular expression that describes the maximum number of calls that the service unit can perform during the clock cycle, for each service unit controlled by the clock and for each clock cycle. To do so, the regular expression defines a sequence of clock values which each correspond to one clock cycle. Clock values are denoted as *n*-tuples of integer numbers, where *n* is the number of service invocation units controlled by the clock unit. Enumerable repetitions of sequences are indicated by a superscript, while infinite repetitions of the last parenthesis are denoted by an "n" superscript. As an example, $(1,1)(2,2)^n$ represents a sequence in which two services are invoked once in the first clock cycle, and then can be invoked at most twice in any subsequent clock cycle. An example of clock function for controlling three service units is: $(3,1,2)(4,0,3)^2(5,1,4)^n$. The clock function can be replaced at runtime, e.g. based on user input.

Every clock has a given *clock frequency* (cycles per second), which determines the time interval between two consecutive pulses to the descendent units. The clock frequency should be related to the average response time of the search services controlled by the clock. A reasonable recommendation is to set the frequency to cater for the execution of the largest of $N_{ij} \times ART_i$, where N_{ij} is the number of calls that unit i is enabled to perform during cycle j and ART_i is the average execution time of

service S_i. This number represents an estimate of the execution time due to the slower service which is controlled by the clock. Otherwise the clock would "enforce" a speed greater than the speed of one controlled service (and as such it can hardly impose any synchronization)[2].

A clock can be *suspended* by any service that it schedules. Services may be slower or faster in producing tuples relative to the plan, and thus the joiner could deviate from the configured ratio. The rationale of suspend/resume is that triangular or rectangular strategies of joins should be faithfully implemented, as they were decided by an optimizer at compilation time by taking into account the features of the services (and attempting a minimization of their access costs), and thus deviations from plans occurring at run-time should be limited. A given amount of permitted deviation (ranging between zero and infinite deviation) is defined as a join parameter. If the allowed deviation is overcome, then the clock is suspended. As a consequence, the clock will not issue any more pulses until it will receive a resume signal, which in turn is sent by the same join unit when the deviation is reduced to an acceptable amount.

Exact and Search Service Units. *Exact service* units produce a finite set of tuples that represent the exact (and thus complete) response to the service call query given the input parameters. The output tuples are neither ranked nor chunked. Nevertheless, exact services produce sequences of chunks, where each chunk corresponds to one service invocation. In the context of pipe joins, these sequences may be ordered in the data flow due to their composition with previous calls to search services. Exact services are triggered by a single input, either a pulse or a data chunk.

- In the first case, denoted as *pulse input*, the pulse produces a single exact service call, performed as soon as the pulse is received. Therefore, a well formed graph should only allow pulse signals to an exact service with the "number of invocations" parameter set to one, which is assumed as default. If an exact service is called only once and independently of data flows, normally its input is filled by "slots" extracted from the query. This situation occurs when the exact service call does not depend on other services. Note that further pulse signals, in this case, should not be allowed by a correct graph, and anyway will have no effect on the unit (i.e. the service call will not be repeated). An EOF marker indicates the end of tuples in the result.
- In the second case, denoted as *chunk input*, the service triggering produces as many calls as there are tuples in the data chunk, performed as soon as the chunk tuples become available and continued until all tuples are consumed. In this situation, the exact service unit implements a pipe join, whose strategy is however rather simple, because it consists in a simple call iteration. The input parameters of each iteration are extracted from the input tuple, and the corresponding result tuples are combined (joined) with the input tuple, thereby producing an output chunk. If the input dataflow is ordered, then the chunks are produced by the service according to the input order.

[2] Setting the clocks' frequency is a delicate service time vs. optimization time trade-off. Currently, we use as default solution setting the frequency exactly to the largest $N_{ij} \times ART_i$ computed on all the clock's edges.

Search Services exhibit a behavior similar to Web search engines: results are unbound, ranked and chunked, and normally there is no interest in obtaining a complete result, but only in obtaining the first chunks. They are triggered by either a pulse or a pair of data chunk and pulse.

- In the first case, denoted as *pulse input*, every pulse corresponds to a given number of service calls, progressively extracting new chunks at each call; N_{ij} denotes the number of allowed calls for service i in a clock cycle j. Normally, input fields for the call are filled by "slots" extracted from the query. This situation occurs when the search service call does not depend on other services.
- In the second case, denoted as *pulse and chunk input*, the same number N_{ij} of service calls is allowed as in the first case, but these calls can either use a new tuple from the input flow to match it with the "first" chunk of results for that tuple, or instead continue with another input tuple T_i that was already used in previous calls (thereby producing a given number C_i of chunks) and produce one or more subsequent chunks for that tuple (i.e. chunks starting with C_i+1). This is the most complex case of pipe join strategy, which iterates over either new or already considered input tuples (which may be unordered or ordered by the first service call) and produces chunks (which, for each input tuple, are relatively ordered by the second service call). A pipe join strategy is used for choosing at each step, which follows either a rectangular or triangular strategy which will be discussed below. In any case, results are produced by chunks (whose size is given by the number of matching tuples produced at each call of the service) and the chunks are ordered.

Pipe joins occur when a dataflow input edge comes into a service call unit of arbitrary nature (either exact or search). A pipe join implements the join between services when the join attributes of the first service are *bound* and the join attributes of the second one are *free*. If the input data consists of the concatenation of N tuples, then the output data will consist of concatenation of N+1 tuples, possibly represented through their keys. If either the input is ordered or the service being called is a search service, then the join output will be ordered.

When the second service being called is a search service, a *pipe join strategy* is needed to control the allocation of service calls to input tuples (as each input tuple is used to provide parameters for a service call, and the same tuple may induce several calls to the service, to find "better" combined results). The join strategy is imposed by performing a pipe join strategy on the downstream service unit, controlled by a clock, called the pipe join's *clock controller***,** whose clock function regulates the behavior of the two services. The input pulse parameters, sent by the clock controller to both services, indicate the number of calls allowed within a given clock cycle, and therefore also of chunks produced in output during a cycle. The pipe join strategy can be either rectangular or triangular, as informally represented in Fig. 4.

- In a *rectangular strategy*, the calls are performed considering every available input tuple in a round robin fashion: chunks are progressively extracted (the first chunk for tuples T_1, T_2, T_3… and then the second chunk for tuples T_1, T_2, T_3…). This strategy is well suited when the first service is an exact service, producing unordered input. A rectangular strategy can be imposed by setting a parameter in

the second service (to *R*) and timed by the join's clock controller, by setting the clock's function to $(1,N)^M(0,N)^n$, where N is the chunk size of the first service, and after the first M calls the first service produces an EOF. Then, calls have to be addressed just to the second service, producing the various layers of the rectangle, by iterating on the result tuples of the first service.

– In a *triangular strategy*, calls are performed in an alternate fashion: the first chunk is extracted for T_1, then the first chunk is extracted for T_2 and the second chunk is extracted for T_1, and so on, as described in the right side of Fig. 4 (same as merge-scan parallel join). This strategy is well suited when the first service is a search service, producing ordered input. A triangular strategy can be imposed by setting a parameter in the second service (to *T*) and timed by the join's clock controller, by setting the clock's function to the sequence (1,N)(1,2N)(1,3N)…, where *N* is the chunk size of the first service, thereby offering to the second service the option to get new chunks both for new tuples and for already available tuples of the first service.[3]

Join Units. Join units support the parallel join between two services, i.e. a join when neither of the join attributes is *bound*. A join unit joins the available information "by chunks". Each chunk combination gives rise to a "tile" of results (i.e. tile (1,1), (1,2), (2,1), (2,2)…), as discussed in the previous chapter. Therefore, it has as predecessor (at least) a pair of search services (producing chunked data). A *join strategy* specifies the order of exploration of tiles, with the aim to process tiles with higher rankings and more matches as fast as possible. A merge-scan strategy is obtained by setting the clock's function to $(N,M)^n$ where *N/M* is the optimal ratio between chunks of the two services. A nested-loop strategy is obtained by setting the clock's function to $(1,N)(1,0)^n$, where *N* calls are required to exhaust the second service and then calls are

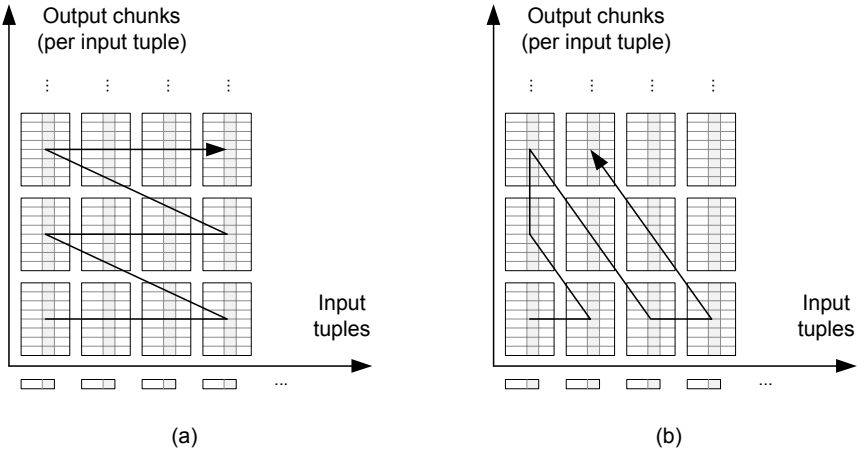

Fig. 4. Rectangular (a) vs. triangular (b) pipe join strategies

[3] A pure triangular strategy can be modified by defining "triangles" more properly, e.g. with an arbitrary proportional alternation. This extension is left for future work.

only addressed to the first service. In both cases, the choice of rectangular or triangular strategy is specified by setting a parameter in the join unit (to *R*/*T*). If either of the join input is not chunked, then the strategy is not needed and the parallel join degenerates to a unit which has a simple implementation, consisting in producing tiles in the only possible order (e.g. (1,1), (1,2), (1,3)…).

A join unit can have a stop parameter, which indicates the maximum number of tuples that should be produced by the join before suspending its execution (and producing an EOF marker)[4].

In addition, a join unit has a pair of local parameters describing the amount of deviation allowed from the planned strategy. Deviation only occurs if the join greedily attempts to produce more chunks than the number allowed by available input and planned strategy. These numbers count how many additional input chunks can be joined from either services, ranging from (0,0) – no deviation for either services – to a given pair (2,3), to unspecified – no deviation control. When the maximum allowed deviation on one input (corresponding to a service running "too fast") is overcome, the clock controlling the join unit is suspended. At that point, the service running "too slow" has some pulses available, and the join unit can concentrate upon the "tiles" which were left behind due to the slowest service, in order to bring the proportion of service calls back within the specified limits. Finally, at that point, the clock is resumed.

Selection Unit. A selection unit receives a (chunked) dataflow in input and produces a (chunked) data flow in output, consisting of all the tuples which satisfy a selection condition (an arbitrary Boolean expression of selection predicates). The selection unit does not re-chunk the output to a given chunk size and, thus, possibly changes the size of the chunks according to the selectivity of the predicate. Equality predicates matching input attributes to constants are used for building service calls, while a selection unit computes additional selections (e.g. comparison operations between attributes). Classical methods are used (by the query optimizer) in order to place the selection unit immediately after the join operation (either pipe or parallel join) which constructs the tuple with the attributes required for computing the predicate.

Sort, Chunker and Cache Units. A *sort unit* gets in input chunks of tuples and produces re-ranked result tuples in output, according to a sort expression. Sort units can be "continuous" (they sort the input chunks one by one, as they are available) or "blocking" (they wait for an EOF marker, and then process the whole input accumulated so far and emit the reordered tuples as a single output chunk). The sort function is a weighted sum of normalized expression (in the [0..1] range) over input tuple attributes, with a sort direction (either ascending or descending).

A *chunker unit* constructs new chunks from input tuples, by ignoring any already existing chunk structure thereupon. It is configured with a desired output chunk size. A chunker emits a new chunk as soon as there are exactly as many tuples as the chunk size. It has a "stop" parameter indicating the number of chunks it should produce before placing an EOF on its output dataflow, which can be interpreted by the output unit as the signal for producing output to the interface. A chunker is normally the last

[4] An EOF marker can be overridden by the user to resume the query plan and produce more results.

unit before the output unit and therefore the suspension stops the computations returning the control to the user, who in turn can resume computations and ask for more outputs.

A *cache unit* stores, within temporary memory, chunks of tuples, retrieved from services, or tuples of their keys forming combinations produced by joins. Conceptually, a cache unit is present after every service or join unit in order to store the service call or join results. However, in order not to overload the representation of an execution plan, we may omit cache units unless they have more than one incoming edge. In this case, all incoming tuples share the same schema and the cache implements the union of these tuples. The cache can also change the order of combinations when used as a union and, hence, its edges are labeled accordingly (e.g. S1/S2). The cache memory uses the normalized schema of the services in order to store service call results, and stores combinations as tuples of keys of the primary table of each service. The keys are system-generated and the tuples are indexed by chunk number and by key.

4 Examples

This section presents examples of execution plan models in increasing complexity. The purpose of the examples is to show, although on a limited sample, that execution plans can support various join strategies, including parallel join, pipe joins, and the Fagin join method which gives top-k guarantees.

We start with a parallel join of search services (Fig. 5), which is discussed at length in Chapter 10. The execution of a merge scan join between two services S_1 and S_2 using a connection pattern C_1 requires a triangular join strategy. If the optimizer determines that the optimal ratio between calls to service S_1 and S_2 is 1/2, it is sufficient to set the clock function to $(1,2)^n$ which means that at each cycle S_1 performs (at most) one call while S_2 performs (at most) two calls. The clock frequency is set so that the slowest call sequence (e.g. the time required for completing either one call of S_1 or two calls of S_2) takes place within about one cycle. The join at each new iteration builds tiles in triangular fashion, e.g. first tiles (1,1), (1,2), then tiles (2,1), (2,2), (1,3), (1,4), and so on. The joiner is allowed to produce

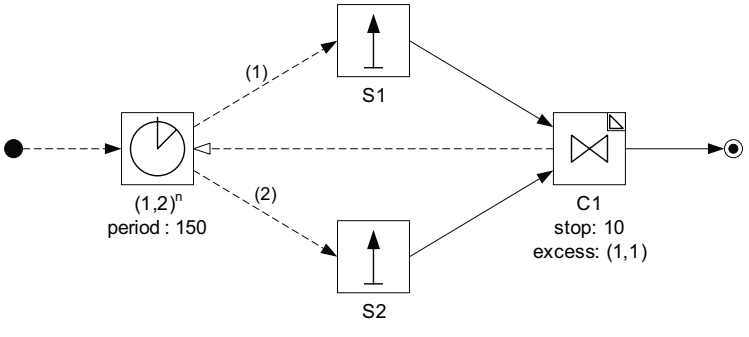

Fig. 5. Parallel join of two search services

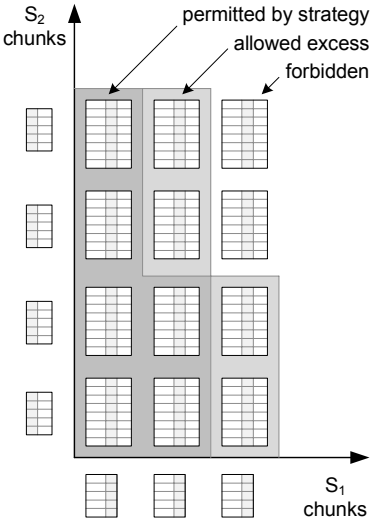

Fig. 6. Join space permitted by the strategy and allowed excess for the parallel join of the example in Fig. 5, in case S_1 returned 3 chunks and S_2 returned 4 chunks

more tiles, e.g. if at a given point of time S_1 has already produced 3 chunks and S_2 has produced only 4 chunks, thus going beyond the 1/2 ratio, the joiner can proceed with the tile (3,1), (3,2) and then (2,3), (2,4), still keeping a triangular strategy. By doing so, it reaches its "allowed excess", which is 1 extra-chunk (see Fig. 6).

In this case, if S_1 produces one more chunk, the joiner signals the clock, and the clock in turn stops sending pulses until S_2 produces 8 chunks, re-establishing the 1/2 ratio. Then, the joiner resumes the clock, and the execution of service calls and joins continues according to the joiner's triangular strategy. The joiner is set to stop its execution, producing an EOF, when 10 result tuples are built. When the EOF is received by the output node, it presents 10 result tuples to the liquid query user interface.

We next illustrate a pipe join on the same services and connection pattern (Fig. 7). We implement a nested loop join, in which we assume that S_1 produces chunks of size 10 and that after 5 calls it produces all relevant results. Then, the ratio between calls to S_1 and S_2 is 1/10 (every tuple of S_1 is an input to S_2) and the number of times this ratio must be iterated is 5. This enables building tiles (1,1)...(50,1), where each tile is obtained for a different tuple in input. At that point, no more calls to S_1 are needed (all 50 tuples are cached) and therefore calls to S_2 must be performed. S_2 then performs the joins, thus producing the second, third, …, and i-th chunk for the 50 cached tuples. The execution is terminated as soon as 20 result tuples are produced and an EOF is produced to transmit the result to the query interface through the output unit.

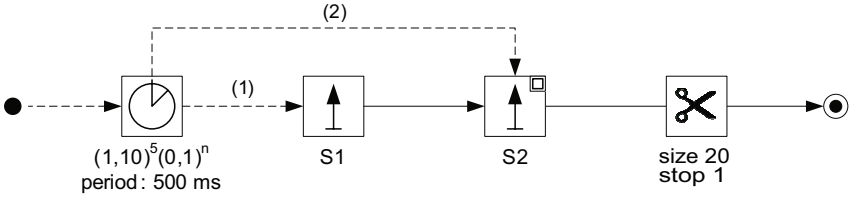

Fig. 7. Pipe Join of two search services

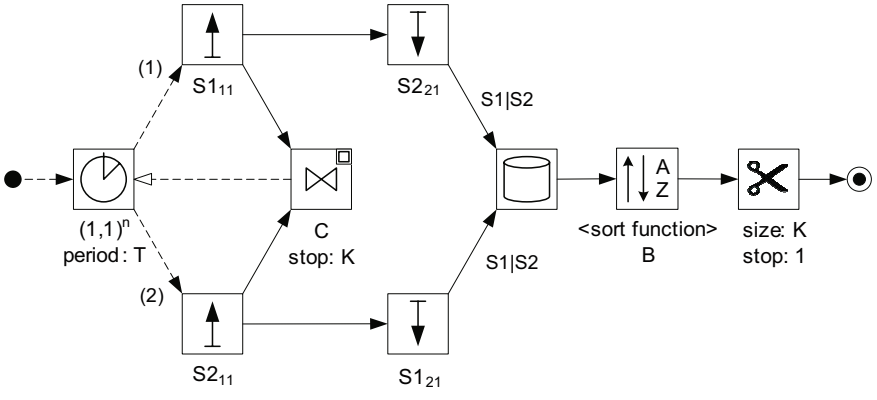

Fig. 8. Execution plan for a Fagin Join

The next example is the Fagin join [9] (Fig. 8). We recall that the method is applicable when both sequential (rank-based) and random (key-based) accesses are available for both of the services involved, and the method guarantees the extraction of *top-K* combination tuples, i.e. the tuples which are the best *K* according to any monotonic function of their relative rankings. We regard the Fagin method very suitable to Search Computing for this generality and for the method's full definability at compile time, although it is suboptimal if compared with the threshold method, as discussed in Chapter 11.

Fig. 8 shows an execution plan for the parallel join of two search services S_1 and S_2 (supporting sequential access) followed by the pipe join of different service interfaces of services S_1 and S_2, supporting direct access (e.g. access by an identifying property). A parallel join serves the purpose of halting the pulses to the search services as soon as *K* tuples are built. Then, by making a direct access for all join result values respectively on S_2 – if the join value comes from S_1 – and on S_1 – if the join value comes from S_2. Results are then reordered and stored into a cache unit which performs their union. Eventually, results are sorted according to the sort function to obtain the single chunk of *K* resulting tuples, which are guaranteed to be top-*K*.

Finally, Fig. 9 shows an execution plan for the running example which queries for a good and recent adventure movie with screenings in a theatre not too far from the user's home and good restaurants nearby. The clock controls in this case a parallel join which is followed by a pipe join. The parallel join combines *Movies* with

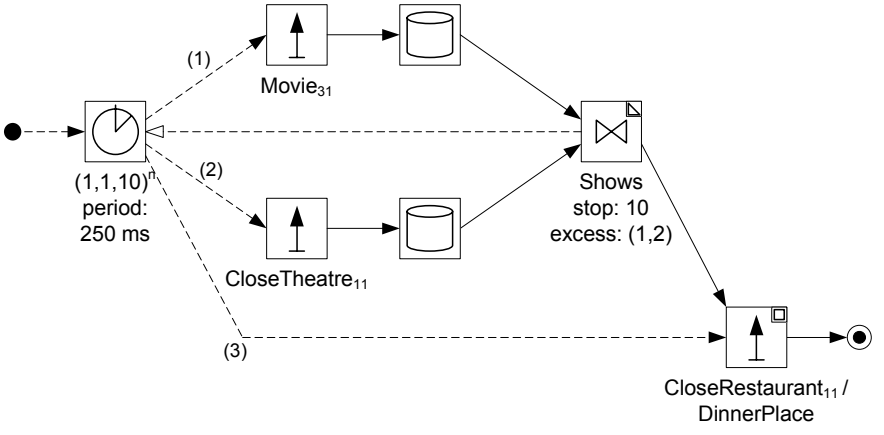

Fig. 9. Running Example

CloseTheatres according to the *Shows* combination pattern. The join combines one chunk of *Movie* with two chunks of *CloseTheatre*, using a triangular strategy, and with allowed excesses also set to (1,2). The join stops after producing 10 combinations of movies and theatres. Meanwhile, the data flow of the join results are sent to the *CloseRestaurant* service through the *DinnerPlace* connection pattern. For each pulse to *Movie*, 10 pulses are sent to *CloseRestaurant*, thereby enabling a tuple-based with 10 input tuples on from the first iteration, so that every matching movie-theatre pair is associated with close-by restaurant of the desired kind. Once 10 pairs are produced with a variable number of matching restaurants, execution is completed and results are transferred, through the output unit, to the user interface.

5 Conclusion

The execution engine described in this chapter supports operations such as service calls, join processing, caching, sorts, and chunking in order to support the efficient execution of the optimal plan selected by the optimizer. The execution engine prototype is a running platform which fosters the experimentation with new ideas and novel join strategies. Its extensible organization allows us to easily introduce new nodes or to change their parameters. The execution engine model is rather preliminary and will be improved while new releases of the environment will be deployed, yet the model resolves most of the technical challenges that are set by Search Computing queries.

References

1. Abadi, D.J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A.S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.: The Design of the Borealis Stream Processing Engine. In: Proceedings of Second Biennial Conference on Innovative Data Systems Research (CIDR 2005), Asilomar, CA, USA (January 2005)

2. Buyya, R., Abramson, D., Giddy, J., Stockinger, H.: Economic Models for Resource Management and Scheduling in Grid Computing. Concurrency and Computation: Practice and Experience 14(13-15), 1507–1542 (2002)
3. Catarci, T., Costabile, M.F., Levialdi, S., Batini, C.: Visual Query Systems for Databases: A survey. Journal of Visual Languages and Computing 8(2), 215–260 (1997)
4. Chaudhuri, S.: Query Optimizers: Time to Rethink the Contract? In: SIGMOD 2009: Proceedings of the 35th SIGMOD international conference on Management of Data, pp. 961–968. ACM, New York (2009)
5. Chaudhuri, S., Narasayya, V., Ramamurthy, R.: A Pay-As-You-Go Framework for Query Execution Feedback. Proc. VLDB Endow. 1(1), 1141–1152 (2008)
6. Cole, R.L., Graefe, G.: Optimization of Dynamic Query Evaluation Plans. In: Snodgrass, R.T., Winslett, M. (eds.) Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, pp. 150–160. ACM Press, New York (1994)
7. Eckerson, W.W.: Performance Dashboards: Measuring, Monitoring, and Managing Your Business. John Wiley & Sons, Chichester (2006)
8. Evrendilek, C., Dogac, A., Nural, S., Ozcan, F.: Multidatabase Query Optimization. Distrib. Parallel Databases 5(1), 77–114 (1997)
9. Fagin, R.: Combining Fuzzy Information from Multiple Systems. J. Comput. Syst. Sci. 58(1), 83–99 (1999)
10. Goodenough, J.B.: Exception Handling: Issues and a Proposed Notation. Commun. ACM 18(12), 683–696 (1975)
11. Gounaris, A., Paton, N.W., Fernandes, A.A.A., Sakellariou, R.: Self-Monitoring Query Execution for Adaptive Query Processing. Data Knowl. Eng. 51(3), 325–348 (2004)
12. Graefe, G.: Query Evaluation Techniques for Large Databases. ACM Comput. Surv. 25(2), 73–169 (1993)
13. Graefe, G.: Iterators, Schedulers, and Distributed-Memory Parallelism. Softw. Pract. Exper. 26(4), 427–452 (1996)
14. Grossniklaus, M., Norrie, M.C.: An Object-Oriented Version Model for Context-Aware Data Management. In: Benatallah, B., Casati, F., Georgakopoulos, D., Bartolini, C., Sadiq, W., Godart, C. (eds.) WISE 2007. LNCS, vol. 4831, pp. 398–409. Springer, Heidelberg (2007)
15. Ives, Z.G., Florescu, D., Friedman, M., Levy, A., Weld, D.S.: An Adaptive Query Execution System for Data Integration. SIGMOD Rec. 28(2), 299–310 (1999)
16. Kabra, N., DeWitt, D.J.: Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. SIGMOD Rec. 27(2), 106–117 (1998)
17. Kossmann, D.: The State of the Art in Distributed Query Processing. ACM Comput. Surv. 32(4), 422–469 (2000)
18. Manolescu, I., Bouganim, L., Fabret, F., Simon, E.: Efficient Querying of Distributed Resources in Mediator Systems. In: Meersman, R., Tari, Z., et al. (eds.) CoopIS 2002, DOA 2002, and ODBASE 2002. LNCS, vol. 2519, pp. 468–485. Springer, Heidelberg (2002)
19. Rao, J., Pirahesh, H., Mohan, C., Lohman, G.: Compiled Query Execution Engine Using JVM. In: ICDE 2006: Proceedings of the 22nd International Conference on Data Engineering, p. 23. IEEE Computer Society, Washington (2006)
20. van Reeuwijk, C.: Maestro: A Self-Organizing Peer-to-Peer Dataflow Framework Using Reinforcement Learning. In: HPDC 2009: Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing, pp. 187–196. ACM, New York (2009)

21. Srivastava, U., Munagala, K., Widom, J., Motwani, R.: Query Optimization Over Web Services. In: Dayal, U., Whang, K.Y., Lomet, D.B., Alonso, G., Lohman, G.M., Kersten, M.L., Cha, S.K., Kim, Y.K. (eds.) VLDB, pp. 355–366. ACM, New York (2006)
22. Tanin, E., Beigel, R., Shneiderman, B.: Incremental Data Structures and Algorithms for Dynamic Query Interfaces. SIGMOD Rec. 25(4), 21–24 (1996)
23. Urhan, T., Franklin, M.J.: XJoin: A Reactively-Scheduled Pipelined Join Operator. IEEE Data Eng. Bull. 23(2), 27–33 (2000)
24. Urhan, T., Franklin, M.J., Amsaleg, L.: Cost-Based Query Scrambling for Initial Delays. SIGMOD Rec. 27(2), 130–141 (1998)
25. Whiting, P.G., Pascoe, R.S.V.: A History of Data-Flow Languages. IEEE Ann. Hist. Comput. 16(4), 38–59 (1994)
26. Wong, E., Youssefi, K.: Decomposition—A Strategy for Query Processing. ACM Trans. Database Syst. 1(3), 223–241 (1976)
27. Yu, J., Buyya, R.: A Taxonomy of Scientific Workflow Systems for Grid Computing. SIGMOD Rec. 34(3), 44–49 (2005)