Ersch. in: Objects and databases : Third International Conference, ICOODB 2010, Frankfurt/Main, Germany, September 28 - 30, 2010; proceedings / Alan Dearle ... (eds.). - Berlin [u.a.] : Springer, 2010. - S. 1-5. - (Lecture notes in computer science ; 6348). - ISBN 978-3-642-16091-2

Search Computing Challenges and Directions

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

Dipartimento di Elettronica e Informazione, Politecnico di Milano
P.za Leonardo Da Vinci, 32
I-20133 Milano, Italy
{ceri,braga,corcoglioniti,grossniklaus,vadacca}@elet.polimi.it

Abstract. Search Computing (SeCo)[1] is a project funded by the European Research Council (ERC). It focuses on building the answers to complex search queries like "Where can I attend an interesting conference in my field close to a sunny beach?" by interacting with a constellation of cooperating search services, using ranking and joining of results as the dominant factors for service composition. SeCo started on November 2008 and will last 5 years. This paper will give a general introduction to the Search Computing approach and then focus on its query optimization and execution engine, the aspect of the project which is most tightly related to "objects and databases" technologies.

1 Introduction

Search engine technology provides worldwide users with the ability to get to the "best" Internet pages with the simplest possible query language. However, this simple query paradigm shows its limits when search is complex and the query cannot be compressed to keywords, or the query results are complex and cannot be included into a single page.

Performing a complex search process with a conventional search engine challenges the user's ability to break the process into several tasks, then interacting with the search engine multiple times, and then mentally reconstructing a global solution. Normally, each task can be made small enough to address a single domain. However, the answer of the global process is usually based upon comparisons and trade-offs which span over the various domains of interest, and require compositional activities performed in the user's mind (maybe augmented with notes). Such processes take place routinely, but they are far from being supported by current technology.

We define *search computing systems* [1] as a new class of systems aimed at responding to multi-domain queries, i.e., queries over multiple semantic fields of interest. Such systems support users in expressing complex queries, then decomposing queries into subqueries that can be addressed to a specific data source, then assembling complete results from partial answers, and making sure that the

[1] `http://www.search-computing.eu`

order in which complete results are produced takes into account their "global" ranking. In addition to generic search engines, this process may also involve more specific data sources and search systems. In accordance with the current software trends, we assume that component systems will be accessible through service interfaces, possibly hiding data management languages.

Building such systems requires solving many research problems. First, data sources must be semantically described so as to enable a query understanding and decomposition process. Then, a rank-aware query execution technology should support scalable and efficient query execution execution. Then, query results should be made available to users, in formats which allow their browsing and their comparative visualization. All these research areas are present in the SeCo project, together with many other research focuses, describing interaction aspects, rank-join theory, application design methods and scenarios, design tools, business and legal models.

This paper focuses on the query optimization and execution engine, which exhibits many interesting properties from a systems' perspective.

2 Query Optimization and Execution Engine

In this section, we describe Panta Rhei, the physical query algebra and runtime used in the SeCo query processor. We start by introducing the underlying data and control models, and then describe the topology of query execution plans, that are graphs consisting of nodes that represent operands (units) and edges that represent the data and control flow. We then define all types of edges and units in detail. Finally, we describe how query plans can be composed by giving a minimal set of recursive rewriting rules to define the concept of a *well-formed* plan. An overview of the concepts of Panta Rhei is given in Figure 1.

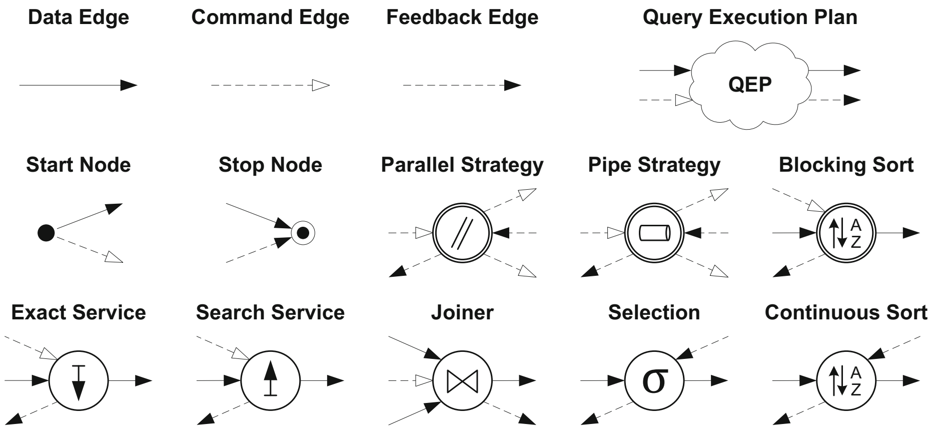

Fig. 1. Panta Rhei concepts

2.1 Data and Control Flow

The *data model* is based on an abstraction that represents the underlying data sources in terms of access patterns that define a list of input, output and ranked attributes which can be atomic or set-valued. Result tuples are progressively composed by using service results as the query evaluation progresses. The *data flow* of a query execution plan consists of **data edges** that form a directed acyclic graph. Every data edge carries tuples whose schema is obtained as the concatenation of all the schemes of the services which are invoked by antecedent nodes of that edge.

The *control model* of the execution engine addresses the fact that, in Search Computing, query processing involves a wide scope of data sources, ranging from traditional databases to Web services and search engines. If the query planner and optimizer can rely on accurate data statistics and estimations of the behavior of the data sources involved in a query, it is possible to completely specify the execution of a query at compile-time. However, if this information is not available at compile-time, the control model must be flexible enough to adapt at run-time. Moreover, plans which want to guarantee optimality (top-k) must adapt their behavior to the actual ranking values which are read from service results. The *control flow* of a query execution plan is bidirectional and comprises **command edges** and **feedback edges** to support both forward and backward scheduling of plans. The forward control flow transports instructions to a query execution plan indicating how the tuples in input must be considered by the plan, the backward control flow reports as feedback statistical data characterizing the plan execution.

2.2 Query Execution Plans

A **query execution plan** (QEP) is a component that accepts an incoming data and control flow edge and produces a data and control flow edge in output. The incoming data edge transmits chunks of tuples in input to the QEP, while the outgoing data edge transmits chunks of tuples to a downstream query execution plan or to the stop node. The incoming control edge expresses how the tuples in input should be processed within the QEP. The outgoing control edge transmits feedback data about the execution of the QEP.

The **start node** injects the constant values specified by the query into the query execution plan along the data flow edge. Additionally, it transmits the “start command” along the control flow edge. Apart from acting as a sink for all feedbacks, the **stop node** collects the results of a query execution plan and makes them available to clients of the execution engine. There is exactly one start and one end node per query.

The **parallel** and **pipe strategy unit** control two query execution plans that perform a parallel or pipe join, respectively, as illustrated in Figure 2. A parallel execution plan is built by two QEPs which are invoked in parallel, followed by a **joiner unit** which receives chunks of tuples from two different QEPs and joins them as instructed by the parallel strategy unit. In contrast, a pipe execution plan is built by two QEPs which are invoked in sequence. The join of results is implicitly performed by the second QEP, whose input data flow is produced by the first QEP. In both cases, the actual strategy of the strategy unit depends on whether the query execution plan is scheduled in forward or backward manner. Forward strategies are static, completely pre-configured by an optimizer. Backward strategies are dynamic and internally generated or altered.

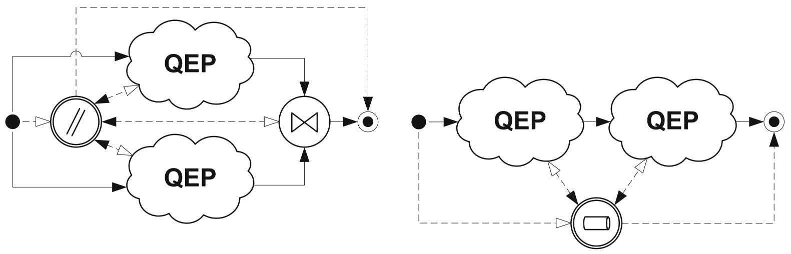

Fig. 2. QEPs for pipe and parallel joins

In our model, data sources are classified in two neatly distinguished categories, called *search* and *exact services*. While this classification is of course a simplification, it is capable of capturing the characteristics of most actual services. Accordingly, the **search** and **exact service units** invoke services of the respective types, using the given input to return result tuples. Search services exhibit a behavior similar to Web search engines: results are unbound, ranked and chunked, and normally there is no need to obtain a complete result, but only the first chunks. Exact services produce a finite set of tuples that represent the exact (and thus complete) response to the service call with the given the input parameters. The output tuples are neither ranked nor chunked.

The **blocking sort unit** buffers all chunk flowing along its input data edge until it receives the EOF message. At that point, it sorts the tuples across all chunks according to a given sort function and emits a sorted output, structured as a series of new chunks of a configurable size. In contrast, the **continuous sort unit** sorts data on a per-chunk basis as the data flows along a data edge. Each chunk in input corresponds to one sorted chunk in output, containing the same tuples. The **selection unit** filters chunks of tuples according to a configured selection predicate. Since the selection unit does not re-chunk the tuples, the chunk size can decrease due to the selectivity of the given predicate.

2.3 Plan Composition

The following set of production rules defines how QEPs can be recursively substituted to compose more complex plans. The axiom QEP consists of a single service unit with a start node as predecessor and a stop node as successor. Rules indicate that the pipe or parallel composition of two QEPs gives a QEP, and that a QEP can be composed with any unary units (i.e. units with a single input and output data and control flow) yielding a QEP. Plans obtained by arbitrary applications of these rules to the axiom are called well-formed and have associated well-defined semantics.

$$QEP := \textcircled{\uparrow} \vee \textcircled{\downarrow} \qquad QEP := QEP \bowtie_{pipe} QEP$$
$$QEP := QEP \cup \{\textcircled{\updownarrow}, \textcircled{\updownarrow}, \textcircled{\sigma}\} \qquad QEP := QEP \bowtie_{parallel} QEP$$

2.4 Example

The example query shown in Fig. 3 searches for a good and recent adventure movie in a theater not too far from the user's home and a good restaurant nearby.

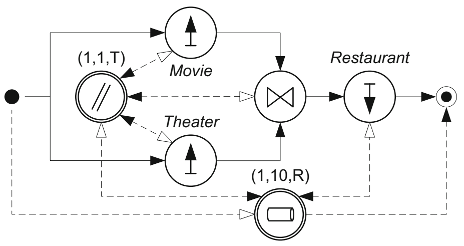

Fig. 3. A simple example query

The query execution plan uses a pipe join strategy unit to schedule the outer join. The first sub-plan of the outer pipe join is the parallel join of the movie and theater services. The results of these fetches are combined by the joiner unit and forwarded to the second sub-plan of the outer pipe join, i.e., the restaurant service.

3 Conclusion

Panta Rhei is the core component of a complete architecture for building search computing systems that process queries over data sources ranging from traditional databases to Web services and search engines. Our engine effectively optimizes the combination of data sources in two ways, namely through pipe and parallel joins. The controllers of these operations implement several strategies that guarantee good performance by limiting the performance bottleneck of service calls. In future work, we plan to port the current implementation to a tuple space environment in order to obtain a scalable Panta Rhei implementation for cloud computing.

References

1. Ceri, S., Brambilla, M. (eds.): Search Computing: Challenges and Directions. Springer, Heidelberg (March 2010)