Zuerst ersch. in : Search Computing : Trends and Developments / Marco Brambilla; Stefano Ceri. - Springer : Berlin, 2011. - S. 141-155. - (Lecture Notes in Computer Science ; 6585). - ISBN 978-3-642-19667-6
http://dx.doi.org/10.1007/978-3-642-19668-3_14

Efficient Computation of Search Computing Queries

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

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

Abstract. This chapter gives a high-level overview of how query processing is carried out in SeCo. At the highest level of abstraction, queries are expressed in a conjunctive declarative query language over service interfaces, named SeCoQL, chosen to be a compact and readable formulation to serve both experts users and system developers. Queries are then expressed at a *logical* level in the form of acyclic invocation workflows, after a compile-time analysis that decides a cost-driven scheduling of service invocations. At a lower, *physical* level queries are then translated into executable specifications that distinguish between the data flow and the control flow, support parallelism, account for stateless and stateful computation tasks, and support backward and forward control. The query engine is implemented as an interpreter of these physical plans. A workbench and testing environment is also available in the form of a tool, to monitor the processing of complex queries by inspecting all phases of their analysis and execution, at all levels of abstraction.

1 Introduction

The query processor deals with the problem of scheduling service calls, taking into account their invocation constraints, pursuing some optimization objectives at compile-time, and then dealing at run-time with their actual possibly unexpected behavior during the execution.

The chapter presents all the abstractions and assumptions used by the query planner and by the execution engine, in the way they are currently supported by the implementation. These concepts are here addressed in top-down order, and the chapter concludes with a short description of the implementation of the query engine and of the characteristics of the query workbench that is available as a demo of the system. The core concepts, abstractions, and system components related to query processing are the following.

– **Multi-domain queries:** independently of the higher-level languages and representations in which queries are formulated at the user interface level, any query is, from the engine's perspective, the specification of a collection of services to be invoked and a set of conjunctive conditions over their

results. SeCoQL, the declarative textual language chosen to represent abstract queries at this stage, serves well as matching point between different components, is easily generated by the UI modules and easily parsed by the underlying modules, and is compact and readable enough to be convenient also for expert users and developers.

- **Logical plans:** a compile-time analysis of the SeCoQL query performs a cost-driven optimization of the scheduling of service invocations. The input of this stage is a SeCoQL query in which the Service Interfaces to be invoked have already been chosen by higher layers, and the join types (pipe vs parallel) are therefore already fixed. The planner exploits the remaining available degrees of freedom to decide the topology, the number and sequence of service invocations, and the join strategies. The output is a logical plan, i.e., a specification of a workflow with quantitative estimates of the size of partial results and of the number of invocations to be performed on each service in order to produce these results.
- **Physical plans:** logical plans are then translated into query plans that are directly executable by the query engine. These plans are expressed in Panta Rhei, a unit-based language with support for parallelism, stateless and stateful execution, and backwards and forward control. Panta Rhei was designed to bridge the gap from the compile-time analysis performed by the query planner at the logical level to the run-time enactment of the query. It was designed with the objective of providing a clear specification of the engine behavior and also enabling runtime adaptivity in the form of reactions to events that do not match the expectations of the user or the assumptions made by the system at compile-time. Distribution, parallelization, and replication issues have also been considered.
- **Query Execution:** the query engine is implemented as an interpreter of Panta Rhei plans. The execution of a query is based on the simple assumption that any query consists of either (a) a simple invocation of a service interface, or (b) the combination of the results of two subqueries. In the former case, the engine supports the invocation of service interfaces that wrap data sources of many different kinds. As for the latter case, the results of the sub-queries can only be joined in series (pipe join) or in parallel (parallel join), and the interleaving of invocations performed on the sub-queries is handled by interpreting the signals sent by operators dedicated to this task, called strategy units. Also, the execution of a query can be monitored in all its stages by means of a workbench tool; a preview of this tool is shown in a demo video on the project website [1].

Figure 1 shows the conceptual architecture of the project and places the three query representation formats in the data flow between modules, from the User Interface (UI), in the upper part of the architecture, down to the invocation of actual services. The rest of the chapter is organized in four sections, dealing with the four items mentioned above.

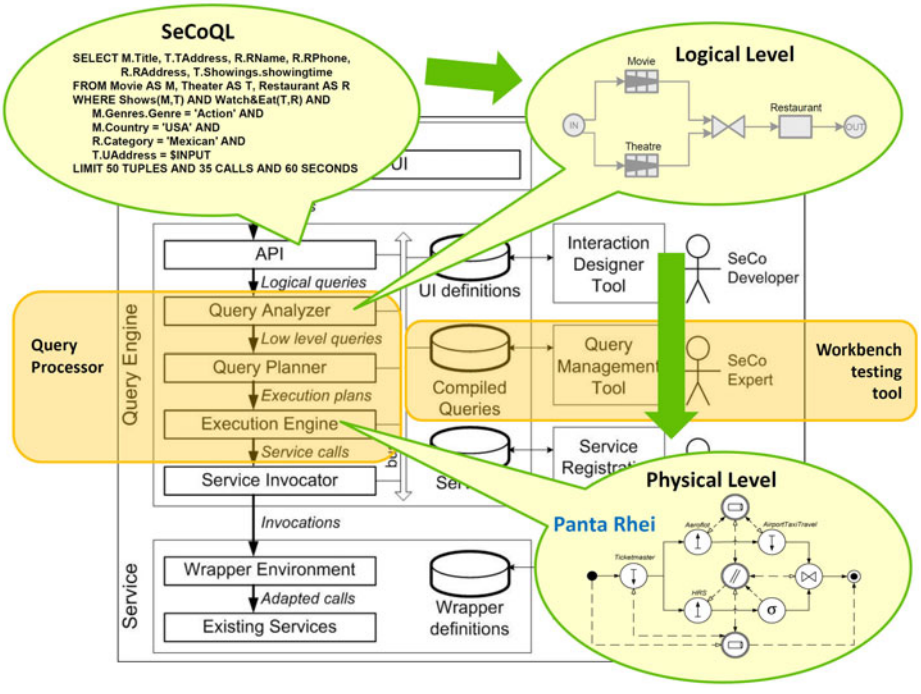


Fig. 1. Query representation formats in the SeCo framework

2 Background on Multi-domain Queries

We recall here the notion of multi-domain query by means of a classical running example: a query that searches for a good and recent American action movie in a theater not too far from the user's home and with a good Mexican restaurant nearby. The query involves three Service Marts (*Movie*, *Theater*, and *Restaurant*), and specifically three Service Invocations that implement three specific Access Patterns of these Marts, having the following schemata, where superscripts I, O, and R respectively identify parameters in input, in output, and in output with ranking.

- **Movie**($Genres.genre^I$, $MCountry^I$, $Title^O$, $Director^O$, $Score^R$, $Year^O$, $Language^O$, $Actor.name^O$)
- **Theater**($UAddress^I$, $TName^O$, $TAddress^O$, $TPhone^O$, $Distance^R$, $Showings.title^O$, $Showings.showingtime^O$)
- **Restaurant**($UAddress^I$, $Category.name^I$, $RName^O$, $RAddress^O$, $RPhone^O$, $MapUrl^O$, $Distance^O$, $Rating^R$)

Note that these Access Patterns are all relative to data sources that return ranked results. Movies are returned in order of the scores assigned by users, theaters are returned in distance order from the user address that is specified in

input, and restaurants are returned in rating order. Reataurants are indeed also retrieved based on a distance range from the address specified in input, but in the case of this access pattern the distance condition only acts as a filter and produces a readable value of distance (the $Distance^O$ attribute), while the actual order in which the results are returned is that of the restaurant rating.

Join conditions on attributes can be denoted by means of connection patterns [2] between service marts, such as *Shows* and *Watch&Eat*, whose definition in terms of equality between attributes is given below.

- **Shows**(*Movie* M, *Theater* T): [(M.Title=T.Showings.title)]
- **Watch&Eat**(*Theater* T, *Restaurant* R): [(T.TAddress=R.UAddress)]

Note that the former connection pattern checks equality between an atomic value (M.Title) and a set of values (T.Showings.title) taken from a multi-valued attribute, thus expressing the containment of the former in the latter.

SeCoQL has a declarative SQL-like syntax, in which the query result is defined as the concatenation of the tuples qualifying from the services listed in the `FROM` clause by means of the evaluation of the predicates listed in the `WHERE` clause and projected according to the list of attributes of the `SELECT` clause. More precisely, every service mentioned and associated to an alias in the `FROM` clause represents one call to a specific Service Interface, and there may be more than one call to the same Interface in the same query, in which case two different aliases are required. SeCoQL supports the specification of join conditions by means of connection patterns, but in the normal form of the language they are always expanded into their explicit form of conjunction of equalities over attributes. The query of the running example may have the following formulation in SeCoQL.

```
SELECT M.Title, T.TAddress, R.RName, R.RPhone, R.RAddress,
       T.Showings.showingtime
FROM Movie AS M, Theater AS T, Restaurant AS R
WHERE Shows(M,T) AND Watch&Eat(T,R) AND
      M.Genres.Genre = 'Action' AND
      M.Country = 'USA' AND
      R.Category = 'Mexican' AND
      T.UAddress = $INPUT
LIMIT 50 TUPLES AND 35 CALLS AND 60 SECONDS
```

Open parameters represent unbound variables whose values are assigned by the execution environment at runtime. All other input variables in the query must either have values specified in the conditions (as it is the case for Genre = "Action" in the example) or get their value from an output attribute (as it is the case for TAddress passed from Theater to Restaurant). In the example, there is only one open parameter, `$INPUT`, that represents the geographic point chosen by the user in order to center the search for theaters. For readability concerns, we restrict open parameters to be specified with identifiers whose first character is a $.

The LIMIT clause, that is fully optional, fixes the query limitations in terms of three limit conditions: number of combinations in the result (keyword TUPLES), number of calls to services during query execution (keyword CALLS), and number of seconds after which the execution is halted by time-out (keyword SECONDS). Query execution terminates as soon as one of the limit conditions is reached.

Based on the access patterns of the service interfaces used in the query, it is already possible at this stage to fix the join types that will be used in the query plan. The planner instantiates pipe joins whenever attributes that are in output from a service are used as input for other services, and instead parallel joins whenever the attributes involved in a join condition are both output attributes. In the example, the join between Movie and Theater can only occur in parallel, and Restaurant must always be invoked after Theater, in order to feed the pipe join with TAddress values, but there are no precedence constraints on the mutual position of Movie and Restaurant in the plan.

The computation of a fully determined topology for the query plans is the main responsibility of the Query Planner, that transforms the SeCoQL query into a physical plan that specifies a workflow of invocations. Also, based on some relevant profile figures taken from the service mart repository, the "optimization" of the query takes place, in a two-step approach (*Logical* and *Physical planss*), as described next, so as to estimate the amount of service calls to be overall performed in order to produce a target number of results.

3 Logical Query Plans

The optimization problem considered in the generation of logical plans, and addressed in this section, is the following: given a SeCoQL query, find the query plan topology that minimizes the *expected* execution cost according to a given *cost metric* in order to obtain k answers. The process of generating a plan starts from the conjunctive query expressed in SeCoQL and ends with a fully instantiated invocation schedule. The choice between alternatives is guided by heuristics.

3.1 Cost Metrics

A cost metric is a function that associates a cost to each query plan. We mainly consider two cost metrics: (a) the *execution time metric*, which measures the (expected) time elapsed from query submission time to the production of the k-th answer, and (b) the *sum cost metric*, which computes the cost of a plan for producing k answers as the sum of the costs of each operator used in the plan. In the former, the time required for producing k tuples takes into account the number of invocations of each unit and the expected elapsed time for the execution of that unit in order to obtain a given number of results, and the cost must account for the slowest path flowing tuples from the input to the output of the plan. For the latter, examples of costs for a service invocation are the cost of computing joins or the cost charged by the service. A special case of the sum cost metric is the request-response cost metric, which consists of considering only

the cost of service invocations required to execute the plan, omitting to consider operation execution costs. This metric is particularly relevant when the transfer of data over the network is the dominating cost factor.

Other cost metrics of interest, though so far not considered in detail in the project are the *bottleneck cost metric*, which gives the execution time of the slowest service in the plan and is relevant in contexts of pipelined execution of continuous queries, and the *time-to-screen* cost metric, which measures the time required to present the user with the first result. The former metric is suitable to contexts with homogeneous services that respond to invocations with "continuous" streams of results. In these cases, the time spent to initialize and load the pipelines is negligible w.r.t. the time spent in stable regime of execution, during which the overall throughput is limited by the throughput of the "slowest" service. This metric is hardly applicable in our context, where search services rarely produce all their tuples and the execution is normally limited to reaching k answers rather than being run as continuous queries. The latter metric is potentially more relevant to the project, as it is suitable for settings in which the user expects a prompt interaction, and would enable an optimization based on estimates of the time that users are ready to spend waiting for the first results, before quitting the task.

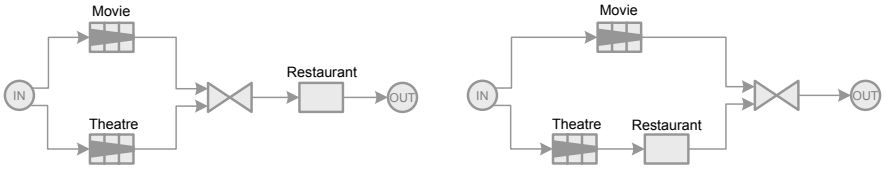
3.2 Heuristic Planning: Topology and Join Strategies

The optimization method adopted to determine the less costly plan explores the combinatorial solution space of all possible translations of the declarative query into fully instantiated invocation schedules. The exploration is organized by means of an incremental construction of the query plans, that takes place in two phases, imposing a discipline in the order in which alternative plans are generated and considered. A detailed description of the approach and of the background for optimization is given in our previous work [3,4]. The graphical notation used in the figures is also taken from these previous works, to which the interested reader is directed.

The first phase in the incremental construction of a logical plan is the selection of a topology for the logical query plan that is compatible with the given choice of service interfaces. This phase fixes the order of invocation of the services, as well as the data flow and the details of join operations. Even if at this stage all access patterns have been fixed, and therefore the nature of joins (pipe vs parallel) is fully determined, there may still be alternative topologies compatible with the precedence constraints that they enforce on the invocation order. The second phase in the construction is the choice of the number of fetches to be performed over all chunked services. This phase allows to fully determine the execution schedule and the join strategies, and therefore to compute its cost according to a given metric. For each phase, there are alternative heuristics for effectively building efficient plans. Once more, the reader is directed to [3] for details. The default heuristics currently adopted are "parallel is better" for the topology and

"square is better" for the fetching ratios, that are respectively meant to minimize the time-to-screen of the first results and to maximize the diversification of results.

In the running example, there are two alternatives. Textually, these topologies can be denoted as "(M∥T)|R" and "M∥(T|R)", where "∥" and "|" stand for parallel and pipe join, respectively. The two logical plans that implement the alternative topologies are shown in Fig. 2.



(a) Topology (M∥T)|R (b) Topology M∥(T|R)

Fig. 2. Alternative query plan topologies

4 Physical Query Plans

Physical plans are expressed in Panta Rhei and are interpreted by the query engine. The design of Panta Rhei follows a well-defined set of design principles. Plans need to be *composable*, i.e., complex plans can be formed from simpler sub-plans. They must lend themselves to *parallelization*, to leverage concurrency as much as possible, and must be *distributable* over a number of computing nodes. We start by introducing the underlying data and control models, and then describe the topology of the plans, that are graphs with nodes that represent physical operators (or units) and edges that represent the data and control flow in the plan. We then define all types of edges and units in detail. Finally, we describe how physical plans can be composed by giving a minimal set of recursive rewriting rules to define the concept of a *well-formed* plan.

4.1 Data and Control Model

The *data model* of the execution engine is based on the Service Mart framework [2] which associates each service with a flat relational schema extended with a controlled use of multi-valued attributes. The schema of the tuples of the results is simply a subset of the schema obtained by concatenating the schemes of all the services that are involved in a query. Some of the attributes are initialized with the constants specified by the user. Result tuples are progressively composed by using service results as the query evaluation progresses.

The *control model* of the execution engine addresses the fact that, in Search Computing, plans need to be highly configurable at compile-time and, to a certain extent, capable of adaptation at run-time. The requirement for configurability stems from the fact that search services may have very different and time

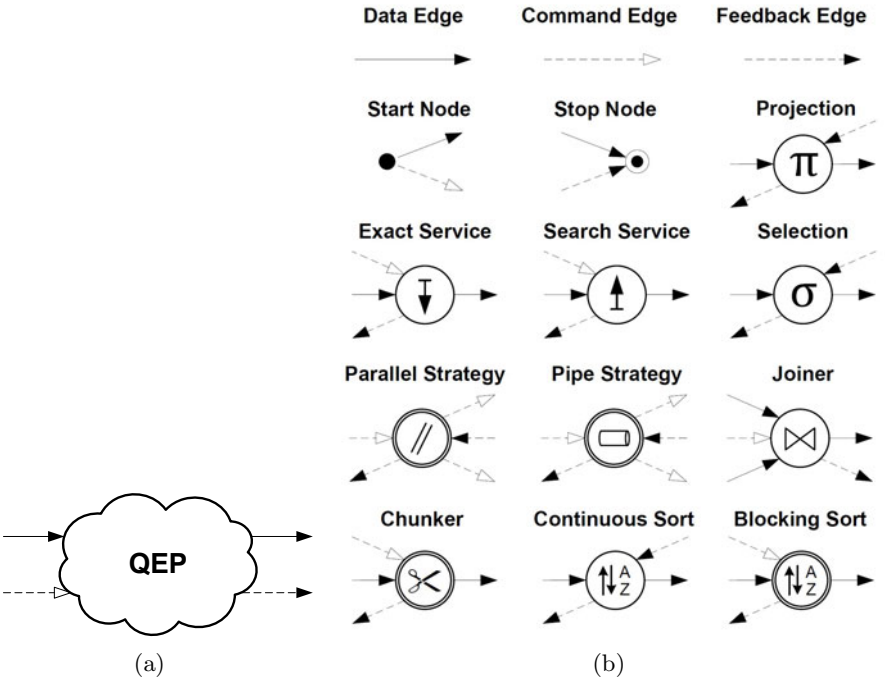


Fig. 3. (a) Query execution plans (QEPs) and (b) their components

varying computational complexity, execution time, or monetary cost. Adaptation is motivated by the fact that the properties of the search services and the data distribution encountered at run-time may be significantly different from the assumptions made by the query planner and the optimizer at compile-time, that are necessarily based on statistics derived from previous observations. Moreover, plans which want to guarantee optimality (top-k) must adapt their behavior to the actual ranking values that are progressively read from service results.

Data Flow. The *data flow* of a physical 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 schemas of the services invoked by antecedent nodes of that edge. Data messages contain chunk identifiers and chunks of tuples.

Control Flow. The *control flow* of a physical plan comprises *command edges* and *feedback edges* to support both forward and backward scheduling. Command edges transport command messages that can be further distinguished into *fetch messages* and *join messages*.

– A fetch message specifies which tuple of which chunk carried by a data edge should be used as input to the query execution plan for performing

the n^{th} fetch operation, where a fetch operation is next defined. We use the shorthand "`3@A2`" to denote that the third fetch is done with tuple `2` from chunk `A`.
- A join message defines which pair of chunks from two data edges in input to a join operator should be joined. Join messages are used to implement different exploration strategies in parallel and pipe joins.

Feedback edges transport feedback messages containing information about the execution of a physical plan. Feedback messages are transmitted after the plan has processed the commands given in input and serve primarily as acknowledgements. Feedback messages can be further classified into *statistics* messages and *exception* messages. Statistics messages contain data such as the cardinality of the computed result, the window of encountered ranking scores, the distribution of monitored attributes (e.g., join attributes), and the EOF marker if no more results can be obtained from a plan. Exception messages are transmitted by a plan in the case it observes an abnormal behavior.

The control flow is, therefore, bidirectional: the forward control flow transports instructions to a 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. As a consequence, the execution engine can support a *purely forward* scheduling of plans, as well as a *mixed forward and backward* scheduling of plans. In the former case, the query optimizer determines the entire execution strategy and configures the engine accordingly. In the latter case, the query execution strategy is dynamically determined based on selected attributes that are monitored by the execution engine.

4.2 Query Execution Plans

A query execution plan (QEP) is a well-formed and executable physical query plan, modeled as a graph, that accepts in input chunks of tuples and control messages, denoted by means of incoming data flow edges and control flow edges respectively, and produces in output chunks of result tuples and feedback messages, denoted by means of outgoing data flow edges and control flow edges respectively. Graphically, a QEP is represented as shown in Fig. 3(a). 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 QEP or to the stop node. The incoming control edge carry messages that regulare how the tuples in input should be processed within the QEP. The outgoing control edge transmits feedback data about the execution of the QEP.

We next discuss the nodes that can appear within QEPs, listed in 3(b), then we give simple compositional rules which explain how well-formed query plans can be generated.

Each node in the plan represents a processing units of one of the following kinds; the behavior of nodes is determined by its input and its state.

- **Start/Stop nodes:** 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. There is only one start node per query. The *stop node* collects the results of a query execution plan and makes them available to clients of the execution engine. Additionally, it acts as a sink for all feedbacks. There is only one stop node per query.
- **Service invocation nodes:** The *exact service invocation node* completes the tuples in input by invoking an exact service. Exact services 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 not ranked. The *search service invocation node* completes the tuples in input by invoking a search service. 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. Both types of service nodes return one chunk of tuples in output for every invocation.
- **Strategy nodes:** A *parallel strategy node* controls two QEPs that are scheduled in a parallel join, while a *pipe strategy node* controls two QEPs that are scheduled in a pipe configuration. Both these unit are assigned a "budget", i.e., a number of invocations that they are allowed to "spend". Spending the budget means breaking it down and passing it to the controlled QEPs to be consumed by considering the commands received in input and distributing the invocation commands to the connected sub-plans accordingly. Optionally, these units also receive feedback from the controlled QEPs, that they may use to tune the spending of the remaining invocation budget.
- **Joiners:** The *joiner node* is always controlled by a parallel strategy node. It receives chunks of tuples on its two different incoming data edges, and evaluates the join predicate in order to join them into new chunks of tuples, according to the join messages received from the strategy node. The joiner per se is a stateless unit whose role is limited to the evaluation of a simple predicate, but coupled with the strategy unit it provides the flexibility to implement many different logical join operations at the physical level.
- **Data flow modifiers:** A *selection node* filters chunks of tuples according to a selection predicate. Since the selection unit does not re-chunk the tuples, the chunk size can decrease in the order of the selectivity of the given predicate. A *projection node* shrinks the schema of the result to the specified list of attributes only. The *chunker node* intercepts chunks of tuples flowing along a data edge and recombines them sequentially into chunks of a given size until it finds the EOF marker. A *continuous sort node* sorts data on a per chunk basis as they flow along a data edge, whereas a *blocking sort node* caches all chunks flowing along a data edge until it receives the EOF marker and then sorts the tuples across all chunks. Both sort nodes are configured with a sort function. While a continuous sort node produces a same-sized output chunk for every input chunk, a blocking sort node is additionally configured with an output chunk size.

Parallel and pipe joins are implemented in a QEP by different stategy nodes, whose parameter setting is determined according to the service interface specifications (their chunk sizes, average response times, and invocation costs). Figure 4 shows the plan configuration for pipe and parallel joins between two generic QEPs.

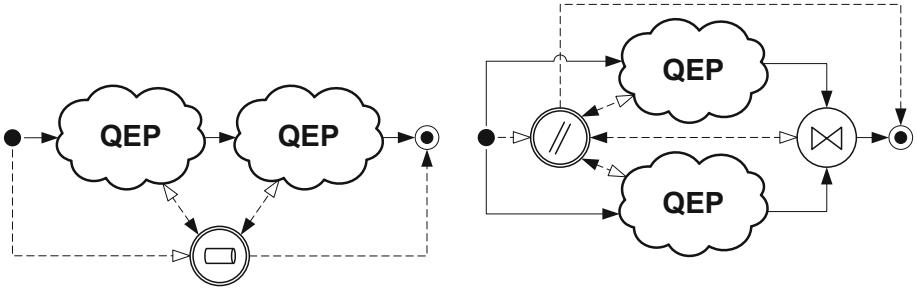


Fig. 4. QEPs for pipe and parallel joins

4.3 Plan Composition

The following set of production rules defines how QEPs can be recursively composed to form more complex plans. The axiom plan consists of a single QEP having 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 modifier operator (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.

$$QEP := \textcircled{\scriptsize ⇕} \tag{1}$$

$$QEP := \textcircled{\scriptsize ⇕} \tag{2}$$

$$QEP := QEP \bowtie_{pipe} QEP \tag{3}$$

$$QEP := QEP \bowtie_{parallel} QEP \tag{4}$$

$$QEP := QEP \cup \{\textcircled{\scriptsize ✂}, \textcircled{\scriptsize ⇅}, \textcircled{\scriptsize ⇅}, \textcircled{\scriptsize σ}\} \tag{5}$$

An example of a QEP composed using all five rules is shown in Fig. 5. Based on Rule 3, the outer-most QEP can be deconstructed into two sub-plans that are combined using a pipe join. The first sub-plan can then be further decomposed based on Rule 4 into two sub-plans that are combined using a parallel join. Applying Rule 1, each of these sub-plans can be substituted by a search service (S_1 and S_2). The second sub-plan of the pipe join can be decomposed into a query execution plan followed by a modifier (selection) based on Rule 5. Finally, this last query execution plan can be substituted by an exact service (S_3) according to Rule 2.

If we instantiate S_1, S_2, S_3 as Movie, Theater and Restaurant respectively, ignoring the application of Rule 5, we also have the Panta Rhei plan for the running example.

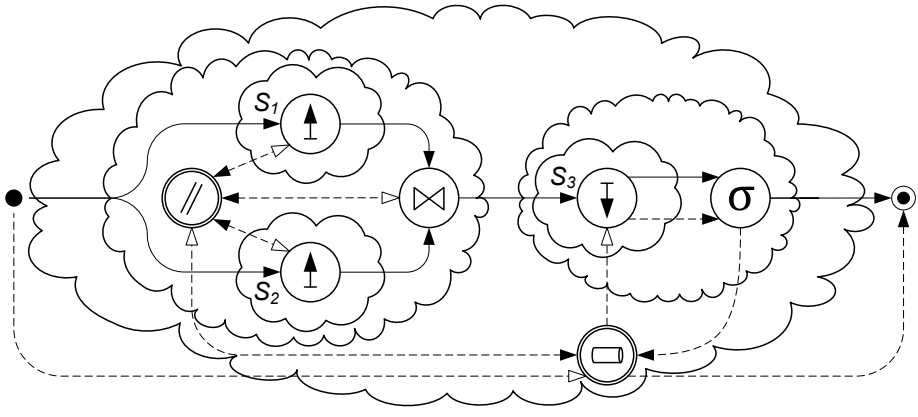


Fig. 5. Nested QEPs

5 The Experimental Workbench

A prototype of an execution engine implementing the functionalities provided by Panta Rhei is currently used for internal tests of the Search Computing framework. The engine is built on top of the Service Mart invocation and wrapping environment shown in Fig. 1, which exposes heterogeneous services, such as Web services, custom services, and relational databases, whose description is out of the scope of this paper.

The system provides both synchronous and asynchronous search mechanisms. End users of the engine can either wait for search results to be produced or be notified of their production. In the first case, the execution engine is directly connected to a user interface, whereas, in the latter case, the execution engine itself is bundled as a service. Fine-grained control is also provided, to allow for interaction with the search process, in order to dynamically orchestrate it, for example to react to the results being produced.

The engine is implemented as a multi-threaded environment that uses a thread pool to support both inter and intra-query parallelism. To support inter-query parallelism, queries can share computation and storage resources as well as reuse cached partial results of already executed invocations, if still valid. Intra-query parallelism is achieved by spawning multiple instantiations of replicable units to distribute the workload of computationally complex and time consuming tasks, such as invocations of services with high latency.

A scheduling algorithm activates the various units, assigning them to threads to handle incoming control messages, according to the priority of the search process. The control flow is implemented by means of message queues, and, in the

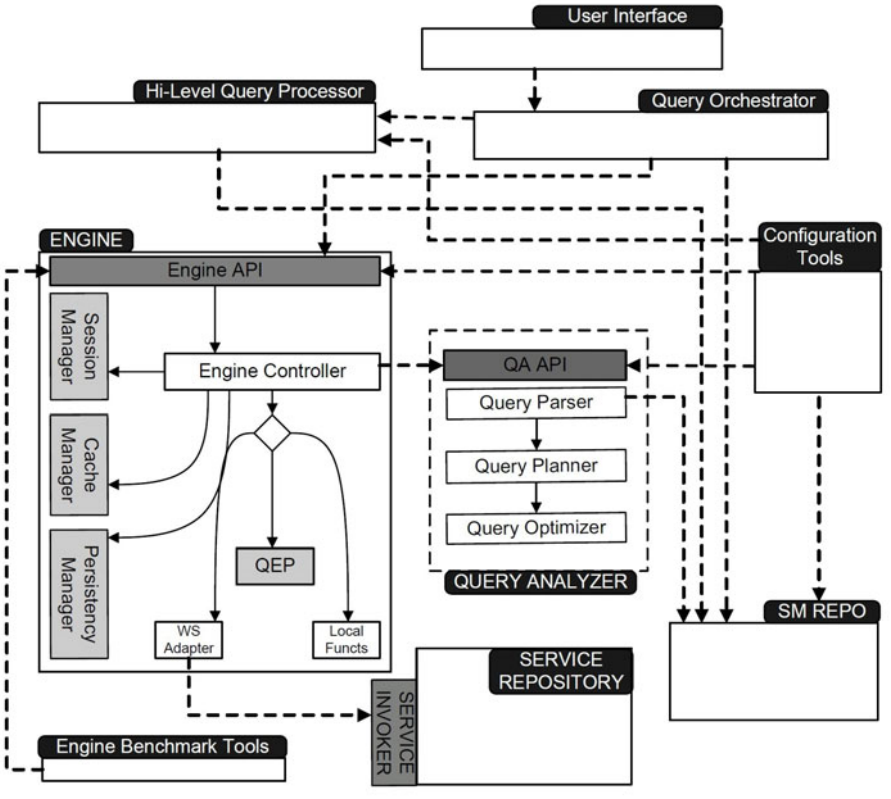


Fig. 6. The architecture of the engine and planner implementation

case of replicable units, messages can be delivered to multiple unit instances, even out-of-order. The data flow is implemented by means of buffers shared between units, according to a producer/consumer paradigm. The implementation of join strategies is fully decoupled from the implementation of the strategy units. This allows new strategies to be plugged in with no impact on the implementation of the units themselves.

Also, we adopt a two-level caching mechanism. On the one hand, we take advantage of the cache implemented at the service level, so as to avoid repeated invocations with the same input. On the other hand, we also cache partial query results at the engine level, which impacts on the performance of multiple executions of the same process and also of the same sub-plan shared between different search processes or reused by the same query in a later execution.

Figure 6 shows the modules of the query engine and the query planner in the context of the overall software architecture of the system. The invocation of the execution of a query via the Engine API causes the instantiation of a Controller for the incoming query. The controller is assigned to the current search session and is given access to both (a) the Cache Manager, that handles the session-based

cache and connects to the inter-session caching system, and possibly (b) to the Persistency Manager, that is in charge of the materialization of query results into a persistent data storage, whenever queries are registered as permanent data sources that need to be periodically updated. The interpretation of a query by the controller follows a simple recursive scheme that is based on the internal structure of the query into QEPs and sub-QEPs. The main functionality of the Controller is to instantiate the threads corresponding to the units in the QEP currently under evaluation, and in turn activate an execution context for any sub-QEP possibly contained into the current QEP. Invocation units are interpreted performing a supervised access to the corresponding Web Service (through an adaptor that connects to the Service Repository) or resorting to Local Functions in some special cases in which the invocation units represent access to locally available computation resources (as it may be the case, e.g., for the transcoding of two geo-referenced locations and the computation of their distance).

Queries can be submitted to the API in the form of already instantiated executable query plans, and in this case they are immediately interpreted, or in the form of SeCoQL statements. In the latter case, they are passed to the Query Analyzer to undergo the transformations described in Sections 3 and be converted in Panta Rhei plans as defined in Section 4.

A demo that consists in a walk through the abstractions presented here is available on the project website [1]. A snapshot of the workbench is shown in Fig. 7. In the shown panel the tool allows the user to control the query results as they are progressively produced and the execution timeline that tracks the activation of the different units.

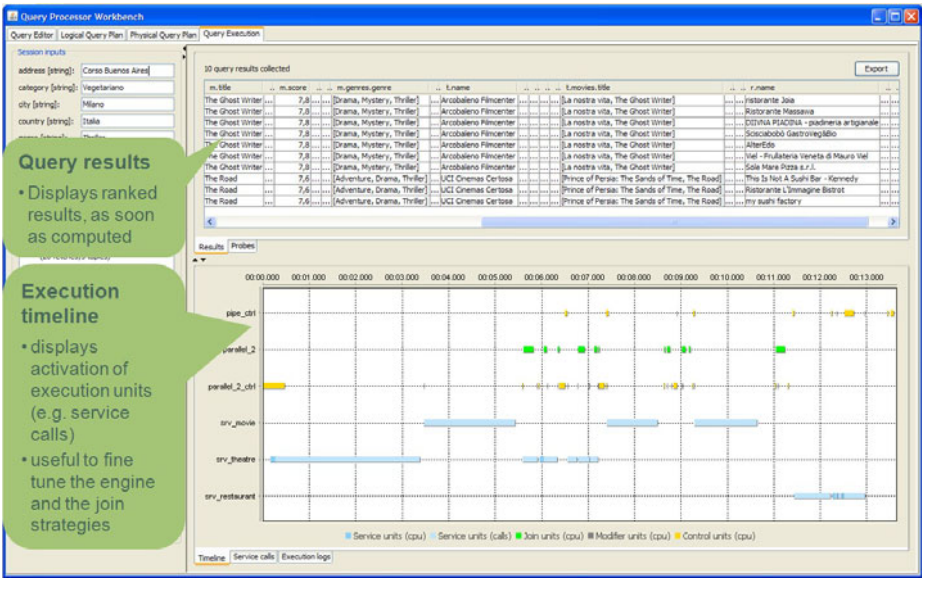


Fig. 7. A snapshot of the Query Processor Workbench

6 Conclusion

In this chapter, we have given a high-level overview of how query processing is carried out in SeCo, addressing the formulation and execution of multi-domain queries at different levels of abstraction, from the expression in SeCoQL, a declarative query language over service interfaces, down to the interpretation of physical query plans by means of a query engine that support parallelism and distribution. We also described the implementation of the query engine as the interpreter of these physical plans, as it is demonstrated in the demo available on the project website [1].

References

1. The Search Computing Project: Demonstrator of the Execution Engine (June 2010), `http://www.search-computing.org/demo/qp`
2. Campi, A., Ceri, S., Gottlob, G., Maesani, A., Ronchi, S.: Service marts. In: Ceri, S., Brambilla, M. (eds.) Search Computing. LNCS, vol. 5950, pp. 163–187. Springer, Heidelberg (2010)
3. Braga, D., Ceri, S., Daniel, F., Martinenghi, D.: Optimization of Multi-domain Queries on the Web. PVLDB 1(1), 562–573 (2008)
4. Braga, D., Ceri, S., Grossniklaus, M.: Join Methods and Query Optimization. In: Ceri, S., Brambilla, M. (eds.) Search Computing. LNCS, vol. 5950, pp. 188–210. Springer, Heidelberg (2010)