Hybrid query plan generation
Carlos-Manuel López-Enríquez, Genoveva Vargas-Solar, José-Luis Zechinelli-Martini, Christine Collet

To cite this version:

HAL Id: hal-00922887
https://hal.archives-ouvertes.fr/hal-00922887
Submitted on 2 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers. L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Hybrid query plan generation

Carlos-Manuel LÓPEZ-ENRÍQUEZ\textsuperscript{1,3}, Genoveva VARGAS-SOLAR\textsuperscript{2},
José-Luis ZECHINELLI-MARTINI\textsuperscript{1}, and Christine COLLET\textsuperscript{3}

\textsuperscript{1} University of the Americas (UDLAP)
\textsuperscript{2} French National Center for Scientific Research (CNRS)
\textsuperscript{3} Grenoble Institute of Technology (Grenoble INP)
carlos.lopez@udlap.mx, genoveva.vargas@gmail.com,
joseluiz.zechinelli@udlap.mx, christine.collet@grenoble-inp.fr

Abstract. A hybrid query is a requirement of data produced by data services and a set of QoS preferences w.r.t. the query execution. In this paper we present the problem of the hybrid query optimization and, in particular, the generation of the search space of hybrid query plans. We show how the constraints that characterize the generation of hybrid query plans are modeled and validated by implementing them in an action language. We present graphs with the experiment results that show the complexity of this generation.

Keywords: Hybrid queries, answer set planning, multi-objective optimization, logic programming, non-monotonic reasoning

1 Introduction

A hybrid query [CV11] is a spatio-temporal query over continuous and on-demand data provided by data services and accessible through interfaces (API). In particular, we focus on the characteristics associated to the on-demand data. Data consumers express their data needs along their QoS preferences. For instance, consider the data need "Where are my friends at this moment?" with the QoS preferences "\texttt{Prioritize the execution price over the execution time and this in turn on the battery consumption}."

Hybrid queries are evaluated with a service coordination approach [CVVC12] that is represented by a query workflow. A query workflow \textsuperscript{4} is a plan that coordinates a set of services with computing capabilities for processing the query. Services have QoS measures associated to their invocation and thus the execution of a query workflow has a cost that is the aggregation of the QoS measures. In centralized data bases the cost of query evaluation is the time, in distributed and parallel data bases the cost is two-folded in execution time and communication cost. The evaluation of hybrid queries in turn has a three dimensional cost: execution time, execution price, and battery consumption.

A finite set of query workflows implements the same hybrid query but there is a subset that satisfies the cost requirements of the user. For example, the

\textsuperscript{4} We refer to query workflow and query plan indistinctly.
cheaper or the faster. This is a combinatorial optimization problem of minimizing the cost of a query workflow $QW$ that is a subset of activities $A$ that implement data operations.

$$\min_{QW \subseteq A} \{\text{cost}(QW): QW \text{is a query workflow}\}$$  \hspace{1cm} (1)

The optimization of queries over data services must tackle challenges such as autonomy, non-durability of data, absence of fine grain data statistics, etc. In [OB04] the authors discuss some non-centralized query optimization approaches such as data mediation systems [TRV95,HIK97], where the cost model only considers the execution time. The evaluation of queries over Web services presented in [SMWM06] proposes a service coordination approach. A service coordination is optimized by ordering the service calls in a pipelined fashion and by tuning the size of data. Nonetheless, the control over data size (i.e., data chunks) and selectivity statistics is a strong assumption because of the autonomy of services and the absence of data statistics in a real scenario. Another aspect to consider during the optimization is the selection of the service instances, which represents a factor for the service coordination cost. In [CAI05,WCS08] the optimal selection of services with a multidimensional cost is proposed for optimizing the complete service coordination. This selection is done by solving a multiobjective assignment problem for a set of abstract services. However, the possibility to modify the order of service invocations is not considered and this is a key issue in the hybrid query optimization.

In this paper we focus on the generation of a search space of query workflows in order to get a finite set of hybrid query implementations. This work is the foundation for adding a cost model in the future and be able to choose the most suitable query workflow w.r.t. user’s preferences. This is important because a search space without constraints is in the order of $n!$ permutations in the number of activities required for evaluating the query. We propose a set of constraints modeled in an action language (e.g., DLV-K) in order to characterize the construction of query workflows with sequential and parallel compositions, and to get only the coherent ones and therefore to reduce the size of the search space.

The paper is organized as follows. Section 2 gives the bases of the environments where hybrid queries take place and explains how the hybrid query plans are modeled with the notion of query workflow. In Section 3 we describe a set of constraints for modeling the query workflow generation using an action language. In Section 4 we show experiment results that confirm the complexity of the problem. Finally, we conclude this paper and present the future work in Section 5.

2 Querying dynamic environments

A dynamic environments is a set of distributed services that provide on-demand data, continuous data and computation functionalities. Queries over dynamic environments, so called Hybrid queries, are expressions of data requirements over
on-demand and continuous data provided by data services and whose execution is constrained by user’s preferences (e.g., price, battery consumption, execution time). For example, the spatio-temporal query presented in the previous section “Where are my friends at this moment?”. In this paper we are interested in the hybrid queries over on-demand data. For example, “What are the interests of my friend Joe”, where only on-demand data are considered.

In [CVVC12] the evaluation of hybrid queries with a service coordination approach is proposed. This approach assumes the existence of computation services with data processing capabilities instead of an off-the-shelf query engine. Service coordination models a plan for evaluating the hybrid query and is represented by a query workflow.

2.1 Data and computing services

A service is a software entity with an abstract functionality described by an Application Program Interface (API) with the specification of methods for accessing data. An operation has a name and a set of typified bound/free parameters [RSU95]. Each parameter has a role identified by a unique name in the method.

For simplicity, we illustrate APIs with the following form

\[ \text{service}_\text{name}:\text{method}_\text{name}(T_1:B_1?,\ldots,T_n:B_n?,T_1:F_1!,\ldots,T_m:F_m!) \]

where the service identified by \( \text{service}_\text{name} \) has a method named \( \text{method}_\text{name} \) with \( n \) bound parameters and \( m \) free parameters. Bound and free parameters will be identified by question and exclamation marks respectively. Each parameter is typified by a data type \( T \) and has a unique name within in the method header.

From this general notion of service, we distinguish two service specializations. (1) On-demand data service provides data in a request-response fashion through synchronous data operations. (2) Computing service provides a set of methods to perform operations over data (e.g., distance estimation, ordering, correlation).

2.2 Query workflow

A query workflow represents a control flow among a set of activities. Each activity is a program that calls a data service method or a computing service method. The control flow is defined by workflow operators such as sequence or parallel.

A query workflow \( QW \) is modeled as a direct acyclic graph \( QW = (V, E, \text{init}, \text{finish}) \) where:

\[ V \text{ is a set of vertices} \]
\[ E \subseteq V \times V \text{ is a set of edges} \]
\[ A \subseteq V \text{ is a set of activities named by unique names} \]
\[ \text{init}, \text{finish} \in A \text{ are the initial init and finish finish activities of the QW} \]
\[ O \subseteq V \text{ is a set of workflow operators of the type } op = \{\text{seq, par}\} \]
\[ \text{tagged by unique labels.} \]

\( ^5 \) Data types are defined in the tuple data model in the thesis [CV11].
There are three activity specializations: the *activity* performs a service method invocation and always has a previous and a next activity, the *init* activity has no previous activity and its only goal is to launch the first *activity* of query workflow, the *finish* activity has no next activity and its only goal is to stop the query workflow execution after the last *activity*.

The compositions of activities $E \subseteq V \times V$ are defined by two workflow operators: sequential and parallel ones. A sequential composition $\text{seq}$ represents the total order of a pair of activities $a_1, a_2$ such that $e_1 = (a_1, a_2)$. It means that the second activity $a_2$ must be executed after $a_1$. A parallel composition $\text{par}$ represents the partial order of activities $a_1, a_2, a_3, a_4$ such that $e_1 = (a_1, a_2), e_2 = (a_1, a_3), e_3 = (a_2, a_4), e_4 = (a_3, a_4)$. It means that $a_2$ and $a_3$ are independent from each other and are executed in parallel after $a_1$ and before $a_4$.

In the next section we present how we model the query workflow generation by abstracting the semantics of activities [CV11]. This semantics is inherited from the data operations implemented by the services.

3 Query workflow generation

Query workflow generation is the process of enumerating all the query workflows in order to have a search space for choosing a subset that satisfies the user's preferences. The enumeration without constraints is in the order of $n!$ and, in fact, most of these query workflows would not be coherent w.r.t. the hybrid query.

The generation is done by constructing query workflows whose activities are subject to query workflow constraints. We model these constraints as action rules in the language DLV-K$^6$ that allows the sequential or parallel execution of query workflow activities.

In DLV-K, planning problems have a set of facts that represents the problem domain named background knowledge. The facts are predicates of static knowledge and are the input of the planning problem. Planning problems are modeled as state machines described by a set of fluents and a set of actions. A fluent is a property of an object in the world and is part of the states of the world [Bar03]. Fluents may be true, false or unknown. An action is executable if a precondition holds in the current state. Once an action is executed, the fluents and the state of the plan are modified. The action rules define the subset of fluents that must be held before the execution of action (i.e., preconditions) and the subset of fluents to be held after the execution (i.e., postconditions). Finally, a goal is a set of fluents that must be reached at the end of plan. A goal is expressed by the conjunction of fluents and by a plan length $l \in \mathbb{Z}_+$.

The mapping from query workflows generation to a planning problem is direct, as shown in Table 1. The APIs and the hybrid query are modeled as facts of the background knowledge. The execution state of a query workflow is modeled as fluents and query workflow activities as actions.

---

$^6$ [http://www.dbai.tuwien.ac.at/proj/dlv/k](http://www.dbai.tuwien.ac.at/proj/dlv/k)
<table>
<thead>
<tr>
<th>Query workflow</th>
<th>Planning problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>APIs, hybrid query</td>
<td>Facts (background knowledge)</td>
</tr>
<tr>
<td>Query workflow states</td>
<td>Fluents</td>
</tr>
<tr>
<td>Query workflow activities</td>
<td>Actions</td>
</tr>
<tr>
<td>Result delivery</td>
<td>Goal: finished? (l ∈ Z+)</td>
</tr>
</tbody>
</table>

Table 1. Mapping from Query workflow to a Planning problem

In the following, we show, along with a simple example, how we represent the background knowledge for query plan generation. Afterwards, we show how the query workflow state and activities are expressed in DLV-K.

3.1 Background knowledge

The background knowledge is the input for query workflow generation and it is represented by a set of facts. The facts are two-folded in (1) service interface representation and (2) hybrid query representation.

**Service interface** Consider the following three methods of service interfaces:

- profile:profile(string: nickname?, int: age!, string: sex!, string: email!) This method provides the user profile formed by age, sex and email of a given user with a nickname.

- interests:interests(string: nickname?, string: tag!, real: score!) This method provides the interests of a given user identified by a nickname. An interest is represented by a tag and has a score that qualifies the interest over the tag.

The service interface and its methods are described by the facts `service_interface/1` and the `operation/2` respectively. We distinguish between bound and free parameters with the facts `bound_p/4` and `free_p/4`. The normal form of a parameter is stated by `parameter/4` and this rule guarantees that the parameter name is unique in the context of the service method.

```prolog
service_interface(profile).
operation(profile, profile).
bound_p(profile, profile, nickname, string).
free_p(profile, profile, age, int).
free_p(profile, profile, sex, string).
free_p(profile, profile, email, string).

s\_i\_inter\_face(interests).
operation(interests, interests).
bound_p(interests, interests, nickname, string).
free_p(interests, interests, tag, string).
free_p(interests, interests, score, real).

parameter(DSN, ON, PN, T):= bound_p(DSN, ON, PN, T).
parameter(DSN, ON, PN, T):= free_p(DSN, ON, PN, T).
```

Listing 1.1. Service interfaces
Hybrid query Now consider the query “What are the interests of my friend Joe” that is represented by the hybrid query in facts:

project (p, nickname, n).
project (i, score, s).
project (i, tag, t).
retrieve (profile, profile, p).
retrieve (interests, interests, i).
select (p, nickname).
join (p, nickname), (i, nickname).

Listing 1.2. Hybrid query in facts

This query expresses the need of data over the methods profile:profile and interests:interests methods, the nickname of the profile is filtered and correlated with the nickname of interests. Finally, the parameters nickname, score and tag are projected.

Observe that the selection over the nickname attribute is indicated only in intention because the equality operators are not significant for the query workflow generation.

3.2 Query workflow activities

The query workflow activities are represented as actions in DLV-K. Below we present the intuition of activities, for more details about the semantics of activities please refer to [CV11]. In general, the activities are predicates that hold if their facts from background knowledge are true. There are also activities that are independent from facts.

init and finish These activities have the special purpose to init and finish the query workflow execution accordingly with our model in subsection 2.2. Thus the semantics is not associated with the query processing and there is no dependency with the background knowledge.

on-demand This activity establishes a connection with a data service method and requires a method of a service and the expressed need of the user to be queried.

on_demand(DS) requires operation (DSN,ON), retrieve (DSN,ON,DS).

Listing 1.3. on-demand activity

bind_selection This activity invokes and retrieves data from a service method.

The invocation is done by a given bound parameter valid in the service interface definition. The hybrid query must express that data are required from this service method respect to a selected bound parameter.

bind_selection(DS,BP) requires operation (DSN,ON),
   retrieve (DSN,ON,DS), bound_p (DSN,ON,BP,_) , select (DS,BP).

Listing 1.4. bind-selection activity

bind_join The required correlation in the hybrid query is performed by this activity. Correlation is binary between data from two service methods on a parameter from each one. The parameter from the outer method must be
bound. This activity is analogous to `bind_selection` but `bind_join` takes the value of a bound parameter from the output of another method (i.e., the inner method).

\[
bind\_join(DS1, P1, DS2, BP2)
\]

\[
\text{requires operation}(DS1, ON1), \text{retrieve}_1(DS1, ON1, DS1),
\]

\[
\text{parameter}(DS1, ON1, P1, _),
\]

\[
\text{operation}(DS2, ON2), \text{retrieve}_1(DS2, ON2, DS2),
\]

\[
\text{bound}_2(DS2, ON2, BP2, _),
\]

\[
\text{join}_3(DS1, P1, DS2, BP2).
\]

Listing 1.5. bind-join activity

`select` The select activity performs the filtering over a valid parameter of a required service method in the hybrid query.

\[
select(DS, P) \text{ requires operation}(DS, ON),
\]

\[
\text{retrieve}_1(DS, ON, DS),
\]

\[
\text{parameter}(DS, ON, P, _),
\]

\[
\text{select}_1(DS, P).
\]

Listing 1.6. select activity

`project` This activity projects a parameter of a service method required in the hybrid query.

\[
\text{project}(DS, P) \text{ requires project}_1(DS, P, _).
\]

Listing 1.7. project activity

The semantics of activities described above is completed with constraints that define their preconditions and postconditions.

### 3.3 Query workflow constraints

The query workflow constraints define the conditions associated to the execution of activities. A condition is a state of knowledge modifiable by the execution of the activities. Conditions also define when the activities can be executed. Hence, we talk about preconditions and postconditions of activities. A state is composed by a set of fluents that occur during the execution of activities. The rules that define when a fluent occurs can be static or dynamic. Static rules are those that occur given the truth-value of a subset of fluents, and dynamic rules occur given the truth-value of a subset of fluents and after the execution of an activity.

Below we present the constraints that describe how the activities can be performed and how a query plan should be constructed.

- **init and finish** The first executable action in a query workflow is `init`. This activity has no previous activity, thus its precondition is that the query workflow is not `initiated` and produces a new state with `initiated`. The last activity is `finish` and there is no other activity executed after this one. The precondition to execute `finish` is that there is not evidence that the query workflow is `finished` and the data is already `delivered` (See output constraint below for details about `delivered`). The postcondition of `finish` is `finished` and this is always the goal for the generation.
executable init if initiated.
caus ed initiated after init.
executable finish if not finished, delivered.
caus ed finished after finish.

Listing 1.8. init and finish activities

- on-demand Once initiated the plan, the data services must be connected (DS). This fluent is produced by the execution of on-demand (DS) activity.

executable on-demand(DS) if initiated.
caus ed connected(DS) after on-demand(DS).

Listing 1.9. on-demand activity

During the execution, all data services must be connected. Therefore, there is a fluent all_connected that is false if there is not evidence that a data service is connected. Otherwise, it is true.

caus ed all_connected if not connected(DS).
caus ed all_connected if not all_connected.

Listing 1.10. all_connected fluent

- bind selection One of the activities that implements data retrieval is bind selection. It is only executable if there is not evidence that the data service DS has already been retrieved and if there is a connection with DS. Once the bind selection is executed, the fluent retrieved(DS) is true. retrieved(DS) is an inertial fluent, thus the re-execution is not possible.

executable bind_selection(DS, BP)
if not retrieved(DS), connected(DS).
caus ed retrieved(DS) after bind_selection(DS, BP).

Listing 1.11. bind_selection activity

- selection The filtering of data is done by the selection activity. It is executable if there is not evidence that the parameter P of DS has already been selected. There is also need that the data from DS has been retrieved and obviously the selection select_(DS,P) must be part of the hybrid query. The execution of the selection makes the fluent true selected(DS,P) and it is inertial, so the re-execution of the selection select_(DS,P) is not possible.

executable select_(DS,P) if not selected(DS,P), retrieved(DS).
caus ed selected(DS,P) after select_(DS,P).

Listing 1.12. select activity

In order to be aware of the state of selection over all the required parameters of a method DS, the all_selected_from(DSName) becomes true if there is no other parameter pending to be selected.

caus ed all_selected_from(DS) if not selected(DS,P), select_(DS,P).
caus ed all_selected_from(DS) if not all_selected_from(DS).

Listing 1.13. all_selected_from fluent

Analogously, all parameters from all data services must be selected when they are required. Therefore, there is the fluent all_selected. This fluent is true if there is no other method DS pending to be selected.
caused \texttt{all\_selected} if \texttt{-all\_selected\_from(DS), select\_\langle DS, P \rangle.}
\[1\]
caused \texttt{all\_selected} if \texttt{-all\_selected\_from(DS), not select\_\langle DS, P \rangle, parameter\_\langle DSN, ON, P, _\rangle, retrieve\_\langle DSN, ON, DS \rangle.}
\[2\]
caused \texttt{all\_selected} if not \texttt{-all\_selected}. \hspace{1cm} \[3\]

Listing 1.14. \texttt{all\_selected} fluent

projection This activity is executable if there is no evidence that the parameter \texttt{P} of \texttt{DSNName} has been projected. There is also need that the data from \texttt{DSNName} be retrieved. The execution of projection makes the fluent \texttt{projected\_\langle DSNName, P \rangle} true and it is inertial, thus the re-execution of the projection is not possible.

executable \texttt{project\_\langle DS, P \rangle} if not \texttt{projected\_\langle DS, P \rangle, retrieved\_\langle DS \rangle.}
\[1\]
caused \texttt{projected\_\langle DS, P \rangle} after \texttt{project\_\langle DS, P \rangle.}
\[2\]

Listing 1.15. \texttt{project} activity

The projected fluent is true once the action \texttt{project\_\langle DS, P \rangle} is done. During the query workflow execution, all the required parameters must be projected. For \texttt{DS} grain the fluent \texttt{all\_projected\_\langle from(DS)\rangle} is true if there is no other parameter from \texttt{DS} pending to be projected. For the entire query, the fluent \texttt{all\_projected} is true if there is no other \texttt{DS} pending to be projected.

caued \texttt{all\_projected\_\langle from(DS)\rangle} if not \texttt{projected\_\langle DS, P \rangle, project\_\langle DS, P, _\rangle.}
\[1\]
caued \texttt{all\_projected\_\langle from(DS)\rangle} if not \texttt{-all\_projected\_\langle from(DS)\rangle after \texttt{project\_\langle DS, P \rangle.}}
\[2\]
caued \texttt{-all\_projected\_\langle from(DS)\rangle, project\_\langle DS, P, _\rangle.}
\[3\]
caued \texttt{-all\_projected\_\langle from(DS)\rangle, not \texttt{-all\_projected\_from(DS).}}
\[4\]
caued \texttt{-all\_projected\_\langle from(DS)\rangle, not \texttt{-all\_projected\_from(DS).}}
\[5\]

Listing 1.16. \texttt{-all\_projected\_from} fluent

output Once the hybrid query is processed, data must be delivered by the activity \texttt{output}. In order to model this precondition, the fluent \texttt{query\_processed} is true if all the other possible activities have been processed. Otherwise, the fluent is \texttt{-query\_processed}.

caued \texttt{-query\_processed} if not \texttt{all\_connected.}
\[1\]
caued \texttt{-query\_processed} if not \texttt{-all\_retrieved.}
\[2\]
caued \texttt{-query\_processed} if not \texttt{all\_selected.}
\[3\]
caued \texttt{-query\_processed} if not \texttt{-all\_projected.}
\[4\]
caued \texttt{-query\_processed} if not \texttt{-query\_processed.}
\[5\]

Listing 1.17. \texttt{-query\_processed} fluent

Once the query is processed, the \texttt{output} activity delivers the result and the fluent \texttt{delivered} becomes true.

executable \texttt{output} if \texttt{query\_processed}, not \texttt{delivered.}
\[1\]
caued \texttt{delivered} after \texttt{output.}
\[2\]

Listing 1.18. \texttt{output} fluent

4 Experiments

We performed experiments with a set of hybrid queries described below. The objective of the experiments is to measure the size of the search space of query
workflows given a fixed number of activities and a goal to define the length of the query workflow.

4.1 Configuration

In order to test the performance of query workflow generation, we have defined four hybrid queries. All are based on the query example of subsection 3.1. In order to make more complex the generation of query plans, we add two data operations. The resulting queries are configured as follows:

<table>
<thead>
<tr>
<th>Query</th>
<th>( \text{select}_{\text{data}} )</th>
<th>( \text{in}_{\text{data}} )</th>
<th>Description</th>
<th># activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>( \times )</td>
<td>( \times )</td>
<td>no additional operators</td>
<td>11</td>
</tr>
<tr>
<td>Q2</td>
<td>( \checkmark )</td>
<td>( \times )</td>
<td>+unary data operator</td>
<td>12</td>
</tr>
<tr>
<td>Q3</td>
<td>( \times )</td>
<td>( \checkmark )</td>
<td>+binary operator</td>
<td>13</td>
</tr>
<tr>
<td>Q4</td>
<td>( \checkmark )</td>
<td>( \checkmark )</td>
<td>+both unary and binary operator</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 2. Query configuration

Another dimension for the test is the max length of plans. The generated plans depend on a required plan length. In DLV-K, one must specifies the desired length that determines the size of the search space of query workflows and thus the required time for the generation.

In the tests we used lengths from 6 to 14, and we will measure (1) the length of the generated plans, (2) the size of the search space, and (3) the execution time of the generation.

4.2 Results

The data retrieved from the experiments show (1) the behavior of the search space growth during the generation and (2) the time required for generating all the search space.

Each hybrid query requires a fixed number of activities accordingly with the number of data operators required to process the query result. Given that the length \( l \) bounds the size of query plans, there are hybrid queries that require a largest \( l \) than others. For example, Q1 with 11 activities, requires at least an \( l = 7 \) to get query workflows with the most possible parallel compositions, and at most \( l = 11 \) to get completely sequential compositions. For reasons of space we invite the reader to visit the URLs http://goo.gl/XKzuL and http://goo.gl/z4iu3 to appreciate examples of query workflows of 7-length and 11-length respectively. The growth of search space is shown in Table 3 and, as was expected, the size of search space tends to be stable once the maximum length is reached. For example, Q1 reach the maximum length with \( l = 11 \). This is analogous for Q2, Q3, and Q4. Nevertheless, the size of search space for each one is considerably
bigger than the others less complex. In the Figure 1(a), you can see that the search spaces have an exponential growth until the max length.

![Graphs](image1.png)

(a) Search space growth respect to the plan length  
(b) Behavior of query workflow generation during the time

Besides the size of search space, the time for processing the query workflow generation is also exponential (See 1(b)) and it is not feasible to generate completely the search space in the context of query optimization. Thus, this enumeration must be done implicitly in order to avoid the combinatorial explosion.

5 Conclusion and future work

In this paper we presented the constraints for generating query workflows that model the plans of hybrid queries. The constraints represent the semantics of
query workflow activities that is analogous to the data operation implemented by the associated service method. We used the action language DLV-K for modeling the planing problem of query workflow generation in order to get a first (and naive) search space generation towards the optimization of hybrid queries. The results show that time complexity for query workflow generation is exponential.

The constraints must be extended in order to consider the continuous data operators for completely modeling hybrid queries. These constraints can be taken to model a more sophisticated query workflow generation. This implies the abstraction of data operators types and their associated constraints in a general form.

Theoretically, query workflow generation can be modeled mathematically by taking the constraints presented here and modeling an objective function. Additionally, the equivalence and coherence of query workflows must be proved.

References


