HAL
open science

Evaluation des requêtes hybrides basées sur la coordination des services

Victor Cuevas Vicenttin

To cite this version:

Victor Cuevas Vicenttin. Evaluation des requêtes hybrides basées sur la coordination des services. Autre [cs.OH]. Université de Grenoble, 2011. Français. NNT : 2011GRENM026 . tel-00630601

HAL Id: tel-00630601

https://theses.hal.science/tel-00630601v1

Submitted on 10 Oct 2011

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.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Doctorat MSTII/informatique**

Arrêté ministériel : 7 août 2006

Présentée par

« Víctor Cuevas-Vicenttín »

Thèse dirigée par **« Christine Collet »** et
codirigée par **« Genoveva Vargas-Solar »**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Evaluation de Requêtes Hybrides Basée sur la Coordination de Services

Thèse soutenue publiquement le **« XX-XX-2011 »**,
devant le jury composé de :

Civilité, Prénom, NOM
Fonction et lieu de la fonction, rôle (Président, Rapporteur, Membre)
Civilité, Prénom, NOM
Fonction et lieu de la fonction, rôle (Président, Rapporteur, Membre)
Civilité, Prénom, NOM
Fonction et lieu de la fonction, rôle (Président, Rapporteur, Membre)
Civilité, Prénom, NOM
Fonction et lieu de la fonction, rôle (Président, Rapporteur, Membre)
Civilité, Prénom, NOM
Fonction et lieu de la fonction, rôle (Président, Rapporteur, Membre)
Civilité, Prénom, NOM
Fonction et lieu de la fonction, rôle (Président, Rapporteur, Membre)

Víctor Cuevas-Vicenttín

EVALUATION OF HYBRID QUERIES BASED ON SERVICE COORDINATION

186 pages.

Remerciements

Je tiens à remercier sincèrement à ma directrice de thèse Christine Collet et à ma codirectrice Genoveva Vargas-Solar pour leur collaboration dans la réalisation de ce travail. Egalement je remercie l'attention des rapporteurs de thèse et des autres membres du jury.

Grenoble, 29 Avril 2011

Victor Cuevas-Vicenttin

Table of contents

Table of contents 7

List of tables 11

List of figures 13

1 Introduction 15
- 1.1 Context and motivation . . . 15
  - 1.1.1 DBMSs and query processing . . . 15
  - 1.1.2 Architecture of DBMSs . . . 17
  - 1.1.3 Query evaluation process . . . 18
- 1.2 Objective and approach . . . 19
  - 1.2.1 Motivation example . . . 20
  - 1.2.2 Prospective approach for evaluating hybrid queries . . . 21
- 1.3 Main contributions . . . 21
- 1.4 Document organization . . . 22

2 Query processing and hybrid queries 25
- 2.1 Query processing in dynamic environments . . . 25
  - 2.1.1 Data models . . . 25
  - 2.1.2 Dynamic environments . . . 27
  - 2.1.3 Data communication patterns . . . 28
  - 2.1.4 Query taxonomy . . . 29
- 2.2 Snapshot queries . . . 30
  - 2.2.1 Factual queries . . . 30
  - 2.2.2 Snapshot spatio-temporal queries . . . 31
  - 2.2.3 Location-aware queries . . . 34
- 2.3 Continuous queries . . . 35
  - 2.3.1 Stream queries . . . 36
  - 2.3.2 Mobile location-aware queries . . . 40
- 2.4 Service-oriented computing and query processing . . . 42
  - 2.4.1 Query processing architectures . . . 43
  - 2.4.2 Service-oriented query processing . . . 45
- 2.5 Conclusions . . . 47

3 A data model for hybrid queries 49
- 3.1 Data and data services types . . . 49
  - 3.1.1 Complex values . . . 49
  - 3.1.2 Services . . . 51
  - 3.1.3 On-demand data services . . . 52

3.1.4 Streaming data services . . . 52
3.2 Operators . . . 54
3.2.1 Window operators . . . 55
3.2.2 Recursive complex value operators . . . 56
3.2.3 Combination operators . . . 62
3.2.4 Nesting and unnesting operations . . . 65
3.2.5 Set operations . . . 67
3.3 Hybrid queries . . . 68
3.3.1 Characterization of hybrid queries . . . 68
3.3.2 HSQL a language for hybrid queries . . . 69
3.4 Conclusions . . . 72

4 Evaluating a Hybrid Query as a Service Coordination 73
4.1 Representation of a hybrid query as a workflow . . . 73
4.1.1 Query workflow model . . . 73
4.1.2 Graph representation of query workflows . . . 74
4.1.3 Hybrid query workflows . . . 75
4.2 Generation of an HSQL query workflow . . . 79
4.2.1 Representation of join dependencies as a hypergraph . . . 81
4.2.2 BP-GYO reduction of the hypergraph to obtain a join parse tree . . . 85
4.2.3 Join workflow generation by parse tree traversal . . . 88
4.3 Evaluation of a query workflow as a service coordination . . . 92
4.3.1 Service provisioning . . . 92
4.3.2 Simple computation services . . . 93
4.3.3 Composite computation services . . . 93
4.3.4 Implemented composite computation services . . . 95
4.3.5 Flexibility of computation services . . . 98
4.3.6 Service interoperation and communication . . . 99
4.4 Composite computation services workflow model . . . 100
4.4.1 Workflow model constructs . . . 100
4.4.2 Workflow graph construction rules . . . 104
4.5 Conclusions . . . 106

5 HYPATIA: a service-based hybrid query processor 109
5.1 Overview of HYPATIA . . . 109
5.2 Complex values and data services implementation . . . 110
5.2.1 Complex values implementation . . . 111
5.2.2 Data services implementation . . . 112
5.3 Generation of hybrid query workflows from HSQL queries . . . 117
5.3.1 Parsing HSQL queries . . . 117
5.3.2 Generation of hybrid query workflows . . . 118
5.4 Execution of hybrid query workflows . . . 121
5.4.1 Data service operators . . . 122
5.4.2 Basic computation services . . . 122
5.4.3 Simple computation services . . . 123
5.4.4 Composite computation services . . . 124
5.4.5 Not-service-based query operators . . . 125

5.5 GUI and workflow visualization . . . 128
5.5.1 Visualization of hybrid query workflows . . . 128
5.5.2 Visualization of operator workflows . . . 130
5.6 Testbeds and experimental results . . . 131
5.6.1 Friend Finder . . . 131
5.6.2 NEXMark . . . 133
5.6.3 Experimental results . . . 134
5.7 Conclusions . . . 136

6 Conclusions and perspectives 137
6.1 Main results and contributions . . . 137
6.2 Perspectives . . . 138

Bibliography 141

A HSQL syntax 161

B ASM syntax 165

C Semantics of ASM-based Service Coordination Model 169
C.1 Overview of service coordination . . . 169
C.2 Coordination model characterization . . . 170
C.3 Basic ASM concepts . . . 172
C.3.1 Abstract states . . . 172
C.3.2 Transition rules . . . 173
C.4 ASM-based service coordination model . . . 174
C.4.1 Restrictions on basic ASMs . . . 174
C.4.2 Extensions on basic ASMs . . . 175
C.4.3 Language semantics . . . 178

D HYPATIA installation 181

List of tables

4.1 Query workflow derivation 80
4.2 Completion of the join query workflow in Figure 4.8 92

5.1 NEXMark queries 135

List of figures

1.1 Historical outline of DBMSs [Adi07] . . . 16
1.2 Shifts in the use of the Web [Mat01] . . . 17
1.3 General architecture of DBMSs [HS05a] . . . 18
1.4 Example hybrid query scenario . . . 20
1.5 Query and operator workflows . . . 23

2.1 Data communication patterns . . . 28
2.2 Query taxonomy . . . 29
2.3 Recent historical outline of reusability in software [Pol06] . . . 42

3.1 Representation of a service . . . 51

4.1 Query workflow model constructs . . . 74
4.2 Directed graph representation of workflow constructs . . . 75
4.3 Rules for query workflow graph construction . . . 77
4.4 Query workflow graph for the query in Example 1.1 . . . 79
4.5 Attribute symbols and query hypergraph . . . 82
4.6 Construction of the query hypergraph . . . 85
4.7 Hypergraph reduction and parse tree construction . . . 88
4.8 Join workflow for the parse tree in Figure 4.7 . . . 91
4.9 a) Data service and b) Simple computation service . . . 93
4.10 Workflow constructs based on the Abstract State Machines (ASM) formalism . . . 94
4.11 Composite computation service . . . 95
4.12 Workflow of the join service . . . 95
4.13 Tuple-based window composite service operator workflow . . . 97
4.14 Time-based window composite service operator workflow . . . 97
4.15 Service interoperation . . . 99
4.16 Communication patterns for the query workflow graph for the query in Example 1.1 . . . 100
4.17 Update rule workflow graph . . . 101
4.18 Parallel composition rule workflow graph . . . 101
4.19 Sequential composition rule workflow graph . . . 101
4.20 Conditional rule workflow graph . . . 102
4.21 Iteration rule workflow graph . . . 102
4.22 Control state workflow graph . . . 103
4.23 a) Extensional and b) symbolic representation of rules . . . 103
4.24 Query types of our taxonomy addressed by our approach . . . 106

5.1 Overview of the class structure of HYPATIA . . . 110
5.2 Runtime view of HYPATIA . . . 111
5.3 Syntax of JSON objects and arrays . . . 112
5.4 a) Runtime view of generic on-demand data service b) On-demand service interface mapping . . . 113

5.5 Runtime view of our stream server a) during subscription and b) during data delivery . . 114
5.6 Class structure of our stream server 115
5.7 Subscription phase of the MultiStreamServer 115
5.8 Data generation phase of the MultiStreamServer 116
5.9 Structure of the classes for the generation of query workflows 117
5.10 Overview of the query workflow generation process by a) parsing and b) reduction and generation 118
5.11 Detailed view of the join workflow generation process 120
5.12 Classes for the execution of query workflows 121
5.13 Execution of query workflows 122
5.14 Query operators and their relation to data and computation services 123
5.15 Class structure of our ASM interpreter 125
5.16 ASM interpreter execution 126
5.17 Class structure of a) our recursive selection and b) recursive projection operators 126
5.18 Screenshot of the HYPATIA GUI 129
5.19 Fragment of a query workflow displayed in our GUI 129
5.20 Fragment of an operator workflow displayed in our GUI 130
5.21 Classes involved in the visualization of operator workflows 130
5.22 Generation of the visualization of an operator workflow 131
5.23 Google Maps web interface for our testbed 133
5.24 Tuple latency for a services-based vs. a single Java application query processor 136

C.1 Components of our coordination model 171

D.1 Software components in our implementation 182
D.2 File structure of the HYPATIA hybrid query processor 183

CHAPTER 1
Introduction

We first discuss the context and motivation of our work, beginning with recent trends in database management systems and information access, and following with the shortcomings of current technologies. We then present our objective and approach. Next we discuss the main contributions of our work. We conclude with an outline of the remainder of this thesis.

1.1 Context and motivation

1.1.1 DBMSs and query processing

Database management systems (DBMSs) emerged as the flexible and cost-effective solution to the information organization, maintenance, and access problems found in business and government. In the present day, the data management market is dominated by the major Object-Relational Database Management Systems (OR-DBMSs) like Oracle [1], Universal DB2 [2], or SQLServer [3]. These systems arose from decades of corporate and academic research pioneered by the creators of System R [ABC+76] and Ingres [SHWK76].

During this time many innovations and extensions have been proposed to enhance DBMSs in power, usability, and spectrum of applications; this process is outlined in Figure 1.1. The introduction of the relational model [Cod70], prevalent today, enabled to address the shortcomings of earlier data models. Subsequent data models, in turn, were relegated or became complementary to the relational model. Further developments focused on transaction processing and extending DBMSs to new types of data (e.g. spatial, multimedia, etc.); data analysis techniques ensued.

The emergence of the Web marked a turning point, since the attention turned to vast amounts of new data outside of the control of a single DBMS. This resulted in an increased use of data integration techniques and exchange data formats such as XML. However, despite its recent development the Web itself has experienced significant evolution, resulting in a feedback loop between database and Web technologies, in which both depart from their traditional dwellings into new application domains.

Figure 1.2 depicts the major shifts in the use of the Web. A first phase saw the Web as a means to facilitate communication between people, in the spirit of traditional media and mainly through email. Afterwards, the WWW made available vast amounts of information in the form of HTML documents, which can be regarded as people-to-machines communication. Advancements in mobile communication

[1] http://www.oracle.com
[2] http://www-01.ibm.com/software/data/db2/
[3] http://www.microsoft.com/sqlserver/2008/en/us/

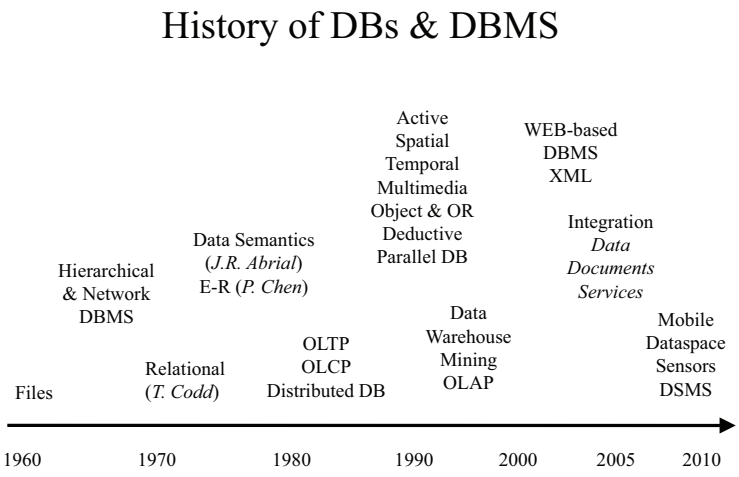


Figure 1.1: Historical outline of DBMSs [Adi07]

technologies extended this notion to mobile devices and a much larger number of users. A more recent trend exemplified by Web Services and later Semantic Web Services, as well as Cloud computing[4], consists of machine-to machine communication. Thus, the Web has also become a means for applications and a plethora of devices to interoperate, sharing data and resources.

In this context, we can identify three major aspects that involve database and Web technologies and that are crucial in satisfying the new information access requirements of users. First, a large number of heterogeneous data sources accessible via standardized interfaces, which we refer to as *data services*. Second, computational resources supported by various platforms that are also publicly available through standardized interfaces, which we call *computation services*. Third, mobile devices that can both generate data and be used to process and display data in behalf of the user.

Information access in a database context is tantamount to query processing [Wie92, DD99]. Our goal is to facilitate query processing in the emerging environments characterized by the three aforementioned aspects, which we henceforth will often refer to as *dynamic environments* for short.

Query processing in dynamic environments must be guided by QoS (quality of service) criteria stemming from (i) user preferences (access cost to data and services); (ii) devices capabilities such as memory, computing power, network bandwidth and stability, battery consumption in relation to operations execution; (iii) data and service pertinence in dynamic contexts i.e., continuously locate providers to guide data and services access considering QoS criteria such as efficiency, results relevance and accuracy.

Therefore, querying in dynamic environments needs mechanisms that integrate data services with query evaluation and data management services, i.e. computation services, for optimally giving access to data according to multiple, changing, and often conflicting QoS criteria. In addition, when changes occur in the execution environment (i.e. connection to a different network, user and services inaccessibility and mobility) alternative services must be matched and dynamically replaced. Consequently, getting the

[4]Cloud computing enables on-demand network access to computing resources managed by external parties.

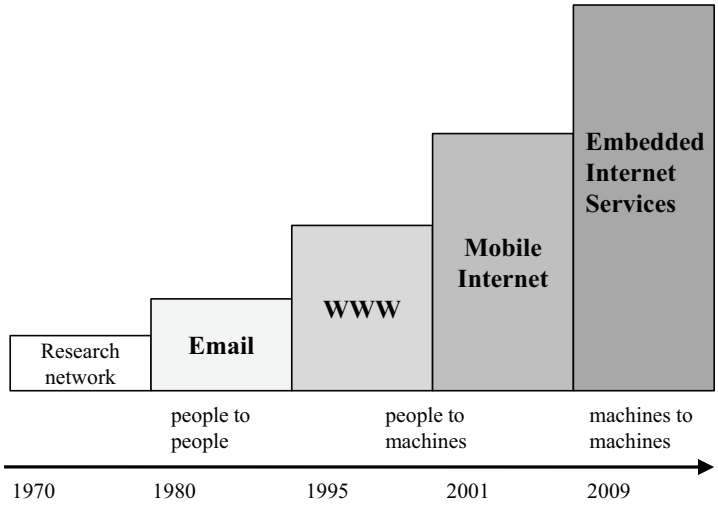


Figure 1.2: Shifts in the use of the Web [Mat01]

right information/functionality implies integrating fault tolerance (data and devices replacement), QoS, location-awareness, mobility, and adaptability in query processing techniques.

Although in the present work we cannot fully address the gamut of issues discussed so far, we can lay a foundation by understanding the fundamental aspects and proposing an adequate solution that can then be developed further. In order to do so we must first understand the capabilities and limitations of current database and query processing technologies. Therefore we address next the architecture of DBMSs and the fundamental process by which they the evaluate queries.

1.1.2 Architecture of DBMSs

An examination of the standard architecture of DBMSs can be useful to determine their viability in new environments and for new applications. We limit ourselves to traditional relational or object-relational DBMSs, whose architecture is presented in Figure 1.3.

A process manager is responsible for regulating the resources utilized by concurrent users and queries. By means of admission control policies users and their respective queries are served by database connections, which in turn are mapped to processes or threads supported by the operating system.

The submitted queries are handled by the query processor, which we will discuss in greater detail shortly, for now it is important to remark its interactions with other components. In regards to our current discussion of traditional DBMSs, queries can take part or be affected by transactions. If that is the case then it is necessary to employ locks on the data and to keep an operations log. These tasks are performed by the transactional storage manager. For its operation, a DBMS and its query processor also require utilities to manage basic resources like memory and disk space. Additional utilities facilitate tasks such as backup and replication as well as administration and configuration.

Form our high-level view of the architecture of DBMSs we can conclude that although they make

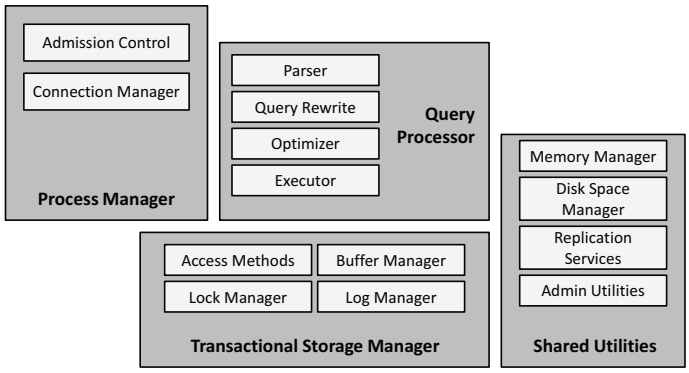


Figure 1.3: General architecture of DBMSs [HS05a]

efficient use of the resources of their underlying environment, they are fixed to that environment as well. In a service-oriented environment, where various hardware and software platforms are hidden behind service interfaces that kind of control is unattainable. Furthermore, for several applications in dynamic environments ACID transactions may not be feasible or required. In addition, DBMSs are not easily portable and often impose a large footprint. A related problem is that they are difficult to evolve and maintain. For these reasons, several researchers have concluded that they in fact exhibit underperformance [SC05] or are even inappropriate [Kos08] for a variety of new applications.

Consequently, the core functionality of the query processor must be adapted for new settings and applications, among which we are particularly interested in dynamic and service-oriented environments. With this purpose in mind we next take a look at the architecture and operation of query processors.

1.1.3 Query evaluation process

The architecture of the query processor component presented in Figure 1.3 and it closely reflects the traditional process [Gra93] for evaluating a query. This process roughly consists of the following phases, each supported by a query processor subcomponent.

1. **Parsing**: when a query is issued in a declarative language such as SQL it is first parsed in order to generate an internal representation of it, such as, an abstract syntax tree or other types of data structures. Overall, parsing is a well understood domain with reliable techniques (e.g. see [ASU86]).

2. **Query rewriting**: the internal representation of the query is employed to analyze it and transform it into a query evaluation plan. The analysis implies verifying that the references of the query are valid (e.g. an attribute is part of its given accompanying relation); also, if views are used these are resolved to their underlying base relations. If the query is valid an evaluation plan is generated, which consists of a tree of algebraic operators that enables to obtain the query result.

3. **Optimization**: usually there are multiple query evaluation plans that all yield the desired query

result, and among these equivalent plans some can exhibit significantly better performance than others. To obtain an optimal plan (strictly speaking often quasi-optimal) we can use two basic strategies. The first is to transform the query evaluation plan generated in the previous phase by reordering its operators, yielding a more efficient plan. The second is to build an optimal plan from the start. In both cases we require a cost model to measure the optimality of each plan in terms of the chosen metrics. This cost model may apply only at the logical level (i.e. algebraic operators) or at the physical level by taking into consideration access paths and specific algorithms.

4. **Execution**: once an optimal plan has been selected, an executable representation of it is generated and carried into execution. This representation can be as low-level as some variant of compiled machine code, but more frequently is a data structure interpreted to guide the flow and processing of data.

The last three phases need to be significantly revised in a dynamic and service-oriented environment with data and computation services; parsing does not fundamentally change since we want to keep the familiar declarative languages. Query rewriting must take into consideration the data service interfaces, since some data may need to be supplied to these in order to retrieve the rest of the data. The existence of a large number of heterogeneous data services may also necessitate the use of data integration techniques. In addition, new types of queries will require the definition of new query operators.

Traditional query optimization techniques are not applicable in this new setting, since the statistics used in cost models are generally not available. Furthermore, resources will be dynamically allocated via computation services, rather than being fixed and easy to monitor and control. Although we do not address query optimization under such context in the present work, we provide a discussion of its related issues and possible solutions in [CVVSCB09].

Finally, query execution must also be reconsidered. First, the means to access the data is via services rather than scanning or employing index structures. Second, to process the data we depend on computation services, instead of a rigid DBMS.

1.2 Objective and approach

Our ultimate goal is to make information, data, and services available everywhere and anytime, thereby democratizing information access. New research challenges for querying techniques are opened from this goal, since as discussed previously, traditional database and query processing techniques do not suffice for dynamic service-oriented environments.

In this thesis, our concrete objective is to enable the efficient evaluation of queries in dynamic environments. Queries posed in a declarative language and that involve the data produced by data services. Furthermore, their evaluation must take advantage of the functionality offered by computation services, thereby facilitating resource reutilization and extensibility.

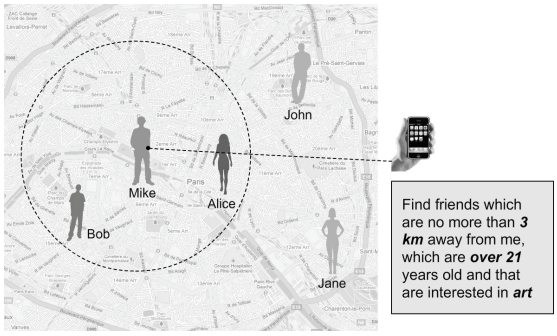


Figure 1.4: Example hybrid query scenario

1.2.1 Motivation example

The following example scenario helps to clarify the challenges that we address.

Example 1.1.
Consider the scenario depicted in Figure 1.4. Multiple users find themselves in an urban area carrying GPS-enabled mobile devices that periodically transmit their location. A user in this scenario may want to *Find friends which are no more than 3 km away from him/her, which are over 21 years old and that are interested in art*.
To answer this query three data services need to be accessed, which produce data in one of two ways: on-demand in response to a given request, or continuously as a data stream. In both cases, the data services expose an interface supported by standardized protocols (e.g. SOAP and WSDL), and which is composed of several *data operations*. Furthermore, data operations produce data in a format in which *complex values* are built from atomic values, nested tuples, and lists.
For instance, the users' location is made available by a stream data service with the (simplified) interface

$$\texttt{subscribe()} \rightarrow \{\texttt{location:}\langle\texttt{nickname, coor}\rangle\}$$

consisting of a subscription operation that after invocation will produce a stream of `location` tuples, each with a nickname that identifies the user and his/her coordinates. A stream is a continuous (and possibly infinite) sequence of tuples ordered in time.
The rest of the data is produced by the next two on-demand data services, each represented by a single operation

$$\texttt{profile(nickname)} \rightarrow \{\texttt{person:}\langle\texttt{age, gender, email}\rangle\}$$
$$\texttt{interests(nickname)} \rightarrow \{\texttt{s_tag:}\langle\texttt{tag, score}\rangle\}$$

The first provides a single `person` tuple denoting a profile of the user, once given a request represented by his/her nickname. The second produces, given the nickname as well, a list of `s_tag` tuples, each with a tag or keyword denoting a particular interest of the user (e.g. music, sports, etc.) and a numeric score indicating the corresponding degree of interest.

In dynamic environments as the one in our example, query processing implies evaluating queries that

address at the same time traditional data providers (e.g., DBMSs), stream data providers, and possibly mobile devices and spatio-temporal aspects as well. For this reason, we refer to queries like the one in Example 4.17 as *hybrid queries*. In our work these queries are expressed in a declarative SQL-like language we refer to as HSQL.

1.2.2 Prospective approach for evaluating hybrid queries

In this thesis we introduce an approach for evaluating hybrid queries in dynamic environments, a problem which as far as we know has not been fully addressed yet, particularly in a flexible and efficient way. Hybrid queries by definition cover many aspects and thus present numerous challenges, accentuated in dynamic environments. Our objective is to propose a unifying approach that addresses them to the broadest extent possible.

By an analysis of existing research we can confirm that numerous solid techniques have been proposed for particular types of queries in specific contexts. However, not much effort has been dedicated to integrate them, focusing instead on following the research trends of the day. Furthermore, if such efforts are to take place, it is unlikely that they will succeed if they are directed towards the development of an enhanced yet monolithic DBMS.

Query processing ultimately involves producers and consumers of data with different characteristics. Such heterogeneity raises the challenge of achieving interoperability. In addition, the vast variety in the types of queries that need to be considered, illustrated by our still non-exhaustive taxonomy presented in Chapter 2, implies that an equally vast variety of functionality needs to be assembled to evaluate them.

We adopt a Service-Oriented Architecture (SOA) approach to deal with the issues of interoperability and acquisition of data and functionality for the evaluation of hybrid queries in dynamic environments. Concretely, we propose to base query evaluation on service coordination. A query is evaluated therefore as a service coordination, where services represent either data sources (data services) or operators that process data (computation services).

This approach was proposed in [CV08a, CV08b, CVVSCB09] and will be developed in the remaining of this work, also being presented in [CVCVSI10a, CVVSC$^+$10] and invited to appear in [EIS11]. Our approach remains prospective to some degree due to the need of incorporating additional aspects out of the scope of our work. These include techniques for service adaptation and substitution, query optimization, and adaptive query evaluation.

1.3 Main contributions

We summarize the main contributions of this thesis as follows.

- *Analysis and taxonomy of queries.* We perform an analysis of the challenges presented by, and of the current techniques employed for, the evaluation of a significant variety of queries. We categorize these various types of queries into a taxonomy in which all fall under the scope of the

hybrid queries that we propose in this thesis. This analysis served to develop our approach and also opens opportunities for future extensions of our work.

- *Data model for hybrid queries.* We develop a data model consisting of on-demand and stream data services that deal with complex values, as presented in our example, and of a series of operators that operate on such data. We provide a formal specification of this data model, and a complementary SQL-like language compatible with an important subset of it, which we call HSQL.

- *Evaluation of a hybrid query as a service coordination.* We do not assume the availability of a pre-built or off-the-shelf DBMS for the evaluation of hybrid queries. Alternatively, our approach for evaluating a hybrid query is based on its representation as a service coordination; which in turn, is specified as a workflow. We introduce a series of construction rules that map operator expressions in our data model to what we refer to as *query workflows*. Furthermore, we introduce the BP-GYO algorithm, an extension of a known algorithm in database theory that enables us to generate a query workflow directly from a HSQL query.

- *Evaluation of query operators as composite computation services.* In our approach, query operators are evaluated by computation services belonging to the service coordination specified as a query workflow. However, in many cases one cannot expect to find a computation service that matches fully with the requirements of an operator.

  For this reason, we introduce a coordination model and language that enables to compose several computation services into a new computation service that enables the evaluation of an operator. This coordination model and language is formally specified and compatible with a workflow representation, which enables to specify composite computation services to evaluate query operators as *operator workflows*. Figure 1.5 depicts intuitively how query and operator workflows specify a service coordination comprised of data and computation services.

- *HYPATIA, a service-based hybrid query processor.* To validate our approach, we developed, HYPATIA, a prototype service-based hybrid query processor that was demonstrated in [CVCVSI10b]. HYPATIA implements the capabilities described above and was tested with two testbeds developed specifically for that purpose. Our experiments showed that although a service-based approach involves certain overhead, as it is to be expected, it remains a viable alternative presenting several advantages.

1.4 Document organization

The remainder of this document is organized as follows.

- **Chapter 2** presents first the elements that we use to characterize query processing in dynamic environments, the most important being our query taxonomy. The subfamilies of queries forming this

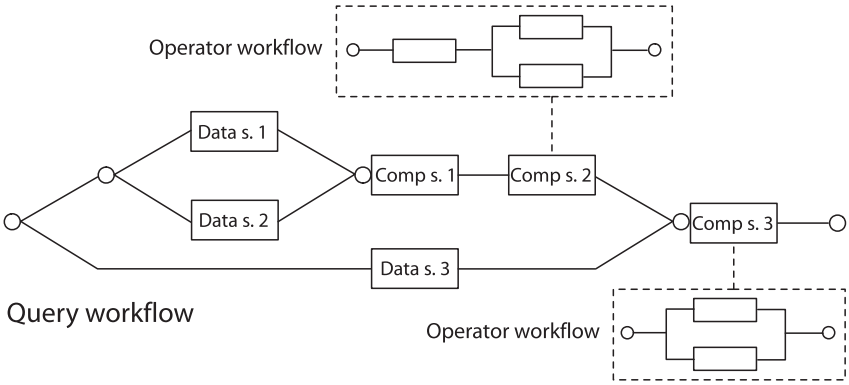


Figure 1.5: Query and operator workflows

taxonomy are then detailed along with the techniques used for their evaluation. Lastly, we present a discussion of query processing architectures with an emphasis on service-oriented architectures and their representative works.

- **Chapter 3** begins by describing complex values and the data services that produce them. Then the various operators included in our data model are presented. A formal characterization of hybrid queries and an overview of the HSQL language follow.

- **Chapter 4** describes first the representation of hybrid queries by query workflows. The BP-GYO algorithm to generate such query workflows is presented next. Afterwards, the evaluation of a query as a service coordination specified by a query workflow is described. Special consideration is given to composite computation services and operator workflows.

- **Chapter 5** presents the design and implementation of HYPATIA beginning by an overview of the system. The implementation of data services is discussed next, followed by the implementation of the generation and execution of query workflows. Finally, the testbeds we developed and the experimental results we obtained are discussed.

- **Chapter 6** presents our conclusions and perspectives, providing a discussion of the challenges that remain and the many possibilities for extending our work.

CHAPTER 2

Query processing and hybrid queries

This chapter presents an overview of the research and developments undertaken in query processing that are relevant to our objective. First, in Section 2.1 we introduce concepts that help us to characterize query processing in dynamic environments, most notably a taxonomy of hybrid queries consisting of two main families of queries: snapshot and continuous. Then we present in detail our query taxonomy, discussing the issues associated with each of its members. Section 2.2 addresses snapshot queries, while Section 2.3 discusses continuous queries. Section 2.4 covers query processing architectures with an emphasis on service-oriented architectures. Finally, we present our conclusions in Section 2.5.

2.1 Query processing in dynamic environments

To properly characterize query processing in dynamic environments, we need to first take into consideration the existent data models over which querying has traditionally been supported.

2.1.1 Data models

Several data models[1] and enhancements over them have been proposed and utilized in DBMSs. Next we briefly discuss the most relevant ones.

- **Relational**: the relational model [Cod70] was introduced to address the inadequacies of the earlier hierarchical and network models, such as redundancy and inflexibility; it enabled to attain data independence and improved query languages. It is based on relations in the mathematical sense, namely sets of tuples formed by elements of a sequence of domains, the domains being sets of values of a particular kind. In practice, however, data are most often viewed as tables and limited to primitive values.

  Querying over relational data takes place predominantly through the Structured Query Language (SQL), whose queries are translated into equivalent algebraic expressions. The mathematical foundations of the relational model enable the definition of a formal semantics and a characterization

[1] A data model provides the means for specifying particular data structures, for constraining the data sets associated with these structures, and for manipulating the data [AHV95a]

of the properties of queries and schemas. Nevertheless, its relative simplicity makes difficult to represent the information of many applications.

- **Complex values**: to obtain greater expressiveness data models consisting of complex values were developed [AHV95b], which however keep strong links with the relational model. The simplest type of complex values is relations with multivalued attributes. Going a step further, the nested relations model [RKS88] enables to arbitrarily nest relations. Taking into account the concepts of sets and tuples present in the relational model, we can obtain still greater flexibility by enabling the arbitrary nesting of both rather than just of relations, thus enabling sets of sets or tuples containing tuples [AB95, BNTW95].

  The definition of algebraic operators over complex values poses several challenges. It is necessary to enable access to the nested structures and there is a very large number of possible data manipulations. All approaches include operators that nest and unnest the complex value structures, which are complemented with adapted versions of the relational operators. A straightforward adaptation of the relational operators can result in deeply nested algebraic expressions that are difficult to understand.

  A solution to this problem is to enable recursion within individual operators. A model consisting of recursive operators accessing data elements by means of recursive projection expressions is presented in [SS86]. A data model that adopts recursion but does not have this type of dependencies in operator expressions is given in [AC88], we consider it to be more in accordance with the established principles of algebraic specifications.

- **Object-oriented models**: these models were proposed to bridge the gap between databases and programming languages. One line of work focused on providing the persistence and transactional support present in DBMSs to object-oriented programming languages. Another aimed at incorporating the features of object-orientation such as encapsulation, class hierarchies, and dynamic binding to databases. These efforts culminated in the development of object database systems such as O2 [Ban92] and SAMOS [DFG$^+$03]. The creation of standards ensued, lead by the Object Data Management Group (ODMG) and resulting in a standard for object databases that included the Object Query Language (OQL).

  Although object databases did not attain a large market share, in part because it was unfeasible to achieve full compatibility with multiple programming languages and applications, object-oriented concepts were incorporated to relational databases. This resulted in object-relational databases that support user-defined types, functions, and operators. Although the type definitions currently used in services are not as rich as in OR-DBMSs, the ability to use user defined functions and operators is highly desirable in the dynamic environments that we address.

- **Semi-structured data models**: they are used to represent data often with no explicit schema and with irregularities. They can be formalized by labeled trees and in some instances by graphs [ABS00]. By far the most common realization of semi-structured data is XML. Despite the fact

that XML is widely used for data exchange and serialization, its use as a data model presents several drawbacks. The prevailing practice is still to avoid irregularities by imposing restrictions on the acquisition of data. Furthermore, in its full generality XML presents some of the shortcomings of the hierarchical and network models [HS05b].

However, with languages such as XQuery and XSL, XML data processing and transformation is greatly facilitated, which is advantageous considering the significant quantities of available XML data. Furthermore, the use of these languages can be automated, an example is given in [CVZMVS06] for the integration of astronomical data sources.

As Chapter 3 shows we opt for a data model consisting of complex values and recursive operators, leaving also the possibility for user-defined operators and functions. We believe that this data model presents adequate levels of expressivity and usability. Furthermore, it is largely compatible with data services; particularly it is compatible with the widely used JavaScript Object Notation (JSON) data format.

2.1.2 Dynamic environments

With the purpose of characterizing the context that we address, we first introduce an example that helps us to identify the characteristics of queries in dynamic environments and the issues at hand. After that, we identify a space of data communication patterns among the actors that participate in the evaluation of a query, which is useful to understand the processes underlying query evaluation in dynamic environments. Both elements serve us to then define a query taxonomy.

Taxonomical example

Let us consider an application for guiding and assisting drivers on highways. We assume several devices and servers connected to various network infrastructures (satellite, Wi-Fi, 3G) give access to services that provide different kinds of information about traffic, weather conditions, rest areas, gas stations, toll lines, accidents, and available hotels or restaurants. Different providers can offer such services, but with different quality criteria and costs. Drivers can then ask themselves, for instance, *which are the rest areas that will be close to me in two hours and that propose a gas station, lodging facilities for two people, a restaurant, and where hotel rooms can be booked on line?*

Such a query includes traditional database aspects (*retrieve the list of hotels and their prices*) as well as continuous spatio-temporal aspects (*determine the position of the car in two hours with respect to traffic and average speed*). It may also make use of different kinds of technical services (look up, matching, data transmission, querying) as well as business services (hotel booking, parking places availability, routing).

Producer \ Consumer		Static		Mobile	
Data		snapshot	continuous	snapshot	continuous
Static	snapshot	Ss-Ss	Ss-Sc	Ss-Ms	Ss-Mc
	continuous	Sc-Ss	Sc-Sc	Sc-Ms	Sc-Mc
Mobile	snapshot	Ms-Ss	Ms-Sc	Ms-Ms	Ms-Mc
	continuous	Mc-Ss	Mc-Sc	Mc-Ms	Mc-Mc

Figure 2.1: Data communication patterns

2.1.3 Data communication patterns

As illustrated by the preceding Example, queries in our context may be concerned with data at a single point in time, which we refer to as a snapshot, or with data subject to continuous evolution during a certain time interval. Thus, we can identify three basic dimensions in our characterization of what we refer to as *data communication patterns*, which are presented in Figure 2.1.

The alternatives for data producers are presented vertically, while those for data consumers are presented horizontally; both can be either static or mobile depending on whether they change their geographical location. In turn, each producer or consumer of data can respectively produce or consume data either in snapshots, or in a continuous fashion. Each of the possible combinations represents a particular data communication pattern.

For instance, a station tracking the location of a car for a given time interval represents a static consumer of data operating in continuous fashion; the data the station consumes is in turn produced continuously by a mobile producer, the car (assuming it reports its location periodically as a stream), thus representing the scenario of quadrant Mc-Sc. Alternatively, the station may only require the location of the car at precise moments, therefore acting as a static snapshot consumer of data. If the car also determines and reports its position given a request, we arrive at the scenario of quadrant Ms-Ss.

In this manner, we obtain an overview of the core aspects that influence querying in dynamic environments. Queries involving entities interacting under different data communication patterns will be associated with different operations, and thus will require different algorithms and techniques.

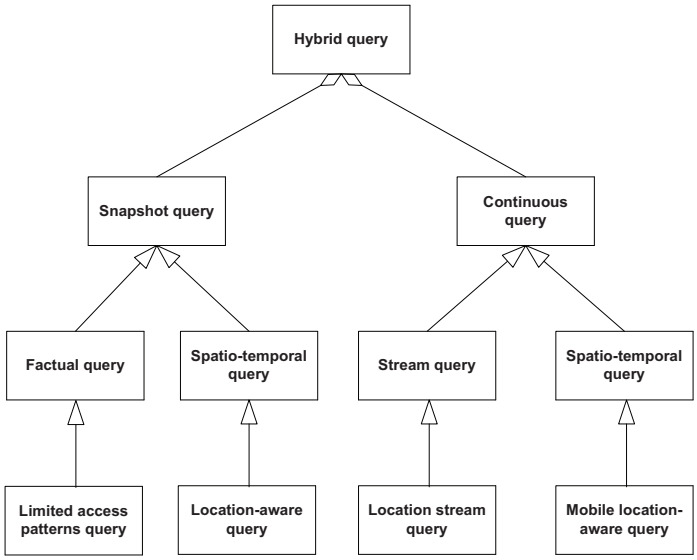


Figure 2.2: Query taxonomy

2.1.4 Query taxonomy

We examine query processing issues and cover the different aspects present in dynamic environments through a taxonomy that extends the work presented in [VSIC+10].

Concretely, we propose a taxonomy of queries in dynamic environments that takes into consideration the dimensions of data communication patterns: the mobility or not of data producers and consumers, the frequency of query execution (i.e., continuous or snapshot), the data production rate (i.e., as a continuous stream or a snapshot), the validity interval of query results with respect to new data production (i.e., freshness), and the results pertinence with respect to the consumer location.

Our taxonomy, depicted in Figure 2.2, consists of two general families of queries defined by the frequency in which a query is executed: snapshot query and continuous query. Subfamilies of queries are then defined for each family according to: the data characteristics (e.g., spatio-temporal properties), the possible mobility of data producers and consumers, and the data production rate (in a continuous stream or in snapshots). In some cases, like in Example 1.1, a single query may incorporate aspects or subqueries pertaining to more than one subfamily, which results in what we call a hybrid query. By convention, queries belonging to a particular subfamily are also considered hybrid queries.

For each subfamily of hybrid queries the existing approaches have introduced data model extensions (to represent streams as well as spatial and temporal attributes), query languages (with special operations for dealing with stream, spatial, and temporal data) and query processing techniques implemented in specific systems. We next discuss these aspects for each of the two main families in their respective section.

2.2 Snapshot queries

A snapshot query is executed once or discretionarily over one or several data producers and its results are transmitted immediately to the consumer. This type of query presents the following specializations.

1. **Factual queries**. These correspond to the queries supported by traditional DBMSs as well as by object databases and mediation systems. They arise in conventional or data integration applications. Typically they involve the derivation by conventional operations of new facts from those stored in the data sources. In some cases, particularly for mediation systems, the data may be under access limitations; meaning that only subsets of the data are obtainable by supplying specific data values, in such cases we speak of **limited access patterns queries**. For example, *give me the titles of the movies released in 2010 that have a rating in the Internet Movie Database*[2] *greater than 7.0*.

2. **Spatio-temporal queries**. These queries involve entities with attributes classified as spatial[3] (e.g. location, shape) and temporal (e.g. validity, duration), over which spatio-temporal predicates are evaluated. For example, *give the name and geographic position of the gas stations located along the highway A48*; or, *which is the average number of cars that traverse the border between 10:00 and 11:00 am*. Often the validity and pertinence of the query results are determined by the location of its consumer, defining what we refer to as **location-aware queries**. For example, *give the identifier and geographic position of the police patrols close to my current position*.

Next, we briefly discuss factual queries while focusing on the limited access patterns variant. Afterwards, in relation to spatio-temporal queries we analyze the evaluation issues for range and k Nearest Neighbor (k-NN) queries involving static objects; then we analyze moving objects and probabilistic queries for location-aware queries in particular.

2.2.1 Factual queries

Traditional relational database queries represent the most widespread type of factual queries. The techniques used to evaluate them efficiently are widely known (e.g., [GMWU99]). Efficient query evaluation plans depend on the minimization of intermediate results, while the efficiency of operator algorithms is measured primarily by the number of disk accesses, the utilized RAM also being a major factor. A particularly relevant and extensively studied type of relational queries is conjunctive queries, these are queries involving the select-project-join operators (with equality predicates only).

The multiplication of databases, each by itself incapable of providing all of the information required by its users, led to the development of data integration systems supporting new forms of factual queries. Federated database systems [SL90] enabled several autonomous and possibly heterogeneous database systems to cooperate, making it possible to issue queries over a federated (or a higher-level external)

[2] http://www.imdb.com/

[3] This term can refer either to the topology or to the exact geometry of objects, e.g. whether one country is connected to another by a border versus the exact delimitations of such border, respectively.

schema defined over the exported schemas of the component databases. Making a large number of databases cooperate in a federation is difficult, and often only querying rather than global transactions is needed. An alternative is to employ a mediator architecture [Wie92], in which several data sources are each accessed by a wrapper, which transforms the retrieved data into the form required by the mediator to combine them, under a mediation schema, into the global results.

The use of wrappers brings two major issues related to their capabilities [MG01]: (i) wrapper query processing capabilities, and (ii) data access restrictions. The first implies that some query processing tasks such as the evaluation of subqueries or query operators can be performed by the wrapper. Each wrapper can have distinct capabilities and it is necessary to employ them in an optimal manner. For this purpose DISCO [Naa99] proposes a model in which wrappers can export cost formulas and statistics by a hierarchy of rules, which can then be used by an optimizer. Garlic [ROH99] adopts a similar approach.

The second issue implies that the data of a source can often only be accessed in a restricted manner. Meaning that it may not be possible to scan the data in its entirety to answer a query, instead some input values need to be supplied in order to obtain their related data subsets. Thus data sources are associated with **limited access patterns**, formalized in [RSU95] as *binding patterns*, in which *bound* attributes must be provided to obtain the *free* attributes of a relation. In [RSU95] it is shown that answers to conjunctive queries with binding patterns can be found in NP time. Some queries are unfeasible, since it is not possible to access all of the necessary data.

An algorithm to obtain an optimal plan for conjunctive queries under binding patterns is presented in [FLMS99]. It is based on extending the traditional dynamic programming algorithm [SAC+79] with branch-and-bound search and a best-first strategy. A similar approach is employed in [BCG10] in the context of queries involving search services and ranking.

2.2.2 Snapshot spatio-temporal queries

A spatio-temporal query[4] involves spatial and temporal relations between objects. Several query subtypes arise from such relations, among which we focus on the next two[5].

1. **Range queries**. A range/region is specified and all objects located in (or intersecting) it are retrieved (recently addressed in [XW03, TWHC04, YK06]). We again assume that the region of interest does not change with time in position or extent. The range can be explicit (e.g., *find the hotels located along highway A48*) or implicit (e.g., *find hotels located within 5 km*, denoting a circular region centered in the user location).

2. **Nearest Neighbor(s) queries**. In their simplest form, based on a particular distance metric retrieve the object that is the closest to a certain object or location (Nearest Neighbour -NN- query [TXC07]). For instance, *find the closest gas station to my current position*. When the closest *k* (a

[4]Some authors define these queries as location-dependent [IMI06] or location-based [HJ04] queries; we take into consideration not only if the data is location related, but also the role played by the current location and context of the user issuing the query, which leads to the location-aware subfamily.

[5]Another important subtype is *join* queries involving spatio-temporal predicates [Pap04].

positive integer) objects are required then the query is a k-NN query. For instance, *which are the two closest gas stations from my current position*. [CAEA03] classifies NN and k-NN queries as static when the reference object is not mobile and dynamic when it is mobile.

2.2.2.1 Data models

Queries involving spatio-temporal predicates have been an object of research since the 1980s, leading to spatial and temporal databases. During this time several data models have been proposed for representing and reasoning on spatial and temporal data [All83, EF91, PT97, AZM99]. Concerning spatial data, objects in space are represented by types such as point, line, poly-line and polygon. Additional constructs enable to represent spatial objects (more) approximately as rectangles (Minimum Bounding Rectangle -MBR-) or as circular regions (Minimum Bounding Circle -MBC-). According to the representation, spatial relations are defined involving direction, topology and metrics.

An alternative paradigm [RSSG03] is to model spatial objects as infinite sets of points, providing a finite representation of these infinite sets with (for instance, linear) *constraints* over a dense domain. Queries can then be expressed as if the infinite extension was stored in the database, and objects can be manipulated by standard set operations (e.g., relational algebra).

For representing time, existing data models are either instant or interval oriented. The former represent duration by a set of instants, where each instant belongs to a time line where the origin is arbitrarily chosen. The latter adopt intervals for representing durations and for reasoning about time. Allen's 13 interval relations [All83] are often used for this purpose.

Most models enable the specification of additional properties associated with objects in spatial and temporal spaces by the use of standards for specifying metadata, like Dublincore [6] or FGDC [7]. Querying can then be performed with respect to the spatial and temporal attributes of the objects and also with respect to the metadata (e.g., processing method, quality, contributor).

2.2.2.2 Spatio-temporal query processing

Some works distinguish between spatio-temporal queries that ask for the geographical position of an object explicitly (position query, *get my current position* [BD05]) or implicitly (*give the geographical positions of the hotels of the 16ème arrondissement* [IMI06]).

In relation to range queries, according to the underlying spatial data model, the range of a query can be given by (i) a rectangular region[8], the most compatible with spatial access methods; or (ii) a circular region, which implies objects within a certain distance of a particular point. A n-body constraint query [XJ07] specifies a constraint by which a predefined set of n objects must lie inside a specified circular range; for example, *are all the city buses within 15 km of the bus station?*

NN and k-NN queries also present several variations. (i) a reverse NN query [BJKS06, WYCT08] retrieves the objects that have the query point as their nearest neighbor. (ii) a reverse k-NN query

[6] http://dublincore.org/
[7] http://www.fgdc.gov/metadata
[8] Some authors like [ZZP+03] use the term window instead, we refrain from it to avoid confusion with stream queries.

[BJKS06, WYCT08] retrieves the objects that have the query point among their k nearest neighbors. (iii) a constraint NN query specifies a range constraint for the list of retrieved objects [FSAA01]; for example, *which are the 5 closest gas stations from my current position located within 120 km.* (iv) a k closest pairs query [CMTV00] retrieves the k pairs of objects with the smallest pairwise distances; for example, *retrieve the 3 gas station and hotel pairs which are closest from each other*.

Navigation queries, beyond the scope of our work, retrieve the best path for an object to get to a destination according to the underlying road network and other conditions such as the traffic [HLCR03, LCH+05]. For example, *how to get from my hotel to the train station*.

The evaluation of spatio-temporal queries relies heavily on spatio-temporal access methods that, like their counterparts in traditional databases, greatly reduce the number of objects that need to be examined to answer a query. The R-tree [Gut84] and its variations has proven to be particularly useful, first for spatial data and later incorporating time. In an R-tree each intermediate node corresponds to a MBR that bounds its children, while leaves contain pointers to the database objects; the MBRs of intermediate nodes can overlap.

For range queries, large numbers of objects can be discarded by comparing the range to the MBRs of the nodes. To evaluate a NN query in an R-tree the tree is navigated downwards from the root, discarding some branches according to predefined rules, until the NN is found. By incorporating branch-and-bound search it is possible to evaluate efficiently also k-NN queries [Pap04].

Research on spatio-temporal access methods remains active today. For example, [TVM00] proposed (i) an access method based on overlapping linear quadtrees to store consecutive historical raster images; (ii) algorithms that exploit that access method to process queries on a database of evolving images. A survey and lineage of spatio-temporal access methods is given in [MGA03, NDAM10].

2.2.2.3 Existing projects and systems

Generations of Geographical Information Systems (GIS), starting in the late 1960s, have brought important contributions for dealing with geospatial data and queries, motivating and incorporating database research. The ArcGIS[9] line of products from ESRI, the leading vendor of GIS, provides interoperability with DBMSs. The golden age of spatio-temporal database systems (around the 1990's) led to the development and popularization of data models, query languages, indexing and optimization techniques [BCT+97]. Presently the major commercial DBMSs have packages for dealing with spatial data and queries.

The emergence of dynamic environments introduces the need of systems that use new techniques for dealing (reasoning) with data that has spatial and temporal properties, its producers and consumers, in different contexts like sensor networks, reactive systems, mobile ad hoc networks (MANETs), among many others. For example, [PZMT03] proposes a solution for sensor network databases based on generating a search region for the query point that expands from the query, which works similarly to Dijkstra's algorithm. The principle of this solution is to compute the distance between a query object and its candidate neighbors on-line.

[9] http://www.esri.com/software/arcgis/

2.2.3 Location-aware queries

The location of the consumer of the query result is used to determine whether an object belongs to that result (location-aware query [SDK01]). For example, *find the police patrols 100 km around my current position*. In [MHMM05] this query is described as moving object database query[10]. A moving object (mobile producer) is any entity whose location is interesting for some data consumers.

2.2.3.1 Data models

Moving object databases [The03, WM05, GS05] extend traditional database technology with models and access methods adapted for efficiently locating and tracking moving objects. Modeling moving objects implies representing their continuous motion. The relational model enables an explicit representation by sampled locations [NST99], but this approach can imply expensive continuous updates to obtain a good precision.

One of the most popular models, MOST (moving objects spatio-temporal), was proposed in the DOMINO Project [SWCD97a]. It represents the current and future expected movement of a moving object by attributes such as starting location, route (a line), direction, and speed. Queries are specified on a SQL-like query language incorporating Future Time Logic (FTL) predicates. It also introduces the notion of *dynamic attributes*, whose values evolve according to the attributes that determine them, even if no explicit updates are executed. Movement histories are represented in [GBE+00] by data types such as moving point, moving line, and moving region; an extension for network constrained movement is described in [DG04]. Both, the current and historical movement lines of work are described in [GS05].

Additionally, an approach based on differential geometry to model moving objects is given in [SXI01]. A model for moving objects on road networks that takes advantage of commercial systems support is described in [VW01]. TQ [MS05] is a constraint-based query language for object trajectories.

2.2.3.2 Location aware query processing

Location-aware queries are classified according to the way moving objects interact among each other: whether objects are aware of the queries that are tracking them, whether they adopt an update policy, whether they follow a predefined trajectory. The update policy is the strategy adopted for locating moving objects, either the moving object itself communicates its location or there is a mechanism for periodically locating it. Such strategies are out of the scope of this work, but the interested reader can refer to [CCMC08] for more details.

Probabilistic querying [SWCD97b, PJ99] implies estimating the location of moving objects (moving producers) using different techniques. Two variants are described in [DYM+05], (i) a threshold probabilistic query retrieves the objects that satisfy the query predicates with a probability higher than some threshold; (ii) a ranking probabilistic query orders the results according to the probability that an object satisfies the query predicates. [TXC07] proposes static probabilistic range queries called probabilistic

[10] Our classification considers that a query in this context is continuous only if the mobility of the consumer of the results (or of the objects of interest) necessitates the proactive generation of new results.

range search, distinguishing between nonfuzzy search over a given region and fuzzy search around a given object. Probabilistic NN queries [KKR07] give the probability of each object of being the NN of a given reference object, constrained probabilistic NN queries [CCMC08] generate only the objects above some threshold, with a given error bound.

In general, the evaluation of location-aware queries requires access to some form of positioning system. It can be satellite-based like the Global Positioning System (GPS); indoor like infrared or radio beacons, as well as RFID tag identification; or network-based like cell identification or uplink time of arrival in wireless telephone networks [Rot04]. The technology employed has an effect on the location update policy, the data accuracy, and the types of queries that can be evaluated.

2.2.3.3 Existing projects and systems

During the last decade, many research works have focused on modeling and querying moving objects [GS05]. DOMINO [TWHC04] evaluates moving object queries employing prediction techniques and incorporating uncertainty operators (e.g., `Always_Definitely_Inside`) that are implemented as user-defined functions in Oracle, the DBMS over which DOMINO is built. SECONDO [GDF+99] is an extensible DBMS that enables the incorporation of diverse data models by its second-order signature formalism and an optimizer implemented in Prolog; the implemented data models include one for moving objects [dAGB06].

A mobile peer-to-peer (P2P) database is a database that is stored in the peers of a mobile P2P network [LWX08]. Such a network is composed of a finite set of mobile peers that communicate with each other via short-range wireless protocols, such as IEEE 802.11. A local database stores and manages a collection of data items on each mobile peer. MOBI-DIC [CWXY05] and MOBIEYES [GL06] focus on query processing on mobile P2P databases. They incorporate data dissemination and partition schemes aimed at reducing the number of required location updates. MOBI-DIC particularly processes queries on vehicles searching for available parking spaces [XOW04]. Reports are generated by vehicles leaving a parking space and diffused to neighboring vehicles. The mobile query processor receives streams of reports and processes them to compute the query result (e.g., *what are the five nearest parking spaces?*).

A new type of query adapted to disconnected mobile networks is introduced in [ZXW08], referred to as spatial-existence query. They explain that processing k-NN queries is too complex in such decentralized contexts. They claim that the most important for the user is not necessarily to get all possible results but rather to know at least one of them. They therefore propose a strategy for disseminating queries in order to retrieve relevant information on remote peers. They also propose a solution to deliver the obtained results to the data consumer (i.e., the query issuer).

2.3 Continuous queries

A continuous[11] query can arise in the next cases

[11] Continuous in our context means that something proceeds without interruption, including the execution of a task periodically.

1. **Stream queries**. A query is evaluated over streams, meaning data that are produced continuously and in significant amounts. The query keeps producing results as long as stream data are generated or the consumer is interested in receiving such results. For example, *how many vehicles have entered and left highway A48 since 7:00 am*; or, *give me the average vehicle speed during the last hour on highway A48*.

2. **Mobile location-aware queries**. A query is issued by a mobile consumer in a context where the validity and pertinence of its results change according to the consumer location, or that of the data producers. Furthermore, the consumer wants to be kept up to date with such changes. For example, *which are (and will be) the nearest gas stations with respect to my current location?*

3. A query is executed repeatedly and at regular time intervals, i.e., periodically. For example, *inform me every hour of the temperature measured in the greenhouse*.

4. A query monitors the occurrence of a particular event and signals it via a notification, possibly subject to additional conditions. For example, *when there is an accident on the road, give me its coordinates, if it is within 20 km of my current location*.

The notion of *complete* query result used for traditional databases makes no sense for continuous queries; since they either produce results with associated validity intervals, or their results are constantly updated for ensuring validity. Both, the data production rate and consumer mobility have an impact on the query results validity and pertinence. These identify the two types of continuous queries on which we focus our attention: stream queries and mobile location-aware queries.

2.3.1 Stream queries

In contrast to data in persistent storage managed by traditional databases, data streams are continuously produced in large amounts and must be processed in real time [BBD+02, GO03]. Queries over data streams are continuous (generate results across time), persistent (stay in execution in the system), long-running (typically from minutes to days), and return results as soon as the conditions of the queries are satisfied. For instance, *every hour report the number of free parking places available in the shopping mall*. As for the other query types, dealing with stream queries requires adequate data models. In the case of streams where data is produced continuously, bounding, aggregating, timestamping and filtering strategies are required in order to query and correlate streams stemming from different producers.

2.3.1.1 Data models

A data stream is an unbounded sequence of data items that can only be read once and following their specified order. Most works use the notion of infinite sequence of tuples for representing a data stream, ordered implicitly by arrival times or explicitly by timestamps. In the latter case tuples can either be timestamped by the stream producer (e.g., with the production instant of the tuple), or by the stream consumer (e.g., with the arrival time of the tuple). The explicit order given by the timestamps can be used

to correlate different stream producers. Synchronization problems can arise due to different production and arrival times as well as diverging clocks. Global clock and distributed event detection strategies can be used for solving these problems.

As it is noted in [JMS+08], a timestamp-based model permits to represent simultaneity (by two values having the same timestamp), while a tuple-based model enables immediate response to events (arrival order overrides equal timestamp values). Both models have their advantages and disadvantages, reflected in the types of queries they can express. Both models can also be represented by incorporating the granularity of *batches* to the reception of a stream. Furthermore, by controlling tuple batches and the order among them new types of queries can be expressed, details can be found in [JMS+08]. This principle is extended in the AStrAL [PLR10] stream algebra to define positional and cross-domain windows (e.g., *the last 10 tuples from the last 100 tuples* and *the last n tuples each m seconds*, respectively).

The major data models have been extended to deal with streams [GO03]: the relational model, the object model, and XML. We next discuss illustrative instances of each in their listed order.

[Gro03] defines a formal abstract semantics using a *discrete, ordered time domain* (enabling simultaneity), where relations are time-varying bags (multisets) of tuples, and streams are unbounded bags of timestamped tuples. Three types of operators are defined: relation-to-relation (selection, projection, join, etc.), relation-to-stream (*Istream* and *Dstream* to represent the streams of insertions and deletions on relations, and *Rstream* to generate a stream from the current state of a relation), and stream-to-relation (windows to bound streams). SQL is extended to create CQL (Continuous Query Language), which enables to express stream queries declaratively.

Instead of employing the *Istream* and *Dstream* operators that hide the explicit changes made to a relation, an alternative approach discussed in [GELA10] is to put those explicit changes directly in the results with *tagged tuples* (a positive tag denotes insertion and a negative tag deletion). This facilitates the composition of streams and enables the definition of *views* over them, analogous to that of traditional databases.

[BGS01] models stream data sources, particularly diverse types of sensors, with Abstract Data Types (ADTs). The interface of each ADT consists of the signal-processing methods it corresponding sensor supports. A language with a SQL-like syntax is adopted, enabling querying over sensor networks.

[KSKR05] defines an XML schema of tuples that represent sensor readings and that can include temporal and spatial attributes. XML data streams are then queried using XQuery expressions.

2.3.1.2 Stream query processing

In contrast to traditional query systems, where each query runs once against a snapshot of the database, stream query systems support queries that continuously generate new results (or changes to results) as stream data continues to arrive [AXYY06]. Infinite tuple streams potentially require unbounded memory space in order to evaluate operators such as full joins, since every tuple in one stream should be compared with every tuple from the other stream. Meaningful results can alternatively be obtained by bounding streams: a *window* specifies a bounded subset of tuples from a stream. With windows, joins, for instance, handle bounded sets of tuples and traditional techniques can be applied, the symmetric hash-join [WA91]

being particularly useful. Extensions of classical query languages have been proposed to deal with these issues, mainly introducing various types of windows and aggregation.

Sliding windows have a fixed size or *range* (in number of tuples or time interval) and continuously move forward (e.g., *the last 100 tuples, tuples within the last 5 minutes*). By adding an explicit *slide* factor other window types can be defined [Kri06]: hopping windows have a slide greater than their range and move by hop (e.g., *5-minute window every 7 minutes*); tumbling windows have the same slide and range and are contiguous (e.g., *5-minute window every 5 minutes*); overlapping windows have a range greater than their slide (e.g., *7-minute window every 5 minutes*). In CQL an additional `SLIDE` clause is used for this purpose, while in [CCD+03] the window upper and lower bounds are defined with a `for` statement similar to that in programming languages. CQL also defines a partitioned window as the union of windows over a partitioned stream based on attribute values (e.g., *the last 5 tuples for every different ID*).

SQuAl, the query algebra of Aurora [ACc+03], distinguishes between order agnostic and sensible operators. The first are: *filter* (basically traditional selection), *map* (apply a user-defined function to all tuples), and *union* (merge streams with common schemas). The latter incorporate window constructs and include: *BSort* creates an approximate ordering of a stream on a given attribute with a semantics related to bubble sort; *Aggregate* applies SQL-style aggregates or user-defined functions to a stream; a *Join* which enables different assumed orderings; and *Resample* that implements interpolation on a stream.

Timestamped data streams can incorporate location data enabling **location stream queries**. For instance, PLACE [MXHA05] proposes spatio-temporal operators and predicate-based windows for such streams. A range operator (by an *INSIDE* clause) and a *k*-NN operator (by a *kNN* clause) can be used to filter the results. Both can be defined either over a static region (for *INSIDE*) or point (for *kNN*) or be linked to mobile objects. Predicate based windows generalize tuple and time-based windows by defining windows based on arbitrary predicates. For instance, *the difference between consecutive timestamp values must be less than a threshold* (timestamps are not directly accessible in some languages like CQL). In [PS07] spatially-aware windows are proposed: extent-based windows (similar to range), *k*-proximity windows (similar to *k*-NN), distance windows, and tessellated windows (which partition objects by a tessellation[12]).

For the time being there is no standard data model and query language defined for streams. As discussed, different semantics are appropriate for particular cases and query types. Also to consider is a languishing interest (at least in publications) on SQL standards since SQL:2003.

2.3.1.3 Existing projects and systems

Stream query processing has attracted much interest in the database community because of a wide range of traditional and emerging applications. For example, trigger and production rules processing [HCH+99], data monitoring [CcC+02], continuous queries [BW01], and publish/subscribe systems [LPT99, CDTW00, PFJ+01, DFK05]. Several stream management systems have been proposed in the

[12] A tessellation (or tiling) of a plane is a collection of plane figures that fills the plane with no overlaps and no gaps.

literature such as STREAM, TelegraphCQ, and Aurora/Borealis for data intensive streams; NiagaraCQ for XML streams; COUGAR, TinyDB, and GSN for sensor networks; Cayuga for complex event monitoring, and PLACE/PLACE* for spatio-temporal streams.

STREAM [Gro03] defines a homogeneous framework for continuous queries over relations and data streams. It employs computation sharing techniques over several continuous queries, enabling the simultaneous execution of a larger number of queries. TelegraphCQ [CCD+03] provides adaptive continuous query processing over streaming and historical data. Adaptive group query optimization is realized by dynamic routing of tuples among (commutative) query operators depending on operator load.

Borealis [AAB+05] extends the work on Aurora [ACc+03] into a Distributed Stream Processing System (DSPS) that enables distributed query processing by defining dataflow graphs of operators in a "box & arrows" fashion. Boxes are computation units that can be distributed, and arrows represent tuple flows between boxes. Adaptive query optimization is done through a load-balancing mechanism. Load-shedding is performed when the rate of the incoming streams are too high, therefore some tuples are discarded in favor of the generation of approximate results.

NiagaraCQ [CDTW00] introduces continuous queries over XML data streams. Queries, expressed using XML-QL[13], are implemented as triggers and produce event notifications in real-time; alternatively, they can be timer-based and produce their result periodically. Incremental evaluation is used to reduce the required amount of computation. Furthermore, queries are grouped by similarity in order to share a maximum of computation between them. They also adopt the notion of "action" to handle query results. An action can be any user-defined function, they are commonly used to send notifications to users (e.g. a "MailTo" action).

COUGAR [DGR+03] and TinyDB [MFHH05] process continuous queries over sensor networks optimizing energy consumption by using in-network query processing. Each sensor has some query processing capabilities, and thus distributed execution plans to handle that can include online data aggregation, which reduces the amount of data transmitted through the network. Global Sensor Networks (GSN) [AHS07] is a middleware for sensor networks that provides continuous query processing facilities over distributed data streams. Continuous queries are specified as virtual sensors whose processing is specified declaratively in SQL. Virtual sensors hide implementation details and homogeneously represent data streams either provided by physical sensors, or produced by continuous queries over other virtual sensors.

Complex Event Processing (CEP) or Event Stream Processing (ESP) techniques express specialized continuous queries that detect complex events from input event streams. Cayuga [DGP+07] is a stateful publish/subscribe system for complex event monitoring where events are defined by SQL-like continuous queries over data streams. It is based on algebraic operators that are translated to non-deterministic finite automatons for physical processing.

PLACE [MXHA05] enables the scalable execution of continuous queries over spatio-temporal data streams, incorporating through the SOLE [MA08] algorithm features like load-shedding. PLACE* [XECA07] follows that line of work and takes query evaluation to a distributed setting, in which multiple

[13] http://www.w3.org/TR/NOTE-xml-ql/

servers collaborate to generate query results.

Academic research has been incorporated into commercial data stream management systems such as StreamBase[14] and Truviso[15]. Their use is limited, however, because for the main target applications, such as high-frequency trading, special-purpose systems are still prevalent.

We next focus on mobile location-aware queries, a subtype of continuous spatio-temporal queries in our taxonomy. Continuous spatio-temporal queries that are not location-aware can involve, for example, weather monitoring or traffic analysis, but we consider them outside the scope of our work. We believe some important challenges remain in (mobile) location-aware queries, such as: dealing with data production rates, the validity and pertinence of query results, the accuracy of location estimations, and the synchronization of data timestamped under different clocks.

2.3.2 Mobile location-aware queries

Mobile location-aware queries are continuous and their results must be refreshed repeatedly because their validity changes as the producers move. For example, *continuously locate the police patrols that are within 10 km of my current position*. Alternative names for these queries given in the literature include: continuously moving location queries [GL06], location-dependent queries [IMI06], and continuous queries on moving objects [PXK+02]. Query subtypes include range-monitoring queries [CHCX06] and continuous nearest neighbor queries [FGPT07, ZLLL07]. The distinction from location stream queries lies in that object locations are not transmitted in streams, or at least not at a high rate. We also note that a related issue not covered in this work is transactions and data consistency in location-dependent data [DK98].

2.3.2.1 Data models

The data models used for snapshot spatio-temporal and location-aware queries can be extended for mobile location-aware queries. [GS05] distinguishes the continuous model that represents moving objects as moving points that start from a specific location with a constant speed vector, and the discrete model where moving objects locations are timestamped. Indexing techniques are associated to these models in order to evaluate queries efficiently. The challenge is to maintain valid results in the presence of continuous motion, both on the reference object and on the objects of interest. The main strategies are to restrain results validity either to a given time interval or to a given region in which the reference object must be located. In addition, prediction models can be used when objects follow trajectories to generate estimated results [MA07].

2.3.2.2 Processing mobile location-aware queries

A moving range query is a query whose result consists of objects that satisfy the spatial restriction specified by the range (e.g., a circular or rectangular area) at a given moment, therefore its results must

[14] http://www.streambase.com/
[15] http://www.truviso.com/

updated repeatedly because the range moves along with the reference object. For instance, *continuously provide me the gas stations located within 5 km of my position*. Since *my position* changes as I am driving, then at different instants in time the query result will contain a different list of gas stations. Furthermore, I do not want to have to issue repeatedly a query to obtain such list. Continuous k-NN queries follow an analogous scheme.

Most of the existing techniques for handling mobile location-aware queries focus on developing high-level algorithms that exploit increasingly improved spatial access methods. For instance, the special case of NN queries with a moving query reference point and static objects of interest is addressed in [ZL01] by employing Voronoi diagrams[16]. However, as noted in [ZZP+03], Voronoi diagrams are expensive to update, and if they are of order-k also expensive to compute and store. The R-Tree is used in [SR01] to evaluate k-NN queries for a moving reference point.

In [LPM02] techniques are given to evaluate range queries involving moving objects (both reference object and objects of interest). Two variants are addressed, predictive queries in which it is possible to estimate the location of the reference object, and non-predictive when there is no such possibility. Range and k-NN queries over moving objects are associated in [ZZP+03] with validity regions. Although precomputed Voronoi diagrams are not used, a means to compute *Voronoi cells* is devised. Both of these works assume the use of R-Trees or another spatial access method with similar properties. Algorithms to evaluate k-NN and reverse k-NN queries over moving objects, taking advantage of the TPR-Tree [vJLL00], are provided in [BJKS06].

The data structures and algorithms used for mobile location-aware queries must be adequate for incremental evaluation, meaning that only updates and not entire new result sets are generated. A useful optimization technique can be to preemptively generate such updates [MA07], for instance, obtain more than k objects or evaluate over a larger range. A related problem to the queries discussed so far is characterized in [HKGT03] by density-based queries. These refer to the number of objects present in a particular region, for example, *notify me of the parking lots that have more than 100 vehicles*.

2.3.2.3 Existing projects and systems

The emergence of handheld devices and wireless networks of the last years has strongly impacted query processing techniques [IN02]. Mobile devices are characterized by limited resources in terms of computing power, memory, storage capacity, data transfer, and battery life. They also undergo intermittent connectivity due to frequent disconnections. In network or indoor-based location, mobile devices can change from one coverage area to another (handoff/handover) [MPKS04, Ses96], thus the infrastructure needs to be taken into consideration. Furthermore, all location methods necessitate a location update policy.

Consequently an important challenge in mobile contexts is to adapt data management and query processing. Mobile databases key differences with traditional database systems include [IB93, PS97, Bar99, PB99]: data fragmentation and replication among mobile units, consistency maintenance, approximate

[16] A Voronoi diagram is a special kind of decomposition of a metric space determined by distances to a specified discrete set of objects in the space.

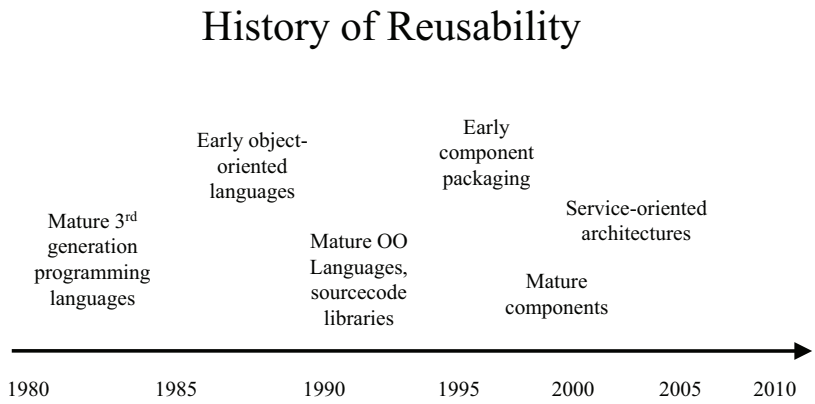


Figure 2.3: Recent historical outline of reusability in software [Pol06]

answers, query fragmentation and routing; as well as query optimization with respect to battery consumption, cost of wireless communication, uncertainty of results. There are two types of mobility according to [IMI06]: (i) software mobility implies transferring passive data (e.g., files) or active data (mobile code, process migration) among computers; (ii) the physical mobility of devices. All of them must be addressed in mobile location-aware (and generally in hybrid) queries.

Most of the existing location-aware query processing systems rely on a client/server architecture mirroring Location-Based Services (LBS), in which location data are sent to a centralized server that will index them and process the queries. SINA (Scalable INcremental hash-based Algorithm [MXA04]) represents an alternative approach to index-based techniques, with it shared query evaluation is abstracted as a spatial join between a set of moving objects and a set of moving queries. Location-aware query processing has also been addressed by projects like DATAMAN [IB93] and in [SDK01]. Adopting a distributed approach, the ISLANDS project proposes evaluation strategies based on query dissemination [TD04]. The LOQOMOTION system [IMI06] performs evaluation based on mobile agents.

2.4 Service-oriented computing and query processing

In this section we provide a brief analysis of the most relevant aspects of service-oriented research and technologies relevant to query processing.

Service-oriented technologies arose primarily from the need for software reuse and system integration in the context of Enterprise Application Integration (EAI) [ACKM04]. Regarding reusability in particular, Figure 2.3 presents a general historical overview covering the last three decades, developments that also influenced DBMSs and therefore are important to analyze.

The third generation programming languages (mainly C) facilitated software development but offered limited opportunities for reuse. Object-oriented languages (mainly C++) offered better prospects for reuse, especially with mature commercial implementations and standardized libraries. These devel-

opments lead to, among other innovations [DG01], extensible DBMSs such as PostgreSQL[17] that enable the incorporation of abstract data types and new index structures; kernel systems (e.g., WISS [CDKK85]) over which full DBMSs can be built; and also toolkits (e.g., EXODUS [CDF+86]) and frameworks (e.g., OPT++ [KD99] and QBF [CV04]) that facilitate DBMSs development.

While OO languages facilitated reuse, they also imposed restrictions. For instance, ObjectGlobe [BKK+01] enabled to share resources and incorporate new capabilities at the granularity of operators, but defined as standard Java classes depending on a custom dynamic class loading infrastructure. Component models pushed reusability forward by enabling components that could be compiled and linked separately, integrated at runtime, and run independently of each other. A component model also permits to encapsulate the behavior of a system by interfaces. For instance, OLE DB, based on Microsoft's COM, defines several DBMS components and was used in SQLServer 7.0.

In [DG01] four types of component-based DBMSs are identified that illustrate the difference between components as units of deployment and services. (i) Plug-in DBMSs use component models to facilitate extensibility. (ii) Database middleware employ wrappers and undertake data integration. (iii) DBMS services make available the functionality of some DBMS components to client applications. (iv) Configurable DBMS can be built to fit new requirements by the aggregation of components. In types (i) and (ii) we deal with a base DBMS, whereas in types (iii) and (iv) no such DBMS exists. Our approach fits to a large extent with type (iv) while also supporting similar functionality as type (ii), but focuses on query processing and follows requirements related to dynamic environments; concretely, we employ data and computation services accessible through standardized interfaces.

The use of services enables greater control on the computations involved in query processing and their distribution. Therefore it is adequate to analyze existing query processing architectures that offer similar capabilities.

2.4.1 Query processing architectures

We limit our analysis to two lines of work that are nevertheless illustrative of the main approaches.

Eddies

An eddy [AH00] operator is a n-ary tuple router interposed between n data sources and a set of query operators. By adaptively changing tuple routing it can impose different operator orderings, in contrast to following a single pre-established query evaluation plan. It effectively merges the optimization and execution phases of query processing favoring adaptivity over best case performance. In this manner it can respond to changes in (i) the cost of operators, (ii) the selectivity of operators, and (iii) the arrival rate of tuples.

However, some algorithms are better suited for reordering than others, which is characterized by their *moments of symmetry*. For instance, for the nested-loops join the outer and inner relations can be reordered every time an inner loop is completed (saving the cursor on the outer relation). In addition, the

[17] http://www.postgresql.org/

adaptivity effectiveness depends on adequate routing policies. A simple policy is to send more tuples to the operators with lower costs; a more sophisticated one is to send the tuples to the operators with greater selectivity, which can vary due to clustering in the inputs.

The granularity can be reduced from operators to *state modules* (SteMs) [RDH03], which roughly correspond to hash tables storing tuples from a single relation (possibly hashed on more than one attribute). With SteMs it is possible to evaluate a n-ary symmetric hash join adaptively. Alternatively to SteMs, STAIRS [DH04] support other adaptive evaluation techniques like query scrambling (aggressively joining together previously-received tuples). The idea behind STAIRs is to expose the state stored in the operators to the eddy, and allow the eddy to manipulate this state (by *promote* and *demote* operations that undo an generate results).

In any case, the use of eddies requires that tuples hold additional data about the operators they have gone through, additional processing time is required to determine routing, and in distributed settings the number of messages increases and the eddy can represent a single point of failure.

Parallel DBMSs

Parallel DBMSs are built to obtain better performance and ensure availability. Three major architectures can be distinguished [Val93]. (i) *Shared memory*, in which all processors have access to any memory or disk module. This architecture facilitates load balancing since there is fine and immediate control of the resources; however, memory conflicts can degrade performance and limit scalability. (ii) *Shared disk*, in which all processors can access the disk but each has its own memory. This facilitates extending the system but the disk can become a bottleneck, it is also necessary to employ cache coherence techniques. (iii) *Shared nothing*, meaning that each processor has its own memory and disk(s). Such configuration favors extensibility and availability, load balancing however becomes dependent on the data location.

Regardless of the architecture, the key to effective parallel DBMSs is pipelined and partitioned parallelism. Pipelined parallelism implies performing simultaneously all of the tasks required in a given sequence, but on different items already subjected to successive subsequences of the process. In the context of databases this means successive operators in a query plan. On the other hand, partitioned parallelism consists of partitioning the data by some technique (the most common being by range, hashing, and round robin), performing a process simultaneously on each part, and then merging the results. Thus the granularity of computations can be controlled, which with adequate algorithms can be applied to query operators.

For instance, Prisma/DB [AvdBF+92] is a shared nothing parallel DBMS that relies on main memory storage of the entire database. It has the characteristic that it is implemented in the Parallel Object-Oriented Language (Pool-X), in which objects behave as processes and can communicate via synchronous and asynchronous messages. This enables a highly modular architecture that supports pipelined and partitioned parallelism. In the context of Cloud computing, and not restricted to databases, the same fundamental principle (although excluding synchronous messages) is applied in the Orleans [BGK+10] framework, where *grains* are analogous to Pool-X objects.

In a dynamic environment the autonomy of data and computation services implies a restricted shared

nothing scenario. Though both pipelined and partitioned parallelism are foreseeable, we limit in our work to the former.

2.4.2 Service-oriented query processing

Next we analyze several representative works involving the use of services in query processing. Although services can take many forms, these projects as well as our own work adopt some variant of Web Services.

SoCQ

Whereas many mediation systems support query evaluation over services as virtualized tables, SoCQ [GLP10, Gri09] (Service-oriented Continuous Query) also enables the evaluation of queries over stream data services. The system is designed for pervasive computing[18] environments and employs REST [Fie00] Web Services to represent data sources and devices such as sensors, hence it is presented as a Pervasive Environment Management System (PEMS). Querying takes place through a SQL-like query language supported by the Serena algebra (Service-enabled algebra). The algebra includes the set operators (union, intersection, and difference), the relational operators (projection, selection, and renaming), and realization operators (assignment and invocation). The operators in the last category enable to access data in the presence of binding patterns.

OGSA-DQP

Services have also proven to be useful in Grid computing[19] settings. The Open Grid Services Architecture (OGSA) makes the functionality of Grid middleware, such as the Globus Toolkit[20], available through Web Service interfaces. OGSA-DQP [LMC+09] takes advantage of this architecture to support distributed query processing. Two important characteristics of this system are: (i) it supports querying over data and analysis resources made available as services; (ii) its internal architecture factors out as services the functionalities related to the construction and execution of distributed query plans, including pipelined and partitioned parallelism.

DBMS based on S3

In [BFG+08] a DBMS built on the Amazon Simple Storage Service (S3)[21] is presented. S3 permits, at a relatively low cost, to store and retrieve (by REST and SOAP Web Service interfaces) objects of variable size identified by an URI and organized into buckets. In their architecture, clients retrieve pages from S3 based on their URIs, buffer the pages locally, update them, and then write them back. This occurs

[18] To a large degree synonymous with ubiquitous computing [Wei91], a human-computer interaction model in which computing is thoroughly integrated into everyday objects and activities.

[19] A Grid is a software infrastructure that supports the discovery, access and use of distributed computational resources [Fos01]. Many similarities exist with Cloud computing, which however is more recent, usually does not require discovery, and involves major corporate support.

[20] http://www.globus.org/ogsa/

[21] http://aws.amazon.com/s3/

through a record manager that in turn depends on a page manager, B-tree indexes are implemented on the latter. Special commit protocols and checkpoint strategies are utilized to deal with a Cloud computing environment; in which, among other characteristics, clients should not be blocked and can fail at any time. The tradeoff incurred implies that only weaker consistency models can be supported, a consequence of the CAP theorem [GL02].

Active XML

The Active XML [ABM08] (AXML) project introduces an alternative approach to gather, process, and present data to the users in order to respond to their information requirements. AXML is a declarative framework that employs Web services for distributed data management, operating under a P2P architecture. The framework is based on two main concepts. First, AXML documents, which are XML documents that may contain embedded Web service calls. Second, AXML services, which are Web services capable of exchanging AXML documents. An AXML peer thus acts both as a client by invoking the embedded service calls, and as a server by providing AXML services, which correspond to queries (expressed in XQuery or XPath) or updates over persistent AXML documents. The use of service calls enables the management of intensional data and introduces several challenges and optimization opportunities, for example, the lazy evaluation of service calls.

Panta Rhei

Panta Rhei [BCGV10] was developed in the context of the Search Computing paradigm, in which multiple search services cooperate and are correlated using database techniques to obtain information. Queries in search computing can be posed in different ways, including a SQL-like language. In Panta Rhei 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. *Clock units* determine when and how many times to invoke services and enforce those decisions via control flows, from which clock units also obtain feedback. Two service types are supported, *exact services* that produce complete and unranked results in chunks, and *search services* that produce unbound ranked results in chunks.

An important remark made in [Wei07] is that DBMSs view query processing as a matching task based on testing logical predicates, whereas information retrieval (the field most closely related to search) views query processing as a ranking task based on statistical models. This has consequences in the definition of the semantics of a query. In relation to query plans in particular, dataflows typically define a Kahn network [Kah74], which is known to be equivalent to function composition. With the inclusion of control flows it is important to determine if this principle still holds, since the semantics of a query can be defined by function composition.

We observe that the existing works do not utilize the full potential and flexibility of service-oriented computing for query processing. This is because they employ services for data acquisition (e.g., SoCQ and Panta Rhei), for modularizing a predefined architecture and achieving platform interoperability (e.g.,

OGSA-DQP); or are restricted to data flow (AXML), while control flow can be required for certain operations. Also, DBMS services are used for aspects such as persistence and consistency (e.g., S3 DBMS), but require building the rest of the functionality from scratch over those services.

2.5 Conclusions

In this chapter we presented an overview of query processing technologies and identified novel aspects that bring multiple new challenges. This lead us to characterize queries in dynamic environments, which involve data and computation services associated with dynamic, distributed, heterogeneous, and mobile infrastructures. We therefore introduced the notion of hybrid queries and its associated taxonomy to better understand the challenges involved as well as the proposed solutions, among which none has addressed satisfactorily the problem in its entirety.

Our analysis leads us to the conclusion that service-oriented computing can provide the foundation for a solution, by offering the capability to integrate the many existing techniques in a flexible and efficient manner. Consequently, rather than depending on an off-the-shelf DBMS, we propose to evaluate hybrid queries via service coordination. Our analysis of existing work shows that for the most part this alternative has not been fully explored. The development of this approach along with its related aspects is the subject of the rest of this work.

Finally, we note that although we did not cover in this chapter the specific techniques and issues related to service coordination, we present an overview and discussion in Appendix C.1. There we discuss our earlier experience, as part of the WebContent Project [22], in the design and implementation of a service coordination engine, which is presented in [CVVSC08] and entirely independent of the work in this thesis. There we also cover the requirements that motivated our service coordination model presented in Section 4.3 and detailed in such appendix.

[22] http://www.webcontent.fr/

CHAPTER 3

A data model for hybrid queries

This chapter presents our data model and query language for hybrid queries. First we present our characterization of data services and their associated data types in Section 3.1. The query operators we propose are described in Section 3.2. Section 3.3 formalizes hybrid queries as expressions built using these operators. In addition, it presents HSQL, the high level language we propose to express hybrid queries; showing by examples how queries in HSQL can be expressed as operation expressions and vice versa. Finally, we present our conclusions in Section 3.4.

3.1 Data and data services types

This section begins with the definition of complex values, which are used to characterize the data produced by data services and manipulated by query operators.

3.1.1 Complex values

We require a representation for data that enables the unification of heterogeneous data sources and that is compatible with Web standards. For these reasons we adopt a characterization of data based on typed complex values. Our typed complex values are compatible with a restricted form of the widespreadly used JavaScript Object Notation, commonly known as JSON[1].

Preamble

We assume a finite set of *domains*, each consisting of a possibly infinite set of values. In particular, we consider the domain $\mathbb{S}$ of strings, $\mathbb{B}$ of booleans ($\{true, false\}$), $\mathbb{Z}$ of integers, and $\mathbb{R}$ of real numbers. We also consider a domain $\mathbb{T}$ defined by the set $\mathbb{N} \cup \{0\}$, i.e. the natural numbers plus zero, which characterizes time values. These can be represented alternatively as $string$, $boolean$, $integer$, $real$, and $time$. In addition, we assume a set $\mathbb{A} = \{A_1, A_2, ...\} \subseteq \mathbb{S}$ of type names.

Complex value types are represented by lower-case letters with hats (e.g. $\hat{t}$) and are defined by a pair $A : def$, where A is the name of the type and def its definition. In order to enable access to

[1] http://www.json.org/

both components we assume the functions $name$ and def, which given a type will return the respective component of the type. For instance, for the type $age : integer$, $name(age : integer) = age$ whereas $def(age : integer) = integer$.

Definition 3.1. [**Complex value types**]
The set $\mathbf{T}$ of all complex value types is defined inductively as follows.

1. if D is a domain, then $A : D$ is an atomic type named A, where $A \in \mathbb{A}$;
2. if $\hat{t}$ is a type, then $A : \{\hat{t}\}$ is a set type named A;
3. if $\hat{t}_1, ..., \hat{t}_n$ are types with distinct names, then $A : \langle \hat{t}_1, ..., \hat{t}_n \rangle$ is a tuple type named A and each $\hat{t}_i$ is an attribute type;
4. if $\hat{t}_1, ..., \hat{t}_n$ and $\hat{t}_o$ are types with distinct names, then $A : \hat{t}_1 \times ... \times \hat{t}_n \rightarrow \hat{t}_o$ is a function type named A, and each $\hat{t}_i$ such that $1 \leq i \leq n$ is an input parameter type while $\hat{t}_o$ is the output parameter type. We will usually write function types as $\hat{f}$.

Every type $\hat{t} \in \mathbf{T}$ denotes a set $[\![\hat{t}]\!]$ of complex value *instances* which is defined inductively as follows.

Definition 3.2. [**Denotation of types**]

- For each atomic type $A : D$, $[\![A : D]\!] = \{A : d \mid d \in D\}$, where we assume the values of the domain D given;
- for set types of the form $A : \{\hat{t}\}$, $[\![A : \{\hat{t}\}]\!] = \{A : S \mid S \in \mathcal{P}([\![\hat{t}]\!])\}$, where $\mathcal{P}$ denotes the power set;
- for tuple types of the form $A : \langle \hat{t}_1, ..., \hat{t}_n \rangle$ where each $\hat{t}_i$ is of the form $A_i : def_i$, $[\![A : \langle \hat{t}_1, ..., \hat{t}_n \rangle]\!] =$

  $\{A : \langle A_1 : v_1, ..., A_n : v_n \rangle \mid A_i : v_i \in [\![\hat{t}_i]\!] \wedge i \in [1..n]\}$.
- for function types of the form $A : \hat{t}_1 \times ... \times \hat{t}_n \rightarrow \hat{t}_o$, $[\![A : \hat{t}_1 \times ... \times \hat{t}_n \rightarrow \hat{t}_o]\!] =$
  $\{A : f \mid f \; is \; a \; function \; with \; signature \; [\![\hat{t}_1]\!] \times ... \times [\![\hat{t}_n]\!] \rightarrow [\![\hat{t}_o]\!]\}$.

By the function signature in the last rule, we mean the function has as domain the cartesian product of the denotations of the input parameters, and as codomain the denotation of the output parameter.

Access to the values of complex value instances

Analogously to complex value types, we define the functions $name$ and val to obtain the components of complex value instances of the form $A : val$. For example, if v denotes $primes : \{num : 2, num : 3, num : 5...\}$ we have $name(v) = primes$ and $val(v) = \{num : 2, num : 3, num : 5...\}$.

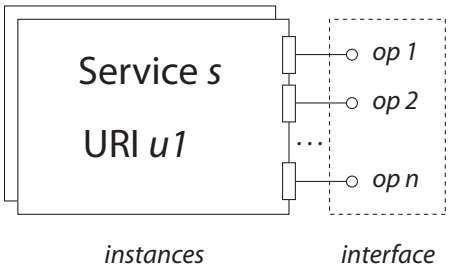


Figure 3.1: Representation of a service

In particular, for tuple values t of the form $A : \langle A_1 : v_1, ..., A_n : v_n \rangle$, we adopt the dot notation $t.A_i$ to access the values v_i of the attributes $A_i : v_i$ of t. For example, if t denotes $item : \langle id : 1, quantity : 5 \rangle$, then $t.id = 1$ and $t.quantity = 5$. We also permit to assign a value v_i to an attribute of t, by writing $t.A_i := v_i$.

In addition, we assume the function $type(A : v)$ that receives a complex value instance $A : v$ and returns the type to which the instance belongs.

Finally, we extend our syntax to express with $\{\hat{t}\}$ not only the definition a set type, but also a stand-alone and unnamed set of complex value instances of type $\hat{t}$, such that $\{\hat{t}\} \subseteq [\![\hat{t}]\!]$. The difference will be clear from context.

3.1.2 Services

We understand by a service an autonomous software entity capable of performing several tasks involving data exchange, which occurs through the invocation of its constituent operations. A service is also accessible through a standardized interface describing such operations and is subject to a single administrative domain.

Furthermore, a given service interface is associated with a set of service instances. The access to service instances can be facilitated, for example, by a simple service catalog or a more sophisticated service discovery mechanism (e.g. UDDI[2]). Figure 3.1 depicts generically a service and its components. Our formal characterization of a service is presented next.

Definition 3.3. [**Service interface**] A service *interface* is represented by a tuple type $serv : \langle \hat{f}_1, ..., \hat{f}_n \rangle$, where each $\hat{f}_i$ is a function type of the form $op_i : f_i^{def}$. The interface thus comprises the name $serv$ of the service and a series of *operations*, where op_i is the name of the ith-operation and f_i^{def} is its function type definition.

Definition 3.4. [**Service instance**] A service *instance* implements a service interface $serv : \langle \hat{f}_1, ..., \hat{f}_n \rangle$. It is represented by a tuple $serv : \langle op_1 : f_1, ..., op_n : f_n \rangle$ where each $op_i : f_i$ is a function complex value of type $\hat{f}_i$. Multiple implementations (instances) are possible for a given interface.

[2]http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm

Service access

Considering the set $S \subseteq \mathbf{T}$ of types specifying service interfaces, we assume a function $registry : S \rightarrow \bigcup_{s_i \in S} \mathcal{P}([\![s_i]\!])$ such that $registry(s) \subseteq [\![s]\!]$ for all $s \in S$. That is, given the type defining a service interface, the registry will provide a set of instances implementing such interface.

In addition, we assume a function $choice : \bigcup_{s_i \in S} \mathcal{P}([\![s_i]\!]) \rightarrow \bigcup_{s_i \in S} [\![s_i]\!]$ such that $choice(I) \in I$ for I a set of service instances implementing the same interface. Thus $choice$ enables to select a specific service instance from the set of instances obtained from the registry.

In this manner we are able to obtain a service instance implementing a service interface via the $registry$ and $choice$ functions. Furthermore, optimality criteria may be integrated into the $choice$ function, enabling us to obtain the most adequate service instance at a given time.

Our previous characterization of services is generic. We now turn our attention into services that specifically produce data to be queried. As discussed in the previous chapter, we consider two kinds of them: on-demand and stream data services.

3.1.3 On-demand data services

On-demand data services provide data in a request-response fashion through synchronous data operations. We characterize their interfaces as follows.

Definition 3.5. [**On-demand data service**] A service is an *on-demand data service* if for its interface $serv : \langle \hat{f}_1, ..., \hat{f}_n \rangle$, each function type $\hat{f}_i$ represents an on-demand data operation, whose definition is presented next.

Definition 3.6. [**On-demand data operation**]

A function type $\hat{f}_i$ represents a data operation if it satisfies the following two conditions.

1. It is of the form $op_i : \hat{a}_1 \times ... \times \hat{a}_m \rightarrow \{\hat{t}_o\}$, where $\hat{t}_o$ is of the form $A : \langle \hat{a}_1, ..., \hat{a}_m, \hat{b}_1, ..., \hat{b}_p \rangle$. Therefore, the output tuples contain attributes of the same type as the input parameters $\hat{a}_j$ of the operation, in one to one correspondence, as well as a series of additional attributes $\hat{b}_k$.

2. Considering a function complex value $op_i : f_i$ corresponding to the operation type $\hat{f}_i$, we have that if $f_i(A_1 : a_1, ..., A_m : a_m) = T$, with each $A_j : a_j \in [\![\hat{a}_j]\!]$ for $1 \leq j \leq m$, it holds that $t.A_j = a_j$ for all tuples $t \in T$. Thus, the values of the attributes with name A_j corresponding to the operation input parameters are the same for all output tuples in the output set, and further, they correspond to the values supplied for the input parameters.

3.1.4 Streaming data services

A streaming data service provides data in a continuous stream once a subscription has been performed by the client. In our model, we abstract from the subscription mechanism and define the interface of a streaming data service as a tuple of stream data operations, each represented by a stream function, as detailed below.

Definition 3.7. [**Streaming data service**] A *streaming data service* has an interface of the form $serv : \langle \hat{f}_1, ..., \hat{f}_n \rangle$, where each function type $\hat{f}_i$ represents a stream function, whose definition is presented next.

Definition 3.8. [**Stream function**] A stream function has a type $\hat{f}_i$ of the form $op_i : \ t : time \rightarrow \{A : \langle \hat{a}_1, ..., \hat{a}_n, \tau : time \rangle\}$ where $\hat{a}_i$ are data attributes and $\tau : time$ a timestamp attribute. The function thus receives a time value and produces a set of *timestamped tuples*.

For the appropriate evaluation of hybrid queries involving stream data operations, it is necessary that their corresponding stream functions adhere to some properties involving order. We define two such order properties next.

Definition 3.9. [**Time-monotonicity of stream functions**] A stream function f, respectively its stream data operation, is *time-monotonic* if it satisfies the following property. If t_1 and t_2 are time values such that $t_1 < t_2$, with $f(t_1) = T_1$ and $f(t_2) = T_2$, then for all tuples $tup_1 \in T_1$ and $tup_2 \in T_2$ we have that $tup_1.\tau \leq tup_2.\tau$.

Consequently, the timestamp values of the tuples produced by a time-monotonic stream function need not coincide with the value for the supplied time attribute, but they will never decrease. This behavior is to be expected in applications where a certain latency is unavoidable, and it can be supported by buffering techniques.

The previous definition implies a notion of order based on time as indicated by the timestamp values in tuples. Nevertheless, order is unspecified for tuples with the same timestamp. Since for some applications it can be important to distinguish between tuples with the same timestamp value, e.g. by order of arrival, we introduce a total order relation in our model, which in fact subsumes time-monotonicity.

Definition 3.10. [**Total order relation $\preceq$ for timestamped tuples**] Considering a set of timestamped tuples T generated by a stream function $f : \ t : time \rightarrow \{\hat{t}\}$, we assume the existence of a function $rank : [\![\hat{t}]\!] \rightarrow \mathbb{N}$, which enables us to define a total order relation over the elements of T.

The $rank$ function adheres to the following two characteristics. *(i)* It is strictly monotonic on timestamp values, i.e., if tup_1 and tup_2 are tuples in $[\![\hat{t}]\!]$ such that $tup_1.\tau < tup_2.\tau$, then $rank(tup_1) < rank(tup_2)$. *(ii)* It is injective, meaning if $tup_1 \neq tup_2$ then $rank(tup_1) \neq rank(tup_2)$. Thus, we can define a total order relation $\preceq$ over T as follows. For all $t_1, t_2 \in T$ such that $t_1 \neq t_2$, $t_1 \preceq t_2$ if and only if $rank(t_1) < rank(t_2)$.

In this work we will focus, without loss of generality, on stream functions that generate sets of totally ordered tuples according to the previous definition. This will enable us to eliminate non-determinism in some circumstances and reflects more accurately real scenarios. Nevertheless, our model allows, via hybrid queries and their operators, the definition of new non-monotonic (resp. non-totally ordered) stream functions based on existent monotonic (resp. totally-ordered) stream functions. This obeys to possible savings in computation costs and that such properties may be not required for some applications. For instance, it is less expensive to evaluate a join between two streams if an order in the results does not have to be enforced.

So far we have defined stream functions and the order among the tuples they generate. However, we are usually interested in effects of the events that lie behind the produced tuples. For this reason we next introduce the concepts of tagged tuples and stream function extents, which together enable to capture the changes reflected by a data stream.

Definition 3.11. [**Tagged tuples**] A tagged tuple is a timestamped tuple of type $A : \langle \hat{a}_1, ..., \hat{a}_n, \tau : time, sign : integer \rangle$, where the values of the *sign* attribute are restricted to the set $\{1, -1\} \subseteq [\![integer]\!]$. We say that a tuple with a sign value of 1 is a *positive* tuple, whereas it is a *negative* tuple if the sign value is -1.

Tagged tuples denote changes on the extent of a stream function, which we define as follows.

Definition 3.12. [**Extent of a stream function**] The *extent* of a stream function f that produces tagged tuples is denoted as $\mathfrak{E}_f$ and created recursively as follows. First, we define the function H that produces the *history* of a stream function s, i.e. all the tuples produced up to a given time, as follows

$$H(s,t) = \bigcup_{i=0}^{t} s(i)$$

Then $\mathfrak{E}_f(t) = calc_ext(t', H(f,t))$ where $t' = max\{tup.\tau | tup \in H(f,t)\}$, i.e. the largest timestamp value in the tuples present in the history of f up to t. In turn, $calc_ext$ is a function that receives a time parameter and a set of tagged tuples, and that is defined as follows.

$$calc_ext(0,T) = \emptyset$$
$$calc_ext(t,T) = (calc_ext(t-1,T) \cup \Delta^+(t,T)) - \Delta^-(t,T)$$

Where Δ^+ and Δ^- are defined as

$$\Delta^+(t,T) = \{tup \in T \mid tup.\tau = t \wedge tup.sign = 1\}$$
$$\Delta^-(t,T) = \{tup \in T \mid tup.\tau = t \wedge tup.sign = -1\}$$

According to the previous definition, the tagged tuples produced at different time instants (in terms of the time parameter of the stream function) are coalesced to form a single set of tagged tuples. Then a new set of data tuples is formed by computing the changes at each possible timestamp value.

Fundamentally, the extent of a stream function represents a view computed from the tagged tuples and linked to time. For some applications it can be useful to compute this view, while for others it may be more convenient to obtain directly the tagged data stream. An example of an application that computes the extent of a stream function is presented in Section 5.6.1.

3.2 Operators

We now present the operators that manipulate the data produced by data services in order to build the result of a hybrid query. For the specification of each operator we introduce first an example of the notation used to apply the operator. Then we present the operation type, which defines the type of complex values the operator can receive and will produce (viewed as a function). Finally, we present the operator semantics, i.e., the output complex values it produces for a given input.

3.2.1 Window operators

The evaluation of a large variety of queries over data streams necessitates a means to bind the data streams, creating subsequences commonly referred to as a windows. Then we can apply to the data in the windows many of the known operations in query processing. We define two types of window operators: time-based sliding windows and tuple-based sliding windows.

Time-based sliding window

Intuitively, a time-based sliding window consists of the latest portion a data stream, particularly those tuples received within the latest time interval of a fixed length T.

- *Notation*: $\mathcal{W}_{T=d}(s)$ where d is an integer greater than or equal to zero denoting the length of the window in time units and s is a stream data operation.
- *Operation type*: $\mathcal{W}_{T=d}(s) : \{A : \langle \hat{a}_1, ..., \hat{a}_n, \tau : time, sign : integer\rangle\} \rightarrow \{A : \langle \hat{a}_1, ..., \hat{a}_n, \tau : time, sign : integer\rangle\}$, thus the operator receives timestamped and tagged tuples and has no effect in their structure
- *Semantics*:

  Recall the function H from Definition 3.12 that produces the history of a stream data operation. The value of the window at time t is then given by

  $$\begin{aligned}\mathcal{W}_{T=d}(s(t)) = &\{tup^+ \mid tup \in H(s,t) \wedge tup.\tau \geq t - T\} \cup \\ &\{tup^- \mid tup \in H(s,t) \wedge tup.\tau = t - T - 1\}\end{aligned}$$

  where tup^+ denotes that the tuple is given a positive sign value (1) whereas tup^- denotes that it is given a negative value (-1). Thus, negative tuples are generated at the moment that the window is said to slide. These negative tuples indicate that the tuple in question is evicted from the window at that time instant.

Example 3.2. Consider the following evaluation of a time-based window of length 2 over a stream function s.
$s(0) = H(s,0) = \{R : \langle data : a, \tau : 0, sign : 1\rangle, R : \langle data : b, \tau : 0, sign : 1\rangle\}$
$\mathcal{W}_{T=2}(s(0)) = \{R : \langle data : a, \tau : 0, sign : 1\rangle, R : \langle data : b, \tau : 0, sign : 1\rangle\} \cup \emptyset$

$s(1) = \{R : \langle data : c, \tau : 1, sign : 1\rangle\}$
$H(s,1) = \{R : \langle data : a, \tau : 0, sign : 1\rangle, R : \langle data : b, \tau : 0, sign : 1\rangle, R : \langle data : c, \tau : 1, sign : 1\rangle\}$
$\mathcal{W}_{T=2}(s(1)) = \{R : \langle data : a, \tau : 0, sign : 1\rangle, R : \langle data : b, \tau : 0, sign : 1\rangle, R : \langle data : c, \tau : 1, sign : 1\rangle\} \cup \emptyset$

$s(2) = \{R : \langle data : d, \tau : 2, sign : 1\rangle\}$
$H(s,2) = \{R : \langle data : a, \tau : 0, sign : 1\rangle, R : \langle data : b, \tau : 0, sign : 1\rangle, R : \langle data : c, \tau : 1, sign : 1\rangle, R : \langle data : d, \tau : 2, sign : 1\rangle\}$

$\mathcal{W}_{T=2}(s(2)) = \{R : \langle data : a, \tau : 0, sign : 1\rangle, R : \langle data : b, \tau : 0, sign : 1\rangle, R : \langle data : c, \tau : 1, sign : 1\rangle, R : \langle data : d, \tau : 2, sign : 1\rangle\} \cup \emptyset$

$s(3) = \{R : \langle data : e, \tau : 3, sign : 1\rangle\}$
$H(s, 3) = \{R : \langle data : a, \tau : 0, sign : 1\rangle, R : \langle data : b, \tau : 0, sign : 1\rangle, R : \langle data : c, \tau : 1, sign : 1\rangle, R : \langle data : d, \tau : 2, sign : 1\rangle, R : \langle data : e, \tau : 3, sign : 1\rangle\}$
$\mathcal{W}_{T=2}(s(3)) = \{R : \langle data : c, \tau : 1, sign : 1\rangle, R : \langle data : d, \tau : 2, sign : 1\rangle, R : \langle data : e, \tau : 3, sign : 1\rangle\} \cup \{R : \langle data : a, \tau : 0, sign : -1\rangle, R : \langle data : b, \tau : 0, sign : -1\rangle\}$

Note that at the time instant 3 the tuples that arrived at time instant 0 are expired by giving them a negative sign.

Tuple-based sliding window

Alternatively to bounding a stream by the tuples received within the last fixed amount of time, we can define a sliding window based on the last N tuples received altogether up to the present time. Recall that we assume that the total order relation $\preceq$ holds in each set of tuples generated by the stream data operation and in the overall history.

- *Notation*: $\mathcal{W}_{N=n}(s)$ where n is an integer greater than zero denoting the size of the window in tuples and s is a stream data operation.
- *Operation type*: $\mathcal{W}_{N=n}(s) : \{A : \langle \hat{a}_1, ..., \hat{a}_n, \tau : time, sign : integer\rangle\} \rightarrow \{A : \langle \hat{a}_1, ..., \hat{a}_n, \tau : time, sign : integer\rangle\}$, thus the operator receives timestamped and tagged tuples and has no effect in their structure
- *Semantics*:

  We rely again on the function H from Definition 3.12 that produces the history of a stream data operation. The value of the window at time t is then given by

  $$\mathcal{W}_{N=n}(s(t)) = \{tup^+ \in S \mid S \subseteq H(s,t) \wedge |S| \leq N \wedge (\nexists tup' \in (H(s,t) - S) | tup \preceq tup')\} \cup \{tup^- \mid tup \in \mathcal{W}_{N=n}(s(t-1)) - S \wedge tup.sign = 1\}$$

  where again tup^+ denotes that the tuple is given a positive sign value (1) whereas tup^- denotes that it is given a negative value (-1). Thus, a subset of the tuples in the history is taken to produce the window, with the characteristics that it is of cardinality at most N and that it consists of the most recently arriving tuples. While negative tuples are generated for the tuples that are evicted at a given time when they are replaced by more recent ones.

3.2.2 Recursive complex value operators

Next we present a series of operators inspired in the traditional relational operators, which however apply to complex values and are further recursive; meaning that through an appropriate expression it is possible to apply the operator to structures nested within a given complex value.

Selection

This operator filters the complex value instances in a set S based on a selection expression σ_{exp}, producing as output the subset of instances that satisfy the expression.

- *Notation*: $\sigma_{exp}(S)$
- *Operation type*: $\sigma : \{\hat{t}\} \rightarrow \{\hat{t}\}$
- *Semantics*: $\sigma_{exp}(S) = \{s \in S \mid s \models \sigma_{exp}\}$

Where the expression σ_{exp} is built as follows. First we define *terms* by the following rules

1. Constant values c for complex value instances $A : c$ are (constant) terms.
2. If A is a type name then A is a (name) term.
3. If f is a function of arity n and $T_1, ..., T_n$ are terms, then $f(T_1, ..., T_n)$ is a (function) term.

By extension constant terms also denote their values, while the functions we consider produce as output only complex values. Next we define *basic expressions* ($bexp$) over terms of the aforementioned type as follows, where T_1, and T_2 represent terms, and A particularly a name term

$$\begin{array}{lll} bexp & ::= & T_1\ \theta\ T_2 \\ \theta & ::= & = \mid \neq \mid < \mid > \mid \leq \mid \geq \mid \subset \mid \subseteq \mid \in \end{array}$$

The relations and comparison operators θ used must be compatible with the values of the terms.

In addition, to facilitate the determination of the truth value of conditions, we define the interpretation of a term T under an arbitrary complex value r, denoted as T^r, as follows

1. For constant terms C, $C^r = C$.
2. For name terms A,

$$A^r = \begin{cases} v, & \text{if } r \text{ is a set or atomic complex value of the form } A : v; \\ r, & \text{if } r \text{ is a tuple complex value of the form } A : v; \\ v', & \text{if } r \text{ is a tuple complex value of the form } B : \langle ..., A : v', ... \rangle; \\ \{v_j | A : v_j \in \text{val(r)}\}, & \text{if } r \text{ is a set value of the form } B : \{A : v_1, ..., A : v_n\}; \end{cases}$$

3. For function terms $f(T_1, ..., T_n)$, $f(T_1, ..., T_n)^r = val(f(T_1^r, ..., T_n^r))$.

A complex value r satisfies of a basic expression $bexp$ of the form $T_1\ \theta\ T_2$ iff $T_1^r\ \theta\ T_2^r$ holds, in which case we write $r \models T_1\ \theta\ T_2$

Next, we define the selection expressions σ_{exp} of our notation, as well as *full expressions* ($fexp$) contained within selection expressions by the following grammar, where A represents a type name, recall that basic expressions $bexp$ were defined earlier

$$
\begin{array}{lcl}
\sigma_{exp} & ::= & \sigma(A, fexp)\ \mid\ \sigma(A, quant\ fexp) \\
fexp & ::= & factor((\wedge|\vee)factor)* \\
factor & ::= & \neg factor\ \mid\ phrase \\
phrase & ::= & '('fexp')'\ \mid\ bexp\ \mid\ \sigma_{exp} \\
quant & ::= & \forall\ \mid\ \exists
\end{array}
$$

The satisfiability of selection expressions consisting of the above subexpressions and logical connectives, under a complex value r, is given by

1. $r \models \neg fexp$ iff $r \models fexp$ does not hold
2. $r \models fexp_1 \wedge fexp_2$ iff $r \models fexp_1$ and $r \models fexp_2$
3. $r \models fexp_1 \vee fexp_2$ iff $r \models fexp_1$ or $r \models fexp_2$

   In turn, the satisfiability of recursive expressions under a complex value r is given by

4. $r \models \sigma(A, fexp)$ iff $A^r \models fexp$
5. $r \models \sigma(A, \forall fexp)$ iff for every $a_i \in A^r$, $a_i \models fexp$, and A^r is a set
6. $r \models \sigma(A, \exists fexp)$ iff for some $a_i \in A^r$, $a_i \models fexp$, and A^r is a set

The satisfiability of the selection expression exp under an input complex value $s \in S$ follows from the satisfiability rules above.

Example 3.3.
Consider the following set of complex values

S = { person:⟨ sex:'M', nickname:'Charles', email:'charles@gmail.com', age:40,
interests:{stag:⟨ tag:'art', score:6.5 ⟩, stag:⟨ tag:'sports', score:7.5 ⟩}⟩,
person:⟨ sex:'F', nickname:'Alice', email:'alice@hotmail.com', age:27,
interests:{stag:⟨ tag:'novels', score:8.2 ⟩, stag:⟨ tag:'fashion', score:6.0 ⟩}⟩ }

The expression $\sigma(person, \sigma(interests, \exists\sigma(stag, tag =$'*art*'$)))(S)$ produces

{ person:⟨ sex:'M', nickname:'Charles', email:'charles@gmail.com', age:40,

interests:{*stag:*⟨ *tag:'art', score:6.5* ⟩, *stag:*⟨ *tag:'sports', score:7.5* ⟩}⟩ }

Since we are interested in those *persons* whose *interests* include *'art'*, and only the first person in the set satisfies the condition.

We trace the evaluation for the first complex value $s_1 \in S$ as follows

$s_1 =$ *person:*⟨ *sex:'M', nickname:'Charles', email:'charles@gmail.com', age:40,*
interests:{*stag:*⟨ *tag:'art', score:6.5* ⟩, *stag:*⟨ *tag:'sports', score:7.5* ⟩}⟩

Given $\sigma(person, \sigma(interests, \exists\sigma(stag, tag =$'*art*'$)))(s_1)$, we apply rule 4 obtaining

$s_1 \models \sigma(person, \sigma(interests, \exists\sigma(stag, tag =$'*art*'$))) \text{ iff } person^{s_1} \models \sigma(interests, \exists\sigma(stag, tag =$'*art*'$))$

we have that $person^{s_1} = s_1$ (by the second case). Then by rule 6

$s_1 \models \sigma(interests, \exists\sigma(stag, tag =$'*art*'$))$ iff for some $a_i \in interests^{s_1}$, $a_i \models \sigma(stag, tag =$'*art*'$)$, and $person^{s_1}$ is a set.

We have that $interests^{s_1} =$ {stag:⟨ tag:'art', score:6.5 ⟩, stag:⟨ tag:'sports', score:7.5 ⟩}, a set. Then, for the first value in the set, $a_1 =$ stag:⟨ tag:'art', score:6.5 ⟩, we need to determine

$a_1 \models \sigma(stag, tag =$'*art*'$)$ then by rule 4

$a_1 \models \sigma(stag, tag =$'*art*'$)$ iff $stag^{a_1} \models tag =$'*art*'

We have that $stag^{a_1} = a_1$ (by the second case). Consequently, we need to determine

$a_1 \models tag =$'*art*', which is a basic expression that holds given that $tag^{a_1} =$'*art*' (by the third case). Therefore the value s_1 satisfies the condition.

Projection

This recursive operator enables to retrieve certain data elements in a complex value instance. Such data elements may be nested and multivalued. The data elements to retrieve are specified in a (possibly recursive) projection expression π_{exp}, which is applied to the input complex value instance s.

- *Notation*: $\pi_{exp}(s)$

  Projection expressions π_{exp} are constructed as follows, we use A to represent type names that occur in the complex value instance

$$\begin{array}{lcl} \pi_{exp} & ::= & \pi\ (\ list\ ) \\ list & ::= & term \mid term\ ,\ list \\ term & ::= & A \mid \pi_{exp} \end{array}$$

- *Operation type*: $\pi : \hat{t} \rightarrow \hat{t'}$, where $\hat{t'}$ is defined below
- *Semantics*: $\pi_{exp}(s)$ is defined as follows.

  First, we define the function $eval(A : v, L)$, where $A : v$ is a tuple complex value of the form $A : \langle ..., A' : v', ... \rangle$ and L an expression term (as defined by the notation third rule above).

  1. If L is of the form A' then $eval(A : v, L) = A' : v'$
  2. If L is of the form $\pi(A', L'_1, ..., L'_n)$ then $eval(A : v, L) = \pi(A', L'_1, ..., L'_n)(A' : v')$

  The value of $\pi_{exp}(s)$ is then given by

  1. If $s = A : \langle A_1 : v_1, ..., A_n : v_n \rangle = A : v$, i.e. s is a tuple complex value, and $\pi_{exp} = \pi(A, L_1, ..., L_n)$, then
     $\pi_{exp}(s) = A : \langle eval(A : v, L_1), ..., eval(A : v, L_n) \rangle$ and
     $\hat{t'}$ is $A : \langle type(eval(A : v, L_1)), ..., type(eval(A : v, L_n)) \rangle$
  2. If $s = A : \{A' : v_1, ..., A' : v_m\}$, i.e. s is a set complex value, and $\pi_{exp} = \pi(A, \pi_{exp'})$ with $\pi_{exp'}$ of the form $\pi(A', L'_1, ..., L'_n)$, then
     $\pi_{exp}(s) = A : \{\pi_{exp'}(A' : v_i) | A' : v_i \in val(s)\}$ and
     $\hat{t'}$ is $A : \{type(\pi_{exp'}(A' : v_j))\}$ for an arbitrary $A' : v_j \in val(s)$

Example 3.4.
Consider the following complex value

s = person:⟨ sex:'M', nickname:'Charles', email:'charles@gmail.com', age:40,
interests:{stag:⟨ tag:'art', score:6.5 ⟩, stag:⟨ tag:'sports', score:7.5 ⟩}⟩

The expression $\pi(person, nickname, age, \pi(interests, \pi(stag, score)))(s)$ produces the value

person:⟨ nickname:'Charles', age:40,
interests:{stag:⟨ score:6.5 ⟩, stag:⟨score:7.5 ⟩}⟩

Renaming

This recursive operator enables to rename certain data elements in a complex value instance. The data elements to rename and their new assigned names are specified in a (possibly recursive) renaming expression ρ_{exp}, which is applied to the input complex value instance *s*.

- *Notation*: $\rho_{exp}(s)$

  Renaming expressions ρ_{exp} are constructed as follows, we use A to represent the type names that occur in the complex value instance, and A' to represent those that will replace such, if so specified

$$\begin{array}{lcl} \rho_{exp} & ::= & \rho\ (\ list\ ) \\ list & ::= & term\ |\ term\ ,\ list \\ term & ::= & A\ |\ A\ :\ A'\ |\ \rho_{exp} \end{array}$$

- *Operation type*: $\rho : \hat{t} \rightarrow \hat{t'}$, where $\hat{t'}$ is defined below
- *Semantics*: $\rho_{exp}(s)$ is defined as follows.

  First we define the function $rname$, which receives a non-recursive renaming expression term L_i and returns the renaming attribute name present in it.

$$rname(L_i) = \begin{cases} A, & \text{if } L_i \text{ is of the form } A; \\ A', & \text{if } L_i \text{ is of the form } A : A'; \end{cases}$$

  We define the function $eval(A : v, L)$, where $A : v$ is a tuple complex value and L is a list of renaming expression terms $L_1, ..., L_n$ such that one of the following rules holds.

  1. There is a unique L_k in L of the form $A : A'$, then $eval(A : v, L) = A' : v$
  2. There is a unique L_k in L of the form $\rho(L'_0, L'_1, ..., L'_n)$ with L'_0 of the form A or $A : A'$ then $eval(A : v, L) = \rho(rname(L'_0), L'_1, ..., L'_n)(rname(L'_0) : v)$
  3. The conditions above do not hold, then $eval(A : v, L) = A : v$

  The value of $\rho_{exp}(s)$ is then given by

  1. If s is a complex value of the form $A : v$, and $\rho_{exp} = \rho(A : B)$, then $\rho_{exp}(s) = B : v$

  2. If $s = A : \langle A_1 : v_1, ..., A_n : v_n \rangle = A : v$, i.e. s is a tuple complex value, and $\rho_{exp} = \rho(L_0, L_1, ..., L_n) = \rho(L_0, L)$, with L_0 of the form A or $A : B$ then $\rho_{exp}(s) = rname(L_0) : \langle eval(A_1 : v_1, L), ..., eval(A_n : v_n, L) \rangle$ and $\hat{t'}$ is $type(rname(L_0) : \langle eval(A_1 : v_1, L), ..., eval(A_n : v_n, L) \rangle)$
  3. If $s = A : \{A' : v_1, ..., A' : v_m\}$, i.e. s is a set complex value, and $\rho_{exp} = \rho(L_0, \rho_{exp'})$ with L_0 of the form A or $A : B$ and then $\rho_{exp}(s) = rname(L_0) : \{\rho_{exp'}(A' : v_i) | A' : v_i \in val(s)\}$ and $\hat{t'}$ is $rname(L_0) : \{type(\rho_{exp'}(A' : v_j))\}$ for an arbitrary $A' : v_j \in val(s)$

Example 3.5.
Consider the following complex value

s = person:⟨ sex:'M', nickname:'Charles', email:'charles@gmail.com', age:40,

interests:{stag:⟨ tag:'art', score:6.5 ⟩, stag:⟨ tag:'sports', score:7.5 ⟩}⟩

The expression

$$\rho(person : user, nickname : username, email : mailto,$$
$$\rho(interests : inter, \rho(stag : scoredtag, score : val, tag : keyword)), sex : gender)(s)$$

produces

user:⟨ gender:'M', username:'Charles', mailto:'charles@gmail.com', age:40,
inter:{scoredstag:⟨ keyword:'art', val:6.5 ⟩, scoredstag:⟨ keyword:'sports', val:7.5 ⟩}⟩

3.2.3 Combination operators

Next we present the operators that enable to combine the data present, in particular, in tuple complex values. This principle applies to all of the combination operators.

Cartesian product

The cartesian product enables to combine the tuple values of two input sets R and S.

- *Notation*: $R \times S$
- *Operation type*: $\times : \{\hat{t}_r\} \times \{\hat{t}_s\} \rightarrow \{\hat{t'}\}$, where if $\hat{t}_r = A : \langle \hat{r}_1, ..., \hat{r}_m \rangle$ and $\hat{t}_s = B : \langle \hat{s}_1, ..., \hat{s}_n \rangle$ then $\hat{t'} = A_B : \langle \hat{r}_1, ..., \hat{r}_m, \hat{s}_1, ..., \hat{s}_n \rangle$

  It is also required that $name(\hat{r}_i) \neq name(\hat{s}_j)$ for all i and j occurring in $\hat{t}_r$ and $\hat{t}_s$
- *Semantics*:

  Given a set R of tuples of the form $A : \langle A_1 : v_1, ..., A_m : v_m \rangle$ as well as a set S of tuples the form $B : \langle B_1 : w_1, ..., B_n : w_n \rangle$, we have that

  $$R \times S = \{A_B : \langle A_1 : r.A_1, ..., A_m : r.A_m, B_1 : s.B_1, ..., B_n : s.B_n \rangle | r \in R \wedge s \in S\}$$

Note that by convention the tuples in the output set are given a name corresponding to the concatenation of the name of the tuples in the two input sets separated by an underscore.

Example 3.6.

Consider the following complex value sets

R = { *person:⟨ person_id:'0', name:'Luitpold Martucci', email:'Martucci@toronto.edu' ⟩* }

S = { *bid:⟨ person_ref:'30', open_auction_id:7, bid:145.00 ⟩*
bid:⟨ person_ref:'17', open_auction_id:9, bid:215.00 ⟩ }

The expression $R \times S$ generates

{ *person_bid:⟨ person_id:'0', name:'Luitpold Martucci', email:'Martucci@toronto.edu',*
person_ref:'30', open_auction_id:7, bid:145.00 ⟩,
person_bid:⟨ person_id:'0', name:'Luitpold Martucci', email:'Martucci@toronto.edu',
person_ref:'17', open_auction_id:9, bid:215.00 ⟩ }

Theta-join

The theta-join enables to combine the tuple values of two input sets R and S based on a logical condition θ. It is equivalent to a cartesian product followed by a selection on the θ condition, which is a basic condition as defined for the selection operator.

- *Notation*: $R \bowtie_\theta S$

- *Operation type*: $\bowtie$: $\{\hat{t}_r\} \times \{\hat{t}_s\} \rightarrow \{\hat{t'}\}$, where if $\hat{t}_r = A : \langle \hat{r}_1, ..., \hat{r}_m \rangle$ and $\hat{t}_s = B : \langle \hat{s}_1, ..., \hat{s}_n \rangle$ then $\hat{t'} = A_B : \langle \hat{r}_1, ..., \hat{r}_m, \hat{s}_1, ..., \hat{s}_n \rangle$

  It is required that $name(\hat{r}_i) \neq name(\hat{s}_j)$ for all i and j occuring in $\hat{t}_r$ and $\hat{t}_s$

- *Semantics*:

  Given a set R of tuples of the form $A : \langle A_1 : v_1, ..., A_m : v_m \rangle$ as well as a set S of tuples the form $B : \langle B_1 : w_1, ..., B_n : w_n \rangle$, we have that

  $$R \bowtie_\theta S = \{t \in \{A_B : \langle A_1 : r.A_1, ..., A_m : r.A_m, B_1 : s.B_1, ..., B_n : s.B_n \rangle \mid r \in R \wedge s \in S\} \mid t \models \theta\}$$

Example 3.7.

Consider the following complex value sets

R = { *person:⟨ person_id:'0', name:'Luitpold Martucci', email:'Martucci@toronto.edu' ⟩*
person:⟨ person_id:'1', name:'Annette Klaiber', email:'Klaiber@propel.com' ⟩ }

S = { *bid:⟨ person_ref:'3', open_auction_id:7, bid:145.00 ⟩*
bid:⟨ person_ref:'1', open_auction_id:12, bid:87.00 ⟩ }

The expression $R \bowtie_{person_id=person_ref} S$ produces

{ *person_bid:⟨ person_id:'1', name:'Annette Klaiber', email:'Klaiber@propel.com',*
person_ref:'1', open_auction_id:12, bid:87.00 ⟩ }

Bind-join

The bind-join enables to combine the data from an input set R containing tuple values with the tuple values that are produced by an on-demand data operation s.

- *Notation*: $R \overset{\rightarrowtail}{\bowtie} s$

- *Operation type*: $\overset{\rightarrowtail}{\bowtie}$: $(\{\hat{t}_R\} \times (\hat{b}_1 \times ... \times \hat{b}_n) \rightarrow \{\hat{t}_s\}) \rightarrow \{\hat{t}_o\}$

  where if $\hat{t}_R = A : \langle \hat{a}_1, ..., \hat{a}_m, \hat{b}_1, ..., \hat{b}_n \rangle$ and $\hat{t}_s = B : \langle \hat{b}_1, ..., \hat{b}_n, \hat{c}_1, ..., \hat{c}_p \rangle$ then
  $\hat{t}_o = A : \langle \hat{a}_1, ..., \hat{a}_m, \hat{b}_1, ..., \hat{b}_n, \hat{c}_1, ..., \hat{c}_p \rangle$

  Thus the operator receives a set of tuples of type $\hat{t}_R$ and, a function representing an on-demand data operation that receives a series of parameters of type $\hat{b}_i$ in order to produce an output set of tuples of type $\hat{t}_s$. The operator then invokes the on-demand data operation with the tuple values obtained from the input set R and combines them with the output set of the on-demand data operation in order to generate a set of output tuples of type $\hat{t}_o$. This process is described next.

- *Semantics*:

  $R \overset{\rightarrowtail}{\bowtie} s = \bigcup_{t \in R} \{A_B : \langle A_1 : u_1, ..., A_m : u_m, B_1 : v_1, ..., B_n : v_n, C_1 : w_1, ..., C_p : w_p \rangle |$
  $A : \langle A_1 : u_1, ..., A_m : u_m, B_1 : v_1, ..., B_n : v_n \rangle = t \wedge$
  $B : \langle B_1 : v_1, ..., B_n : v_n, C_1 : w_1, ..., C_p : w_p \rangle \in s(v_1, ..., v_n)\}$

  where each $A_i : u_i$ is a value of type $\hat{a}_i$, each $B_j : v_j$ is a value of type $\hat{b}_j$, and each $C_k : w_k$ is a value of type $\hat{c}_k$. Note that the values $B_j : v_j$ correspond to the input parameters of the on-demand data operation, and that the tuples in the output set are given the name of the tuples in the input set.

Example 3.8.

Consider the following complex value set

R = { *person:*⟨ *person_id:'0', name:'Luitpold Martucci', email:'Martucci@toronto.edu'* ⟩ }

and a function s representing an on-demand data operation of the following type:

person_id:integer → { *bid:*⟨ *person_id:integer, open_auction_id:integer, amount:real* ⟩ }

using the value 0 obtained from the single input tuple in the set R for the $person_id$ input parameter, through the invocation $s(0)$ we obtain

s(0) = { *bid:*⟨ *person_id:'0', open_auction_id:7, amount:145.00* ⟩,
bid:⟨ *person_id:'0', open_auction_id:9, amount:550.00* ⟩ }

The expression $R \overset{\mapsto}{\bowtie} s$ thus yields:

{ *person_bid:*⟨ *name:'Luitpold Martucci', email:'Martucci@toronto.edu',*
person_id:'0', open_auction_id:7, amount:145.00 ⟩,
person_bid:⟨ *name:'Luitpold Martucci', email:'Martucci@toronto.edu',*
person_id:'0', open_auction_id:9, amount:550.00 ⟩ }

3.2.4 Nesting and unnesting operations

We introduce two operators that enable us to nest and unnest tuple complex values. These operators take into consideration common values occuring in several tuples, therefore facilitating grouping or ungrouping them (hence the operators' names). In particular, these operators can be used to flatten or deflatten complex values into or from tuples in the relational model.

Group

Intuitively, grouping a set of tuple complex values R over a set of attributes X implies aggregating the tuples that are equal in all attributes except those in X to create a single tuple, which will contain a new set attribute with new tuples containing all of the X-values of the aggregated input tuples. This set attribute is given a new name, as are the tuples built from the X attributes that are contained in it; both of which are specified in the group expression.

- *Notation*: $group_{exp}(R)$

  Group expressions exp are constructed as follows, we use A to represent the type names that occur in the complex value instances, and B and B' to represent the new names of the grouped tuples set and its constituent tuples, respectively

$$\begin{array}{lcl} exp & ::= & group\,(A,\ B : list\,[B']\,) \\ list & ::= & A \mid A\,,\ list \end{array}$$

- *Operation type*:

  $group:\ \{A : \langle \hat{a}_1, ..., \hat{a}_m, \hat{b}_1, ..., \hat{b}_n \rangle\} \rightarrow \{A : \langle \hat{a}_1, ..., \hat{a}_m, B : \{B' : \langle \hat{b}_1, ..., \hat{b}_n \rangle\}\rangle\}$

- *Semantics*:

  $group_{exp}(R) =$
  $\{A : \langle A_1 : v_1, ..., A_m : v_m, B : w\rangle \mid ($
  $\exists t \in R \mid \forall_{i|1 \leq i \leq m}\ t.A_i = v_i \wedge w =$
  $\{B' : \langle B_1 : u_1, ..., B_n : u_n\rangle | A : \langle A_1 : v'_1, ..., A_m : v'_m, B_1 : u_1, ..., B_n : u_n\rangle$
  $\in R \wedge \forall_{i|1 \leq i \leq m}\ t.A_i = v'_i\}$
  $)\ \}$

  where all values $A_i : v_i$ and $A_i : v'_i$ are of type $\hat{a}_i$ and all values $B_i : u_i$ are of type $\hat{b}_i$.

Example 3.9.
Consider the following set of tuple complex values representing data (in flattened form) from the `interests` data service in Example 4.17

$R = \{$ *person:⟨ nickname:'Bob', tag:'sports', score:6.5 ⟩*
person:⟨ nickname:'Bob', tag:'cars', score:8.0 ⟩
person:⟨ nickname:'Alice', tag:'fashion', score:7.0 ⟩
person:⟨ nickname:'Alice', tag:'novels', score:8.5 ⟩ $\}$

The expression $group(person, interests : tag, score[s_tag])(R)$ thus yields:

$R' = \{$ *person:⟨ nickname:'Bob',*
interests:{s_tag:⟨ tag:'sports', score:6.5 ⟩,
s_tag:⟨tag:'cars', score:8.0 ⟩ },
person:⟨ nickname:'Alice',
interests:{s_tag:⟨ tag:'fashion', score:7.0 ⟩,
s_tag:⟨tag:'novels', score:8.5 ⟩ } }

Ungroup

Ungrouping performs the opposite transformation as grouping, i.e., given a set of tuple complex values R containing a set attribute, which in turn contains tuples consisting of the attribute set Z, the ungroup operation generates a new set of tuple complex values with the series of attributes in Z replacing the single set attribute.

- *Notation*: $ungroup(A, B)(R)$

  where A is the name of the tuple complex values in R and B the name of the set attribute to be ungrouped

- *Operation type*:

  $group: \ \{A : \langle \hat{a}_1, ..., \hat{a}_m, B : \{B' : \langle \hat{b}_1, ..., \hat{b}_n \rangle\}\rangle\} \rightarrow \{A : \langle \hat{a}_1, ..., \hat{a}_m, \hat{b}_1, ..., \hat{b}_n \rangle\}$

- *Semantics*:

  $$\begin{array}{l} ungroup(A, B)(R) = \\ \{A : \langle A_1 : v_1, ..., A_m : v_m, B_1 : u_1, ..., B_n : u_n \rangle \mid (\\ \quad \exists t \in R \mid \forall_{i \mid 1 \leq i \leq m} \, t.A_i = v_i \wedge \\ \qquad B' : \langle B_1 : u_1, ..., B_n : u_n \rangle \in t.B \\ ) \, \} \end{array}$$

  where all values $A_i : v_i$ and $A_i : v'_i$ are of type $\hat{a}_i$ and all values $B_i : u_i$ are of type $\hat{b}_i$.

Example 3.10.
Considering the set R' of tuple complex values produced as result in our example of the $group$ operator, the expression

$$ungroup(person, interests)(R')$$

yields the input set R of the same example, thus reversing the effect of the $group$ operator.

3.2.5 Set operations

Next we present the set operations that apply to complex values. In accordance to their mathematical foundations, they can only be applied to set of complex values of the same type.

Intersection

Computes the intersection of two input sets R and S of complex values of the same type $\hat{t}$.

- *Notation*: $R \cap S$

- *Operation type*: $\cap : \ \{\hat{t}\} \times \{\hat{t}\} \rightarrow \{\hat{t}\}$

- *Semantics*: $R \cap S = \{t | t \in R \wedge t \in S\}$

Union

Computes the union of two input sets R and S of complex values of the same type.

- *Notation*: $R \cup S$
- *Operation type*: $\cup : \ \{\hat{t}\} \times \{\hat{t}\} \rightarrow \{\hat{t}\}$
- *Semantics*: $R \cup S = \{t | t \in R \vee t \in S\}$

Set difference

Computes the set difference of two input sets R and S of complex values of the same type. Unlike the intersection and union operators, this operator is not commutative.

- *Notation*: $R - S$
- *Operation type*: $- : \ \{\hat{t}\} \times \{\hat{t}\} \rightarrow \{\hat{t}\}$
- *Semantics*: $R - S = \{t | t \in R \wedge t \notin S\}$

3.3 Hybrid queries

The operators that we have presented previously, which manipulate data in the form of complex values, enable us to characterize hybrid queries as query operator expressions. In addition, our characterization of on-demand and streaming data services as functions, which also applies to our query operators, enables us to define the semantics of such expressions representing hybrid queries based on the principle of function composition. Both of these aspects are detailed next.

3.3.1 Characterization of hybrid queries

We propose the stream-complex-value operators, $\mathcal{SC}$-$\mathcal{O}$, for the specification of hybrid queries. The following grammar specifies how $\mathcal{SC}$-$\mathcal{O}$ operator expressions E in are generated from on-demand and streaming data service operations, represented as o and s, respectively

D	$::=$	$o(v_1, ..., v_n) \mid s \mid F$	on-demand and (windowed) streaming operations
F	$::=$	$W_{N=n}(s) \mid W_{T=t}(s)$	tuple and time-based windows
E	$::=$	D	single operation query expression
	$\mid$	$E \overset{\rightarrowtail}{\bowtie} o$	bind-join on on-demand operation o
	$\mid$	$E \bowtie_\theta E$	theta-join under expression θ
	$\mid$	$\sigma_{exp}(E)$	selection under expression σ_{exp}
	$\mid$	$\pi_{exp}(E)$	projection under expression π_{exp}
	$\mid$	$\rho_{exp}(E)$	renaming under expression ρ_{exp}
	$\mid$	$group_{exp}(E)$	grouping under expression $group_{exp}$

$\mid\ ungroup(A, B)(E)$	ungroup set attribute B in A
$\mid\ E \times E$	cartesian product
$\mid\ E \cup E$	set union
$\mid\ E \cap E$	set intersection
$\mid\ E - E$	set difference

According to our specification, only on-demand data operations (invoked with the appropiate constant values) or stream data operations can produce data on their own. The various operators that we have presented can then be applied to the retrieved data.

In particular, windows can only be applied to streaming data operations; for the bind-join one of its arguments has to be an on-demand data operation. Otherwise, the operators can be applied to one or two arbitrary subexpressions depending on whether the operator is unary or binary, respectively. The composition of the operators and the data services must also adhere with the type rules of each of the operators in order for the expression to be valid.

Relying on our specification, hybrid queries and their semantics are formalized as follows.

Definition 3.13 (Hybrid query). A given $\mathcal{SC}$-$\mathcal{O}$ expression E over a series of on-demand and streaming data service operations $o_1, ..., o_m, s_1, ..., s_n$, represents a *hybrid query* whose result is given by the stream function resulting from the composition of the constituent operators of E and specific instances of the data operations o_i and s_j obtained via the $registry$ and $choice$ functions. Additional constant values v_k may be supplied for the on-demand data operations. Thus, the query can be represented as: $E(o_1(v_1, ..., v_p), ..., o_m(v_1, ..., v_q), s_1(t), ..., s_n(t))(t)$.

Therefore a hybrid query is a stream function that will produce at time t a set of tagged and timestamped tuples representing the query result. The result depends on the specific data operation instances that produce the data and the operators that process them. In the particular case that only on-demand data operations (accessed via constant values) and no stream data operation is involved, then the query result is invariant at all times.

Additional research is required to determine whether our operators properly form an algebra and thus respect its related properties. Further work is also needed to study the consistency and completeness of expressions build from our query operators. We consider both of these aspects to be part of future work.

3.3.2 HSQL a language for hybrid queries

We present a declarative language for expressing hybrid queries. Our language is based on the familiar SQL syntax and is called HSQL, which stands for Hybrid SQL. Queries in HSQL are issued over on-demand and stream data services, and can be transformed into operation expressions consisting of the operators presented in the previous section, this process is detailed in Chapter 4.

We illustrate the language and its capabilities through examples, focusing on the aspects involving data streams and complex values, each presented over the data services associated with a particular

scenario. The specification of the HSQL syntax is presented in grammar form in Appendix A.

To describe the capabilities of HSQL for querying data streams we introduce a scenario based on the NEXMark benchmark [TTPM02]. It consists of a series of data services that enable access to information about ongoing online auctions. The data services in question are all stream data services, each represented by a single stream data operation, whose simplified interfaces (timestamps are omitted for brevity) are presented next.

Example 3.11.

$$
\begin{aligned}
&person: \ \ t : time \rightarrow \\
&\quad \{person : \langle person_id : int, name : str, phone : str, email : str, income : real, \tau : time \rangle\} \\
&auction: \ \ t : time \rightarrow \\
&\quad \{auction : \langle open_auction_id : int, seller_person : int, category : int, quantity : int \rangle\} \\
&bid: \ \ t : time \rightarrow \\
&\quad \{bid : \langle person_ref : int, open_auction_id : int, amount : real \rangle\}
\end{aligned}
$$

The first operation represents persons that join the auction at a given time in order to both buy and sell items. Each person is given a unique $person_id$ and personal information such as their phone number, email, and annual income are made available. Each ongoing auction is identified by a $open_auction_id$ and offers a certain quantity of items belonging to a particular category. The items in question are offered by a person identified by a $seller_ref$, which corresponds to the $person_id$ in the person operation. Bids are made on the items of a particular auction for a specific currency $amount$, each bid also contains a reference to the corresponding auction and to the person that is making the bid. Next we present example tuples of each of the stream operations.

person: ⟨ *person_id:0, name:'Luitpold Martucci', phone:'+56(52)3418151',*
email:'Martucci@toronto.edu', income:59178.78 ⟩

auction: ⟨ *open_auction_id:9, seller_person:21, category:202, quantity:5* ⟩

bid: ⟨ *person_ref:30, auction_id:7, bid:145.00* ⟩

Based on the stream data services just discussed we present the following example queries, each with a brief description and its expression in HSQL. We also present an operation expression equivalent to the HSQL query, which however needs not be unique or optimal.

Example 3.12.
Among the bids made in the last 5 seconds, find those whose amount is between 50 and 100 euros

```
SELECT bid.person_ref, bid.auction_id, bid.amount
FROM bid [RANGE 5]
WHERE bid.amount > 50.0 and bid.amount < 100.0;
```

Time-based sliding windows are specified generically using the syntax `operation_name [RANGE T]`, where `T` is the time interval over which the window is created (in seconds). The `bid` stream is bounded by a 5 second window and the conditions in the `WHERE` clause enable to retrieve the desired tuple complex values, the operation expression for this query is as follows

$$\sigma(bid, amount > 50.0 \wedge amount < 100.0)(\mathcal{W}_{T=5}(bid))$$

Example 3.13.

For the persons joining and the products offered during the last minute, generate the name and email of the person along with the id of the product(s) he/she offers

```
SELECT P.name, P.emailaddress, A.open_auction_id
FROM auction [RANGE 60] as A, person [RANGE 60] as P
WHERE A.seller_person = P.person_id;
```

We allow the use of aliases to refer to relations as is usual in SQL. In this case a join is performed on the bounded relations according to the condition in the `WHERE` clause, as the corresponding operation expression is

$$\pi(person, name, email, open_auction_id)$$
$$(\sigma(person, person_id = seller_person)(\mathcal{W}_{T=60}(person) \times \mathcal{W}_{T=60}(auction)))$$

Example 3.14.

For the last 30 persons and 30 products offered, retrieve the bids of the last 20 seconds greater than 15 euros

```
SELECT bid.person_ref, bid.auction_id, bid.amount
FROM bid [RANGE 20], auction [ROWS 30], person [ROWS 30]
WHERE bid.auction_id = auction.open_auction_id AND
auction.seller_person = person.person_id AND bid.amount > 15;
```

Tuple-based sliding windows are specified using the syntax `operation_name [ROWS N]`, where `N` is the size of the window, i.e., the maximum number of tuples it contains at a given time. The equivalent of two equijoin operations are performed for this query, as specified in the next operation expression

$$\pi(person, person_ref, auction_id, amount)$$
$$(\sigma(person, person_id = seller_person \wedge auction_id = open_auction_id \wedge amount > 15)$$
$$(\mathcal{W}_{N=30}(person) \times \mathcal{W}_{N=30}(auction)) \times \mathcal{W}_{t=20}(bid))$$

3.4 Conclusions

In this chapter we have presented a data model for hybrid queries. Our model is defined over on-demand and stream data services that produce typed complex values. A series of operators enable to manipulate such data, and their composition permits to express hybrid queries and to compute their result. By the use of recursive expressions in certain operators we are able to operate over nested complex values in a clear and concise manner.

Our model addresses many of the challenges associated with limited access patterns and streams that were discussed in the previous chapter. Continuous queries in particular are represented by stream functions, in turn, their result is represented by a tagged data stream over which we can compute the stream function extent, a time-varying dataset that reflects all updates. Finally, we introduced HSQL, a declarative language with a SQL-like syntax for expressing a significant subset of hybrid queries.

Two major issues need to be addressed in order to facilitate the evaluation of an HSQL query. First it is necessary to transform the query into an operator expression, while verifying that such expression is valid; meaning that it satisfies the type rules and the data dependencies associated with on-demand data services. Second, according to our approach an operator expression needs to be transformed into a workflow representing a service coordination. Both of these issues are addressed in the next chapter.

CHAPTER 4

Evaluating a Hybrid Query as a Service Coordination

In this chapter we describe how to build a service coordination (specified via a workflow) to evaluate a given hybrid query. Section 4.1 presents how a hybrid query $\mathcal{SC}$-$\mathcal{O}$ operator expression can be represented as a query workflow. The procedure to obtain a query workflow from a given HSQL query and to verify that it satisfies the desired properties is presented in Section 4.2. Section 4.3 shows how the operators in a query workflow are evaluated by simple and composite computation services, the latter of which are specified as service coordinations themselves, either by operator workflows or in a coordination language. In Section 4.4 we formalize operator workflows and describe how they can be generated from the specifications of composite computation services given in our coordination language. Finally, we present our conclusions in Section 4.5.

4.1 Representation of a hybrid query as a workflow

In this section we present how a hybrid query as described in the previous chapter can be represented as a workflow (which can be used to specify a service coordination). With this purpose in mind, we first introduce a query workflow model that supports such a representation.

4.1.1 Query workflow model

An $\mathcal{SC}$-$\mathcal{O}$ operator expression states how data services and query operators are composed to form a hybrid query. Formally, the expression represents a composition of functions that operate over sets of typed complex values, yielding a tagged data stream as the query result. We require a representation that respects those semantics but that is more suitable for execution, particularly in a service-based environment. For this purpose we adopt a workflow model to represent queries, and which we further use to specify service coordinations that enable their execution.

Our *query workflow model* is based on the parallel and sequential composition of activities; its constructs are presented in Figure 4.1. As we can see, activities are represented by rectangles and given a

Construct	Graphical representation	Description
Activity	A	The activity labeled *A*; it may be executed multiple times
Sequential composition	A → B	A dependency exists between activity *A* and activity *B*
Parallel composition	A, B	No dependency exists between activities *A* and *B*, but both are requried to be executed

Figure 4.1: Query workflow model constructs

unique name that identifies them. An activity represents a particular task that needs to be executed, possibly several times. The sequential composition of two activities A and B denotes a dependency according to which activity A must be completed before activity B. The parallel composition of two activities A and B implies that there is no dependency between them, consequently they can in principle be executed in parallel. It is still required that both are executed before the execution of the rest of the workflow can proceed.

4.1.2 Graph representation of query workflows

We now present a formalization of query workflows based on the constructs presented in Figure 4.1. First, we show how these constructs can be represented by directed graphs that we refer to as workflow graphs. In our context, a directed graph $G = (V, E)$ consists of a set V of vertices and a set E of edges with $E \subseteq V \times V$. Thus the pair $e = (v_1, v_2)$ denotes an edge between the vertices v_1 and v_2. The definition of a workflow graph is therefore as follows.

Definition 4.1. [**Workflow graph**] A *workflow graph*, WG, is a quadruple (V, E, in, out), where

- V is a set of vertices;
- $E \subseteq V \times V$ is a set of edges;
- $in, out \in V$ are start and end vertices;
- $A \subseteq V$ is a set of activity vertices, represented by their unique names;
- $P \subseteq V$ is a set of parallel composition vertices of the form par_l, $endpar_l$, such that $l \in L$, L being a set of unique labels.

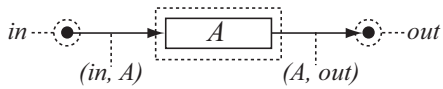
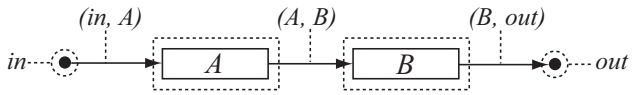
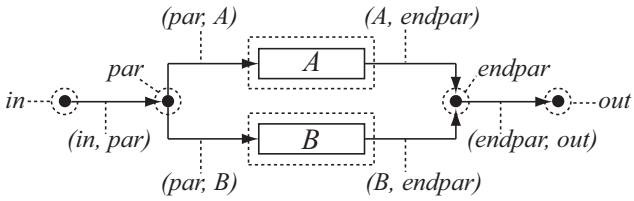
Construct	Graph-based representation
Activity	
Sequential composition	
Parallel composition	

Figure 4.2: Directed graph representation of workflow constructs

Figure 4.2 depicts how this definition applies to the constructs of our query workflow model. Note that vertices are shown with dashed rectangles and circles.

In the case of a single activity, a vertex is created to represent the activity. The vertex has the same name as the activity (A in Figure 4.2, top row). Two vertices named $input$ and $output$ are created to represent the start and end points of the workflow. Two edges link the start point, activity, and end point, thus forming a workflow consisting of a single activity. The sequential composition extends this principle, applied in Figure 4.2 (middle row) to two activities, although it can be extended to *n* activities. A dependency between two sequential activities is specified by an edge between the respective activity vertices.

For the parallel composition of activities two additional parallel composition vertices are used, *par* and *endpar*. An edge is created between the *in* start point vertex and the *par* vertex. Next, the *par* vertex is linked to the parallel activities by the corresponding edges, indicating that the activities can be executed from that point on. Afterwards the parallel activities' vertices are linked to the *endpar* vertex, which indicates that they must be completed. Finally, the *endpar* vertex is linked to the end point of the workflow, as shown in Figure 4.2 (bottom row).

4.1.3 Hybrid query workflows

Based on our workflow model just described we can derive a workflow representation for hybrid queries. First we will show how the workflow can be constructed in conjunction with the operator expression, thus establishing the relation between hybrid query operator expressions and query workflows. The workflow construction is based on a series of construction rules that map each operator subexpression to its respective workflow. When applied to the expression in its entirety, these construction rules yield a query workflow equivalent to a given operator expression.

We restate the stream-complex-value operators, $\mathcal{SC}$-$\mathcal{O}$, of Section 3.3.1 for the specification of hybrid queries. We add parentheses to the binary operator expressions to avoid ambiguity, also providing an additional label to identify the expression. The following grammar thus specifies how $\mathcal{SC}$-$\mathcal{O}$ expressions E are generated.

D	$::=$	$o(v_1, ..., v_n) \mid s \mid F$	on-demand and (windowed) streaming operations
F	$::=$	$W_{N=n}(s) \mid W_{T=t}(s)$	tuple and time-based windows
E	$::=$	D	single data operation expression
	$\mid$	$(E \overset{\rightarrowtail}{\bowtie} o)^l$	bind-join on on-demand operation o
	$\mid$	$\sigma_{exp}(E)$	selection under expression exp
	$\mid$	$\pi_{exp}(E)$	projection under expression exp
	$\mid$	$\rho_{exp}(E)$	renaming under expression exp
	$\mid$	$group_{exp}(E)$	grouping under expression exp
	$\mid$	$ungroup(A, B)(E)$	ungroup set attribute B in A
	$\mid$	$(E \bowtie_\theta E)^l$	theta join
	$\mid$	$(E \cup E)^l$	set union
	$\mid$	$(E \cap E)^l$	set intersection
	$\mid$	$(E - E)^l$	set difference

In the following definition we make the assumption that vertices representing data operations, as well as query operators, are annotated with the necessary parameters to enable the evaluation of the query. Such parameters include, for example, the conditions of selection and theta join operators. Furthermore, the annotations make those vertices unique, whereas the par and $endpar$ vertices of parallel compositions are made unique using the labels l of the rules above.

Definition 4.2. [**Query workflow graph**] For an $\mathcal{SC}$-$\mathcal{O}$ operator expression or subexpression w the function *qwfg* specifies the *query workflow graph* such that *qwfg*$(w) = (V, E, in, out)$, as follows

$$\text{if } w = r \text{ then } \begin{cases} V = \{r, in, out\} \\ E = \{(in, r), (r, out)\} \\ in, out \text{ are new vertices and } r \text{ is an on-demand or stream data operation name} \end{cases}$$

$$\text{if } w = W_\beta(s) \text{ then } \begin{cases} \text{Let } qwfg(s) = (V', E', in', out'),\ s \text{ being a stream data operation name} \\ V = V' \cup \{W_\beta\} \\ E = (E' - \{(v, out') | v \in V'\}) \cup \{(v, W_\beta) | (v, out') \in E'\} \cup (W_\beta, out) \\ \text{and } in = in',\ out = out';\ W_\beta \text{ denotes a window with } \beta \text{ being either } N = n \text{ or } T = t \end{cases}$$

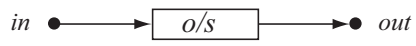
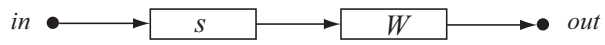
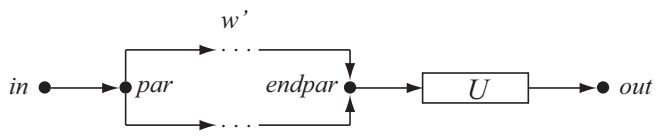
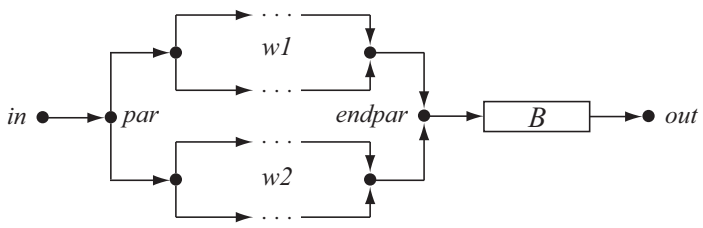
Rule	**Query workflow representation**
On-demand or stream data operation *w = o* or *w = s*	
Window operator *w = W(s)*	
Unary operator *w = U(w')*	
Binary operator *w = (w1 B w2)*	

Figure 4.3: Rules for query workflow graph construction

In the following rules let U denote a unary operator (σ_{exp}, π_{exp}, ...) and B a binary operator ($\bowtie_\theta$ $, \cup, \cap, -$, ...), while w', w_1 and w_2 denote subexpressions.

$$\text{if } w = U(w') \text{ then } \begin{cases} \text{Let } qwfg(w') = (V', E', in', out') \\ V = V' \cup \{U\} \\ E = (E' - \{(v, out') | v \in V'\}) \cup \{(v, U) | (v, out') \in E'\} \cup (U, out) \\ \text{and } in = in', out = out' \end{cases}$$

$$\text{if } w = (w_1 B w_2)^l \text{ then } \begin{cases} \text{Let } qwfg(w_1) = (V_1, E_1, in_1, out_1) \text{ and } qwfg(w_2) = (V_2, E_2, in_2, out_2) \\ V = (V_1 - \{out_1\}) \cup (V_2 - \{in_2\}) \cup \{par_l, endpar_l\} \cup \{B\} \\ E = \{(in_1, par_l), (endpar_l, B), (B, out_2)\} \\ \cup\{(par_l, v_1) | (in_1, v_1) \in E_1\} \cup \{(par_l, v_2) | (in_2, v_2) \in E_2\} \\ \cup\{(v'_1, endpar_l) | (v'_1, out_1) \in E_1\} \cup \{(v'_2, endpar_l) | (v'_2, out_2) \in E_2\} \\ \cup(E_1 - (\{(in_1, v_1) | v_1 \in V_1\} \cup \{(v'_1, out_1) | v'_1 \in V_1\})) \\ \cup(E_2 - (\{(in_2, v_2) | v_2 \in V_2\} \cup \{(v'_2, out_2) | v'_2 \in V_2\})) \\ par_l, endpar_l \text{ are new vertices and } in = in_1, out = out_2 \end{cases}$$

Intuitively, on-demand and data stream operations are represented by a query workflow with a single activity to access the data as well as its corresponding in and out vertices. If a window is defined over a stream operation, it is represented by a window operator next in sequence to the stream operation vertex. Unary operators over subexpressions are treated in a similar manner, inserting them in sequence at the end of the workflow and reestructuring the edges to the in and out vertices.

In the case of binary operators involving two subexpressions, their respective workflows are composed in parallel, with a new activity vertex representing the operator inserted after the $endpar_l$ vertex. Figure 4.3 depicts the query workflow construction process as defined by the rules of Definition 4.2.

The following examples illustrate how a query workflow can represent a given operator expression as well as the HSQL query from which the operator expression originates. The generation of the operator expression from an HSQL query is detailed in the next section.

Example 4.15.
We present next an HSQL specification of the query presented in the scenario of Example 1.1.
Find friends which are no more than 3 km away from my current location (48.85889, 2.29583) considering their locations of the last 10 min, which are also over 21 years old and that are interested in art.

```
SELECT p.nickname, p.age, p.gender, p.email
FROM profile AS p, location [range 10] AS l, interests AS i
WHERE p.age >= 21 AND l.nickname = p.nickname AND i.nickname = p.nickname
AND i.tag='art' AND distance(lat, lon, 48.85889, 2.29583) <= 3.0;
```

Example 4.16. The operator expression corresponding to the query presented in Example 4.15 is of the following form

$$\pi(\sigma_{tag}(\sigma_{age}(\sigma_{distance}(\mathcal{W}_T(location)) \overset{\rightarrowtail}{\bowtie} profile) \overset{\rightarrowtail}{\bowtie} interests))$$

Where the annotations on the operators have been omitted for simplicity. The expression applies a window to the `location` stream data service and the produced data is then joined with that of the

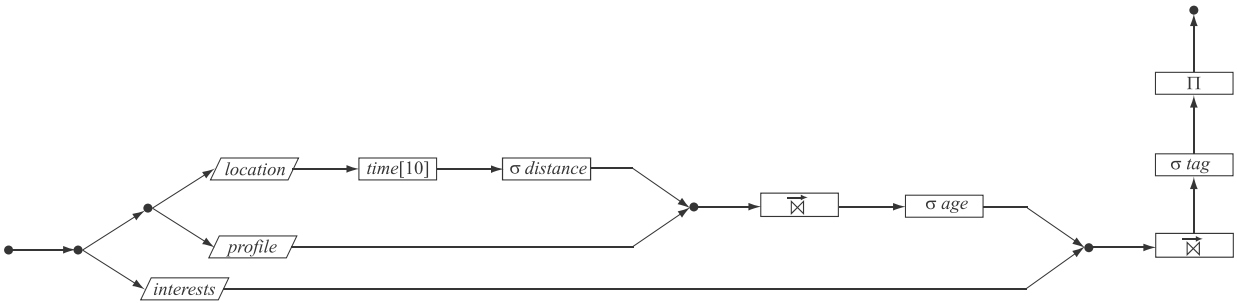


Figure 4.4: Query workflow graph for the query in Example 1.1

`profile` and `interests` on-demand data services through bind-joins. The conditions of the `WHERE` clause of the query are applied by selection operators, one which involves the `distance` function. A projection operator then produces the desired attributes in the result.

The previous expression generates the query workflow graph presented in Figure 4.4. We present in Table 4.1 the derivation of its following subexpression via the four construction rules of Definition 4.2.

$$\sigma_{age}(\sigma_{distance}(\mathcal{W}_T(location)) \overset{\rightarrowtail}{\bowtie} profile)$$

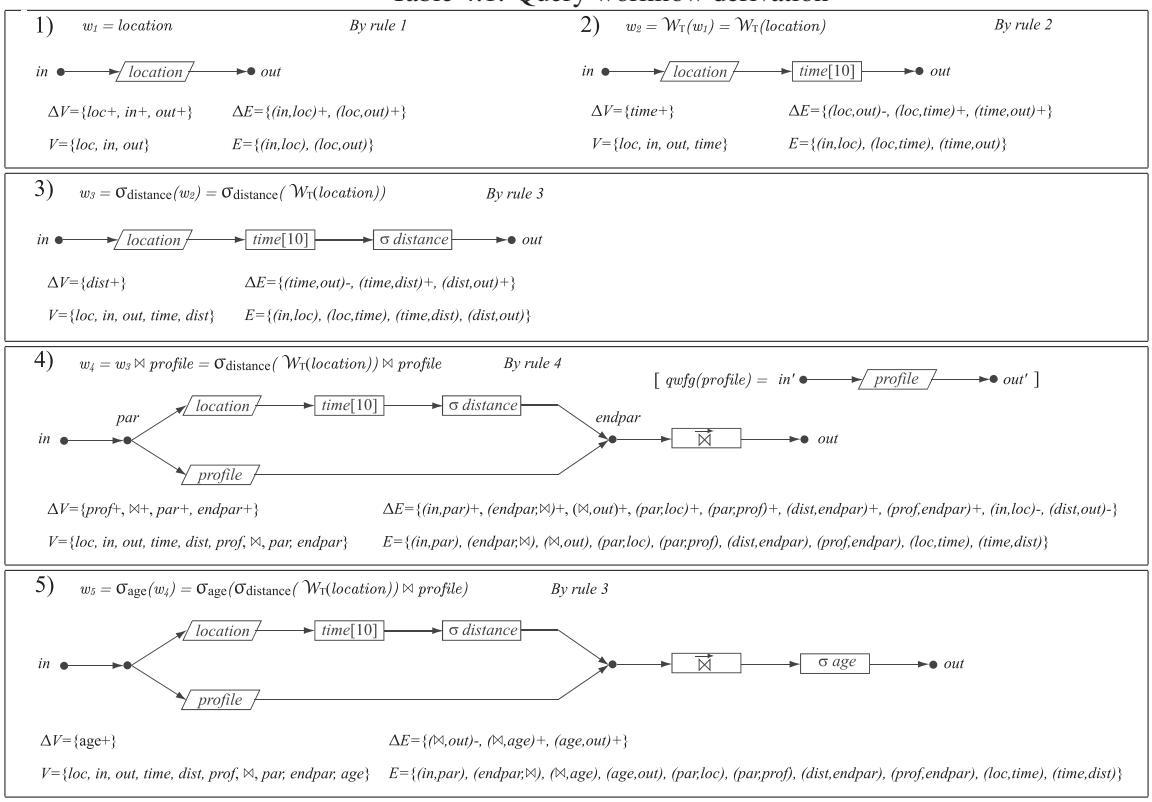
We begin with empty sets V and E of vertices and edges, respectively, in accordance to Definition 4.2. Then for each subexpression w_i we apply the corresponding rule of Definition 4.2, obtaining sets ΔV and ΔE that denote, respectively, the changes to the set of vertices and edges obtained at the completion of the previous step (and the empty initial sets in the case of the first step). A positive sign indicates that the vertex or edge is added to the set, whereas a negative sign indicates it is removed from the set. In this manner at the last step of Table 4.1 the workflow corresponding to the aforementioned subexpression.

4.2 Generation of an HSQL query workflow

We now address the issue of how to generate a query workflow from declarative query in HSQL. First, let us recall that not all syntactically correct operator expressions (and hence their equivalent query workflows) specify valid queries. This is due to the fact that we may have type rule violations such as a union over different types, or the projection of a non-existent attribute. In addition, we have to take into consideration that on-demand data services can only be accessed through their interfaces. Therefore, if values for the input parameters of a particular data operation are not available, then none of the data can be retrieved.

In this section we present an algorithm that given a HSQL query and the interfaces of the data services it concerns, will generate (if possible) a query workflow to evaluate the query. If the generation of the query workflow is possible we say that the query is *feasible*.

Table 4.1: Query workflow derivation



Assumptions

We present an algorithm to construct HSQL query workflows, currently it supports only queries involving windows, selection, projection, bind-join, and equi-join operators; and these cannot be applied over nested subelements, e.g., a set attribute can be projected but no selection can be defined over its constituent elements. The extension of the algorithm to the rest of the query operators and the complex values they manipulate forms part of future work.

We will make use of *binding pattern* annotations over data service interfaces. These will indicate which attributes obtained from on-demand data operations correspond to input parameters and hence are *bound*, i.e. are required to obtain the data; whereas *free* attributes are provided when the data is retrieved with no special requirement. Concretely, an on-demand data operation f annotated with binding patterns is of the form

$$f(A_1 : v_1, ..., A_m : v_m) \rightarrow \{A : \langle A_1^b : v_1, ..., A_m^b : v_m, B_1^f : u_1, ..., B_n^f : u_n \rangle\}$$

where the A_i attributes are denoted by ^{b} as bound and the B_j attributes are denoted by ^{f} as free. For the purposes of this section we can abbreviate the above interface as

$$A : \langle A_1^b, ..., A_m^b, B_1^f, ..., B_n^f \rangle$$

In the case of data stream operations, all of their attributes are free by default.

Example 4.17.
Consider the following data operations from Example 1.1, the first is a data stream operation and the next two are on-demand data operations

$$location(t{:}time) \rightarrow \{location{:}\langle\ nickname^f{:}str,\ lat^f{:}real,\ lon^f{:}real\ \rangle\}$$

$$profile(nickname{:}str) \rightarrow \{profile{:}\langle\ nickname^b{:}str,\ age^f{:}int,\ gender^f{:}str,\ email^f{:}str\ \rangle\}$$

$$interests(nickname{:}str) \rightarrow \{interests{:}\langle\ nickname^b{:}str,\ tag^f{:}str,\ score^f{:}real\ \rangle\}$$

these annotated data operations can be abbreviated as

$$location{:}\langle\ nickname^f,\ lat^f,\ lon^f\ \rangle$$

$$profile{:}\langle\ nickname^b,\ age^f,\ gender^f,\ email^f\ \rangle$$

$$interests{:}\langle\ nickname^b,\ tag^f,\ score^f\ \rangle$$

Overview of the generation process

Given a HSQL query multiple tasks need to be performed in order to generate a query workflow. Among these tasks, possibly the most challenging is to determine the appropriate joins. We propose for this purpose an algorithm based on the Graham-Yu-Ozsoyoglu (GYO) algorithm [Yan81] used in database theory to determine if a relational query is acyclic. We extended that algorithm to take into consideration the binding patterns associated with on-demand data services, therefore we refer to our algorithm as BP-GYO. The query workflow generation process consists of three main phases that are described in the remainder of this section.

1. Represent the join dependencies by a hypergraph, which will be used to generate a parse tree.
2. Process the hypergraph by BP-GYO reduction, yielding a parse tree denoting the valid join orders.
3. Traverse the parse tree, deriving a join workflow to which the remaining operators are added to obtain the full query workflow.

To illustrate each of these phases we will use the HSQL query presented in Example 4.15.

4.2.1 Representation of join dependencies as a hypergraph

A HSQL query like the one in Example 4.15 defines a series of join dependencies in its `WHERE` clause. In particular, using the aliases `p`, `l`, and `i` for the `profile`, `location`, and `interests` data services, respectively, it states

```
l.nickname = p.nickname AND i.nickname = p.nickname
```

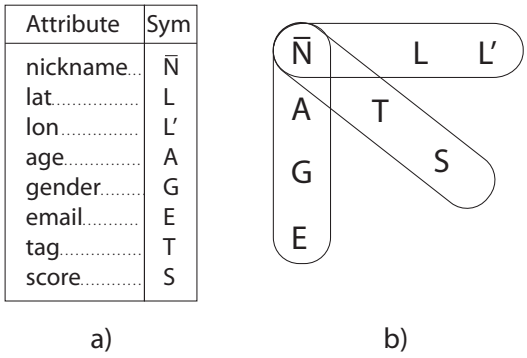


Figure 4.5: Attribute symbols and query hypergraph

Each of these two conditions implies a bidirectional join dependency between two data operations; for instance, the `nickname` attribute of `l` has to be joined with the `nickname` attribute of `p`, which can be seen vice versa. These dependencies further propagate transitively (in this case via `p` at the right). Thus, a join dependency among the three data services arises, in particular, based on the common `nickname` attribute.

In the general case, the join dependencies between several data service operations can be represented by a *hypergraph*, formed by vertices and hyperedges; the latter which unlike ordinary graph edges, can be sets of any number of vertices. Intuitively, a hypergraph is a set of sets of vertices with multiple overlappings. Formally, we define a hypergraph as follows.

Definition 4.3. [**Hypergraph**] A hypergraph H is a pair $H = (V, \mathcal{E})$ where V is a set of vertices and $\mathcal{E}$ a family of sets whose members represent hyperedges, such that for all $e \in \mathcal{E}$, it holds that $e \subseteq V$ and $|e| > 0$.

With a hypergraph we are able to represent join dependencies involving multiple attributes. In order to construct the hypergraph, the attributes from the data operations need to be given first a symbol that corresponds to a vertex in the hypergraph. In so doing we can represent the joins by giving the same symbol to all of the attributes in a join chain. For illustration purposes, we use symbols consisting of single letters and give the attributes that take part in a chain of joins always symbols distinguished by a bar (e.g. $\bar{N}$). The symbols given to the attributes of the data operations from Example 4.17 based on the query in Example 4.15 are presented in Figure 4.5 a).

Once the symbols have been generated, then we substitute the original attribute names in the data operations for their symbols. As we will discuss later, these new data operation representations are used to construct the hypergraph. In Example 4.18 we apply this transformation to the data operations of the query in Example 4.17.

Example 4.18.

location:$\langle\ \bar{N}^f, L^f, L'^f\ \rangle$

profile:$\langle\ \bar{N}^b, A^f, G^f, E^f\ \rangle$

interests:$\langle \bar{N}^b, T^f, S^f \rangle$

Only the `nickname` attributes take part in a join and hence are represented by $\bar{N}$. The rest of the attributes are simply abbreviated by their corresponding symbols. The corresponding hypergraph is depicted in Figure 4.5 b), showing the join dependency on the `nickname` attribute. Note that by the use of the $\bar{N}$ symbol we are able to represent in the hybergraph the joins involving the `nickname` attribute in the three data operations, which in other circumstances may have different names.

Next, we describe how to generate the symbols for the joined attributes and then how to construct the query hypergraph.

4.2.1.1 Generation of attribute symbols for the hypergraph

Consider a set C of conditions of the form $leftTerm = rightTerm$ obtained from the `WHERE` clause of the query, both terms being of the form *op_name.att_name* denoting the attribute *att_name* of the data operation *op_name*. The algorithm in Listing 4.1 generates the symbols for all of the attributes involved in a join. We assume the existence of the functions `getSymbol` and `setSymbol` to obtain and assign the symbol of an attribute, as well as a function `genSymbol` to generate random symbols. Finally, we also assume a function `attsWithSymbol` that enables to obtain the set of all attributes that up to a certain point have been given a particular symbol.

Listing 4.1: Generation of attribute symbols

```
1  foreach join_condition c in C
2      if(getSymbol(c.leftTerm) = nil) {
3          if(getSymbol(c.rightTerm) = nil) {
4              setSymbol(c.leftTerm, genSymbol())
5              setSymbol(c.rightTerm, getSymbol(c.leftTerm))
6          }
7          else {
8              setSymbol(c.leftTerm, getSymbol(c.rightTerm))
9          }
10     }
11     else {
12         if(getSymbol(c.rightTerm) = nil) {
13             setSymbol(c.rightTerm, getSymbol(c.leftTerm))
14         }
15         else {
16             Set R := attsWithID(c.leftTerm)
17             foreach attribute r in R
18                 setSymbol(r, getSymbol(c.leftTerm))
19         }
20     }
```

The algorithm basically processes all join conditions in sequence and verifies which attributes have been given a symbol. If both of the attributes do not have a symbol, then a symbol is generated and assigned to both. If one of the attributes has a symbol and the other one does not, the existing symbol is used for the unassigned attribute.

The most complex case is when both attributes have symbols, and thus two join subchains need to be merged. In this case, by convention, the symbol of the left attribute overrides that of the right attribute, and it is also applied to all of the attributes that have the old symbol of the right attribute. In this manner, the symbols are assigned transitively and thus cover the entirety of the new join chain.

Example 4.19.
Consider the join conditions of the query in Example 4.15 and apply to them the algorithm in Listing 4.1.

```
C = {l.nickname = p.nickname, i.nickname = p.nickname}
Symbols = { }
1) l.nickname = p.nickname                line 1
  getSymbol(l.nickname) = nil             line 2
  getSymbol(p.nickname) = nil             line 3
  setSymbol(l.nickname, N̄)                line 4
  setSymbol(p.nickname, N̄)                line 5
Symbols = {l.nickname→ N̄, p.nickname→ N̄}
2) i.nickname = p.nickname                line 1
  getSymbol(i.nickname) = nil             line 2
  getSymbol(p.nickname) = N̄               line 3
  setSymbol(i.nickname, N̄)                line 8
Symbols = {l.nickname→ N̄, p.nickname→ N̄, i.nickname→ N̄}
```

Besides the symbols we also need the binding patterns, since we need to generate query workflows that are congruent with the input parameters required for on-demand data services. Therefore we assume the existence of a function $bp(op_name, symbol) \rightarrow \{b, f\}$ which enables to determine the bound or free property of a particular attribute (specified via its symbol) in a particular data operation, in accordance to its respective interface (see the Assumptions of this chapter).

4.2.1.2 Construction of the hypergraph

Once the join attribute symbols have been generated, the construction of the hypergraph is straightforward. Recall that a hypergraph consists of a set V of vertices and a family of sets $\mathcal{E}$ of hyperedges. Figure 4.6 shows the algorithm to construct the hypergraph. We use D to represent the set of all data operations that appear in the query. We also assume the existence of a function *getAttributes* that receives the name of a data operation and returns the set of symbols of its constituent attributes.

In the algorithm we add the symbols of all of the attributes of each data operation as vertices. Each of these sets of attribute symbols also represents a hyperedge, which is added to the set family $\mathcal{E}$.

1 *Let* $V := \emptyset$ *and* $\mathcal{E} := \emptyset$
2 ***foreach*** *data_operation op* ***in*** *D*
3 $V := V \cup getAttributes(op)$
4 $\mathcal{E}' := \{\, e \mid e \in \mathcal{E} \vee e = getAttributes(op) \,\}$
5 $\mathcal{E} := \mathcal{E}'$

Figure 4.6: Construction of the query hypergraph

Example 4.20.
Consider the data operations in Example 4.18 and apply to them the algorithm in Figure 4.6.
$V := \emptyset$, $\mathcal{E} := \emptyset$
$D = \{location, profile, interests\}$

1)	$op = location$	line 2
	$V := \emptyset \cup \{\bar{N}, L, L'\}$	line 3
	$\mathcal{E}' := \{\{\bar{N}, L, L'\}\}, \quad \mathcal{E} := \mathcal{E}'$	lines 4-5
2)	$op = profile$	line 2
	$V := \{\bar{N}, L, L'\} \cup \{\bar{N}, A, G, E\}$	line 3
	$\mathcal{E}' := \{\{\bar{N}, L, L'\}, \{\bar{N}, A, G, E\}\}, \quad \mathcal{E} := \mathcal{E}'$	lines 4-5
3)	$op = interests$	line 2
	$V := \{\bar{N}, L, L', A, G, E\} \cup \{\bar{N}, T, S\}$	line 3
	$\mathcal{E}' := \{\{\bar{N}, L, L'\}, \{\bar{N}, A, G, E\}, \{\bar{N}, T, S\}\}, \quad \mathcal{E} := \mathcal{E}'$	lines 4-5

At termination, $V = \{\bar{N}, L, L', A, G, E, T, S\}$ and $\mathcal{E} = \{\{\bar{N}, L, L'\}, \{\bar{N}, A, G, E\}, \{\bar{N}, T, S\}\}$ which corresponds to the hypergraph in Figure 4.5 b).

4.2.2 BP-GYO reduction of the hypergraph to obtain a join parse tree

Once the query hypergraph has been built, we perform our BP-GYO reduction algorithm. By reduction we mean that hyperedges that meet a particular criteria are removed from the hypergraph one after the other. If by the end of the process a single hyperedge remains, then the query is considered to be well formed (at least with respect to joins) and the construction of the query workflow can proceed.

Formally, the reduction of the hypergraph down to a single hyperedge is a sufficient and necessary condition for the query to be acyclic [Yan81]. Also, in general terms, the fact that a query is acyclic implies that it produces a meaningful result [BFMY83]. We consider that these properties hold in the presence of binding patterns since these only affect how the data can be retrieved.

The objective of the BP-GYO algorithm, besides verifying that the query is acylic, is to obtain a join parse tree denoting the valid join orders. This tree will then be used to generate a join workflow.

The next two definitions present two concepts that are fundamental to understand BP-GYO reduction.

Definition 4.4. [**Ear hyperedge**] Given a hypergraph $H = (V, \mathcal{E})$ we say a hyperedge $e \in \mathcal{E}$ is an *ear* if and only if e is divisible into two subsets of vertices:

1. vertices that appear in e and no other hyperedge
2. vertices that are contained in another hyperedge f.

Formally, we have $e = e_1 \cup e_2$ where $|e_1| > 0$ and $e_1 \not\subseteq f$ for all $f \in \mathcal{E}$ such that $f \neq e$; and $\exists f' \in \mathcal{E}$ such that $e_2 \subseteq f'$ and $f' \neq e$. We also say that f' can *consume* the ear e.

Definition 4.5. [**Ear consumption**] Given a hypergraph $H = (V, \mathcal{E})$ in addition to an ear hyperedge e and a consumer hyperedge f', as specified by Definition 4.4, the *consumption* of ear e by f' results in a hypergraph $H' = (V', \mathcal{E}')$ such that

- $V' = V - e_1$
- $\mathcal{E}' = \{f \in \mathcal{E} | f \neq e\}$.

Intuitively, an ear is a hyperedge with *isolated* vertices (e_1) and shared vertices (e_2). When the ear is consumed by a *consumer* hyperedge (f'), the isolated vertices are completely removed from the hypergraph. While the ear hyperedge is also removed, the shared vertices remain since they are contained in (at least) the consumer hyperedge.

The reduction process suffices to determine if the query is well formed, but not to generate a join workflow. For this purpose we generate during the reduction a *parse tree*, which specifies which hyperedge consumed which other hyperedge(s). Concretely, a directed edge (f', e) in the tree indicates that the hyperedge f' consumed the ear e. The result is a tree in which non-leaf nodes can have more than two children. Recall that hyperedges represent data operations, because of this (as we will see in the next subsection) the parse tree encapsulates the valid join orders for the query.

The algorithm in Listing 4.2 performs our BP-GYO reduction over a given hypergraph $H = (V, E)$. It takes into consideration the binding patterns and produces a parse tree containing the join orders.

Listing 4.2: BP-GYO reduction and parse tree generation

```
tree t := createTree()
while( E.size > 1 ) {
    h_edge ear := findEar(H)
    if(ear = nil)
        return nil
    h_edge c := findConsumer(H, ear)
    H := remove(H, ear)
    t := addEdge(t, c, ear)
    setAttsFree(c)
}
```

The algorithm tries to remove ears until a single hyperedge remains. In the case that no suitable ear is found, the algorithm returns nil which means that the query is cyclic and no query workflow can be generated. Ears are found by the function *findEar*, which applies the criteria of Definition 4.4. If multiple ears exist at a given time, the function selects one arbitrarily.

For the selected ear a suitable consumer is found, similarly multiple consumers are possible and one is chosen arbitrarily. For the selection of the consumer we check, in addition to the criteria of Definition 4.5, that the consumption is *feasible* considering the binding patterns of both operations, which we will detail shortly.

The ear is removed from the hypergraph and the corresponding consumer-ear edge is added to the parse tree. In addition, the binding patterns of the consumer (as specified by the function bp described in the previous subsection) are all set to free in the data operation corresponding to the consumer c. This is because the data produced by a possible bind-join between the ear and consumer can now be used to access data from other operations involved in the query. Therefore, c can consume additional ears in the next iterations that were previously unfeasible. We present next the criteria for feasibility between an ear and a consumer.

Definition 4.6. [**Feasible ear consumption**] Given an ear hyperedge e and a consumer hyperedge c, as specified by Definitions 4.4 and 4.5, the consumption of e by c is said to be *feasible* if the following criteria are met. We assume the functions $Bound(h_edge)$ and $Free(h_edge)$ that receive a hyperedge and return the set of symbols of all bound or free attributes, respectively, in the data operation corresponding to the hyperedge, as specified by the function bp.

- If both the ear and the consumer have no bound attributes, i.e. $|Bound(e)| = 0$ and $|Bound(c)| = 0$, then consumption is feasible.
- If both the ear and the consumer have bound attributes, i.e. $|Bound(e)| > 0$ and $|Bound(c)| > 0$, then consumption is not feasible.
- If the ear has bound attributes and the consumer does not, i.e. $|Bound(e)| > 0$ and $|Bound(c)| = 0$, then consumption is feasible if and only if for all symbols $b \in Bound(e) \exists f \in Free(c)$ such that $b = f$.
- If the ear has no bound attributes but the consumer has, i.e. $|Bound(e)| = 0$ and $|Bound(c)| > 0$, then consumption is feasible if and only if for all symbols $b \in Bound(c) \exists f \in Free(e)$ such that $b = f$.

To illustrate the algorithm in Listing 4.2 let us consider the hypergraph presented in Figure 4.5 b) corresponding to the query in Example 4.15.

First, we identify $\bar{N}C$ as an ear, whose attributes are divided into the sets $\{C\}$ and $\{\bar{N}\}$ of isolated and shared vertices, respectively. Recall that isolated vertices are removed entirely, while shared vertices remain in the hypergraph but not as part of the ear, which is removed. A hyperedge that contains the shared subset of vertices of the ear is then selected to consume the ear. In the example, the $\bar{N}C$ ear denoted by a dashed line in Figure 4.7 a) is consumed by the hyperedge $\bar{N}AGE$, resulting in the hypergraph presented in b) of the same figure. Note that vertex C is removed but the vertex $\bar{N}$ is kept.

Each time a hyperedge is selected to consume an ear, the consumption feasibility criteria are checked. In addition, whenever a hyperedge consumes an ear we add a subtree to the parse tree of the hypergraph.

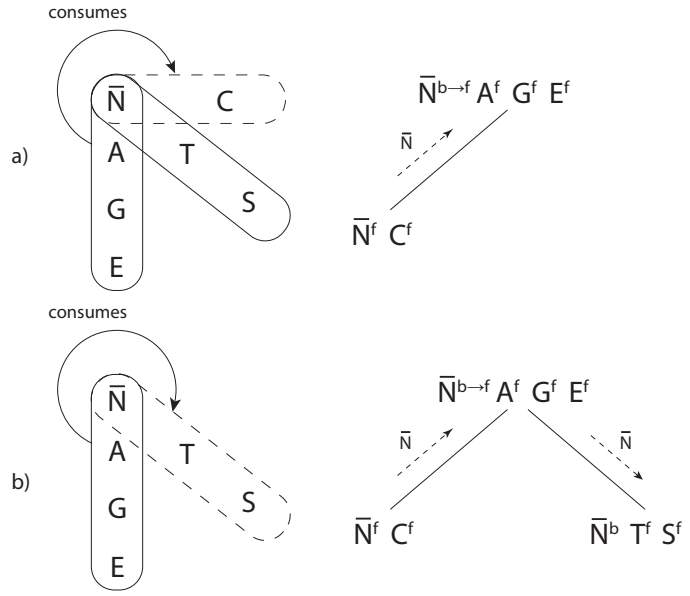


Figure 4.7: Hypergraph reduction and parse tree construction

The consumer $\bar{N}AGE$ becomes parent of the ear $\bar{N}C$, as shown in Figure 4.7 a). The reduction process proceeds in this manner until a single hyperedge remains, and a full parse tree is built, as shown in Figure 4.7 b) after $\bar{N}AGE$ also consumes $\bar{N}TS$.

4.2.3 Join workflow generation by parse tree traversal

Once the BP-GYO reduction has been performed and the parse tree has been generated, we can construct the join workflow and then the full query workflow. As noted in the previous subsection, the parse tree denotes which hyperedges consumed other hyperedges, which also denotes which data operations can be joined with other data operations. If we ignore binding patterns, every bottom-up order of the tree nodes (representing data operations) defines a valid join order. In [Yan81] it is noted that a conventional GYO reduction join order also guarantees desirable properties such as the polynomial increase of intermediate results. Since binding patterns only restrict the way data is accessed, effectively reducing the set of valid plans, we consider that these properties still hold in BP-GYO.

In our approach, we first generate an operator expression, to which then we can apply the construction rules presented in Section 4.1.3. To generate the operator expression we perform a special traversal on the parse tree generated by the BP-GYO reduction. During the traversal we use the information stored in the parse tree nodes, which is simply the names of the data operations they represent. We create new nodes that contain join expressions involving the data operations, however, no new tree structure is built, only an expression. The algorithm is presented in Listing 4.3.

Listing 4.3: Join expression generation from parse tree

```
createJoinExp(Node node) {
    if( node.children.size = 0 )
        return node
    else {
        Queue q := createQueue()
        foreach( Node child in node.children )
            add(q, child)
        while( q.size > 0 ) {
            Node leftNode := node
            Node rightNode := createJoinExp(dequeue(q))
            Node joinedNode := createJoinNode(leftNode.expr, rightNode.expr)
            node := joinedNode
        }
        return node
    }
}
```

To apply the algorithm we invoke the $createJoinExp$ function on the root of the parse tree, its behavior can be described as follows

- If the tree consists of a single node, then the result corresponds to that single node (and thus its associated expression).
- If the node has children, these are stored in a queue in order to be processed from first to last as follows
  - Considering the node received as a parameter, remove one of its children from the queue and obtain recursively the join tree that corresponds to that node.
  - Join the two nodes creating a new node along with its associated join expression.
  - The procedure continues with the remaining children of the node received as a parameter, however, the result nodes that correspond to those children, and that are also obtained recursively, are merged (i.e. as a single expression node) with the previously generated join node.
  - When all of the children have been processed, a single join node remains, which corresponds to the join expression node of the parse subtree supplied as a parameter.

The function $createJoinNode$ receives two subexpressions and creates a new node with an expression of the form $'(leftExpr \bowtie rightExpr)'$. It is important to note that the children are processed always from first to last, which guarantees that we are respecting the dependencies that were followed for the generation of the parse tree during the BP-GYO reduction process. Recall that when a consumer

hyperedge consumes an ear, the binding patterns of the consumer are all set free, which enables that hyperedge to consume other ears.

When the algorithm is applied to the parse tree in Figure 4.7, we obtain the following join expression

$$((\bar{N}C \bowtie \bar{N}AGE) \bowtie \bar{N}TS)$$

Once a join expression has been obtained, we can use it to generate the query workflow. We perform this task in two basic steps

- Transform the join expression into a join workflow
- Add the remaining operators to the join workflow to complete the query workflow.

Transformation of the join expression into a workflow

The construction rules of Subsection 4.1.3 can be applied directly to the join expression to generate the join workflow, provided an adequate parsing mechanism exists. However, this approach would be unnecessarily complex, given that we deal only with *infix* expressions involving joins. In an alternative method to apply these rules, we first transform the infix expression generated by the algorithm in Listing 4.3 into a *postfix* expression, which is more amenable to machine processing. For this purpose we use a general algorithm to transform infix expressions into postfix expressions presented in [Knu97].

For instance, the postfix expression corresponding to the infix expression generated from the parse tree in Figure 4.7, is as follows

$$\bar{N}C\ \bar{N}AGE \bowtie \bar{N}TS \bowtie$$

Then, the well-known algorithm to evaluate postfix expressions can be modified to generate a join workflow based on our construction rules, which is the purpose of the algorithm in Listing 4.4.

Listing 4.4: Generation of a join workflow from a postfix join expression

```
createJoinWF(Expr postfixExp) {
    Stack stack := createStack()
    while( s := nextToken(postfixExp) != nil ) {
        if( isOperand(s) )
            push(stack, createVertex(s))
        else if ( isOperator(s) ) {
            Vertex v1 := pop(stack)
            Vertex v2 := pop(stack)
            Vertex v' := createJoinSubWF(v1, v2)
            push(stack, v')
```

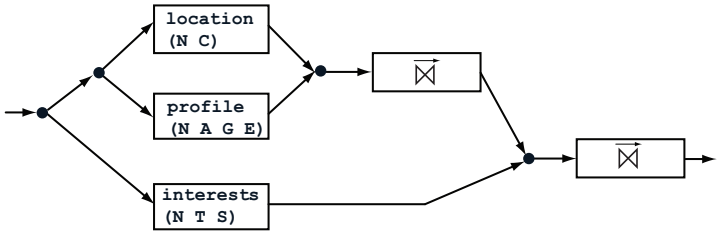


Figure 4.8: Join workflow for the parse tree in Figure 4.7

```
        }
    }
    return pop(stack)
}
```

The function $createJoinSubWF$ will create a sub-workflow from the two vertices it is provided, which represent either data operations or subworkflows, this according to the rules in Definition 4.2. Furthermore, taking into consideration if the join in question is an equi-join or a bind-join.

In the case of the postfix expression above resulting from the parse tree of Figure 4.7 we obtain the join workflow depicted in Figure 4.8. It consists of a bind-join between the `location` and `profile` data operations, and an additional bind-join between the result of that subworkflow and the `interests` data operation.

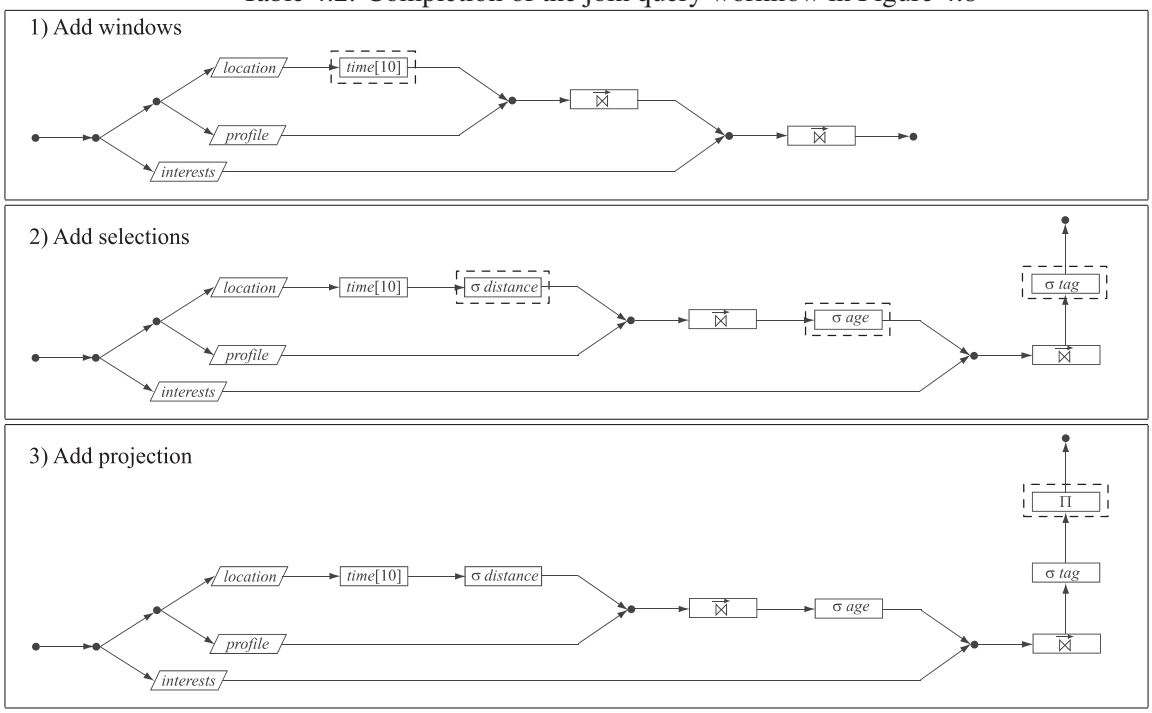
Addition of operators to the join workflow to complete the query workflow

To generate a complete query workflow we only need to add to the join workflow the rest of the operators required by the original HSQL query. We present next the guidelines and heuristics for each of these operators.

- Time and tuple based windows are placed next in sequence to data stream operations, if they were specified in the `FROM` clause of the query.
- Selections are *pushed-down*, i.e. placed close in sequence to the data operations that produce the required data. Special consideration is required for selections that include a function invocation, and therefore involve possibly multiple attributes from multiple data operations, for instance, the selection based on the distance predicate in Example 4.15.
- A single projection operation is added at the end of the query workflow. Alternatively, several projections can be added and pushed-down to reduce the size of intermediate results.

By adding the window, selection, and projection operators to the join workflow in Figure 4.8 we obtain the query workflow presented in Figure 4.4, this process is depicted in Table 4.2.

Table 4.2: Completion of the join query workflow in Figure 4.8



4.3 Evaluation of a query workflow as a service coordination

We have shown how to generate a query workflow from a given declarative hybrid query in HSQL; such query workflow comprises data operations that produce data and query operators that process them. In our approach, we view the query workflow as a service coordination whose enactment will enable us to obtain the query result, hence we refer to it as *query coordination*. Concretely, the data operations enable access to data produced by data services, while computation services can perform the tasks required from query operators, i.e. processing the data to generate the query result.

Evaluating a hybrid query by its corresponding query coordination thus involves: (i) service provisioning, i.e., acquiring the appropriate data and computation services; (ii) utilizing the acquired data services to obtain data and computation services to evaluate individual query operators; (ii) enabling the services communication and interoperation in order to execute the full query coordination, thus obtaining the query result. We deal with all of these aspects and their implications next.

4.3.1 Service provisioning

In Section 3.1.2 we described how data service instances can be accessed by a $registry$ and a complementary $choose$ procedure. We extend the application of these concepts to computation services as well, which enable the evaluation of query operators.

First recall that a given data access activity in a query workflow accesses only a single data operation from a given data service instance, which may be an on-demand data operation or a subscription oper-

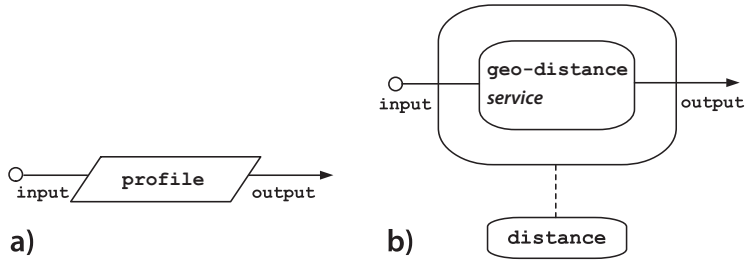


Figure 4.9: a) Data service and b) Simple computation service

ation in case of stream data services. Thus data services can be visualized as in Figure 4.9 a), in which a data operation from the service instance is invoked with a particular input to produce output complex values, the example shows particularly the *profile* data operation of Example 4.17.

Computation services can also be accessed by the $registry$ and $choice$ functions, as their interfaces take the same form as the interfaces of on-demand data operations, i.e. multiple data operations that receive a series of input parameters and produce complex values. However, a single computation service instance or a single of its operations may not suffice to carry out all of the tasks associated with a query operator. Due to the varying complexity of the different operators, we require what we call simple and composite computation services, which we describe next.

4.3.2 Simple computation services

A computation service is considered *simple* whenever the execution of its corresponding query operator is performed by a single operation invocation. For example, consider the selection relying on the *distance* function in the query workflow in Figure 4.4, which is aimed at considering the users only at a distance no larger than 3 km, as specified in the query in Example 4.15.

Figure 4.9 b) illustrates a `distance` service to evaluate that selection. The `distance` computation service relies on a `geo-distance` service, which computes the geographical distance between two points, for instance, by using Vincenty's formulae[1]. This is achieved by the invocation of its `distance` operation with the appropriate parameters obtained from the input complex value, after which the selection condition is evaluated and if the complex value satisfies it, the value is sent to the output.

4.3.3 Composite computation services

A computation service is considered *composite* if for the execution of its corresponding operator it employs service coordination, i.e., it performs multiple operation invocations, possibly also from different service instances. In our approach, the operation invocations and additional activities required to evaluate the query operator are specified in an *operator workflow* that in turn specifies the composite computation service.

[1] http://en.wikipedia.org/wiki/Vincenty%27s_formulae

Construct	Graphical representation	Notation	Description
Computation activity	A	data_item := new_value [1] data_item := s.op(params)[2]	activity *A* modifies a data item (1), or invokes a service operation (2)
Conditional rule	cond, T, F, A, B	**if** (*cond*) **then** *A* else *B*	execute *A* if *cond* is satisfied, otherwise execute *B*
Parallel composition	A, B	**par** *A B* **endpar**	execute *A* and *B* in parallel, changes are visible until both complete
Sequential composition	A, B	**seq** *A B* **endseq**	execute *A* followed by *B*, changes made by *A* are visible to *B*
Iteration	F, cond, T, A	**iterate**(*cond*) *A*	execute *A* repeatedly if *cond* is satisfied, changes are visible after each iteration
Control state	A	**if** (*cst_item* = *val*) **then** *A*	structure activities and monitor execution via the *cst_item* control state data item

Figure 4.10: Workflow constructs based on the Abstract State Machines (ASM) formalism

The operator workflows specifying composite computation services are based on a workflow model whose constructs are presented in Table 4.10. The semantics of these constructs corresponds to the Abstract State Machines [Gur95] (ASM) formalism, which is presented in Appendix C. Consequently, composite computation services can be specified alternatively as ASMs. We defined a series of construction rules analogous to those in Definition 4.2 that establish the equivalence between ASM specifications and their workflow counterparts, these rules are presented in the next section as part of our formalization of the operator workflow model.

We will first describe informally the operator workflow specification of composite computation services. It is important to note that having the means to visualize a service coordination can significantly facilitate its understanding for developers, bringing important advantages in debugging, evolution, and reutilization. Our workflow representation can be applied at varying levels of abstraction, and it can also be incorporated into development tools, a task which however, is beyond the scope of this work.

An operator workflow involves computation activities that perform service operation invocations and reads and updates on local data items. Such data items are visible only to the operator workflow activities during the workflow execution, their main purpose is to maintain relevant data to guide and enable the evaluation of the operator. We remark that operator workflows are executed repeatedly, based on the recently arriving input as well as on the current state of the data items. In addition, special data items called control states are used, which enable to determine by logical conditions the activities to perform during each execution step.

As an example of a composite computation service let us consider a service to evaluate an equi-join operator. Figure 4.11 gives an overview of a composite computation service evaluating the equi-join operator based on the symmetric hash-join algorithm [WA91]. This algorithm essentially builds two hash indexes simultaneously, one for each input, and probes them with complex values from the

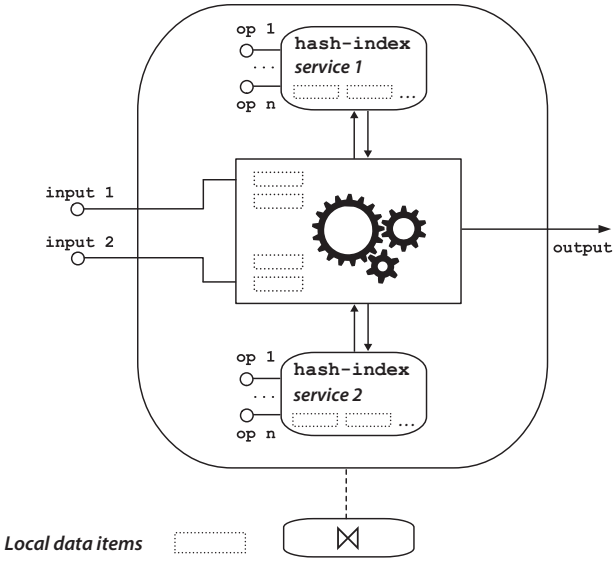


Figure 4.11: Composite computation service

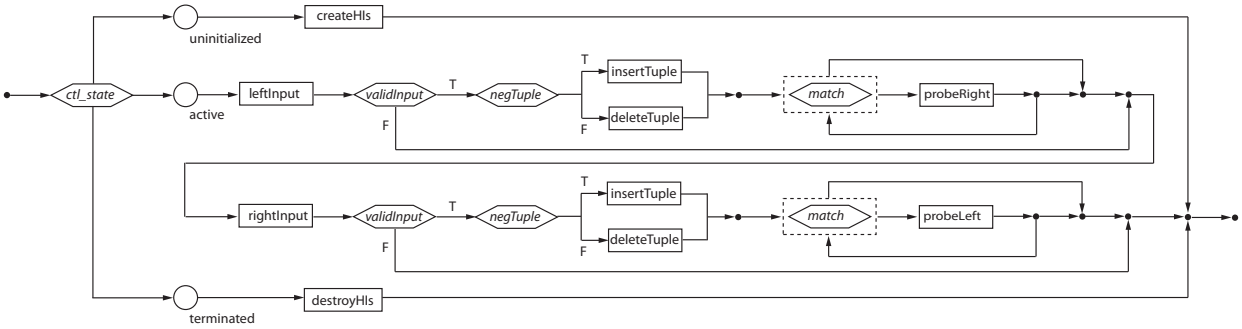


Figure 4.12: Workflow of the join service

complementary input, thus avoiding blocking.

As Figure 4.11 depicts, the `join` service performs multiple internal activities; invoking operations of the `hash-index` services as well as accessing and modifying local data items. This is because several interrelated operation invocations, on both service instances, are necessary to find the tuple matches that form part of the join result. Also note that the `hash-index` service is *stateful*, i.e., state data are maintained by the service across operation invocations. The complete operator workflow specification of the `join` composite computation service (albeit in a simplified form) is shown in Figure 4.12.

4.3.4 Implemented composite computation services

We present next the implementation of the two composite computation services, in addition to the equi-join computation service presented as an example above, that we implemented to validate our approach.

Concretely, these composite computation services enable the evaluation of the tuple and time-based window operators. Their relative simplicity enables us to provide their full ASM and operator workflow specifications.

Listing 4.5: ASM specification for the tuple-based window

```
if( ctl_state = 'active')
  seq
     inTuple := readTuple()
     if(inTuple = nil)
        skip
     else
        seq
           output(inTuple)
           q.enqueue(inTuple)
           count := count + 1
           if(count = range + 1)
              seq
                 outTuple := q.dequeue()
                 outTuple.sign = -1
                 count := count - 1
                 output(outTuple)
              endseq
        endseq
  endseq
```

Tuple-based window operator

Listing 4.5 presents the ASM specification for the composite computation service evaluating the tuple-based window operator. Again, the formalism is described in Appendix C, however the algorithm is for the most part understandable as pseudocode. Essentially, the algorithm stores the input tuples in a *queue service*, keeping track of the number of tuples stored in respect to the size of the window. When it is necessary to evict tuples from the window, this is performed by assigning a negative value to the *sign* attribute of the evicted tuple. The corresponding operator workflow is depicted in Figure 4.13.

Time-based window operator

The ASM specification for the time-based window composite computation service is presented in Listing 4.6. This operator uses a *calendar queue service*, an advanced variant of the priority queue which stores data items by ranges in their priority values, which in this case corresponds to the timestamp value of the tuples. This service enables to examine data elements without removing them with the *peekFirst*

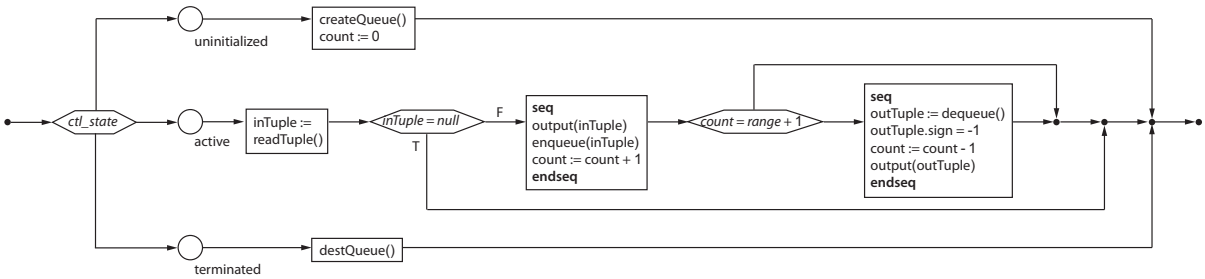


Figure 4.13: Tuple-based window composite service operator workflow

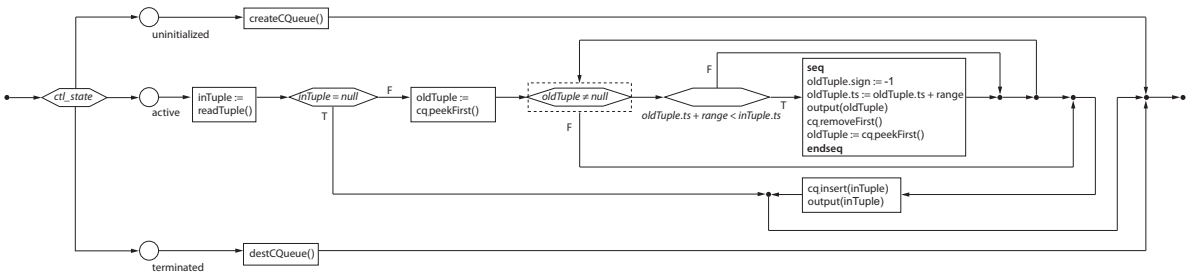


Figure 4.14: Time-based window composite service operator workflow

operation. The algorithm implements the formal characterization of tuple-based windows presented in Definition 3.2.1. Again, evicted tuples are given a negative sign attribute. The corresponding operator workflow is depicted in Figure 4.14.

Listing 4.6: ASM specification for the time-based window

```
if( ctl_state = 'active')
  seq
     inTuple := readTuple()
     if(inTuple = nil)
        skip
     else
        seq
           oldTuple := cq.peekFirst()
           iterate(oldTuple != nil)
              if(oldTuple.ts + range < inTuple.ts)
                 seq
                    oldTuple.sign := -1
                    oldTuple.ts := oldTuple.ts + range
                    output(oldTuple)
                    cq.removeFirst()
```

```
                    oldTuple := cq.peekFirst()
                endseq
            pq.enqueue(inTuple)
            output(inTuple)
        endseq
    endseq
```

Additional composite computation services can be specified to evaluate the rest of the operators in our data model, as well as potentially others that currently do not form part of it. What is required is the appropriate base computation services and an understanding of the operator algorithms. Our experience, as well as that of ASMs research in general, suggests that the operator algorithms can then be specified succinctly and clearly in ASM or operator workflow form. Next we discuss important benefits of this approach.

4.3.5 Flexibility of computation services

Our approach provides several advantages among which we remark flexibility, a property intensely pursued in service-oriented applications research (e.g. [HEOP08]). This property is highly desirable when evaluating hybrid queries over dynamic and mobile data services, which are subject to intermittent availability.

Our query evaluation approach based on service coordination is geared towards flexibility. First, the use of service coordination offers the capability to dynamically acquire resources by late binding (i.e., deferring service location and binding until execution); the best services available can therefore be bound and then used at execution time. As the query and its evaluation may be continuous, access to services that are subject to frequent disconnections is an important issue to consider. Our approach can be extended by enabling the replacement of services whenever such failures arise.

In our approach, if some data or computation service becomes unavailable, the evaluation of the query can still proceed by discovering another service offering the same (or similar) data or functionality. For instance, if the `geo-distance` service discussed previously becomes unavailable, we can rely on an alternative service. Concretely, a service offering a spatial index based on an R-Tree could be used instead. In this case we can determine the tuples that satisfy the spatial condition by creating an index over them.

In addition, we can define new functions that can be incorporated into query operators. In [CVVSC+10] we present a detailed example consisting of the definition of an alternative notion of distance between users to that of Example 1.1. This alternative distance is represented by the similarity between the interests of the users as denoted by their tags, instead as of the geographic distance between their locations. The use of computation services incorporating functionality associated with information retrieval systems would enable the evaluation of selections using this new type of distance function.

If appropriate computation services are not available for a particular operator, such computation services can be developed from scratch and integrated into the service coordination generation process. This

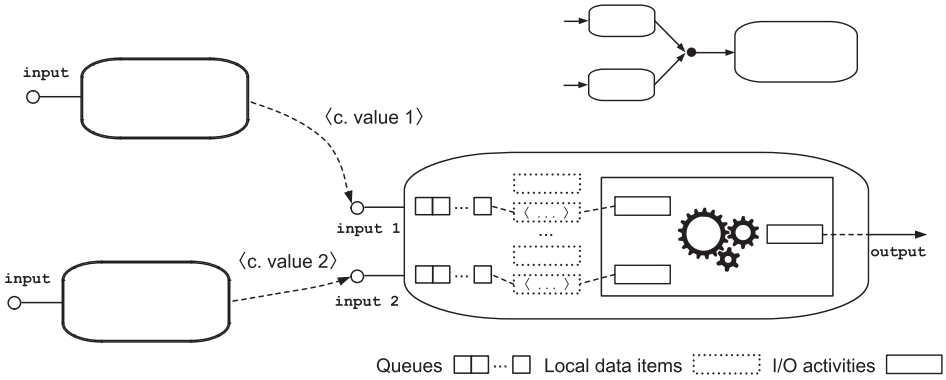


Figure 4.15: Service interoperation

task can still be greatly facilitated with the appropriate *blueprints*, an example being the implementation of the recursive operators of our data model that we will present in Section 5.4.5.

4.3.6 Service interoperation and communication

We now describe how the various data and computation services that comprise a query coordination interoperate and communicate with each other to generate the query result.

The data produced by stream data services is accessed by special stream access services that implement asynchronous *input operations*, which are exposed by their executing environment. Our particular implementation of this mechanism is detailed in Section 5.2.2.2. The data produced by on-demand data services, on the other hand, is accessed by synchronous calls to their specific data operations. The computation services implementing operators such as the bind-join perform such operation calls.

Computation services implementing whole operators communicate with one another asynchronously through input operations. In order to take advantage of this mechanism, computation services must be implemented in such a way that they access the input data sent to them by their preceding computation service(s), and that they then send the output to their successor computation service.

When a service invokes the input operation of its successor, it will in turn store the input complex value(s) in a queue forming part of its state, in order to process it accordingly. This behavior is predefined for simple computation services. In the case of composite computation services, input and output activities will retrieve and send the complex values, which can be associated with local data items. For instance, `inTuple := readTuple()` in Listing 4.5 is an input activity that associates the input with the `inTuple` data item. Figure 4.15 depicts this process.

In this manner, a query coordination can support the data communication patterns (see Section 2.1.3) associated with its query. These communication patterns apply to each successive pair of services in the query coordination, and serve to characterize the query evaluation process.

For instance, the communication patterns for the query in Example 1.1 are presented in Figure 4.16. The location stream data service follows an Sc-Sc (Static continuous - Static continuous) communication

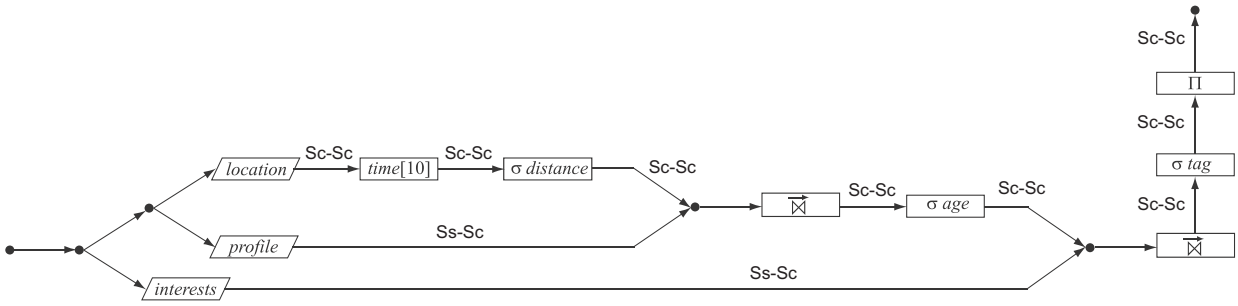


Figure 4.16: Communication patterns for the query workflow graph for the query in Example 1.1

pattern with its successor time-based window operator. The presence of this data stream results in mostly Sc-Sc patterns between operators, since evaluation must be continuous. An exception are the on-demand data services with their respective bind-joins, which follow an Ss-Sc (Static snapshot - Static continuous) pattern.

4.4 Composite computation services workflow model

We introduce in full the operator workflow model by its correspondence to service coordination ASMs, which are detailed in Appendix C and that we employ in particular to specify composite computation services to evaluate query operators, as described in the previous section.

We focus our description at the activity and controlflow level, since developers mostly work at this cognitive level, while dataflow follows as a result. Our workflow model is again formalized by graphs as presented in Definition 4.2. First, we will introduce the graphs for each of the ASM constructs. Subsequently, we will present the construction rules that link the ASM syntax to the workflow model. Thereby we obtain a formal characterization that can serve to study the properties of query coordinations and possibly for their optimization.

4.4.1 Workflow model constructs

We present next the workflow graph representation for each of the ASM constructs used in our service coordination model for composite computation services, which are also depicted in simplified form in Figure 4.10.

Informally in the following descriptions, update rules correspond to computation activities and dynamic functions to local data items. For emphasis, vertices are shown with dashed rectangles and circles.

4.4.1.1 Update rules

Updates over dynamic functions, specified as $f := g(t_1, ..., t_n)$, are represented by a block vertex containing the textual specification. The block is adjacent to start and end vertices, as Figure 4.17 shows.

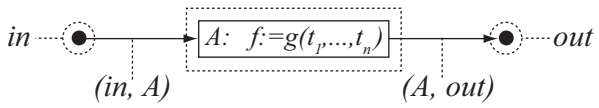


Figure 4.17: Update rule workflow graph

4.4.1.2 Parallel composition

The parallel composition of n rules specified as **par** R_1 ... R_n **endpar** is represented by branches in a graph that originate at a start vertex and are rejoined at an end vertex, as depicted in Figure 4.18.

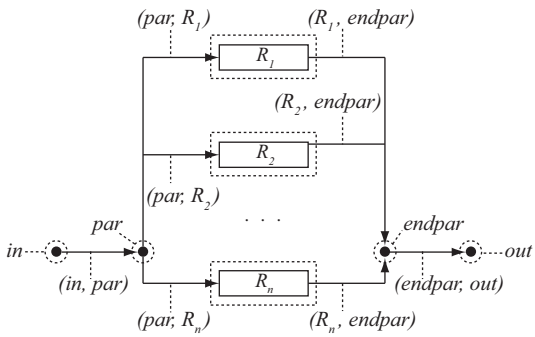


Figure 4.18: Parallel composition rule workflow graph

4.4.1.3 Sequential composition

The sequential composition of n rules specified as **seq** R_1 R_2 ... R_n **endseq** is represented by successive edges between the rule blocks, as seen in Figure 4.19.

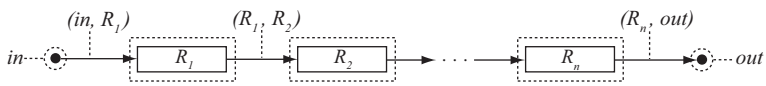


Figure 4.19: Sequential composition rule workflow graph

4.4.1.4 Conditional rule

A conditional rule of the form **if** $cond$ **then** R **else** S is represented by a branching graph consisting of a hexagon shaped vertex containing the condition and two branches with labels T and F, leading to the rules that should be executed depending on whether the condition was satisfied or not. This type of rule is depicted in Figure 4.20, where the condition is identified by a label c.

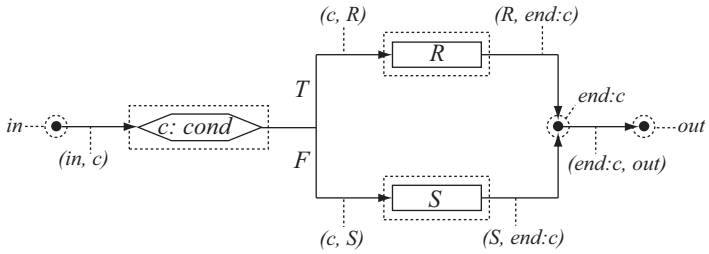


Figure 4.20: Conditional rule workflow graph

4.4.1.5 Iteration rule

An iteration rule of the form **iterate** $(cond)$ R is represented by a condition vertex linked to the R rule vertex, which in turn is linked to a vertex ending the condition. The condition vertex is represented by a dashed hexagon in order to distinguish it from an ordinary condition. The T edge of the iteration condition leads to the R rule vertex, while an additional edge links the end of the condition vertex back to the condition, thus establishing an iteration. On the other hand, the F edge leads directly to the end of the iteration rule vertex, as illustrated in Figure 4.21.

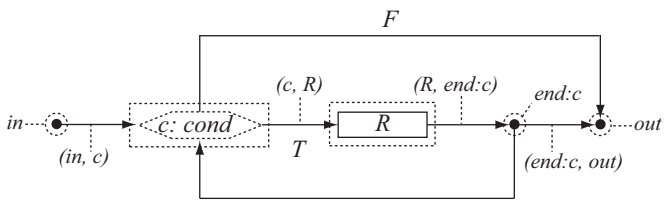


Figure 4.21: Iteration rule workflow graph

4.4.1.6 Control states

Dynamic functions can be used in combination with conditional rules to structure an ASM into several clearly defined tasks. These dynamic functions, referred to as *control states*, are specified in conditional rules of the form:

if *(ctl_state = 1)* **then**
  $rule_1$
  ctl_st $:= s_1$
if *(ctl_state = 2)* **then**
  $rule_2$
  ctl_st $:= s_2$
  . . .
if *(ctl_state = n)* **then**
  $rule_n$

$ctl_st := s_n$

The dynamic function *ctl_st* is used to determine which rules to execute depending on its current value. Note that the value of *ctl_st* is subject to change in the nested rules, thus resulting in new control states and thus different actions in subsequent executions.

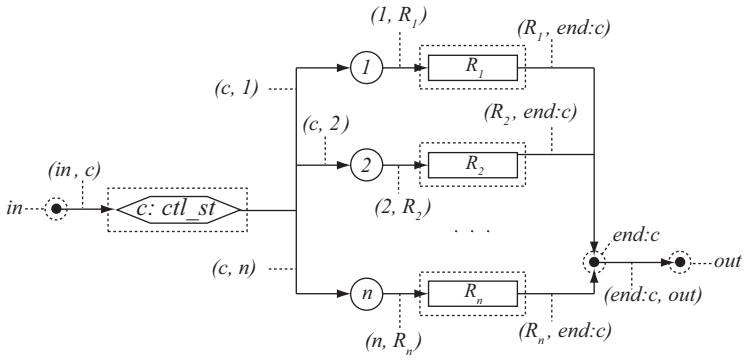


Figure 4.22: Control state workflow graph

These rules can be given a workflow graph representation to help visualize the operation of a service coordination. Figure 4.22 presents such workflow graph representation. A condition vertex is linked to multiple control state vertices represented by circles, which contain the value for which a particular action is to be executed. Each control state vertex leads to rules to execute, which then are joined on the end vertex of the condition.

4.4.1.7 Abstraction of rules

To facilitate the visualization of a service coordination workflow rules can be expressed in abstract form. Regardless of the type of a particular rule (conditional, iteration, parallel composition, etc.), it can be represented by a block containing the rule either in *extensional* of *symbolic* form, as depicted in Figure 4.23. The extensional representation corresponds to the full rule specification, while in the symbolic representation the rule is replaced by a particular symbol, like, R in the figure.

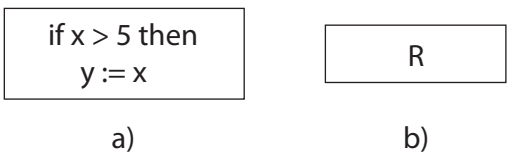


Figure 4.23: a) Extensional and b) symbolic representation of rules

4.4.2 Workflow graph construction rules

We now present the construction rules that enable to generate a workflow graph from a given abstract state machine (ASM), which in turn consists of the composition of rules from the various types presented previously. These construction rules, being analogous to the rules presented in Definition 4.2 to generate query workflow graphs, can be used to generate composite computation services workflows. In this case, a parsing mechanism was developed so that these construction rules can be applied, the process is described in Section 5.5.2.

In the following definition we make the assumption that the vertices representing rules and conditions have annotations that make those vertices unique, whereas the par and $endpar$ vertices of parallel compositions are made unique using labels l that can be generated automatically.

Definition 4.7. [**ASM workflow graph**] For an ASM M, the function *asmwfg* specifies the *ASM workflow graph* such that *asmwfg*$(M) = (V, E, in, out)$, as follows. We use R to represent rules of different types.

Update rule:

$$\text{if } R = \ f := g(t_1, ..., t_n) \text{ then } \begin{cases} V = \{R, in, out\} \\ E = \{(in, R), (R, out)\} \\ in, out \text{ are new vertices} \end{cases}$$

Parallel composition:

$$\text{if } R = \ \textbf{par} R_1 ... R_n \textbf{endpar} \text{ then } \begin{cases} \text{Let } asmwfg(R_i) = (V_i, E_i, in_i, out_i), \text{ for all } i \in [1..n] \\ V = (V_1 - \{out_1\}) \cup \bigcup_{j=2}^{n-1} (V_j - (\{in_j, out_j\})) \\ \cup (V_n - \{in_n\}) \cup \{par_l, endpar_l\} \\ E = \{(in_1, par_l), (endpar_l, out_n)\} \\ \cup \bigcup_{j=1}^{n} E_j - \{(in_j, v_j), (v'_j, out_j) | v_j, v'_j \in V_j\} \\ \cup \bigcup_{j=1}^{n} \{(par_l, v_j) | (in_j, v_j) \in E_j\} \\ \cup \bigcup_{j=1}^{n} \{(v'_j, endpar_l) | (v'_j, out_j) \in E_j\} \\ par_l, endpar_l \text{ are new vertices defined with the label } l \\ \text{also } in = in_1 \text{ and } out = out_2; \end{cases}$$

Sequential composition:

$$\text{if } R = \textbf{seq} R_1 ... R_n \textbf{endseq} \text{ then } \begin{cases} \text{Let } \textit{asmwfg}(R_i) = (V_i, E_i, in_i, out_i), \text{ for all } i \in [1..n] \\ V = (V_1 - \{out_1\}) \cup \bigcup_{j=2}^{n-1} (V_j - (\{in_j, out_j\})) \\ \cup (V_n - \{in_n\}) \\ E = \{(in_1, v_1) | (in_1, v_1) \in E_1\} \cup \{(v_n, out_n) | (v_n, out_n) \in E_n\} \\ \cup \bigcup_{j=1}^{n} E_j - \{(in_j, v_j), (v'_j, out_j) | v_j, v'_j \in V_j\} \\ \cup \bigcup_{j=1}^{n-1} \{(v_j, v'_{j+1}) | (v_j, out_j) \in E_j, (in_{j+1}, v'_{j+1}) \in E_{j+1}\} \\ \text{where } in = in_1 \text{ and } out = out_n; \end{cases}$$

Conditional rule:

$$\text{if } R = \textbf{if } cond \textbf{ then } S_1 \textbf{ else } S_2 \text{ then } \begin{cases} \text{Let } \textit{asmwfg}(S_1) = (V_1, E_1, in_1, out_1) \text{ and} \\ \textit{asmwfg}(S_2) = (V_2, E_2, in_2, out_2) \\ V = (V_1 - \{in_1, out_1\}) \cup (V_2 - \{in_2, out_2\}) \cup \{cond_c, endcond_c\} \\ E = \{(in, cond_c)\} \cup \{(endcond_c, out)\} \\ \cup \{(cond_c, v_1) | (in_1, v_1) \in E_1\} \\ \cup \{(v'_1, endcond_c) | (v'_1, out_1) \in E_1\} \\ \cup \{(cond_c, v_2) | (in_2, v_2) \in E_2\} \\ \cup \{(v'_2, endcond_c) | (v'_2, out_2) \in E_2\} \\ \cup \bigcup_{j=1}^{n} E_j - \{(in_j, v_j), (v'_j, out_j) | v_j, v'_j \in V_j\} \\ in, out \text{ are new vertices} \\ \text{also } cond_c \text{ and } endcond_c \text{ are new vertices defined with the label } c \end{cases}$$

Iteration rule:

$$\text{if } R = \textbf{iterate } (cond)\ R \begin{cases} \text{Let } asmwfg(R) = (V_R, E_R, in_R, out_R) \\ V = V_R \cup \{cond_c, endcond_c\} \\ E = \{(in, cond_c)\} \cup \{(endcond_c, out)\} \\ \cup\{(cond_c, v_R) | (in_R, v_R) \in E_R\} \\ \cup\{(v'_R, endcond_c) | (v'_R, out_R) \in E_R\} \\ \cup\{(cond_c, out)\} \cup \{(endcond_c, cond_c)\} \\ \cup E_R - \{(in_R, v_R), (v'_R, out_R) | v_R, v'_R \in V_R\} \\ \text{where } in = in_R \text{ and } out = out_R; \\ \text{also } cond_c \text{ and } endcond_c \text{ are new vertices defined with the label } c \end{cases}$$

4.5 Conclusions

In this chapter we first introduced a workflow representation for hybrid queries which is equivalent to the operator expressions described in the previous chapter. A series of construction rules formalize the transformation from one representation to the other. The goal of deriving a workflow representation of a hybrid query is to enable its evaluation via service coordination.

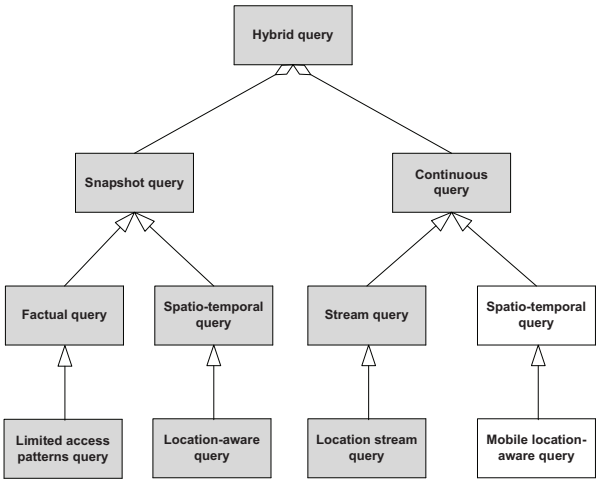


Figure 4.24: Query types of our taxonomy addressed by our approach

Furthermore, we addressed two major issues in this chapter. First, we bridged the gap between the declarative level of HSQL queries and our operator expression characterization of hybrid queries. Concretely, with the use of our extended GYO reduction algorithm, BP-GYO, we are able to generate query workflows for conjunctive queries over on-demand and stream data services.

Second, we showed how the query operators present in query workflows can be associated to compu-

tation services that enable the evaluation of a query as a service coordination. We enable the utilization of two types of computation services, simple and composite. Simple computation services evaluate a query operator by a single operation invocation. On the other hand, composite computation services can make use of multiple service instances and a more complex logic, representing therefore a service coordination by themselves. The creation of composite computation services is supported by a workflow service coordination model based on the ASM formalism. A workflow can be generated for a given ASM by means of our graph-based representation and construction rules.

Our approach enables the evaluation of queries belonging to an important subset of hybrid queries as defined by our taxonomy. Concretely, the types of queries highlighted in gray in Figure 4.24. By this we do not mean the entirety of those queries or the existing techniques for their evaluation. In a service-based environment the viability to evaluate a particular query depends on the data and computation services available. We also can conclude that the possible mobility of data producers and consumers remains a major challenge.

In this manner, we are able to evaluate a hybrid query expressed in a declarative language such as HSQL by the coordination of data and computation services. Additionally, we provide a means to construct new computation services via composition. This is a significant advantage when it is desired to utilize services to reduce development cost, increase the available resources, and evaluate new types of queries.

CHAPTER 5

HYPATIA: a service-based hybrid query processor

This chapter presents HYPATIA, a system that implements and validates the various aspects described in the previous chapters. First, we present an overview of our system in Section 5.1. Next, we show how it implements complex values and data services in Section 5.2. The implementation of the construction of hybrid query workflows from HSQL queries, and its associated algorithms, is described in Section 5.3. Then Section 5.4 presents the implementation of the execution mechanism for hybrid query workflows, including query operator execution via computation services and the coordination model that we employ to construct composite computation services. In Section 5.5 we describe the graphical user interface of HYPATIA and its visualization capabilities. Afterwards, we introduce in Section 5.6 two testbeds that enabled us to validate our approach and then discuss the experimental results. Finally, we present our conclusions in Section 5.7.

5.1 Overview of HYPATIA

We now describe how the various aspects developed so far are joined together to derive a fully-functional hybrid query processor based on services that we name HYPATIA.

We adopt the *Views and Beyond* methodology [GBI+10] to describe the architecture of HYPATIA; according to which an architecture is described in terms of multiple views and how they are interrelated. The views include: (i) *Modular view*, describing the principal implementation units of a system and the relations among them. (ii) *Runtime view*, showing the runtime elements and how they are interconnected to accomplish tasks. (iii) *Deployment view*, specifying how the software artifacts are organized and supported in the hardware. We focus on views (i) and (ii) whereas view (iii) is presented in Appendix D describing also the installation of HYPATIA.

Figure 5.1 presents the modular view of HYPATIA through the class structure of the system, which is implemented with the Java Platform. The class that controls the execution of a query is `HybridQP`, relying on the `QEPBuilder` to build the query workflow. In turn, `QEPBuilder` depends on the `HSQLParser` to parse the query and on `GYOReduction` to generate the join workflow. To generate the

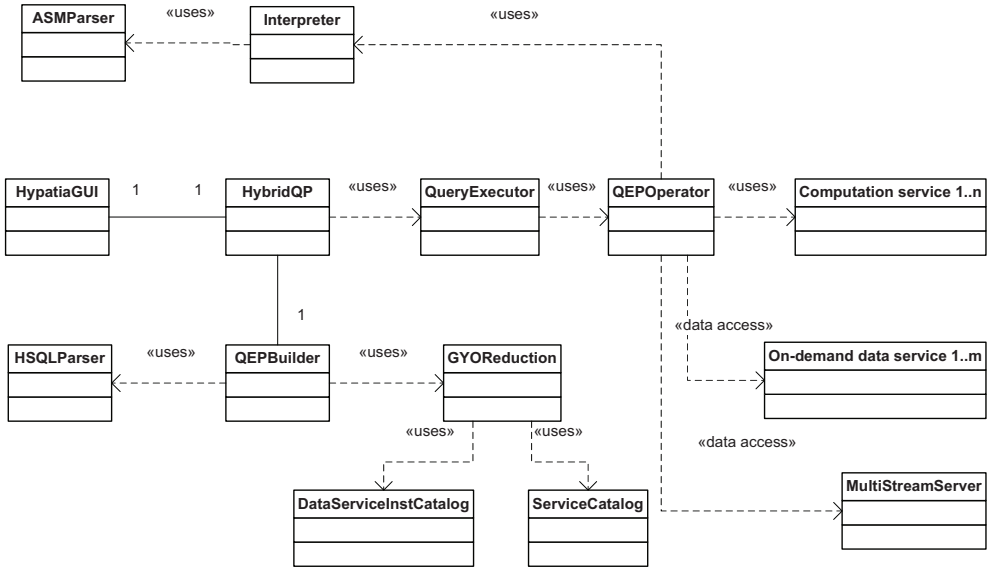


Figure 5.1: Overview of the class structure of HYPATIA

join workflow the `GYOReduction` class also needs information about the data service interfaces and the addresses of their instances, managed by the `ServiceCatalog` and `DataServiceInstCatalog`, respectively.

The `HybridQP` class then employs a `QueryExecutor` object to execute the generated query workflow. The `QueryExecutor`, in turn, will manage the execution of the various query operators, all subclasses of `QEPOperator`. Some of the operators represent composite computation services and thus will employ an ASM `Interpreter`, which uses an `ASMParser`. The data and computation services are external but coordinated by HYPATIA. For stream data services in particular, a `MultiStreamServer` is required.

For evaluating a particular query supplied via the graphical user interface (GUI), these classes are instantiated and communicate following the scheme just described and as indicated in Figure 5.2, which represents a runtime view of HYPATIA. It is important to note that three types of operators are instantiated: (i) data access operators to retrieve or receive the data from data services, (ii) computation services to process the data, and (iii) an output operator to send the result back to the user.

Communication between the different subsystems of HYPATIA takes place via Java method calls, while invocation of data and computation services occurs via SOAP on HTTP. In the next sections we describe each of the subsystems and their main components in detail.

5.2 Complex values and data services implementation

We present the implementation of the complex values specified in our data model, followed by on-demand and stream data services.

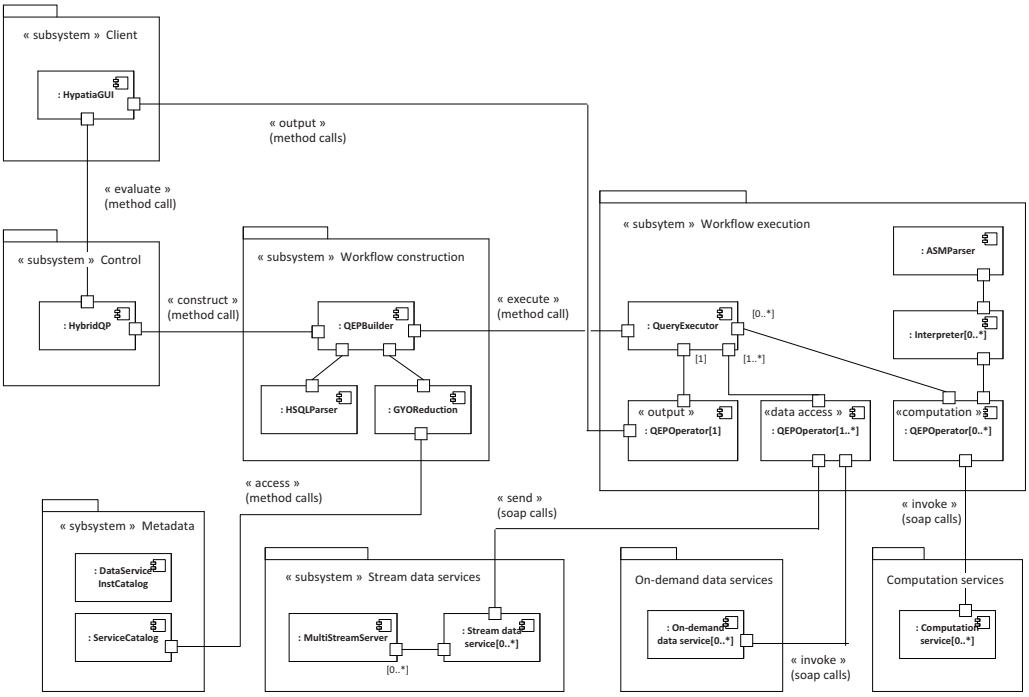


Figure 5.2: Runtime view of HYPATIA

5.2.1 Complex values implementation

Our typed complex values are compatible with the JavaScript Object Notation, commonly referred to as JSON. In principle, objects and arrays in JSON can represent the tuples and sets in our model, respectively. We took advantage of the reference Java implementation of JSON[1] to implement our complex values using this popular data format.

We illustrate how the representation of complex values via JSON takes place by a series of simple rules and an example. First, the syntax of JSON objects and arrays is presented in Figure 5.3[2]. JSON *objects* are represented as pairs of names and values inside curly braces, while *arrays* are lists of values enclosed within square braces. *Values* in both cases can be JSON objects, arrays or primitive values. This enables us to represent complex values in JSON according to the following rules

1. A tuple complex value of the form $A : \langle A_1 : v_1, ..., A_n : v_n \rangle$ is represented by two nested JSON objects of the form `{"A":{"A1":v1,...,"An":vn}}`

2. A set complex value of the form $A : \{A' : v_1, ..., A' : v_n\}$ is represented by a JSON array of JSON objects, nested within another JSON object and having the form `{"A":[{"A1":v1},...,{"An":vn}]}`

[1]http://www.json.org/java/
[2]Images obtained from http://www.json.org/

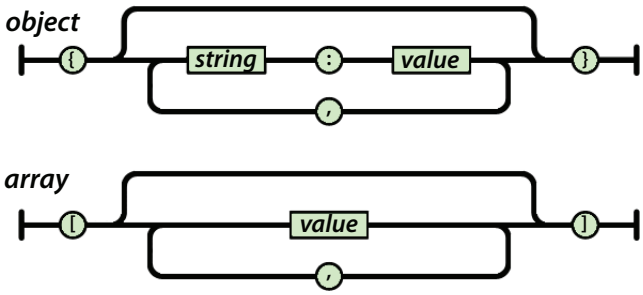


Figure 5.3: Syntax of JSON objects and arrays

Since JSON values do not have an associated name, except for the values enclosed within an object, we employ a wrapping JSON object to specify the name of complex value tuples and sets. Although this may appear somewhat cumbersome for manually writing JSON data, it does not represent a problem for a programming implementation. The following example illustrates how these rules are applied.

Example 5.21.
The complex value

person:⟨ nickname:'Charles', email:'charles@gmail.com', sex:'M', age:40,
interests:{stag:⟨ tag:'art', score:6.5 ⟩, stag:⟨ tag:'sports', score:7.5 ⟩}⟩

can be represented in JSON as

```
{"person":{"nickname":"Charles", "email":"charles@gmail.com",
 "sex":"M", "age":40, "interests":[
     {"stag":{"tag":"art", "score":6.5}},
     {"stag":{"tag":"sports", "score":7.5}} ]}}
```

5.2.2 Data services implementation

For the implementation of both on-demand and streaming data services we adopt the W3C's Web Services standards. Concretely, data service interfaces are specified in WSDL and service requests and responses are transmitted in SOAP via HTTP. For this purpose, we use in particular the Java Web Services Reference Implementation (JAX-WS RI)[3] version 2.1.5.

5.2.2.1 On-demand data services

On-demand data services are represented by the operations of a WSDL interface, which are mapped by the JAX-WS middleware to methods in an annotated[4] Java class referred to as *service bean*. At

[3] http://jax-ws.java.net/
[4] `@WebService` and `@WebMethod` annotations specify the methods in a class that are published as a Web Service.

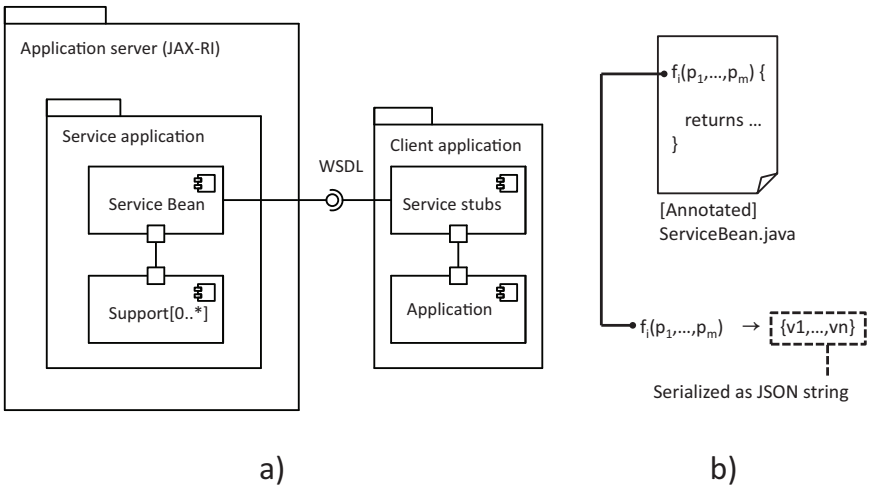


Figure 5.4: a) Runtime view of generic on-demand data service b) On-demand service interface mapping

runtime, as shown in Figure 5.4 a), a service application supports a service bean Java object while it may also employ additional support objects. The service application is hosted in a compliant application server (e.g. Tomcat) or a `javax.xml.ws.Endpoint` object, and it can access resources such as files, databases, or devices. For a client application, access to the service operations requires only the stub classes generated automatically from the service WSDL description.

In this manner, the selected methods of the Java service bean are presented as WSDL operations that correspond to on-demand data operations in our model. A client, in our case a query processor, can then access the data operations by providing the required primitive value input parameters, in order to obtain as output a set of complex values. For convenience, we expect the JSON output data to be serialized as an ordinary string, which we can then use to construct objects with the standard JSON Java library. This process is depicted in Figure 5.4 b).

The service implementation scheme outlined in Figure 5.4 a) can be used to build on-demand data services in a variety of scenarios. In Section 5.6 we present two implemented testbeds. However, in HYPATIA third party services supported by other platforms can be accessed as on-demand data services as well, provided they adhere to the aforementioned syntactic restrictions.

5.2.2.2 Stream data services

The implementation of stream data services requires a different approach than that for on-demand data services. This is because it is necessary to support a subscription mechanism and to provide data continuously and with minimal delay. In addition, for experimental settings, it is desirable to have control over the stream rates and the possible data dependencies among the various data streams. Therefore our aim is to create a stream data services implementation that is compliant with the Web Services standards and that is also configurable and flexible; representing by itself a platform for experimentation on data streams.

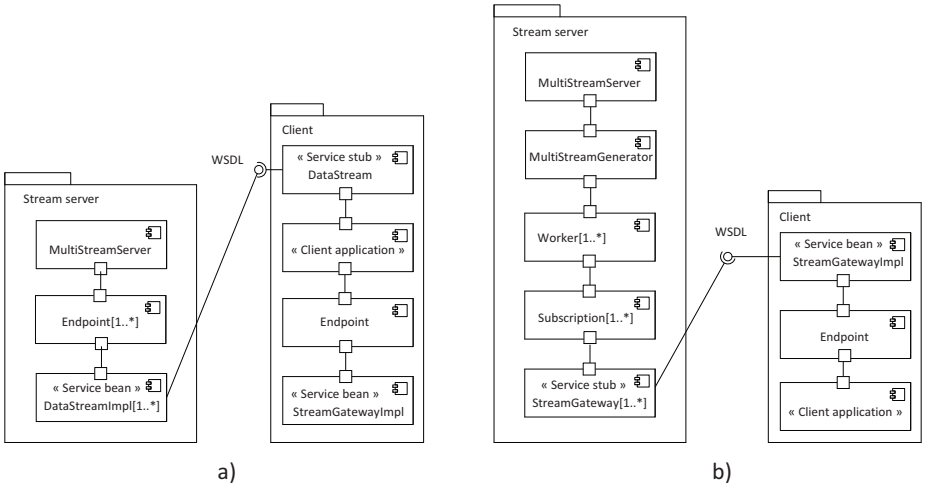


Figure 5.5: Runtime view of our stream server a) during subscription and b) during data delivery

Our implementation consists of two main elements, a `MultiStreamServer` that will continuosly send data corresponding to multiple data streams to the clients, and a `StreamGateway` service that the clients employ to receive the data. The interaction between the stream server and a client, concerning the subscription process, is presented in Figure 5.5 a); while Figure 5.5 b) presents the interaction during the data delivery process. The class structure of our `MultiStreamServer` is presented in Figure 5.6. Next, we discuss in greater detail the subscription and data delivery phases.

Subscription to data streams

For subscription, the server creates a subscription service for each of the published data streams via `DataStreamImpl` objects. The client will first set up a `StreamGatewayImpl` object as a service to receive the data stream and then invoke the subscription service of the server. Both the server and the client use the `javax.xml.ws.Endpoint` class to publish their services.

The concrete actions to take to setup the server and handle the subscriptions are presented in the sequence diagram of Figure 5.7. First, the server reads from a configuration file the data streams that need to be published and their respective addresses; then a subscription service is published for each of the data streams at the corresponding address. When a client subscribes to a data stream it sends the address of the `StreamGateway` service that will receive the data stream. The server is notified of the subscription and a connection is established with the gateway.

A client may require that no data stream sends data until a subscription has been performed on *all* of the data streams accessed by the client, so that all of the data can begin to be processed at unison. For this reason, if more than one data stream is being published the server launches a monitor thread after the first subscription. The monitor thread will check if all of the expected subscriptions have been performed, when that is the case then the stream generator will be started.

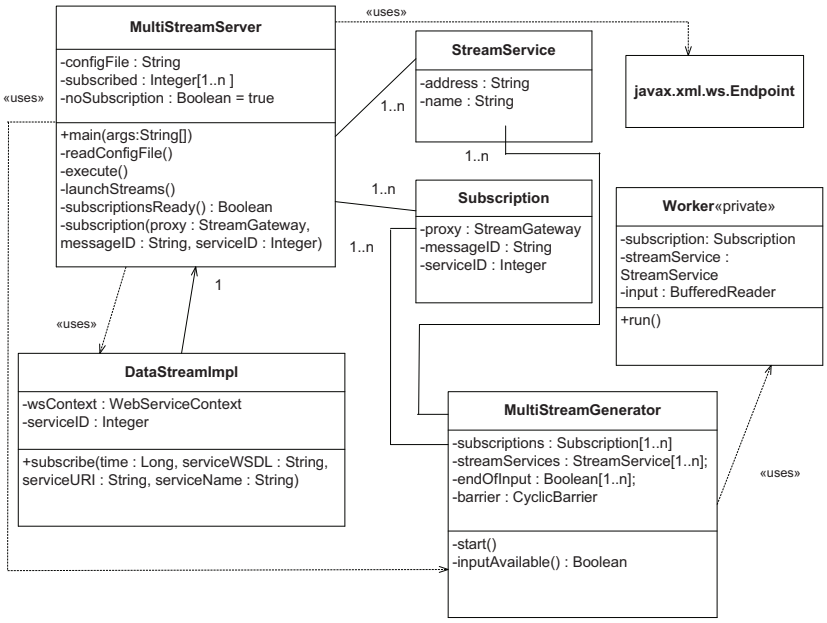


Figure 5.6: Class structure of our stream server

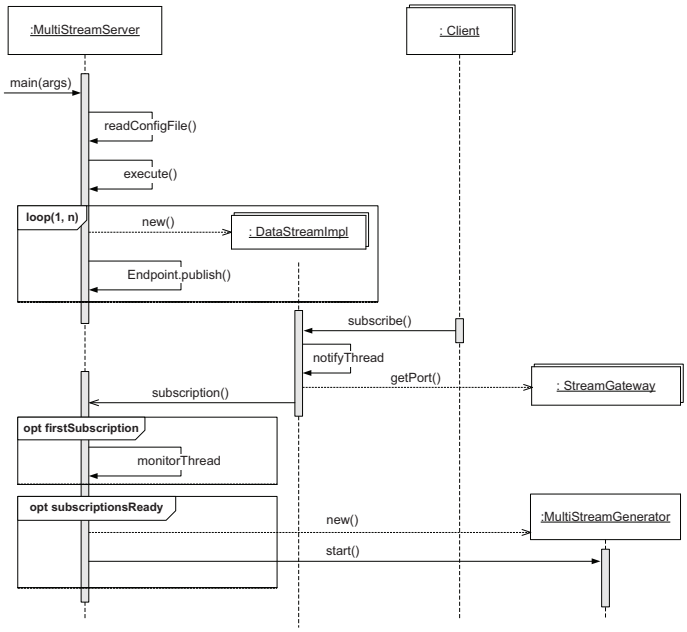


Figure 5.7: Subscription phase of the MultiStreamServer

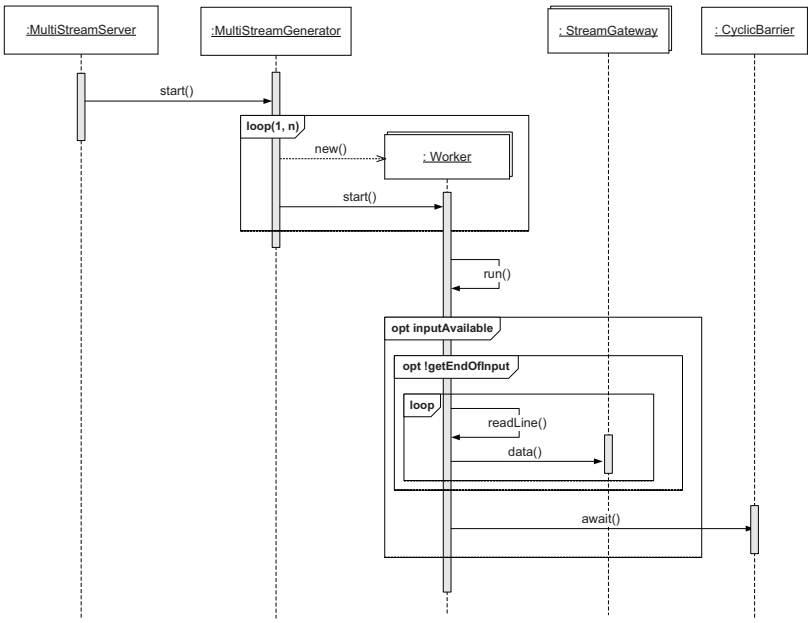


Figure 5.8: Data generation phase of the MultiStreamServer

Delivery of data streams

For data delivery, the main application, `MultiStreamServer`, relies on a `MultiStreamGenerator` to produce the stream data. The generator in turn launches a series of `Worker` threads to send the data. Figure 5.8 shows the activities of the stream server just before and once it has started to produce data. A `Worker` thread is launched for each data stream that is being accessed. The `Worker` thread will read the data from a file and send the data to the respective `StreamGateway` service in the client.

It is important to remark that the stream data stored in the files is organized in *batches*, i.e. subsequences of complex values delimited by a special separator in the file. This is due to several reasons. First, it facilitates compatibility with our stream function formalization of stream data services presented in Section 3.1.4, in which a particular time value is mapped to a set of complex values. In addition, it facilitates the control of data rates and utilization of resources in experimental settings. A simple data stream is a special case in which there is only one batch.

Accordingly, synchronization is enforced between the data streams over the batches; a data stream can only proceed to the next batch when *all* of the data streams have completed the current batch. The outer condition in Figure 5.8 verifies that there is a next batch in all of the data streams. The next inner condition and loop processes each of the complex values in the batch; sending the complex value via the `data` operation of the `StreamGateway`. A certain (configurable) amount of time is allotted for processing each batch, which is spread uniformly among the complex values in the batch. Synchronization is enforced by a `CyclicBarrier` (one of the standard Java concurrency classes) whose `await` method is called by each `Worker` after the completion of a batch.

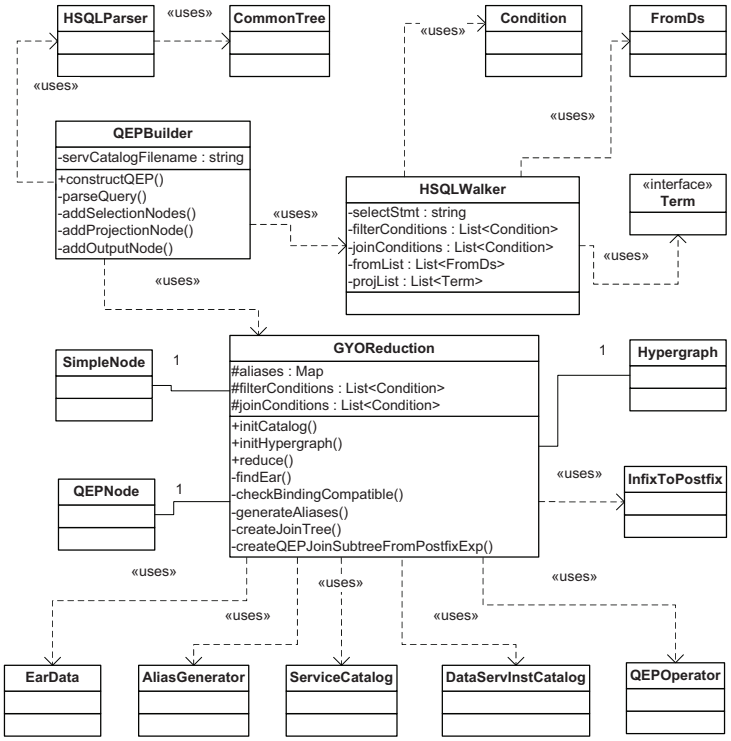


Figure 5.9: Structure of the classes for the generation of query workflows

5.3 Generation of hybrid query workflows from HSQL queries

Next we describe how a given HSQL query is parsed and from a series of intermediary representations it is then transformed into a hybrid query workflow. First we discuss the implementation of our HSQL parser, followed by the implementation of the algorithms to generate hybrid query workflows described in Section 4.2.

5.3.1 Parsing HSQL queries

The HSQL parser of HYPATIA consists of two subparsers, each generated from an ANTLR grammar. The first of the two grammars is presented for reference and in simplified form in Appendix A, since it is useful to understand the language structure.

The two subparsers are depicted in the top of Figure 5.9 along with their associated classes. The first parser, `HSQLParser`, generates given the query string an abstract syntax tree representing the query. The AST consists of nodes belonging to the `CommonTree` class of ANTLR. The second parser, `HSQLWalker`, traverses the generated AST and applies a series of rules that map subtree structures into specialized objects. These specialized objects are designed to represent the different query elements in a form that facilitates the generation of a query workflow.

Concretely, `HSQLWalker` generates two lists of `Condition` objects to represent filtering and join

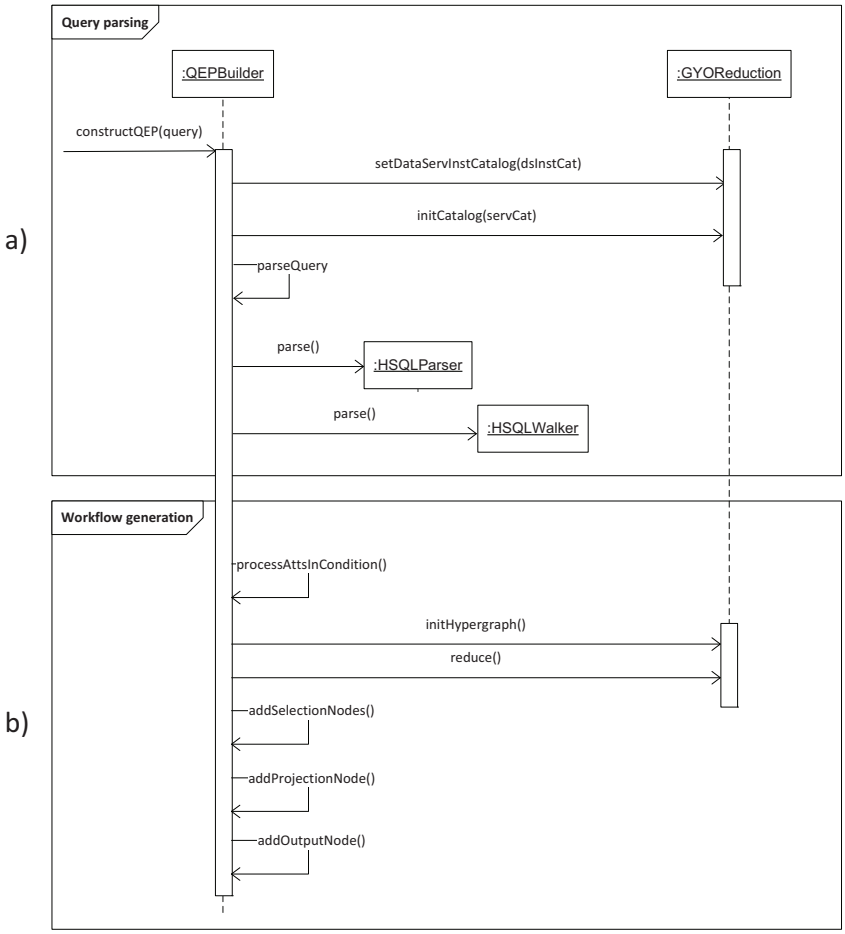


Figure 5.10: Overview of the query workflow generation process by a) parsing and b) reduction and generation

conditions in the WHERE clause of the query. A list of `FromDs` objects represents the on-demand and stream data services appearing in the FROM clause; while a list of objects implementing the `Term` interface represent the data elements that are projected in the SELECT clause. Finally, a `selectStmt` variable specifies the type of projection (* or attribute list, possibly including `distinct`). The parsing process is controlled by the `QEPBuilder` class and is depicted in Figure 5.10 a).

5.3.2 Generation of hybrid query workflows

Figure 5.9 shows the classes that take part in the generation of hybrid query workflows. Most of the work is performed by the `QEPBuilder` class in colaboration with the `GYOReduction` class, while the additional classes enable to represent the data appropriately and to perform support tasks. To generate a query workflow we require first the metadata describing the data service interfaces, which we discuss next.

5.3.2.1 Representation of data service interfaces

We represent data service interfaces by binding patterns, which in our implementation are specified in a text file which is processed to construct a `ServiceCatalog`. Listing 5.1 shows how we specify the binding patterns of Example 4.17.

Listing 5.1: Specification of data service binding patterns

```
profile:profile(string nickname?, int age!, string sex!, string email!)
interests:interests(string nickname?, string tag!, real score!)
location:location(string nickname!, real lat!, real lon!)
```

A colon is used to separate the name of the service from the name of the data operation, in this case all of the services have a single operation that is given the identical name as the service. Inside the parentheses we specify the type and name of each data attribute. A question mark indicates that the parameter is bound, while an exclamation mark denotes that it is free. For instance, the location data operation has only free attributes since it is a stream data operation.

The `ServiceCatalog` class that manages the binding patterns of data operations relies on a `ServiceSignature` parser to obtain the data. This parser is, like the others in our implementation, specified via an ANTLR grammar; in this case the grammar is rather simple so we omit its description.

The `QEPBuilder` class controls the creation of the `ServiceCatalog` from the text files and then makes it available to a `GYOReduction` object, as indicated in Figure 5.10 a). A `DataServInstCatalog` is also created, which contains the addresses of the service instances.

5.3.2.2 BP-GYO reduction and query workflow generation

We will describe the generation of a query workflow from the intermediate representation generated after parsing at two levels. The first corresponds to the overview of the generation process presented in Figure 5.10 b), the second is a more detailed view of the BP-GYO reduction activities within that process.

As discussed previously, the parsing process generates an object structure representing the query. In particular, lists representing the join and selection conditions of the WHERE clause, as well as representations of the terms in the SELECT and FROM clauses of the query. These objects are transmitted by the `QEPBuilder` to a `GYOReduction` object. Furthermore, since aliases may have been used in the FROM clause of the query, it is necessary to resolve them in the join and selection conditions, this is done by the `processAttsInCondition` method.

Then a hypergraph is created by the `GYOReduction` object to represent the joins in the query, as required by our algorithm. Next, the hypergraph is reduced to ultimately generate a join workflow. Once a join query workflow is generated, it is necessary to add the selection and projection operators, as well as a special output operator used to transmit the results, as shown in the bottom of Figure 5.10 b).

Next we discuss in greater detail the BP-GYO reduction process and the subsequent join workflow generation, which occurs in the `GYOReduction` object and is depicted in Figure 5.11.

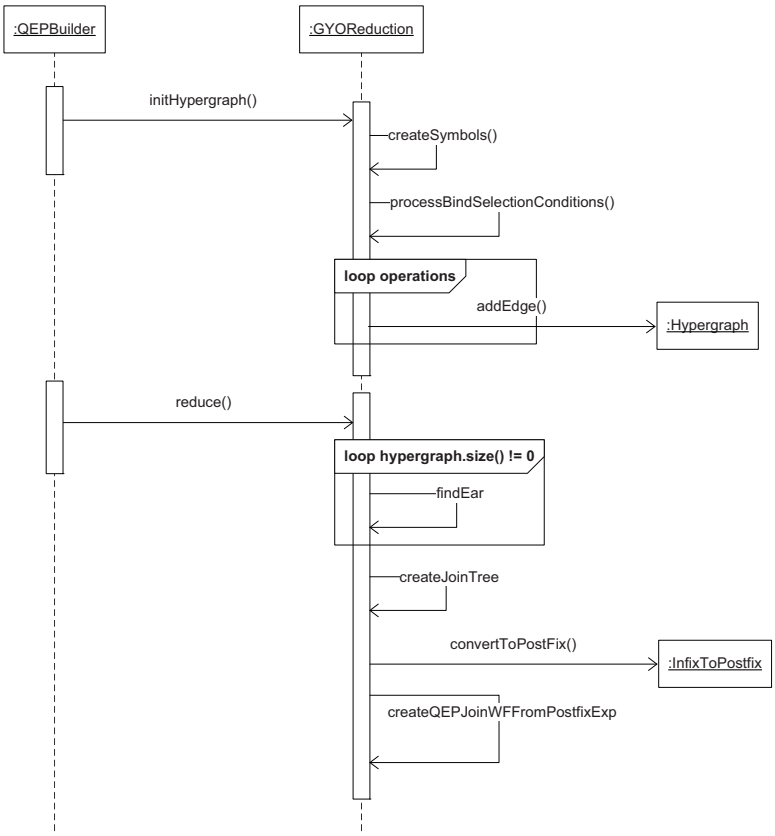


Figure 5.11: Detailed view of the join workflow generation process

First, in order to initialize the hypergraph it is necessary to generate the attribute symbols. Then, it is necessary to identify the selection conditions linked to a bound attributes, since these imply that the attribute is now free. Next we proceed with the construction of the hypergraph, facilitated by our `Hypergraph` class. A hyperedge is added for each of the data operations involved in the query.

When the reduction process is triggered by the `QEPBuilder` we proceed as specified by the algorithm, namely we find an ear and remove it until the hypergraph disappears, constructing the parse tree along the process. The inorder join expression is then derived, which is translated into postfix form by the `InfixToPostfix` class. Once we have a postfix join expression we are able to derive the join workflow, which is then completed by the `QEPBuilder`.

By the process just described HYPATIA generates an executable representation of a query in the form of a hybrid query workflow. This workflow structure employs the classes presented in Figure 5.12. Each `QEPNode` in the query workflow is bound to a `QEPOperator` object that implements the operator associated with the node. All operators implement the `DataDriven` interface consisting of three methods: `init` to initialize the operator (e.g. bind the service stubs), `execute` to process the data, and `destroy` to perform cleanup. In the next section we discuss how this workflow is executed to obtain the query result.

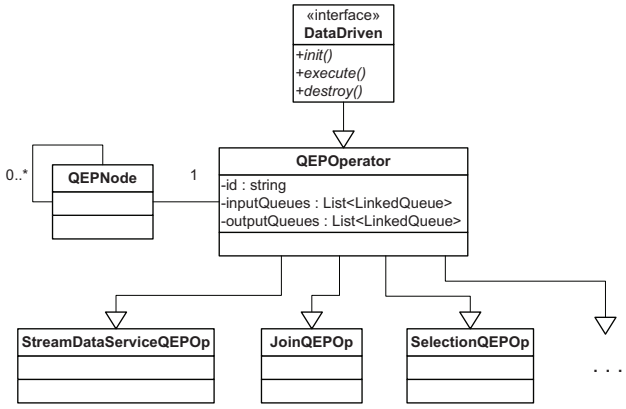


Figure 5.12: Classes for the execution of query workflows

5.4 Execution of hybrid query workflows

.

This section discusses how hybrid query workflows and their operators are activated and executed, also describing in particular the use of computation services in the evaluation of query operators.

The execution of query workflows follows the dataflow specified in their structure, in which operators associated with data and computation services are linked according to the expression corresponding to the HSQL query. The operators communicate with one another via queues. Each operator has a list of input and output queues. All operators use a single output queue; unary operators use one input queue while binary operators two input queues. Although the present communication mechanism relies on shared memory and does not adopt service protocols, a service-compliant Message-Oriented Middleware (MOM) could be adopted to fully support the communication mechanism described in Section 4.3.6.

The concrete steps undertaken to execute a query workflow are presented in Figure 5.13. The process is triggered by the `HybridQP` class, which delegates execution to the `QEPExecutor` class that runs as a thread. The `QEPExecutor` will first create the operator queues and initialize the operators. During initialization, an operator may, for instance, set up the workflow engine in case it is a composite computation service.

The operators are then scheduled. Currently we employ only a round-robin scheduling policy, in which the operators are arranged in a list beginning with the leftmost operators in the workflow; each operator in the list is executed after its predecessor, suspending the execution thread temporarily to avoid blocking the CPU. In this manner we are able to evaluate a query workflow and to obtain the query result.

Next we will discuss query operators in detail and their relation with data and computation services. We begin by discussing the query operators that access data services. Then we will describe basic computation services, simple computation services, and composite computation services, as well as their associated operators. Finally, we address the query operators that are not service-based. All of these concepts are illustrated in Figure 5.14.

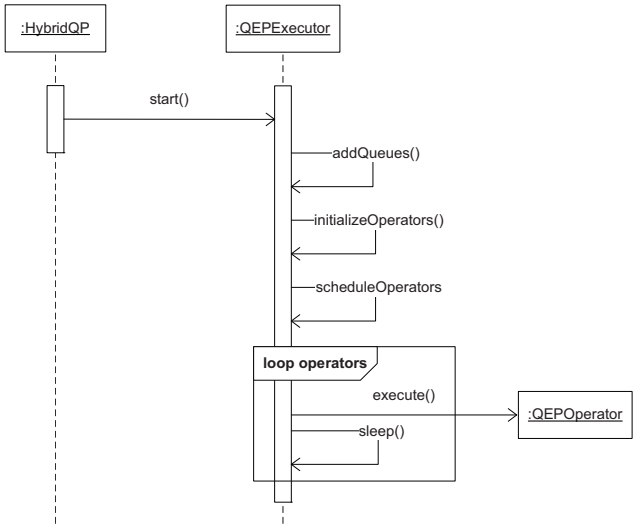


Figure 5.13: Execution of query workflows

5.4.1 Data service operators

Recall that we consider stream and on-demand data services in hybrid queries. Stream data services are accessed by a `StreamDataServiceQEPOp` object that acts as the client in Figure 5.5, therefore it establishes an `Endpoint` with an associated `StreamGateway` service and performs a subscription.

On the other hand, on-demand data services are accessed by the bind-selection and bind-join operators. Both of them use a service access mechanism by which given the name of the service and the signature of the data operation, stubs and method calls are generated via Java reflection. Consequently, no special operators need to be developed to access new on-demand data services.

A bind-selection takes place when in the query all of the input parameters of the on-demand data service are mapped to constant values, in which case a `BindSelectionQEPOp` object performs the service call by the aforementioned method.

Our current implementation of the bind-join in the `BindJoinQEPOp` class follows the definition of Section 3.2.3. It essentially invokes an on-demand data operation with parameters obtained from tuples stemming from one of its inputs, as specified in the join condition, and then combines the input tuples with the tuples obtained from the on-demand data operation invocation. Performance improvements could be obtained by caching the values obtained from the on-demand data operation, as discussed in [MBFS01]. A caching computation service could be used in this case, yielding a composite computation service implementation of the bind-join.

5.4.2 Basic computation services

Basic computation services enable to perform data processing tasks that alone, or as part of a more complex logic in a composition, make it possible to evaluate a query operator. The basic computation

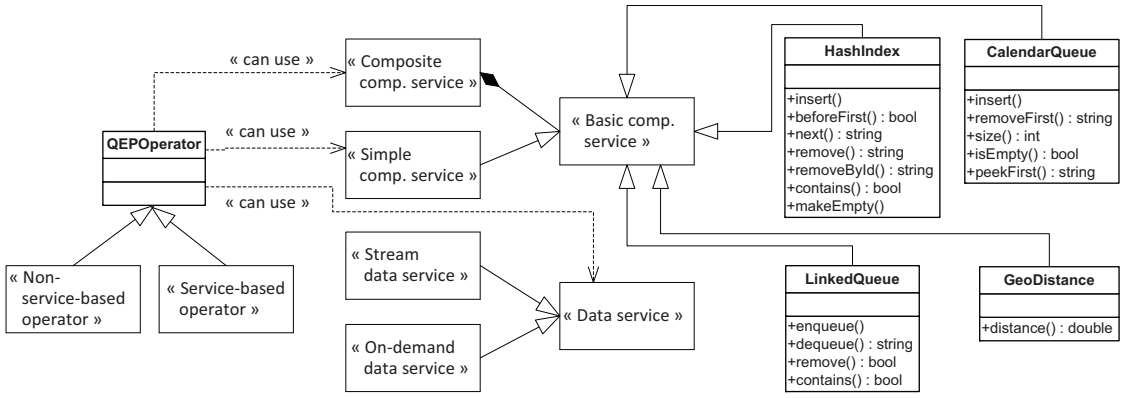


Figure 5.14: Query operators and their relation to data and computation services

services used in our current implementation are presented in Figure 5.14 (right).

The `HashIndex` service enables us to evaluate equi-joins using the symmetric hash join algorithm, as described in Section 4.3. This type of index is used, for example, in PostgreSQL[5]. It is based on the dynamic hash tables algorithm presented in [Lar88]. However, unlike in a traditional hash-table, a cursor is maintained on the hashed data items. In particular, the method `beforeFirst` places this cursor before the first match of the supplied key. In this manner we can iterate over the matches by moving the cursor, which can then be used to generate joined complex values.

As described in Section 4.3 the tuple-based window operator uses a simple queue service. Its interface is `LinkedQueue` and as it name suggests it is based on the standard Java implementation of a linked list. On the other hand, the time-based window uses a sophisticated type of priority queue known as a calendar queue, whose implementation we adopted from the JiST[6] high-performance discrete event simulation engine. Fundamentally, it creates an additional level of indexation over a series of heaps in order to optimize access to data elements by a numeric value such as time.

Finally, the `GeoDistance` service enables to determine the distance in meters between two geographical locations specified by latitude and longitude coordinates. This calculation appears as part of a function-selection in one of our scenarios, and it is made using Vincenty's formulae[7]. For this purpose we relied on an open source Java geodesy library[8].

5.4.3 Simple computation services

Recall from Section 4.3 that simple computation services involve a single operation invocation of a basic computation service. Currently, the only operator that fits this category is the `FunCallSelection` operator, which given a selection condition involving a function (e.g. *distance(lat, lon, 48.85889, 2.29583) <= 3.0* in Example 4.15) will evaluate the function and determine whether the condition is satisfied.

[5] http://www.postgresql.org/
[6] http://jist.ece.cornell.edu/index.html
[7] http://en.wikipedia.org/wiki/Vincenty%27s_formulae
[8] http://www.gavaghan.org/blog/free-source-code/geodesy-library-vincentys-formula-java/

Two main tasks need to be performed to evaluate a function call selection. First, the terms representing the function call parameters must be evaluated, these terms can be either constant values defined in the query or values obtained from an input complex value. Second, the service must be invoked to obtain the result so that it can be compared with the other side of the (in)equality. The service invocation occurs via the same reflection-based mechanism used for on-demand data services. In this manner, a single operator can be used to filter tuples based on the results produced by various functions belonging to different services.

5.4.4 Composite computation services

As described in Section 4.3 some operators can require multiple operation invocations from possibly multiple computation services. Consequently, we employ a workflow service coordination model based on abstract state machines (ASMs) to build composite computation services.

Our currently implemented composite computation services (equi-join as well as tuple and time based windows) were described in Section 4.3 along with the workflow model, of which we give further details in Appendix C. Therefore, we only give an overview of the components of the interpreter that enable us to execute a service coordination, along with additional constructs that enable us to derive a GUI visualization of ASM-specified workflows.

ASM interpreter

The classes that build and interact with our ASM interpreter are presented in Figure 5.15. The `Interpreter` relies on an `ASMParser` to obtain an abstract syntax tree (AST) representation of an ASM specified according to our ASMs grammar, which we provide for reference in Appendix B. The state of the ASM, i.e. the value of variables, and the changes performed to it are handled by the `MemorySpace` classes.

Concretely we employ two subclasses of `UpdateSpace` for this purpose, which handle updates in parallel and sequential compositions. Their precise semantics is described in Appendix C, but we note that sequential updates are performed immediately while parallel updates are performed at the end of the composition, and there must not be a conflict in the generated parallel updates. By the use of stacks we ensure that the updates are performed at the appropriate moment. Currently we do not support true parallelism in the execution of rules.

Our `Interpreter` is designed to be embeddable, meaning that it can be used within an arbitrary object that we refer to as a *controller*. In addition, the interpreter can interact with another object that will handle the function calls performed in ASM rules, which we refer to as the *environment*. In the case of query operators extending `QEPOperator`, the single operator object plays both roles.

The main events during the execution of a service coordination implementing a query operator are presented in Figure 5.16. When the `QEPOperator` receives a method call to initialize, it in turns calls the `initInterpreter` method on the interpreter, which will parse the operator ASM specification stored in a text file to generate the AST. Afterwards, each time when the `QEPOperator` is scheduled to execute, it will order the interpreter to execute a step.

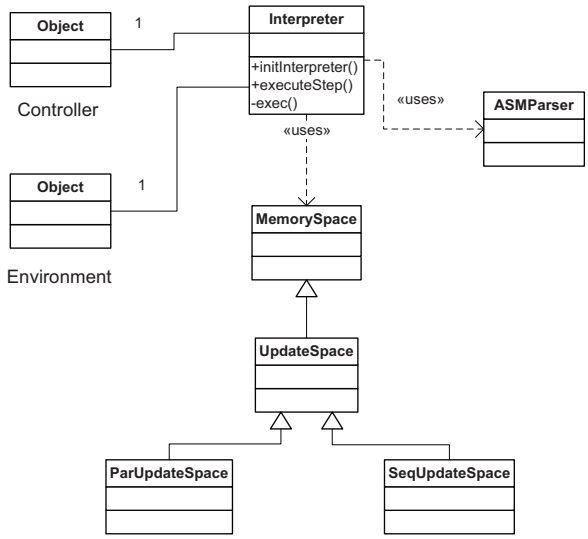


Figure 5.15: Class structure of our ASM interpreter

During the execution of each step, input data will be read, possibly output data may be generated, and also computation service operations may be invoked. All of these occur via function calls in ASM rules that the interpreter will translate into method calls on the environment, which in this case is the query operator. The method calls are performed via reflection, therefore the environment only needs to have an implementation of the corresponding method, instead of relying on a difficult to evolve ad-hoc mechanism.

5.4.5 Not-service-based query operators

Our service coordination approach for the construction of query operators offers many advantages, such as flexibility and the allocation of new resources. However, in a concrete implementation some operators may employ local functionality not accessed as computation services, unless it was developed expressly in that way. Particularly for our data model, this applies to the unary operators that manipulate complex values according to our mathematical specifications: projection, selection, and renaming.

As we mentioned in Chapter 4, we address the full evaluation process only for queries involving tuples with attributes consisting of atomic complex values. Therefore, we employ relatively straightforward standard Java implementations of the selection and projection operators, while renaming is not currently implemented.

Although, as stated, we currently do not support the evaluation of queries addressing nested values within tuple attributes, we have implemented the specifications of the recursive selection, projection, and renaming operators presented in Section 3.2. The implementation enables us to validate the specification of the operators, it also makes significantly easier the incorporation of recursive operators in the future. Next, we describe the implementation of the selection and projection operators, each of which represents

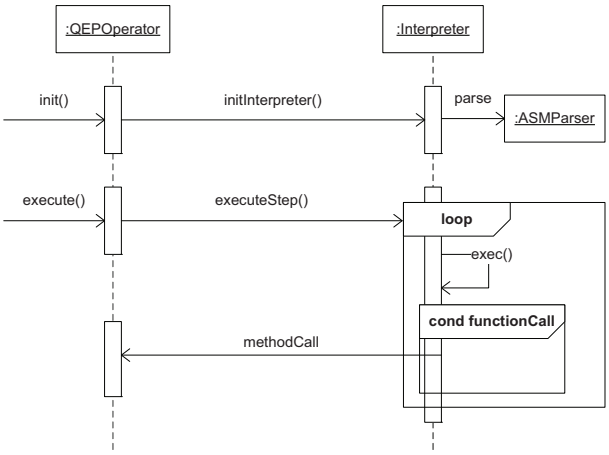


Figure 5.16: ASM interpreter execution

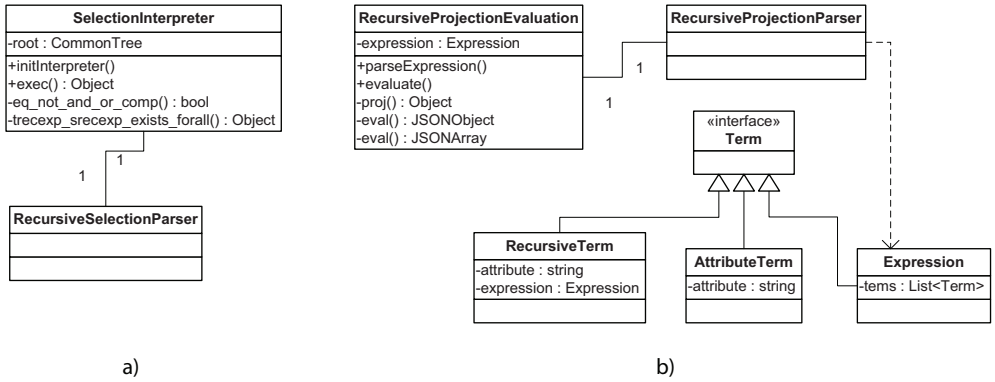


Figure 5.17: Class structure of a) our recursive selection and b) recursive projection operators

a particular implementation pattern we identified.

Recursive selection implementation

Our implementation of the recursive selection follows the specification presented in Section 3.2, although it currently does not support function call terms and permits only comparisons with basic value types (e.g. string or integer). It is based on two main components: a `RecursiveSelectionParser` that builds an abstract syntax tree (AST) representing the selection expression, and a `SelectionInterpreter` that uses the AST to implement the satisfiability rules of our selection specification. These components are depicted in Figure 5.17 a).

The parser was built using the Java-based ANTLR 3.3[9] parser generator, which we also used for other parsers for our work. Next we present in Listing 5.2 the simplified specification of selection expressions

[9] http://www.antlr.org/

in the ANTLR syntax.

Listing 5.2: Recursive selection implementation grammar

```
fexp       : term ((AND|OR) term)*;
term       : NOT term | phrase;
phrase     : '(' fexp ')' | bexp | rexp;
bexp       : atom ( (LT|LE|GT|GE|EQ|NE) atom);
rexp       : trexp | srexp;
trexp      : SIGMA '(' ID ',' fexp ')';
srexp      : SIGMA '(' ID ',' quantifier fexp ')';
quantifier : EXISTS | FORALL;
```

Words in boldface represent terminal symbols. A full expression (*fexp*) consists of the conjunction or disjunction of (possibly negated) *terms*. Terms in turn consist of boolean (*bexp*) or recursive (*rexp*) expressions. Boolean expressions are comparisons of *atoms* (basic type values). Recursive expressions are of two kinds, tuple (*trexp*) and set (*srexp*), the latter of which use quantifiers.

Once an AST has been constructed from a given selection expression, via the parser generated from the above grammar, the AST and the input JSON object are passed to the interpreter. The `exec` method will then process the tree and the JSON object, starting from the root and invoking recursively the methods corresponding to each type of AST node. For illustration, we present in Listing 5.3 the simplified code that evaluates the recursive set expressions with an universal quantifier.

Listing 5.3: Method to evaluate universally quantified set subexpressions

```
private Object forall(JSONArray set, SelAST expr) {
    boolean valid = true;
    for(int i = 0; i < set.length(); i++) {
        JSONObject vali = set.getJSONObject(i);
        Object valieval = exec(expr, vali);
        if( !((Boolean)valieval).booleanValue() )
            valid = false;
    }
    return new Boolean(valid);
}
```

The method receives the set of complex values and the subexpression. It evaluates the subexpression for each value in the set and returns true only if all of the complex values satisfy the subexpressions. The recursive invocation of methods of this type over the input JSON complex value, as well as over its nested complex values, enables to determine if the input complex value satisfies the entire selection condition.

Example 5.22.

Given the following complex value (in our model notation)

person:⟨ sex:'M', nickname:'Charles', email:'charles@gmail.com', age:40,
interests:{stag:⟨ tag:'art', score:6.5 ⟩, stag:⟨ tag:'sports', score:7.5 ⟩}⟩

The following expression enables to determine that the person has at least one interest with a score greater than 7.0.

```
SIGMA(person, SIGMA(interests, exists SIGMA(stag, score > 7.0) ))
```

Recursive projection implementation

The implementations of the recursive projection and renaming operators adopt a different approach from that of the selection operator. Concretely, they build a custom object structure from a given expression, instead of an AST. This structure is then used to process recursively the input JSON complex value to obtain the desired result. In this case we do not employ an interpreter to determine the invocation of functions, since they coordinate among each other.

In the case of the recursive projection operator, a `RecursiveProjectionEvaluation` class implements the processing functions. It relies on a `RecursiveProjectionParser` to build the expression representation, which in turn consists of objects from several `Term` subclasses. The main classes involved are presented in Figure 5.17 b).

5.5 GUI and workflow visualization

A query can be issued via a GUI, supported by the `HypatiaGUI` class, which also enables to visualize its results along with the hybrid query workflow and the operator workflows. We present a screenshot of this GUI in Figure 5.18. The user poses the query in the top text area and triggers its evaluation via the `Run` button. The query results will appear in the bottom text area.

In addition, the left panel presents the generated hybrid query workflow; data services appear in yellow and computation services in blue. The right panel enables to visualize the operator workflows. The particular operator workflow to visualize can be selected via a menu. In both cases, to graphically display workflow graphs in the GUI we use the JGraph[10] library.

5.5.1 Visualization of hybrid query workflows

The `QEPNodeToJGraph` class is in charge of generating a JGraph representation of a hybrid query workflow to be displayed in the GUI. The algorithm it employs is based on a traversal of the hybrid query workflow, we omit additional details since the process is relatively straghtforward. A partial closeup of a displayed workflow graph is presented in Figure 5.19.

[10] http://www.jgraph.com/

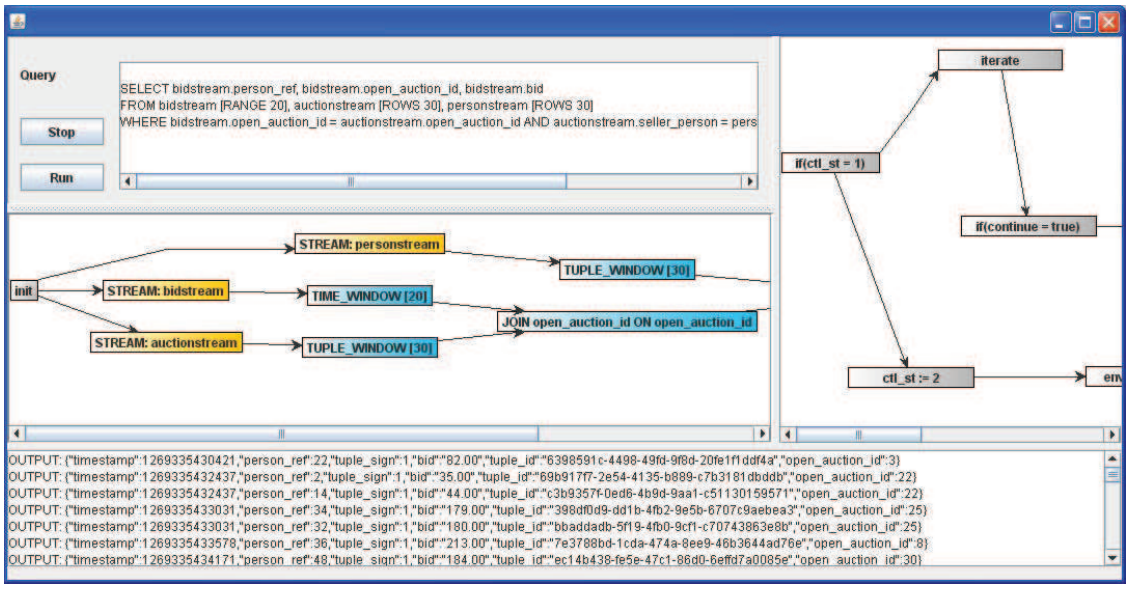


Figure 5.18: Screenshot of the HYPATIA GUI

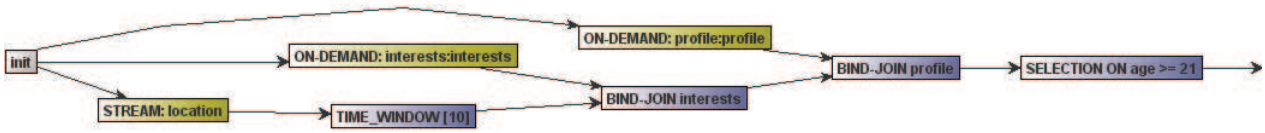


Figure 5.19: Fragment of a query workflow displayed in our GUI

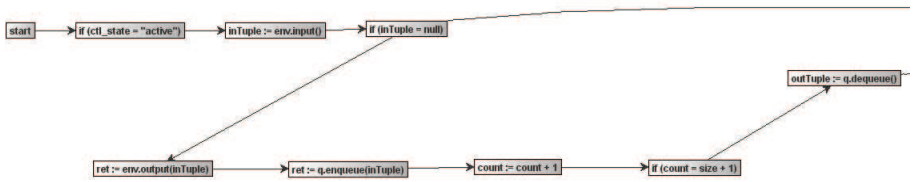


Figure 5.20: Fragment of an operator workflow displayed in our GUI

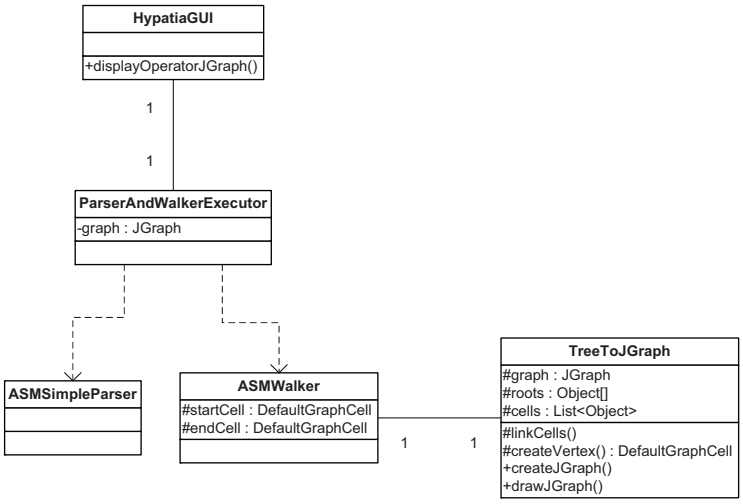


Figure 5.21: Classes involved in the visualization of operator workflows

5.5.2 Visualization of operator workflows

The visualization mechanism for operator workflows, which are specified as ASMs, differs significantly from that we use for hybrid query workflows. In this case a visualization is generated from the textual ASM specification and is fully independent of the structures generated for the execution of the composite computation service by the interpreter. Figure 5.20 presents a close caption of part of a time-based window operator workflow.

For the generation of the visual representation of an operator workflow we employ a two phase parsing process. In the first step we use a parser that generates an abstract syntax tree representation of the ASM similar to the one used for its execution. A second parser then processes the generated AST to construct a `JGraph` object that can be displayed in the GUI. The second parser essentially implements the construction rules presented in Section 4.4.

This two-phase process is controlled by a `ParserAndWalkerExecutor` object, while a `TreeToJGraph` object supports the construction rules. The classes involved are presented in Figure 5.21. As required by the JGraph library, a *model* of the workflow is constructed first, which is represented by the `roots` and `cells` attributes of the `TreeToJGraph` class. This model is then used to produce the `JGraph` visualization object that is presented in the GUI. The process is outlined in Figure 5.22.

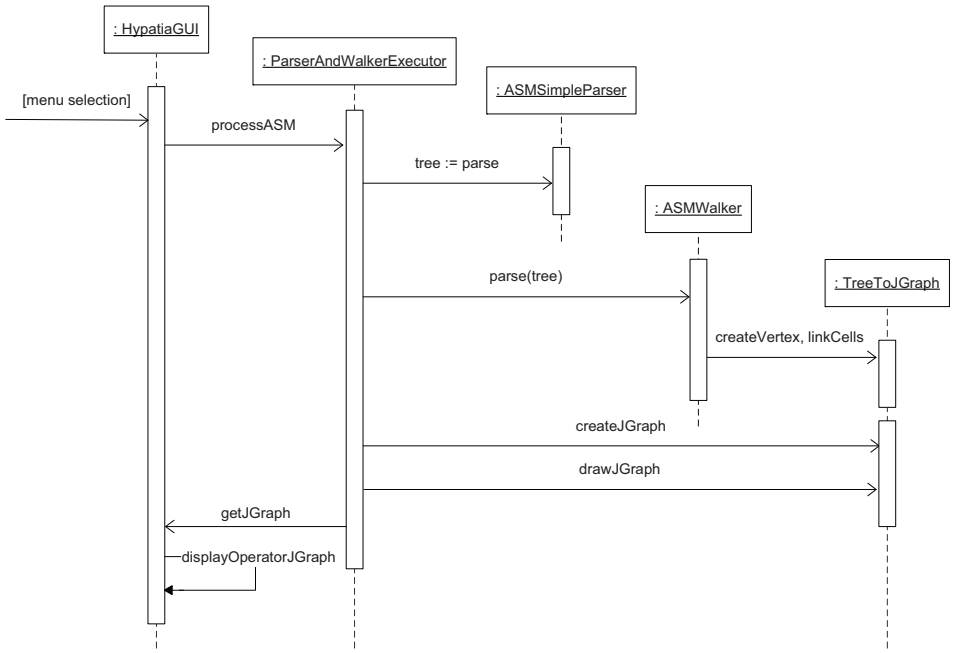


Figure 5.22: Generation of the visualization of an operator workflow

5.6 Testbeds and experimental results

In this section we present the two testbeds that we implemented to validate our approach and implementation. One of them enabled us to obtain experimental results of which we present a discussion.

5.6.1 Friend Finder

Our first testbed is based on the Friend Finder scenario described in our Introduction. We implemented the data services presented in Example 4.17, which comprises the *location*, *profile*, and *interests* data services. This testbed was used in the demonstration of our system at [CVCVSI10b], where along with the Hypatia GUI we presented an interface based on Google Maps[11] to visualize the query results. Next, we present first how we implemented the data services and generated the test data, after which we discuss the mapping interface.

Data services

Recall the Friend Finder data service interfaces presented in abbreviated form in Example 4.17 along with their binding patterns.

$$location{:}\langle\ nickname^f,\ lat^f,\ lon^f\ \rangle$$

$$profile{:}\langle\ nickname^b,\ age^f,\ gender^f,\ email^f\ \rangle$$

[11] http://maps.google.com/

interests:$\langle$ *nickname*^{b}*, tag*^{f}*, score*^{f}

A `GenerateData` class is responsible to generate the data available through these service interfaces. It employs a series of seed data files for this purpose, which in conjunction with a randomized algorithm enables us to automatically generate data sets with various parametrized characteristics.

First we generate the *profiles* of the users, whose nicknames are obtained from files with lists of common male and female names. Their ages are assigned randomly while their email addresses are constructed by the combination of their nickname with a set of email providers. In the case of the *interests* data service, we first generate a series of scored tags which are then assigned randomly to a particular user. The scored tags are obtained from Amazon's most popular tags list[12]. The score of a tag for a particular user is determined by a combined metric of the tags popularity at Amazon (reflected in its font size) and a random number.

The greatest challenge corresponds to the generation of the *location* data. Aiming to present a real world use case, we incorporated the GPS-tracks data provided by multiple users at EveryTrail[13]. This is a site that enables users to share the data generated by GPS devices during walks, bicycle rides, driving, etc. We chose in particular 58 GPS tracks for the city of Paris of an above medium length.

The data are originally available in GPS eXchange Format[14] (GPX) files, we translate them into JSON by the use on an XSL stylesheet. Then we perform a simulation in which a certain number of users join the location stream every certain time interval. Each of these users is randomly assigned a particular location track which advances during the simulation. The cummulated location notifications of the active users are then shuffled and added to the location stream.

The generated location stream is saved into a file using our batch format, so that it can be published by our `MultiStreamServer`. On the other hand, the profiles and interests data is saved into a MySQL[15] database, which is then made available as a service by a Java application running on the Tomcat application server.

Mapping interface

We implemented a web interface based Google Maps to visualize the results of Friend Finder queries involving location. A screenshot of this interface is presented in Figure 5.23. It is supported by an application that computes the extent of the data stream generated by the query, which (recall Definition 3.12) fundamentally corresponds to a view defined over the tagged data stream.

The HYPATIA query processor publishes the result stream via a Web Service, which is accessed by the client application of the web interface, it running on a browser and implemented in JavaScript. The JavaScript stubs for the result data stream service were generated with Apache CXF 2.2[16]. The client web application periodically accesses the result data stream service, which enqueues the query results

[12] http://www.amazon.com/gp/tagging/cloud
[13] http://www.everytrail.com/
[14] http://en.wikipedia.org/wiki/GPS_eXchange_Format
[15] http://www.mysql.com/
[16] http://cxf.apache.org/

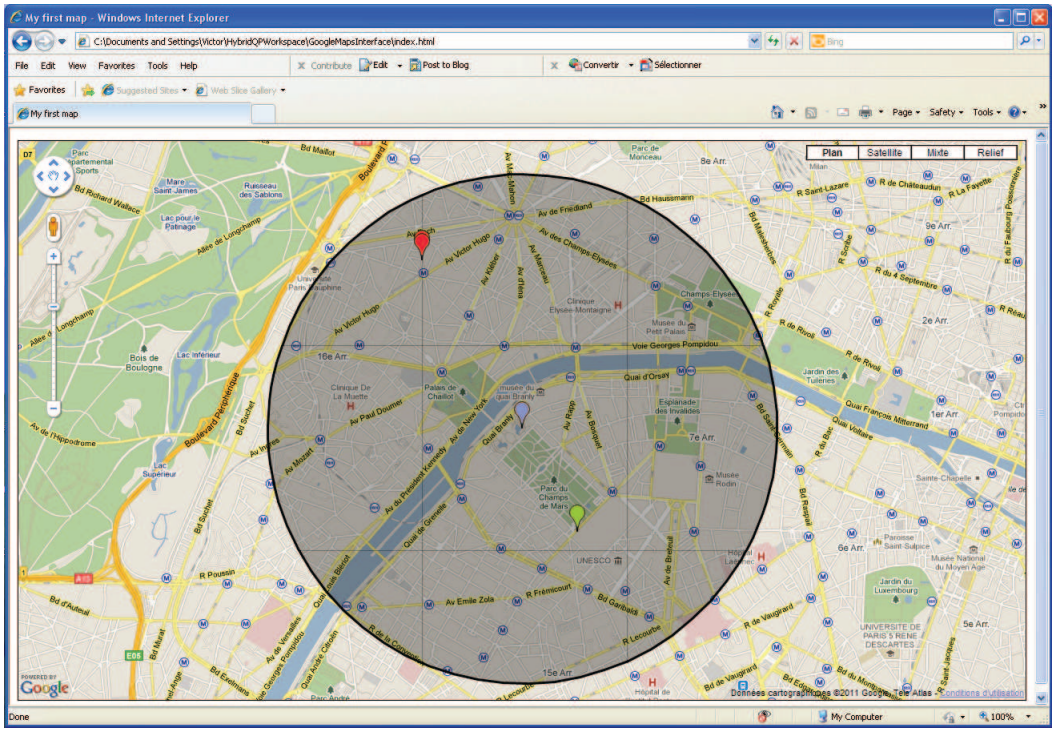


Figure 5.23: Google Maps web interface for our testbed

obtained up to a given point.

By the use of the Google Maps API[17] we first display a map corresponding to a predefined location, over which we can overlay a circular area of a predefined radius and center. In Figure 5.23 we present a map of Paris centered at the location of the Eiffel tower, at which we assume an user is issuing a query like that of Example 4.15.

As the result data stream is obtained, the web interface continuously adds or removes markers that represent the location of a particular user, as shown in Figure 5.23. The colors of the markers are assigned in such a way as to avoid or minimize collisions. The results of Figure 5.23 correspond to a particular instant for a query to find all the users with an age greater than 21 and located at no more than 2 km from the center of the circle, which has a predefined radius of also 2 km.

5.6.2 NEXMark

A second scenario was developed to measure the efficiency of our current implementation in a more reliable manner. It is based on the NEXMark benchmark [18], originally implemented for streaming XML data but that we modified to generate JSON complex values. Our NEXMark testbed follows as close as possible the original auctions scenario, and consists of three stream data services supported by our `MultiStreamServer`: **`person`**, **`auction`** and, **`bid`**; which export the following interfaces.

[17] http://code.google.com/apis/maps/documentation/javascript/
[18] http://datalab.cs.pdx.edu/niagara/NEXMark/

person:⟨person_id^f, name^f, phone^f, email^f, income^f⟩
auction:⟨open_auction_id^f, seller_person^f, category^f, quantity^f⟩
bid:⟨person_ref^f, open_auction_id^f, bid^f⟩

A variable number of persons join the auctions periodically. These persons propose products to sell in an auction, while at the same they or other persons can make offers on the auctioned products by means of bids for a certain amount.

Our main goal was to measure the overhead of using services, so we measured the total latency (i.e. the total time required to process a tuple complex value present in the result) for a set of six queries that are presented in Table 5.1. They are stated using our operators and HSQL expressions, and presented along to the query service coordination that implements them (generated by HYPATIA).

We measured latency first for our service-based system, HYPATIA, and then for an equivalent system that had at its disposal the same functionality offered by the computation services, but supported by objects inside the same Java Virtual Machine.

5.6.3 Experimental results

For our experiments we used as a local machine a Dell D830 laptop with an Intel Core2 Duo (2.10 GHz) processor, equipped with 2 GB of RAM. The remote machine was a Dell Desktop PC with a Pentium 4 (1.8 GHz) processor and 1 GB of RAM. In both cases running JSE 1.6.0_17, the local machine under Windows XP SP3 and the remote machine under Windows Server 2008.

Our first experimental setting corresponds to a query processor using the same functionality of our computation services, but as plain java objects in the same virtual machine. In the second we used our computation services, which are based on the JAX-WS reference implementation, by making them run on a Tomcat container in the same machine as the query processor. For the third setting we ran the Tomcat container with the computation services on a different machine connected via intranet to the machine running the query processor.

The results are shown in Figure 5.24, and from them we are able to derive two main conclusions. First, the use of services instead of shared memory resulted in about twice the latency. Second, the main overhead is due to the middleware and not to the network connection, since the results for the local Tomcat container and the remote Tomcat container are very similar. We believe that in this case the network costs are balanced-out by resource contingency on the query processor machine, when that machine also runs the container.

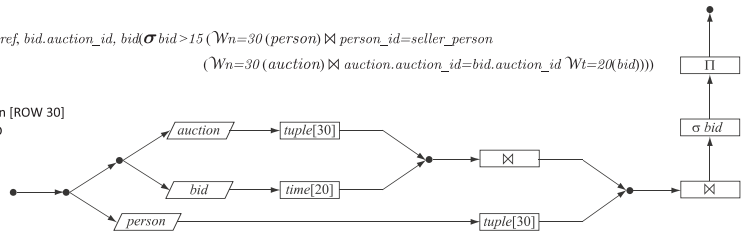
We consider the overhead to be important but not invalidating for our approach, especially since in some cases the best alternative can be to use services to acquire the required functionality. In addition, with the incorporation of optimization techniques the cost and associated overhead can be reduced further.

Table 5.1: NEXMark queries

- For the last 30 persons and 30 products offered, retrieve the bids of the last 20 seconds greater than 15 euros

1)

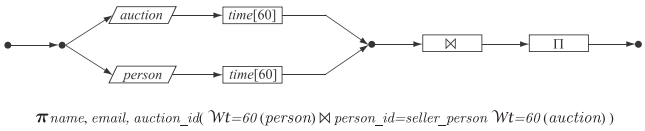
SELECT bids.person_ref, bid.auction_id, bid.bid
FROM bid [RANGE 20], auction [ROW 30], person [ROW 30]
WHERE bid.auction_id = auction.auction_id AND
auction.seller_person = person.person_id
AND bid.bid > 15;



- For the persons joining and the products offered during the last minute, generate the name and email of the person along with the id of the product he/she offers

2)

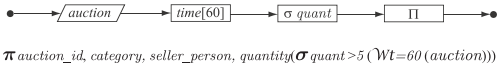
SELECT P.name, P.email, A.auction_id
FROM auction [RANGE 60] as A, person [RANGE 60] as P
WHERE A.seller_person = P.person_id;



- For the last 50 products, find those whose quantity is greater than 5

3)

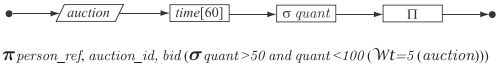
SELECT A.auction_id, A.category, A.seller_person, A.quantity
FROM auction [ROWS 50] as A
WHERE A.quantity > 5;



- Among the bids made in the last 5 seconds, find those whose amount is between 50 and 100 euros

4)

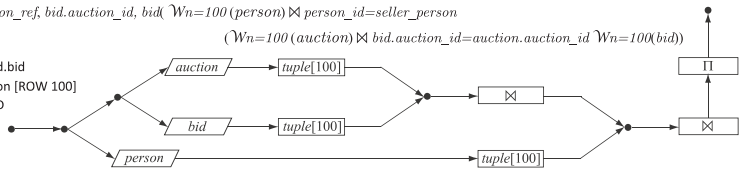
SELECT bid.person_ref, bid.auction_id, bid.bid
FROM bid [RANGE 5]
WHERE bid.bid > 50.0 and bid.bid < 100.0;



- For the last 100 persons, products and bids; give the id of the seller person, the id of the product, and the amount of the bid

5)

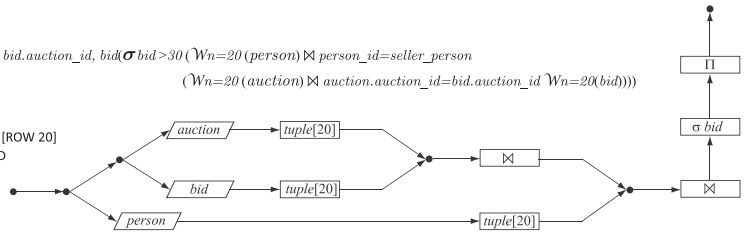
SELECT bid.person_ref, bid.open_auction_id, bid.bid
FROM bid [ROW 100], auction [ROW 100], person [ROW 100]
WHERE bid.auction_id = auction.auction_id AND
auction.seller_person = person.person_id;



- For the last 20 persons, products and bids; give the id of the seller person, the id of the product, and the amount of the bid, whenever that amount is greater than 30

6)

SELECT bid.person_ref, bid.auction_id, bid.bid
FROM bid [ROW 20], auction [ROW 20], person [ROW 20]
WHERE bid.auction_id = auction.auction_id AND
auction.seller_person = person.person_id
AND bid.bid > 30;



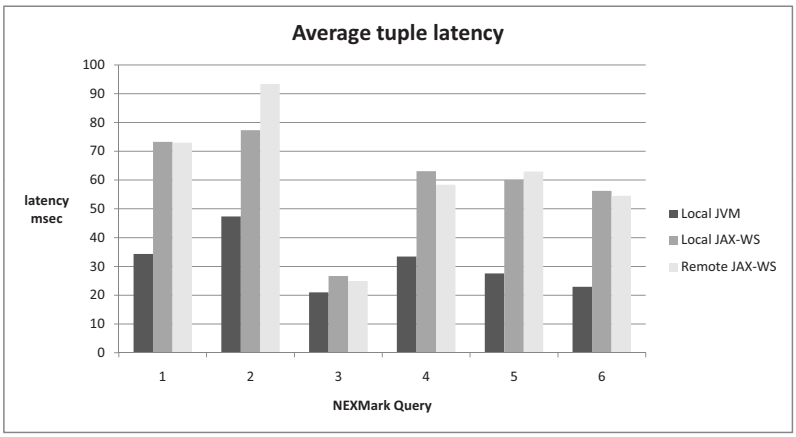


Figure 5.24: Tuple latency for a services-based vs. a single Java application query processor

5.7 Conclusions

We have presented the implementation of the concepts presented in Chapters 3 and 4. Taken together, they enabled us to create HYPATIA, a service-based hybrid query processor. By our implementation and validation we demonstrated that it is possible to evaluate hybrid queries by relying on service coordination, and therefore without necessarily using a full-fledged DBMS. Furthermore, our experimental results show that although there is some penalty in efficiency for our approach, which is to be expected, this nevertheless is acceptable when considering its advantages.

CHAPTER 6
Conclusions and perspectives

We present our conclusions and discuss the perspectives for possible future work.

6.1 Main results and contributions

In this thesis we have presented a solution to the problem of efficiently evaluating queries in dynamic environments. This entails declarative querying over data produced by data services, taking advantage of computation services to process the data and obtain the query result. Thereby gaining in flexibility and extensibility.

Concretely, in our solution we introduced the notion of hybrid queries, defined a data model for this type of queries, and developed an approach for their evaluation based on service coordination. Although many challenges remain, our solution demonstrated to be efficient and adequate. Next we present our conclusions in regard to the specific contributions of this thesis as discussed in our introduction.

- *Analysis and taxonomy of queries.* By an analysis of the characteristics of the queries associated with dynamic environments we derived a query taxonomy and arrived at the notion of hybrid queries. A further analysis of the techniques employed for their evaluation helped us to incorporate many of them in our approach. Although our analysis and taxonomy are far from exhaustive, they can serve for the further development of our approach and be a useful tool for understanding the vast domain that query processing has become.

- *Data model for hybrid queries.* We introduced a data model for hybrid queries that meets multiple goals. First it formalizes on-demand and stream data services, capturing the differences between the two but enabling both to partake in hybrid queries. The adoption of complex values provides a greater flexibility than the relational model, which our model subsumes, but still presents a level of structure that makes it easy to use in applications.

  We presented a series of operators that operate on complex values and that we believe provide sufficient power for a large class of queries. Our experience confirms the need of formal specifications given the many subtleties that usually arise. We also introduced HSQL, a SQL-like language for hybrid queries compatible with data services and an important subset of our query operators. Achieving full compatibility, however, lies beyond the scope of our work.

- *Evaluation of a hybrid query as a service coordination.* We proposed an approach in which a hybrid query is evaluated by a workflow-specified service coordination comprising data and computation services. In this manner, query evaluation takes place without the need of an off-the-shelf DBMS and in a flexible way. This approach is compatible with our data model as demonstrated by our construction rules that map operator expressions to query workflows. In addition, our approach proved to be applicable to declarative languages. By means of the BP-GYO algorithm we are able to generate a query workflow directly from an HSQL query in an efficient manner.

- *Evaluation of query operators as composite computation services.* The applicability of our service-based evaluation approach would be severely restricted if a direct match between query operators and *a priori* available computation services was required. For this reason we provide the means to construct new computation services from component computation services. This process is enabled by a coordination model and language that has its foundations in the abstract state machines formalism, and that is compatible with a workflow representation. This model proved adequate to construct several query operators belonging to our model, based on realistic component computation services.

- *HYPATIA, a service-based hybrid query processor.* Our approach was validated by HYPATIA, in which we implemented our query workflow construction mechanism for HSQL queries based on the BP-GYO algorithm, as well as the service-based evaluation of hybrid queries by query workflows. Also including query operators implemented by composite computation services specified via our ASM-based coordination language. The Friend Finder and NEXMark testbeds proved our techniques to be successful in concrete scenarios. In addition, our experiments showed that although our service-based approach implies a certain overhead, it is acceptable and does not severely limit scalability.

6.2 Perspectives

Although we regard the results obtained from this thesis as satisfactory, many challenges and possible improvements remain, covering various aspects of our solution. Next we discuss the most relevant ones.

- *Query optimization* Our approach opens many opportunities for optimization related to the following aspects of query evaluation

  (i) Generation of optimal query workflows. For the evaluation of a given hybrid query, it is highly desirable to examine equivalent alternative query workflows, since operator reorderings can lead to significantly better performance. The BP-GYO algorithm could potentially facilitate the exploration of the search space of query workflows, because different choices of the hyperedges that are consumed (which are now arbitrary) lead to different query workflows. It is important to consider, however, that in a dynamic environment the statistics commonly used for optimization will not be readily available, and QoS criteria become predominant. Currently these issues are being

addressed as part of the thesis work of Carlos Manuel Lopez-Enriquez, beginning with a Prolog implementation of the BP-GYO algorithm.

(ii) Choice of operator algorithms. In our approach, query operators can be evaluated via composite computation services that implement operator algorithms. In a dynamic environment, different basic computation services can be available that could serve to implement, via different composite computation services, different algorithms for the same operator. The choice of different computation services thus represents an optimization opportunity, in which the optimal choice is not only linked to the algorithm *per se*, but also to the basic computation services that support it (i.e., their cost, access time, etc.).

(iii) Scheduling and localization of operators. Once a query workflow has been created it needs to be executed in one or several machines by means of multiple processes. The scheduling of operators determines when and how much time each of the various operators assigned to a particular machine or process is executed. An adequate scheduling policy represents an opportunity for optimization. Operator localization, on the other hand, implies determining which operators will be assigned to each machine available. In this case the problem is more complex than in traditional settings, since not only query operators but the various computation services they use are involved.

We believe that both scheduling and localization could benefit from a cost model specified using queueing theory. Then the various policies and algorithms available in the literature could be tested for scheduling. For operator localization, constraint-based logic programming could be employed to determine an optimal solution.

(iv) Caching over data and computation services. In some cases, the data obtained from data or computation services can be cached, thus avoiding potentially expensive operation calls. This can occur, for instance, if different input complex values generate the same or similar input parameters. This problem is currently under study in the context of Web Service Mashups as part of the thesis of Mohamad Othman-Abdallah.

(v) Global optimization. A highly ambitious alternative involves optimization over a global view of a query workflow. This implies considering all the interdependencies between the activities of the various operator workflows that form part of the query workflow. Optimization techniques could then be applied at this lower-level granularity. In this case the techniques used for the optimization of programming languages could prove useful.

- *Intra-operator parallelism* Further performance improvements could be obtained by the adoption of intra-operator parallelism. In principle, our approach is compatible with intra-operator parallelism, since we specify query operators at a finer granularity by employing composite computation services. With the appropriate parallel algorithms handling data partitioning, a greater number of computation services could be exploited to evaluate query operators.

- *Extend the capabilities of HSQL* Our current specification of HSQL supports only a subset of the operators in our data model and does not fully exploit the capabilities of complex values. By

extending HSQL to fully comply with complex values we could take advantage of our implementations of the recursive operators presented in Section 5.4.5. This could be accomplished by the incorporation of variables to the language as, for instance, in [CDLR90].

Another possibility for enhancing the language, and our implementation of it, is to support additional predicates via our mechanism of selections with function calls, as illustrated in Section 4.3.5. In this manner, predicates related in particular to spatio-temporal relations could be incorporated. They could then be supported if the adequate computation services are available.

- *Adopt alternative architectures* Some of the aforementioned enhancements could be better implemented in alternative architectures. For instance, parallelization could benefit from migrating HYPATIA to a cloud architecture based on an infrastructure such as Amazon Web Services. A Grid architecture could be useful to test the effectiveness of operator localization techniques. Another interesting perspective it to implement our approach in an alternative programming language and platform, for example, Erlang due to its concurrency capabilities.

- *GUI specification of query and operator workflows* Currently in HYPATIA hybrid queries are specified in HSQL and composite computation services in our ASM-based coordination language, these specifications are then used to generate a visual representation of query and operator workflows, respectively. The visualization capabilities of the HYPATIA GUI could serve as the base to develop an improved GUI which would allow a user to manually construct a query or operator workflow, in a manner similar to some Web Service mashup systems or tools used in business process management (BPM). We have taken a first step in this direction by starting the development of a demonstration of HYPATIA in a BPM context.

Bibliography

[AAB+05] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, January 2005.

[AB95] Serge Abiteboul and Catriel Beeri. The power of languages for the manipulation of complex values. *The VLDB Journal*, 4:727–794, October 1995.

[ABC+76] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System r: relational approach to database management. *ACM Trans. Database Syst.*, 1:97–137, June 1976.

[ABM08] Serge Abiteboul, Omar Benjelloun, and Tova Milo. The active xml project: an overview. *The VLDB Journal*, 17:1019–1040, August 2008.

[ABS00] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: from relations to semistructured data and XML*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[AC88] Michel E. Adiba and Christine Collet. Management of complex objects as dynamic forms. In *Proceedings of the 14th International Conference on Very Large Data Bases*, VLDB '88, pages 134–147, San Francisco, CA, USA, 1988. Morgan Kaufmann Publishers Inc.

[ACc+03] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12:120–139, August 2003.

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer, Berlin, 2004.

[Adi07] Michel Adiba. Ambient, Continuous & Mobile Data. Presentation, 2007.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. *SIGMOD Rec.*, 29(2):261–272, 2000.

[AHS07] Karl Aberer, Manfred Hauswirth, and Ali Salehi. Infrastructure for data processing in large-scale interconnected sensor networks. In *MDM '07: Proceedings of the 2007 International Conference on Mobile Data Management*, pages 198–205, Washington, DC, USA, 2007. IEEE Computer Society.

[AHV95a] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[AHV95b] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[All83] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[AvdBF+92] P. M. G. Apers, C. A. van den Berg, J. Flokstra, P. W. P. J. Grefen, M. L. Kersten, and A. N. Wilschut. Prisma/db: A parallel, main memory relational dbms. *IEEE Trans. on Knowl. and Data Eng.*, 4:541–554, December 1992.

[AXYY06] Pankaj K. Agarwal, Junyi Xie, Jun Yang, and Hai Yu. Scalable continuous query processing by tracking hotspots. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 31–42. VLDB Endowment, 2006.

[AZM99] Michel E. Adiba and José-Luis Zechinelli-Martini. Spatio-temporal multimedia presentations as database objects. In *DEXA '99: Proceedings of the 10th International Conference on Database and Expert Systems Applications*, pages 974–985, London, UK, 1999. Springer-Verlag.

[Ban92] François Bancilhon. The o2 object-oriented database system. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, SIGMOD '92, pages 7–7, New York, NY, USA, 1992. ACM.

[Bar99] Daniel Barbará. Mobile computing and databases-a survey. *IEEE Trans. on Knowl. and Data Eng.*, 11(1):108–117, 1999.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16, New York, NY, USA, 2002. ACM.

[BCG10] Daniele Braga, Stefano Ceri, and Michael Grossniklaus. Join methods and query optimization. In Paul Maglio, Mathias Weske, Jian Yang, and Marcelo Fantinato, editors, *Service-Oriented Computing*, volume 6470 of *Lecture Notes in Computer Science*, pages 715–716. Springer Berlin / Heidelberg, 2010.

[BCGV10] Daniele Braga, Francesco Corcoglioniti, Michael Grossniklaus, and Salvatore Vadacca. Panta rhei: Optimized and ranked data processing over heterogeneous sources. In Paul Maglio, Mathias Weske, Jian Yang, and Marcelo Fantinato, editors, *Service-Oriented Computing*, volume 6470 of *Lecture Notes in Computer Science*, pages 715–716. Springer Berlin / Heidelberg, 2010.

[BCT+97] Elisa Bertino, C, Kian-Lee Tan, Beng Chin Ooi, Ron Sacks-Davis, Justin Zobel, and Boris Shidlovsky. *Indexing Techniques for Advanced Database Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[BD05] Christian Becker and Frank Dürr. On location models for ubiquitous computing. *Personal Ubiquitous Comput.*, 9(1):20–31, 2005.

[Bel04] Khalid Belhajjame. *Defining and orchestrating open services for building distributed information systems*. Thèse de Doctorat, Institut Polytechnique de Grenoble, 2004.

[BFG+08] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a database on s3. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 251–264, New York, NY, USA, 2008. ACM.

[BFMY83] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30:479–513, July 1983.

[BGK+10] Sergey Bykov, Alan Geller, Gabriel Kliot, James Larus, Ravi Pandya, and Jorgen Thelin. Orleans: A framework for cloud computing. Technical Report MSR-TR-2010-159, Microsoft Research, 2010.

[BGS01] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards sensor database systems. In *Proceedings of the Second International Conference on Mobile Data Management*, MDM '01, pages 3–14, London, UK, 2001. Springer-Verlag.

[BJKS06] Rimantas Benetis, S. Jensen, Gytis Karĉiauskas, and Simonas Ŝaltenis. Nearest and reverse nearest neighbor queries for moving objects. *The VLDB Journal*, 15(3):229–249, 2006.

[BKK+01] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Seltzsam, and K. Stocker. Objectglobe: Ubiquitous query processing on the internet. *The VLDB Journal*, 10(1):48–71, 2001.

[BNTW95] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149:3–48, September 1995.

[Bul08] Tevfik Bultan. Service choreography and orchestration with conversations. In *Proceedings of the 19th international conference on Concurrency Theory*, CONCUR '08, pages 2–3, Berlin, Heidelberg, 2008. Springer-Verlag.

[BW01] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30(3):109–120, 2001.

[BZ83] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30:323–342, April 1983.

[CAEA03] Hae Don Chon, Divyakant Agrawal, and Amr El Abbadi. Range and knn query processing for moving objects in grid model. *Mob. Netw. Appl.*, 8(4):401–412, 2003.

[CcC+02] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: a new class of data management applications. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 215–226. VLDB Endowment, 2002.

[CCD+03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: continuous dataflow processing. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668, New York, NY, USA, 2003. ACM.

[CCMC08] Reynold Cheng, Jinchuan Chen, Mohamed Mokbel, and Chi-Yin Chow. Probabilistic verifiers: Evaluating constrained nearest-neighbor queries over uncertain data. *Data Engineering, International Conference on*, 0:973–982, 2008.

[CDF+86] Michael J. Carey, David J. DeWitt, Daniel Frank, M. Muralikrishna, Goetz Graefe, Joel E. Richardson, and Eugene J. Shekita. The architecture of the exodus extensible dbms. In *Proceedings on the 1986 international workshop on Object-oriented database systems*, OODS '86, pages 52–65, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[CDKK85] H.-T. Chou, David J. Dewitt, Randy H. Katz, and Anthony C. Klug. Design and implementation of the wisconsin storage system. *Softw. Pract. Exper.*, 15:943–962, October 1985.

[CDLR90] S. Cluet, C. Delobel, C. Lécluse, and P. Richard. Reloop, an algebra based query language for an object-oriented database system. *Data Knowl. Eng.*, 5:333–352, October 1990.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: a scalable continuous query system for internet databases. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 379–390, New York, NY, USA, 2000. ACM.

[CHCX06] Ying Cai, Kien A. Hua, Guohong Cao, and Toby Xu. Real-time processing of range-monitoring queries in heterogeneous mobile databases. *IEEE Transactions on Mobile Computing*, 5(7):931–942, 2006.

[Cia96] Paolo Ciancarini. Coordination models and languages as software integrators. *ACM Comput. Surv.*, 28(2):300–302, 1996.

[CMTV00] Antonio Corral, Yannis Manolopoulos, Yannis Theodoridis, and Michael Vassilakopoulos. Closest pair queries in spatial databases. *SIGMOD Rec.*, 29(2):189–200, 2000.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13:377–387, June 1970.

[CV04] Christine Collet and Tuyet-Trinh Vu. Qbf: A query broker framework for adaptable query evaluation. In Henning Christiansen, Mohand-Said Hacid, Troels Andreasen, and Hendrik Legind Larsen, editors, *Flexible Query Answering Systems*, volume 3055 of *Lecture Notes in Computer Science*, pages 362–375. Springer Berlin / Heidelberg, 2004.

[CV08a] Víctor Cuevas-Vicenttín. Hybrid query processing through services composition. In *Proceedings of the 2008 EDBT Ph.D. workshop*, Ph.D. '08, pages 75–82, New York, NY, USA, 2008. ACM.

[CV08b] Victor Cuevas-Vicenttin. Towards multi-scale query processing. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering Workshop*, pages 137–144, Washington, DC, USA, 2008. IEEE Computer Society.

[CVCVSI10a] Víctor Cuevas-Vicenttín, Christine Collet, Genoveva Vargas-Solar, and Noha Ibrahim. Hypatia: Flexible processing of hybrid queries using service coordination. In *BDA 2010: 26èmes Journées Bases de Données Avancées*, 2010.

[CVCVSI10b] Víctor Cuevas-Vicenttín, Christine Collet, Genoveva Vargas-Solar, and Noha Ibrahim. The hypatia system for processing hybrid queries. In *Demonstration in BDA 2010: 26èmes Journées Bases de Données Avancées*, 2010.

[CVVSC08] Victor Cuevas-Vicenttín, Genoveva Vargas-Solar, and Christine Collet. Web services orchestration in the webcontent semantic web framework. In *Proceedings of the 2008*

Mexican International Conference on Computer Science, pages 271–282, Washington, DC, USA, 2008. IEEE Computer Society.

[CVVSC+10] Víctor Cuevas-Vicenttín, Genoveva Vargas-Solar, Christine Collet, Noha Ibrahim, and Christophe Bobineau. Coordinating services for accessing and processing data in dynamic environments. In *Proceedings of the 2010 international conference on On the move to meaningful internet systems - Volume Part I*, OTM'10, pages 309–325, Berlin, Heidelberg, 2010. Springer-Verlag.

[CVVSCB09] Victor Cuevas-Vicenttin, Genoveva Vargas-Solar, Christine Collet, and Paolo Bucciol. Efficiently coordinating services for querying data in dynamic environments. *Mexican International Conference on Computer Science*, 0:95–106, 2009.

[CVZMVS06] Víctor Cuevas-Vicenttín, JoséLuis Zechinelli-Martini, and Genoveva Vargas-Solar. Andromeda: Building e-science data integration tools. In Stéphane Bressan, Josef Küng, and Roland Wagner, editors, *Database and Expert Systems Applications*, volume 4080 of *Lecture Notes in Computer Science*, pages 44–53. Springer Berlin / Heidelberg, 2006.

[CWXY05] Hu Cao, Ouri Wolfson, Bo Xu, and Huabei Yin. Mobi-dic: Mobile discovery of local resources in peer-to-peer wireless network. *IEEE Data Eng. Bull.*, 28(3):11–18, 2005.

[dAGB06] Victor Teixeira de Almeida, Ralf Hartmut Güting, and Thomas Behr. Querying moving objects in secondo. In *Proceedings of the 7th International Conference on Mobile Data Management*, MDM '06, pages 47–, Washington, DC, USA, 2006. IEEE Computer Society.

[DD99] Ruxandra Domenig and Klaus R. Dittrich. An overview and classification of mediated query systems. *SIGMOD Rec.*, 28(3):63–72, 1999.

[DFG+03] Klaus R. Dittrich, Hans Fritschi, Stella Gatziu, Andreas Geppert, and Anca Vaduva. Samos in hindsight: experiences in building an active object-oriented dbms. *Inf. Syst.*, 28:369–392, July 2003.

[DFK05] Jens-Peter Dittrich, Peter M. Fischer, and Donald Kossmann. Agile: adaptive indexing for context-aware information filters. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 215–226, New York, NY, USA, 2005. ACM.

[DG01] Klaus R. Dittrich and Andreas Geppert, editors. *Component database systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[DG04] Zhiming Ding and Ralf Hartmut Güting. Managing moving objects on dynamic transportation networks. In *SSDBM '04: Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, page 287, Washington, DC, USA, 2004. IEEE Computer Society.

[DGP+07] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422. www.crdrdb.org, 2007.

[DGR+03] Alan Demers, Johannes Gehrke, Rajmohan Rajaraman, Niki Trigoni, and Yong Yao. The cougar project: a work-in-progress report. *SIGMOD Rec.*, 32(4):53–59, 2003.

[DH04] Amol Deshpande and Joseph M. Hellerstein. Lifting the burden of history from adaptive query processing. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 948–959. VLDB Endowment, 2004.

[DK98] Margaret H. Dunham and Vijay Kumar. Location dependent data and its management in mobile databases. In *DEXA '98: Proceedings of the 9th International Workshop on Database and Expert Systems Applications*, page 414, Washington, DC, USA, 1998. IEEE Computer Society.

[DM06] Peter J. Denning and Thomas W. Malone. Coordination. In Dina Goldin, Scott A. Smolka, and Peter Wegner, editors, *Interactive Computation*, pages 415–439. Springer Berlin Heidelberg, 2006.

[DYM+05] Xiangyuan Dai, Man Lung Yiu, Nikos Mamoulis, Yufei Tao, and Michail Vaitis. Probabilistic spatial queries on existentially uncertain data. In Claudia Bauzer Medeiros, Max Egenhofer, and Elisa Bertino, editors, *Advances in Spatial and Temporal Databases*, volume 3633 of *Lecture Notes in Computer Science*, pages 400–417. Springer, 2005.

[EF91] Max J. Egenhofer and Robert D. Franzosa. Point-set topological spatial relations. *International Journal of Geographical Information Systems*, 5(2):161–174, 1991.

[EIS11] *Enterprise Information Systems*, 2011. Special Issue: Information Systems for Enterprise Integration, Interoperability and Networking: Theory and Applications.

[FGPT07] Elias Frentzos, Kostas Gratsias, Nikos Pelekis, and Yannis Theodoridis. Algorithms for nearest neighbor search on moving object trajectories. *Geoinformatica*, 11(2):159–193, 2007.

[Fie00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. Thèse de Doctorat, 2000.

[FLMS99] Daniela Florescu, Alon Levy, Ioana Manolescu, and Dan Suciu. Query optimization in the presence of limited access patterns. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, SIGMOD '99, pages 311–322, New York, NY, USA, 1999. ACM.

[Fos01] Ian T. Foster. The anatomy of the grid: Enabling scalable virtual organizations. In *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, Euro-Par '01, pages 1–4, London, UK, 2001. Springer-Verlag.

[FSAA01] Hakan Ferhatosmanoglu, Ioana Stanoi, Divyakant Agrawal, and Amr El Abbadi. Constrained nearest neighbor queries. In *SSTD '01: Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, pages 257–278, London, UK, 2001. Springer-Verlag.

[GBE+00] Ralf Hartmut Güting, Michael H. Böhlen, Martin Erwig, Christian S. Jensen, Nikos A. Lorentzos, Markus Schneider, and Michalis Vazirgiannis. A foundation for representing and querying moving objects. *ACM Trans. Database Syst.*, 25:1–42, March 2000.

[GBI+10] David Garlan, Felix Bachmann, James Ivers, Judith Stafford, Len Bass, Paul Clements, and Paulo Merson. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2nd edition, 2010.

[GDF+99] Ralf Hartmut Güting, Stefan Dieker, Claudia Freundorfer, Ludger Becker, and Holger Schenk. Secondo/qp: Implementation of a generic query processor. In *Proceedings of the 10th International Conference on Database and Expert Systems Applications*, DEXA '99, pages 66–87, London, UK, 1999. Springer-Verlag.

[GELA10] Thanaa M. Ghanem, Ahmed K. Elmagarmid, Per-Åke Larson, and Walid G. Aref. Supporting views in data stream management systems. *ACM Trans. Database Syst.*, 35(1):1–47, 2010.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002.

[GL06] Bugra Gedik and Ling Liu. Mobieyes: A distributed location monitoring service using moving location queries. *IEEE Transactions on Mobile Computing*, 5(10):1384–1402, 2006.

[GLP10] Yann Gripay, Frédérique Laforest, and Jean-Marc Petit. A simple (yet powerful) algebra for pervasive environments. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 359–370, New York, NY, USA, 2010. ACM.

[GMWU99] Hector Garcia-Molina, Jennifer Widom, and Jeffrey D. Ullman. *Database System Implementation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[GO03] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, 2003.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25:73–169, June 1993.

[Gri09] Yann Gripay. *A Declarative Approach for Pervasive Environments: Model and Implementation*. Thèse de Doctorat, 2009.

[Gro03] The STREAM Group. Stream: The stanford stream data manager. Technical Report 2003-21, Stanford InfoLab, 2003.

[GS05] Ralf Hartmut Güting and Markus Schneider. *Moving Objects Databases (The Morgan Kaufmann Series in Data Management Systems) (The Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[Gur95] Yuri Gurevich. Evolving algebras 1993: Lipari guide. pages 9–36, 1995.

[Gut84] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[HCH+99] Eric N. Hanson, Chris Carnes, Lan Huang, Mohan Konyala, Lloyd Noronha, Sashi Parthasarathy, J. B. Park, and Albert Vernon. Scalable trigger processing. *Data Engineering, International Conference on*, 0:266, 1999.

[HEOP08] Tan Hanh, Javier-Alfonso Espinosa-Oviedo, and Alberto Portilla. Building reliable mobile services based applications. In *In Proc. of the DS2ME'08 Workshop in conjunction with ICDE'09*, pages 121–128, Cacun, Mexico, avril 2008. IEEE.

[HJ04] Xuegang Huang and Christian S. Jensen. Towards a streams-based framework for defining location-based queries. In Jörg Sander and Mario A. Nascimento, editors, *STDBM*, pages 73–80, 2004.

[HKGT03] Marios Hadjieleftheriou, George Kollios, Dimitrios Gunopulos, and Vassilis J. Tsotras. On-line discovery of dense areas in spatio-temporal databases. In *In Proc. SSTD*, pages 306–324, 2003.

[HLCR03] Dick Hung, Kam-Yiu Lam, Edward Chan, and Krithi Ramamritham. Processing of location-dependent continuous queries on real-time spatial data: The view from retina. In *DEXA '03: Proceedings of the 14th International Workshop on Database and Expert Systems Applications*, page 961, Washington, DC, USA, 2003. IEEE Computer Society.

[Hoh07] Gregor Hohpe. Let's have a conversation. *IEEE Internet Computing*, 11:78–81, May 2007.

[HS05a] Joseph M. Hellerstein and Michael Stonebraker. *Anatomy of a Database System*. The MIT Press, 2005.

[HS05b] Joseph M. Hellerstein and Michael Stonebraker. *What Goes Around Comes Around*. The MIT Press, 2005.

[IB93] Tomasz Imieĺinski and B. R. Badrinath. Data management for mobile computing. *SIGMOD Rec.*, 22(1):34–39, 1993.

[IMI06] Sergio Ilarri, Eduardo Mena, and Arantza Illarramendi. Location-dependent queries in mobile contexts: Distributed processing using mobile agents. *IEEE Transactions on Mobile Computing*, 5:1029–1043, 2006.

[IN02] Tomasz Imielinski and Badri Nath. Wireless graffiti: data, data everywhere. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 9–19. VLDB Endowment, 2002.

[JMS+08] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. Towards a streaming sql standard. *Proc. VLDB Endow.*, 1:1379–1390, August 2008.

[Kah74] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[KD99] Navin Kabra and David J. DeWitt. Opt++ : an object-oriented implementation for extensible database query optimization. *The VLDB Journal*, 8:55–78, April 1999.

[KKL+04] M. Kloppmann, D. König, F. Leymann, G. Pfau, and D. Roller. Business process choreography in websphere: combining the power of bpel and j2ee. *IBM Syst. J.*, 43:270–296, April 2004.

[KKR07] Hans-Peter Kriegel, Peter Kunath, and Matthias Renz. Probabilistic nearest-neighbor query on uncertain objects. In *DASFAA'07: Proceedings of the 12th international conference on Database systems for advanced applications*, pages 337–348, Berlin, Heidelberg, 2007. Springer-Verlag.

[Knu97] Donald E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[Kos08] Donald Kossmann. Building web applications without a database system. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, EDBT '08, pages 3–3, New York, NY, USA, 2008. ACM.

[Kri06] Saileshwar Krishnamurthy. *Shared query processing in data streaming systems*. Thèse de Doctorat, Berkeley, CA, USA, 2006. AAI3253935.

[KSKR05] Richard Kuntschke, Bernhard Stegmaier, Alfons Kemper, and Angelika Reiser. Streamglobe: processing and sharing data streams in grid-based p2p infrastructures. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 1259–1262. VLDB Endowment, 2005.

[Lar88] Per-Åke Larson. Dynamic hash tables. *Commun. ACM*, 31:446–457, April 1988.

[LCH+05] Feifei Li, Dihan Cheng, Marios Hadjieleftheriou, George Kollios, and Shang-Hua Teng. On trip planning queries in spatial databases. In Claudia Bauzer Medeiros, Max J. Egenhofer, and Elisa Bertino, editors, *SSTD*, volume 3633 of *Lecture Notes in Computer Science*, pages 273–290. Springer, 2005.

[LMC+09] Steven Lynden, Arijit Mukherjee, Hu Alastair C., Alvaro A. A. Fernandes, Norman W. Paton, Rizos Sakellariou, and Paul Watson. The design and implementation of ogsa-dqp: A service-based distributed query processor. *Future Gener. Comput. Syst.*, 25:224–236, March 2009.

[LPM02] Iosif Lazaridis, Kriengkrai Porkaew, and Sharad Mehrotra. Dynamic queries over mobile objects. In *EDBT '02: Proceedings of the 8th International Conference on Extending Database Technology*, pages 269–286, London, UK, 2002. Springer-Verlag.

[LPT99] Ling Liu, Calton Pu, and Wei Tang. Continual queries for internet scale event-driven information delivery. *IEEE Trans. on Knowl. and Data Eng.*, 11(4):610–628, 1999.

[LWX08] Yan Luo, Ouri Wolfson, and Bo Xu. Mobile local search via p2p databases. In *Portable 2008: Proceedings of the 2nd IEEE International Interdisciplinary Intersociety Conference on Portable Information Devices (PIDs)*, pages 1–6, 2008.

[MA07] M.F. Mokbel and W.G. Aref. Location-aware query processing and optimization. In *Mobile Data Management, 2007 International Conference on*, page 229, May 2007.

[MA08] Mohamed F. Mokbel and Walid G. Aref. Sole: scalable on-line execution of continuous queries on spatio-temporal data streams. *The VLDB Journal*, 17(5):971–995, 2008.

[Mat01] Friedmann Mattern. Ubiquitous computing. Presentation, 2001.

[MBFS01] Ioana Manolescu, Luc Bouganim, Françoise Fabret, and Eric Simon. Efficient Data and Program Integration Using Binding Patterns. Research Report RR-4239, INRIA, 2001. Projet CARAVEL.

[MFHH05] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[MG01] Ioana Manolescu-Goujot. *Optimization techniques for querying heterogeneous distributed data sources*. Thèse de Doctorat, Université de Versailles Saint-Quentin-en-Yvelines, 2001.

[MGA03] Mohamed F. Mokbel, Thanaa M. Ghanem, and Walid G. Aref. Spatio-temporal access methods. *IEEE Data Engineering Bulletin*, 26:40–49, 2003.

[MHMM05] N. Marsit, A. Hameurlain, Z. Mammeri, and F. Morvan. Query processing in mobile environments: A survey and open problems. In *DFMA '05: Proceedings of the First International Conference on Distributed Frameworks for Multimedia Applications*, pages 150–157, Washington, DC, USA, 2005. IEEE Computer Society.

[MPKS04] Antonis Markopoulos, Panagiotis Pissaris, Sofoklis Kyriazakos, and Efstathios D. Sykas. Efficient location-based hard handoff algorithms for cellular systems. In Nikolas Mitrou, Kimon Kontovasilis, George N. Rouskas, Ilias Iliadis, and Lazaros Merakos, editors, *NETWORKING 2004*, volume 3042 of *Lecture Notes in Computer Science*, pages 476–489. Springer Berlin / Heidelberg, 2004.

[MS05] Hoda Mokhtar and Jianwen Su. A query language for moving object trajectories. In *SSDBM'2005: Proceedings of the 17th international conference on Scientific and statistical database management*, pages 173–182, Berkeley, CA, US, 2005. Lawrence Berkeley Laboratory.

[MXA04] Mohamed F. Mokbel, Xiaopeing Xiong, and Walid G. Aref. Sina: scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 623–634, New York, NY, USA, 2004. ACM.

[MXHA05] Mohamed F. Mokbel, Xiaopeng Xiong, Moustafa A. Hammad, and Walid G. Aref. Continuous query processing of spatio-temporal data streams in place. *Geoinformatica*, 9:343–365, December 2005.

[Naa99] Hubert Naacke. *Modèles de coût pour médiateurs de bases de données hétérogènes*. Thèse de Doctorat, Université de Versailles Saint-Quentin-en-Yvelines, 1999.

[NDAM10] Long-Van Nguyen-Dinh, Walid G. Aref, and Mohamed F. Mokbel. Spatio-temporal access methods: Part 2 (2003 - 2010). *IEEE Data Eng. Bull.*, 33(2):46–55, 2010.

[NST99] Mario A. Nascimento, Jefferson R. O. Silva, and Yannis Theodoridis. Evaluation of access structures for discretely moving points. In *STDBM '99: Proceedings of the International Workshop on Spatio-Temporal Database Management*, pages 171–188, London, UK, 1999. Springer-Verlag.

[Pap04] Apostolos N. Papadopoulos. *Nearest Neighbor Search: A Database Perspective*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004.

[PB99] Evaggelia Pitoura and Bharat Bhargava. Data consistency in intermittently connected distributed systems. *IEEE Trans. on Knowl. and Data Eng.*, 11(6):896–915, 1999.

[PFJ+01] ao Pereira, Jo Françoise Fabret, Hans-Arno Jacobsen, François Llirbat, and Dennis Shasha. Webfilter: A high-throughput xml-based publish and subscribe system. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 723–724, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[PJ99] Dieter Pfoser and Christian S. Jensen. Capturing the uncertainty of moving-object representations. In *SSD '99: Proceedings of the 6th International Symposium on Advances in Spatial Databases*, pages 111–132, London, UK, 1999. Springer-Verlag.

[PLR10] Loïc Petit, Cyril Labbé, and Claudia Lucia Roncancio. An algebric window model for data stream management. In *Proceedings of the Ninth ACM International Workshop on Data Engineering for Wireless and Mobile Access*, MobiDE '10, pages 17–24, New York, NY, USA, 2010. ACM.

[Pol06] Thomas Pole. Software Reuse: History 1980 to 2005. Presentation, 2006.

[PS97] Evaggelia Pitoura and George Samaras. *Data Management for Mobile Computing*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[PS07] Kostas Patroumpas and Timos Sellis. Semantics of spatially-aware windows over streaming moving objects. In *Proceedings of the 2007 International Conference on Mobile Data Management*, pages 52–59, Washington, DC, USA, 2007. IEEE Computer Society.

[PT97] Dimitris Papadias and Yannis Theodoridis. Spatial relations, minimum bounding rectangles, and spatial data structures. *International Journal of Geographical Information Science*, 11(2):111–138, 1997.

[PXK+02] Sunil Prabhakar, Yuni Xia, Dmitri V. Kalashnikov, Walid G. Aref, and Susanne E. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Transactions on Computers*, 51:1124–1140, 2002.

[PZMT03] Dimitris Papadias, Jun Zhang, Nikos Mamoulis, and Yufei Tao. Query processing in spatial network databases. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 802–813. VLDB Endowment, 2003.

[RDH03] Vijayshankar Raman, A. Deshpande, and J.M. Hellerstein. Using state modules for adaptive query processing. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 353–364, 2003.

[RKS88] Mark A. Roth, Herry F. Korth, and Abraham Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Trans. Database Syst.*, 13:389–417, October 1988.

[ROH99] Mary Tork Roth, Fatma Ozcan, and Laura M. Haas. Cost models do matter: Providing cost information for diverse data sources in a federated system. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 599–610, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[Rot04] Jöorg Roth. *Data collection*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[RSSG03] Philippe Rigaux, Michel Scholl, Luc Segoufin, and Stéphane Grumbach. Building a constraint-based spatial database system: model, languages, and implementation. *Information Systems*, 28(6):563 – 595, 2003.

[RSU95] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering queries using templates with binding patterns (extended abstract). In *PODS '95: Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 105–112, New York, NY, USA, 1995. ACM.

[SAC+79] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34, New York, NY, USA, 1979. ACM.

[SC05] Michael Stonebraker and Ugur Cetintemel. "one size fits all": An idea whose time has come and gone. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 2–11, Washington, DC, USA, 2005. IEEE Computer Society.

[SDK01] Ayse Y. Seydim, Margaret H. Dunham, and Vijay Kumar. Location dependent query processing. In *MobiDe '01: Proceedings of the 2nd ACM international workshop on Data engineering for wireless and mobile access*, pages 47–53, New York, NY, USA, 2001. ACM.

[Ses96] Srinivasan Seshan. Low-latency handoff for cellular data networks. Rapport technique, Berkeley, CA, USA, 1996.

[SHWK76] Michael Stonebraker, Gerald Held, Eugene Wong, and Peter Kreps. The design and implementation of ingres. *ACM Trans. Database Syst.*, 1:189–222, September 1976.

[SL90] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 22:183–236, September 1990.

[SR01] Zhexuan Song and Nick Roussopoulos. K-nearest neighbor search for moving query point. In *SSTD '01: Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, pages 79–96, London, UK, 2001. Springer-Verlag.

[SS86] H. J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137 – 147, 1986.

[SWCD97a] A. Prasad Sistla, Ouri Wolfson, Sam Chamberlain, and Son Dao. Modeling and querying moving objects. In *ICDE '97: Proceedings of the Thirteenth International Conference on Data Engineering*, pages 422–432, Washington, DC, USA, 1997. IEEE Computer Society.

[SWCD97b] A. Prasad Sistla, Ouri Wolfson, Sam Chamberlain, and Son Dao. Querying the uncertain position of moving objects. In *Temporal Databases, Dagstuhl*, pages 310–337, 1997.

[SXI01] Jianwen Su, Haiyan Xu, and Oscar H. Ibarra. Moving objects: Logical relationships and queries. In *SSTD '01: Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, pages 3–19, London, UK, 2001. Springer-Verlag.

[tBBG07] Maurice ter Beek, Antonio Bucchiarone, and Stefania Gnesi. Web service composition approaches: From industrial standards to formal methods. In *Proceedings of the Second International Conference on Internet and Web Applications and Services*, pages 15–, Washington, DC, USA, 2007. IEEE Computer Society.

[TD04] Marie Thilliez and Thierry Delot. Evaluating location dependent queries using islands. In Félix F. Ramos, Herwig Unger, and Victor Larios, editors, *ISSADS*, volume 3061 of *Lecture Notes in Computer Science*, pages 125–136. Springer, 2004.

[The03] Yannis Theodoridis. Ten benchmark database queries for location-based services. *The Computer Journal*, 46:713–725, 2003.

[TTPM02] P. A. Tucker, K. Tufte, V Papadimos, and D. Maier. Nexmark - a benchmark for querying data streams. Rapport technique, 2002.

[TVM00] Theodoros Tzouramanis, Michael Vassilakopoulos, and Yannis Manolopoulos. Overlapping linear quadtrees and spatio-temporal query processing. *Comput. J.*, 43(4):325–343, 2000.

[TWHC04] Goce Trajcevski, Ouri Wolfson, Klaus Hinrichs, and Sam Chamberlain. Managing uncertainty in moving objects databases. *ACM Trans. Database Syst.*, 29(3):463–507, 2004.

[TXC07] Yufei Tao, Xiaokui Xiao, and Reynold Cheng. Range search on multidimensional uncertain data. *ACM Trans. Database Syst.*, 32(3):15, 2007.

[Val93] Patrick Valduriez. Parallel database systems: open problems and new issues. *Distrib. Parallel Databases*, 1:137–165, April 1993.

[vJLL00] Simonas Šaltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the positions of continuously moving objects. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, pages 331–342, New York, NY, USA, 2000. ACM.

[VSIC+10] Genoveva Vargas-Solar, Noha Ibrahim, Christine Collet, Michel Adiba, Jean-Marc Petit, and Thierry Delot. *Querying Issues in Pervasive Environments*. IGI Global, 2010.

[VW01] Michalis Vazirgiannis and Ouri Wolfson. A spatiotemporal model and language for moving objects on road networks. In *SSTD '01: Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, pages 20–35, London, UK, 2001. Springer-Verlag.

[WA91] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *PDIS '91: Proceedings of the first international conference on Parallel and distributed information systems*, pages 68–77, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[Wei91] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3):66–75, September 1991.

[Wei07] Gerhard Weikum. Db&ir: both sides now. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 25–30, New York, NY, USA, 2007. ACM.

[Wie92] Gio Wiederhold. Mediators in the architecture of future information systems. *Computer*, 25(3):38–49, 1992.

[WM05] Ouri Wolfson and Eduardo Mena. Applications of moving objects databases. In Yannis Manolopoulos, Apostolos Papadopoulos, and Michael Vassilakopoulos, editors, *Spatial Databases*, pages 186–203. Idea Group, 2005.

[WYCT08] Wei Wu, Fei Yang, Chee Yong Chan, and Kian-Lee Tan. Continuous reverse k-nearest-neighbor monitoring. In *MDM '08: Proceedings of the The Ninth International Conference on Mobile Data Management*, pages 132–139, Washington, DC, USA, 2008. IEEE Computer Society.

[XECA07] Xiaopeng Xiong, Hicham G. Elmongui, Xiaoyong Chai, and Walid G. Aref. Place*: A distributed spatio-temporal data stream management system for moving objects. *Mobile Data Management, IEEE International Conference on*, 0:44–51, 2007.

[XJ07] Zhengdao Xu and Arno Jacobsen. Adaptive location constraint processing. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 581–592, New York, NY, USA, 2007. ACM.

[XOW04] Bo Xu, Aris Ouksel, and Ouri Wolfson. Opportunistic resource exchange in inter-vehicle ad-hoc networks. *Mobile Data Management, IEEE International Conference on*, 0:4, 2004.

[XW03] Bo Xu and Ouri Wolfson. Time-series prediction with applications to traffic and moving objects databases. In *MobiDe '03: Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access*, pages 56–60, New York, NY, USA, 2003. ACM.

[Yan81] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB '1981: Proceedings of the seventh international conference on Very Large Data Bases*, pages 82–94. VLDB Endowment, 1981.

[YK06] Byunggu Yu and Seon Ho Kim. Interpolating and using most likely trajectories in moving-objects databases. In Stéphane Bressan, Josef Küng, and Roland Wagner, editors, *DEXA*, volume 4080 of *Lecture Notes in Computer Science*, pages 718–727. Springer, 2006.

[ZL01] Baihua Zheng and Dik Lun Lee. Semantic caching in location-dependent query processing. In *SSTD '01: Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, pages 97–116, London, UK, 2001. Springer-Verlag.

[ZLLL07] Baihua Zheng, Wang-Chien Lee, and Dik Lun Lee. On searching continuous k nearest neighbors in wireless data broadcast systems. *IEEE Transactions on Mobile Computing*, 6(7):748–761, 2007.

[ZXW08] Xinjuan Zhu, Bo Xu, and Ouri Wolfson. Spatial queries in disconnected mobile networks. In *GIS '08: Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*, pages 1–10, New York, NY, USA, 2008. ACM.

[ZZP+03] Jun Zhang, Manli Zhu, Dimitris Papadias, Yufei Tao, and Dik Lun Lee. Location-based spatial queries. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 443–454, New York, NY, USA, 2003. ACM.

Annex

ANNEX A

HSQL syntax

Listing A.1: HSQL grammar

```
query :       select_clause from_clause opt_where_clause? ';'

select_clause :
              SELECT DISTINCT non_mt_projterm_list |
              SELECT non_mt_projterm_list |
              SELECT DISTINCT '*' |
              SELECT '*' ;

non_mt_projterm_list :  projterm ( ',' projterm )* ;

projterm :  term;

from_clause : FROM non_mt_relation_list ;

non_mt_relation_list :  relation_variable ( ',' relation_variable )* ;

relation_variable :
             ID |
             ID AS ID |
             ID '[' window_type ']' |
             ID '[' window_type ']' AS ID ;

window_type :
             RANGE INT |
             ROWS INT |
             RANGE UNBOUNDED ;

opt_where_clause :  WHERE non_mt_cond_list ;
```

```
non_mt_cond_list :  condition ( AND condition )* ;

condition :
            term LT term |
            term LE term |
            term GT term |
            term GE term |
            term EQ term |
            term NE term;

term  :
            attr |
            funCall |
            INT |
            REAL |
            TEXT_STRING;

fterm :
            attr |
            INT |
            REAL |
            TEXT_STRING;

attr    :
            ID '.' ID |
            ID;

ftermList :      fterm ( ',' fterm )* ;

funCall   :      attr '(' ftermList ')' ;


LETTER   :    'a'..'z' |'A'..'Z' ;
DIGIT    :    '0'..'9' ;
NUM      :    DIGIT+ ;
INT      :    ('-' | '+' )? NUM ;
REAL     :    (INT '.' NUM) ( ('E'|'e') INT )? ;
ID       :    LETTER(LETTER | DIGIT | '_' | '$')* ;
WS       :    (' '|'\r'|'\n'|'\t')+ ;
```

```
TEXT_STRING : ('\''   (  ~('\'' | '\r' | '\n' )  )*   '\'' ) ;

AND       :   'AND' | 'and' ;
SELECT    :   'SELECT' | 'select';
FROM      :   'FROM' | 'from';
WHERE     :   'WHERE' | 'where';
DISTINCT  :   'DISTINCT' | 'distinct';
AS        :   'AS' | 'as' ;
RANGE     :   'RANGE' | 'range' ;
UNBOUNDED :   'UNBOUNDED' | 'unbounded' ;
ROWS      :   'ROWS' | 'rows' ;

LT  :  '<' ;
LE  :  '<=' ;
GT  :  '>' ;
GE  :  '>=' ;
EQ  :  '=' ;
NE  :  '!=' | '<>';
```

Annex B
ASM syntax

Listing B.1: ASM grammar

```
asm :               initDefinition rule+ EOF ;

initDefinition :    INIT '{'
                            vardef*
                    '}' ;

vardef :            ID ASSIGN atom ;


rule :   ID ASSIGN expr
       | IF ( expr ) THEN BEGIN c=rule+ END (ELSE BEGIN el=rule+ END)?
       | PAR rule+ ENDPAR
       | SEQ rule+ ENDSEQ
       | ITERATE expr rule
       | PRINT^ expr
       | SKIP
       ;

call : qid '(' (expr (',' expr )*)? ')' ;

qid : ID DOT ID ;

expr : relexpr ( (AND|OR) relexpr)* ;

relexpr : addexpr ( (LT|LE|GT|GE|EQ|NE) addexpr)? ;

addexpr : mulexpr (('+'|'-') mulexpr)* ;

mulexpr : unaryexpr (('*'|'/'|'%') unaryexpr)* ;
```

```
unaryexpr :   '-' atom
            | NOT atom
            | atom
            ;

atom :
           INT
         | CHAR
         | REAL
         | STRING
         | ID
         | NULL
         | TRUE
         | FALSE
         | '(' expr ')'
         | qid
         | call
         ;


LETTER :             'a'..'z' |'A'..'Z' ;

DIGIT :              '0'..'9' ;

IntegerNumber :      '0'
                   | '1'..'9' (DIGIT)*
                   ;

RealNumber :         (DIGIT)+ '.' (DIGIT)+ Exponent? ;

Exponent :           ( 'e' | 'E' ) ( '+' | '-' )? ( DIGIT )+ ;

ID :                 LETTER (LETTER | DIGIT | '_')*  ;

CHAR :  '\'' . '\'' ;

STRING :  '\"' .* '\"' ;
```

```
INT :  IntegerNumber ;

REAL :  RealNumber ;

WS :  ( ' ' | '\t' | '\r' | '\n' )+ ;

SL_COMMENT :  '#' ~('\r'|'\n')* ;
```

Annex C

Semantics of ASM-based Service Coordination Model

We present in detail our service coordination model based on abstract state machines (ASMs) and its semantics. First, we present a general overview of service coordination in Section C.1, followed by a characterization of our service coordination model in Section C.2. Afterwards, we describe the basic concepts of ASMs in Section C.3. Subsequently, in Section C.4 the restrictions and extensions adopted to create an ASM-based service coordination model and language are presented, along with the language semantics.

C.1 Overview of service coordination

Coordination concerns multiple agents acting together (interacting) to achieve a common goal [DM06]. The goal in the case of service coordination is to perform a computation, which can have effects in the real world as with business processes, or generate a desired result related to a calculation or query. Service coordination occurs in two main forms, both based on message exchanges and that can be complementary [Hoh07, Bul08]: choreography and orchestration.

A *choreography* specifies fundamentally a protocol, which implies the valid message exchanges among the various participant services. Therefore, a choreography is not necessarily executable and represents the viewpoint of a neutral observer, since no single participant is in control. Services that implement such protocol need to be built or found to enact the choreography. An *orchestration*, on the other hand, defines an executable process that specifies how and when message exchanges with other participants occur, in addition to possibly additional execution logic. Consequently, an orchestration represents the viewpoint of a entity in control of the process. If the process is exposed as a service, we also speak of service composition.

The W3C proposed standard for Web Services choreography is WS-CDL[1], while WS-BPEL[2], proposed by OASIS, is the leading standard for Web Services orchestration. Both are XML languages, which we believe can be disadvantageous for the development of a service-based query processor.

[1] http://www.w3.org/TR/ws-cdl-10/
[2] http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

Our reasoning is based on our experience in the development of a Web Services orchestration engine in the context of the WebContent Project[3], which represents a separate line of research from the present work that has its origin in [Bel04]; the details of the orchestration engine are presented in [CVVSC08].

We adopted a simplified form of the XML Process Definition Language (XPDL)[4] to specify a workflow model. It is standardized in terms of general syntax, while the engine enforces our particular semantics. Although XML languages can benefit from GUI tools, we find that they are cumbersome for conventional coding, which is fundamental for the development of a query processor. Nevertheless, XML documents can facilitate the expression and processing of information needs when complemented with service calls, as is the case with ActiveXML [ABM08]; that is, however, a different querying approach that the one we take favoring a declarative language.

We also noted significant differences between a service orchestration used for business process management and what is required for query processing. A business process can take weeks or more to be completed, with related periods of inactivity, whereas queries may be snapshot and short-lived or continuous and remain active. Also, while query results are transient, business processes require persistency as well as concurrency management. In our implementation we addressed these requirements with Java EE, as has been done for other orchestration engines (e.g., [KKL+04]).

The operators in a query coordination communicate via input/output operations supported by asynchronous queues, which can be said to represent a choreography; simple from the point of view of its execution, but not from its derivation. To build composite computation services to evaluate the query operators, on the other hand, an orchestration model and language with the following three main features (in decreasing order of importance) is required.

(i) Turing completeness, since we need to compute a great variety of query operators. This is relatively easy to satisfy, for example, finite state automata with FIFO queues are Turing complete [BZ83]. (ii) Ease of use for developers and sophisticated users. (iii) Formal semantics, intrinsic or specified in some formalism. Commonly used formalisms in service coordination include automata, Petri nets, and process algebras like the π-calculus (a discussion of them is given in [tBBG07]).

We adopt Abstract State Machines (ASMs) [Gur95] since we find that they satisfy the above requirements the best. Consequently, we employ them for specifying composite computation services as described in Chapter 4, where we also presented their workflow graph representation. In this appendix we present in more detail this formalism, as applied for service coordination, and its semantics. First we present a characterization of our overall approach in terms of the specific problem that we address.

C.2 Coordination model characterization

To clarify and summarize our service coordination approach we employ the constructive method to specify coordination models given in [Cia96], where coordination models are presented in terms of the three main types of components that conform them: coordination entities, media, and rules. The components

[3] http://www.webcontent.fr/
[4] http://www.wfmc.org/xpdl.html

	Inter-operator	*Intra-operator*
Coordination entities	• Operators • Processes	• Computation services • Operator state
Coordination media	• Asynchronous queues • Messaging infrastructure	• Shared memory • Service middleware
Coordination rules	• Query workflow • Scheduling policy	• ASM-based orchestration language

Figure C.1: Components of our coordination model

of our coordination model following this scheme are presented in Figure C.1, subdivided into two categories, inter-operator and intra-operator; corresponding to the coordination of the overall evaluation plan and of each individual operator, respectively. We discuss next each type of component and how they integrate to our overall approach.

- **Coordination entities** are the entity types that need to be coordinated, in general terms they may involve objects, processes, or even users. For query processing, concerning inter-operator coordination the entities that need to be coordinated are the operators and the processes (or threads) in charge of their execution.

  Regarding intra-operator execution, the coordination entities are primarily computation services, such as indexing, storage, or calculation services that are used to evaluate a particular operator. Operators can also have program state associated with them, such as the data items used in composite computation services.

- **Coordination media** enable communication among the participating agents, also facilitating their synchronization. For inter-operator coordination, the main coordination media are asynchronous queues, since they support input/output operations. To support distributed execution, we can incorporate a messaging infrastructure capable of operating in dynamic environments (e.g., a mesh network).

  The internal execution of operators relying on computation services requires an appropriate middleware that supports the service standards in use. In addition, assuming local execution, the data items associated with a given operator state can be coordinated through shared memory.

- **Coordination rules** describe how agents coordinate themselves through the coordination media and using several coordination primitives. These can be embedded in an architecture or expressed

in a coordination language. For inter-operator coordination they are expressed in the query workflow, which specifies the operators to be executed and the dependencies between them. The execution of each operator is in turn controlled by a scheduler, that determines when and for how much time a certain operator will execute.

For intra-operator coordination, it is necessary to specify the logic of each of the operators that is evaluated with computation services. This logic, as discussed previously, is specified by ASMs defining a service orchestration. Each of the ASM constructs specifies and associated rule.

C.3 Basic ASM concepts

The service coordination model that we use to build composite computation services, as described in Chapter 4, is based on the abstract state machines formalism, ASMs for short. ASMs were introduced for, in contrast with Turing machines, specify algorithms at an arbitrary level of abstraction. They have the characteristic that they capture mathematically the fundamental notions of computing, yet they can be understood as pseudocode. In our discussion of ASMs as adopted for our service coordination model, we omit their full mathematical definition and describe instead the fundamental concepts they are based upon as well as their operation.

C.3.1 Abstract states

ASMs act upon data represented by abstract elements of universes (also called domains), which are associated with operations represented by functions. In this manner, the state of an ASM represents an algebraic structure, hence their original denomination of evolving algebras. The function names form a *vocabulary* Σ. Each function has an arity representing the number of arguments the function takes. For example, we can consider the universe $\mathbb{N}$ of natural numbers and the successor ($S : N \rightarrow N$) and addition ($+ : N \times N \rightarrow N$) functions, with arities of 1 and 2, respectively. Functions with zero arity are called *nullary*. In addition, functions can be static or dynamic, depending on whether their interpretations (see below) are subject to change. Static nullary functions are also referred to as constants.

At a given time the functions of the vocabulary Σ are associated with a *state* $\mathfrak{S}$, which consists of a set X including elements from all of the domains of the functions in Σ, therefore called the *superuniverse* of $\mathfrak{S}$, and also a series of *interpretations* of the function names in Σ. An interpretation of a function f of arity n, denoted by $f^{\mathfrak{S}}$ is a function from X^n to X. For example, an interpretation of the function $+ : N \times N \rightarrow N$ corresponds to the usual addition function defined as follows: $a + 0 = a, a + S(b) = S(a + b)$.

The function names in Σ are used to build expressions referred to as *terms* that are generated as follows

1. Variables $v_0, v_1, ..., v_n$ are terms

2. If f is an n-ary function name of Σ and $t_1, ..., t_n$ are terms, then $f(t_1, ..., t_n)$ is a term.

Some example terms are $x * (r(3) + 1)$ and $y - (s * 2)$, where x and y are variables and r and s are functions. Terms can be given an interpretation that depends only on the value of the variables. For this purpose, we assume a variable assignment $\mathfrak{V}$ which assigns to each variable v_i an element of the superuniverse of $\mathfrak{S}$. Then the interpretation of terms t in state $\mathfrak{S}$ under $\mathfrak{V}$ denoted by $[\![t]\!]^{\mathfrak{S}}_{\mathfrak{V}}$ is defined as follows

1. $[\![v_i]\!]^{\mathfrak{S}}_{\mathfrak{V}} := \mathfrak{V}(v_i)$
2. $[\![f(t_1, ..., t_n)]\!]^{\mathfrak{S}}_{\mathfrak{V}} := f^{\mathfrak{S}}([\![t_1]\!]^{\mathfrak{S}}_{\mathfrak{V}}, ..., [\![t_n]\!]^{\mathfrak{S}}_{\mathfrak{V}})$

Intuitively, the variables take the value specified by the assignment; whereas the dynamic functions take the value obtained by first obtaining the value of each of the terms, and then using those values as the arguments for the function. Abstract states in ASMs are subject to changes in the interpretation of the dynamic functions of the underlying signature, these changes arise from transition rules, which are described next.

C.3.2 Transition rules

Transition rules are expressions over the vocabulary Σ that generate *updates* over the state $\mathfrak{S}$. An update over a state $\mathfrak{S}$ is represented by a triple $(f, (a_1, ..., a_n), b)$ where f is an n-ary function name from Σ, and $a_1, ..., a_n$ as well as b are elements from the superuniverse of $\mathfrak{S}$. Under such an update, the interpretation of the function f in state $\mathfrak{S}$ is changed so that it produces the value b under the arguments $a_1, ..., a_n$. Updates can be generated from rules of the following form

1. Skip rule:
   skip
   Perform no action.

2. Update rule:
   $f(t_1, ..., t_n) := s$
   Where f is a n-ary dynamic function name from Σ and $t_1, ..., t_n$ as well as s are terms over Σ. The value of the function f for the arguments obtained from $t_1, ..., t_n$ (if f is non-nullary) is updated to s in the next state.

3. Block rule:
   $R\ S$
   Rules R and S are executed in parallel generating a single set of updates.

4. Conditional rule: **if** ϕ **then** R **else** S If the condition ϕ (a basic first-order logic formula) is true, then execute R, otherwise execute S. The **else** clause may be omitted.

An ASM M consisting of a finite set of transition rules T over the vocabulary Σ, a variable assignment $\mathfrak{V}$, and an initial state $\mathfrak{S}$ begins its execution when the transition rules are executed in parallel generating an update set U. The update set U is *consistent* if there are no updates of the form $(f, (a_1, ..., a_n), b)$

and $(f, (a_1, ..., a_n), c)$ where $b \neq c$. If the update set U is consistent, then a new state $\mathfrak{S}'$ is generated having the same superuniverse as $\mathfrak{S}$ but reflecting the new interpretations of the updated functions, i.e., if $(f, (a_1, ..., a_n), b) \in U$ then $f^{\mathfrak{S}'}(a_1, ..., a_n) = b$. For a particular rule R, its interpretation $[\![R]\!]$ is also identified with the update set it generates.

The execution process then proceeds analogously from the state $\mathfrak{S}'$ generating a new update set U', and stops when the generated update set is empty. Thus a *run* is defined over the ASM M consisting of a series of states $\mathfrak{S}_0, \mathfrak{S}_1, \mathfrak{S}_2, ...$ over Σ where $\mathfrak{S}_0 = \mathfrak{S}$.

C.4 ASM-based service coordination model

It has been demonstrated that ASMs as presented previously suffice to specify any sequential algorithm. However, their applicability to yield executable specifications in many contexts is limited for two main reasons. First, they lack explicitly some of the widely used constructs from structured programming, such as iteration. Second, their elementary constructs in their full generality differ in several aspects from those of typical programming languages.

The particular context we are interested in is service coordination. By coordination we understand the management of the dataflow and controlflow dependencies between several software entities accessible as services, in order to accomplish a particular task. The use of services has as a consequence that service-oriented applications are constructed at a higher level of abstraction than traditional applications. By adapting the ASM formalism through some extensions as well as restrictions, some of which are known in literature, we can retain its abstraction and expressiveness capabilities and use it as a domain-specific language for service coordination.

Next we present the extensions and restrictions that we regard as essential for the adoption of ASMs as a model and language for service coordination.

C.4.1 Restrictions on basic ASMs

We introduce two restrictions over basic ASMs as defined previously. The objective of these restrictions is to yield a language more familiar to most developers accustomed to traditional programming languages. In addition, the implementation of the resulting language is simplified, while at the same time opportunities arise to improve efficiency.

1. All dynamic functions are nullary
   Only update rules of the form $f := g(t_1, ..., t_n)$ are allowed. Where f and g are function names and $t_1, ..., t_n$ terms over the vocabulary Σ. Such nullary dynamic functions f are also referred to as *locations*.

2. Use of ground terms in condition formulas
   Only ground terms, i.e. terms with no variables and not subject to quantifiers, may be used in conditional rules. Concretely, in conditional rues of the form **if** ϕ **then** R **else** S, logical formulas

in ϕ are defined using ground terms and the $\vee$, $\wedge$, and $\neg$ connectors; representing disjunction, conjunction, and negation, respectively. Ground terms may also include the comparison operators $>$, $<$, $\leq$, $\geq$, $=$, and $\neq$.

The first restriction makes dynamic functions analogous to variables in traditional programming languages. Additionally, dynamic functions may no longer represent a conceptually unbounded memory storage. The second restriction serves to relate the logic of the specified algorithm to a concrete implementation, whose behavior the user can be expected to be familiar with.

In this manner we guarantee ASM specifications that will be more similar to the programs developers are accustomed to, without sacrificing expressivity in a significant manner. However, an analysis of the full consequences of these restrictions over the known properties of the ASM formalism is beyond the scope of this work.

C.4.2 Extensions on basic ASMs

In addition to the aforementioned restrictions on basic ASMs, we adopt the following extensions in order to derive a more appropriate service coordination language. The first two are only syntactical conventions to represent control structures that are inconvenient to express in the original formalism. The third results from the reinterpretation of computation from the evaluation of computable functions to a interactive process among distinct entities. These three extensions have been published before in the literature. Finally, we present an extension to incorporate services.

C.4.2.1 Sequential composition

Rules in ASMs are evaluated in parallel by default, since sequential execution as occurs in monoprocessor computers is really a machine restriction and not intrinsic to the logic of the algorithms. However, there are many cases in which programmers contemplate performing a certain action and storing its resulting value, in order to then use it in a subsequent step. To facilitate the specification of algorithms in sequential steps, a sequential composition construct has been proposed for ASMs.

The sequential composition R **seq** S of two rules R and S represents a single rule that is interpreted as follows.

$$[\![R \textbf{ seq } S]\!]^{\mathfrak{S}} = [\![R]\!]^{\mathfrak{S}} \oplus [\![S]\!]^{\mathfrak{S}'}$$

Where $\mathfrak{S}'$ is an intermediate state obtained from applying the update set generated from R in state $\mathfrak{S}$, if the updates of R are consistent. The operation $U \oplus V$ denotes the merging of the update set V with the update set U, where updates in V overwrite updates in U. Merging occurs only if U is consistent, thus

$$U \oplus V = \begin{cases} \{(loc, val) | (loc, val) \in U \wedge loc \notin locs(V)\} \cup V, & consistent(U) \\ U, & otherwise \end{cases}$$

Where $locs(V)$ denotes the locations refered to in the update set V.

Intuitively the updates made by R are visible to S. Afterwards both update sets are merged to generate a single update set, giving preference to the updates from S. The sequential composition of rules is associative and thus can be generalized to an arbitrary number of rules. For this reason, in our language syntax we adopt the notation **seq** R_1 R_2 ... R_n **endseq** to denote the sequential compositions of rules R_1 R_2 ... R_n.

C.4.2.2 Iteration

Iteration is one of the most widely used control structures in programming languages. It can be succinctly specified for ASMs with the use of the sequential composition construct just described. In particular, iteration is defined in terms of the execution of a certain rule successively as follows

$$R^n = \begin{cases} \textbf{skip}, & n = 0 \\ R^{n-1}\textbf{seq}R, & n > 0 \end{cases}$$

We denote by $\mathfrak{S}_n$ the state obtained by applying the update set of the rule R^n in state $\mathfrak{S}$, if consistent. Two termination conditions are possible in this case, which arise either when the generated update set is empty or it is inconsistent. Thus the interpretation of iteration can be specified in the following manner

$$[\![\textbf{iterate}(R)]\!]^{\mathfrak{S}} = \lim_{n\to\infty} [\![R^n]\!]^{\mathfrak{S}}, \textit{ if } \exists n \geq 0 \textit{ such that } [\![R]\!]^{\mathfrak{S}_n} = \varnothing \vee [\![R]\!]^{\mathfrak{S}_n} \textit{ is inconsistent}$$

Iteration may also be associated with a condition that helps to determine when no more updates are generated. Aiming to provide a more familiar language to programmers, we allow the iteration of a guarded rule of the form **if** $cond$ **then** R as **iterate**($cond$) R.

C.4.2.3 Ordinary interaction

ASMs as discussed so far fundamentally specify the means to compute a certain function, provided its component functions are themselves computable. However, it is often more common to use computing devices not to perform calculations, but to provide the means for communication and interaction among different entities.

The study of the viability to introduce different forms of interactivity to the ASM formalism has resulted in different characterizations of interactivity as well as of the extended ASMs to deal with them. In our context, we adopt in particular the notion of *ordinary* interactive ASMs, which have the following characteristics

- An ordinary interactive ASM is able to obtain information from external entities by issuing *queries*. A query q is issued through the invocation of a function $f(t_1, ..., t_n)$ where f is a *external* function from the ASM vocabulary Σ and $t_1, ..., t_n$ are terms over the same vocabulary.

- The ASM cannot successfully complete a step as long as a query issued during the same step remains unanswered.

- The only information the ASM receives from the environment during a step consists of the answers to the queries issued during the step.

In our context the external functions correspond to data and computation services. The underlying execution environment will receive the service calls along with their parameters, perform the service calls to obtain the results, and update the state of the ASM accordingly. In this manner, the coordination of services according to the application logic specified as an ASM is supported.

The combination of restrictions and extensions to the basic ASM formalism we have introduced result in an effective model and language for services coordination. Next we will discuss the language semantics, which is crucial for developers to understand for an adequate use of the language for building service-based applications.

C.4.2.4 Services

Recal from Section 3.1.2 that we understand by a service a software entity capable of performing several tasks, which is accessible through a standardized interface and subject to a single administrative domain. In addition, service instances are accessible via *registry* and *choice* mechanisms. Concretely, a service instance implements a service interface and is represented by a tuple $serv : \langle op_1 : f_1, ..., op_n : f_n \rangle$ where each $op_i : f_i$ is a function complex value of type $\hat{f}_i$.

The operations of a service are thus represented as functions that receive a set of input parameters and produce a single output parameter. As stated in Definition 3.2, each of the input and output parameters of the operation take values associated with the denotation of a particular type. In terms of abstract state machines, these values correspond to elements in a universe or domain; for instance, the domain $\mathbb{S}$ of finite strings over a particular alphabet.

For the utilization of service operations in ASMs, the name op of the operation is prefixed by the name s of the service yielding a function name of the form $s.op$, which can be integrated as part of the vocabulary Σ of a service coordination ASM. If that is the case, then the term $s.op(t_1, ..., t_n)$ represents a query, as defined for ordinary interaction in ASMs, that corresponds to the invocation of operation op of service s. The input parameters are obtained by interpreting the terms t_i and the result of the query corresponds to the operation output parameter.

The execution environment is responsible for accessing a particular instance of the service (identified by an URI) and performing the operation invocation.

Example C.23.

Consider the interface of a service named $mathlib$ that performs mathematical operations. In particular, this service offers the operations $power : \ x : \mathbb{R} \times n : \mathbb{N} \rightarrow res : \mathbb{R}$, $sqrt : \ x : \mathbb{R} \rightarrow res : \mathbb{R}$, and $fact : \ n : \mathbb{N} \rightarrow res : \mathbb{N}$; to compute the n-power, square root, and factorial functions. Where $\mathbb{N}$ and $\mathbb{R}$ are the domains of natural and real numbers, respectively.

C.4.3 Language semantics

The objective of this subsection is to describe more concretely the semantics of our ASM-based service coordination language. In particular we adopt a dataflow perspective, explaining the execution of ASM rules and aiming to prevent ambiguities that may arise due to the use of parallelism as a default in ASMs.

To understand the semantics of ASM rules it is essential to take under consideration the different roles that dynamic functions can assume, these are identified as follows

- We say that a function f is $updated$, denoted as upd, in a rule R of an ASM M if f appears on the left-hand side of an update $f := g(t_1, ..., t_n)$ of R, i.e. its value is updated. We also define $upd(R)$ as the set of all dynamic functions that are updated in rule R.
- similarly, we say that a function f is in *computation use*, denoted as *c-use*, in a rule R of an ASM M if f is used in the computation of an update of R, and thus appears on the right-hand side of an update, e.g. $r := g(a, h(f))$. We also define *c-use*(R) as the set of all dynamic functions that are under computation use in rule R.
- finally, we say that a function f is in *predicate use*, denoted as *p-use*, in a rule R of an ASM M if f is used in a predicate of R, and thus appears on the condition of a conditional rule, e.g. **if** $f = 1 \wedge g > x$ **then** P. We also define *p-use*(R) as the set of all dynamic functions that are under predicate use in rule R.

Next we describe the semantics of parallel and sequential composite rules in terms of their effects on dynamic functions under the different roles just defined. First, recall that the interpretation of a term $f(t_1, ..., t_n)$ without variables in a state $\mathfrak{S}$ and denoted by $[\![f(t_1, ..., t_n)]\!]^{\mathfrak{S}}$ is defined as $f^{\mathfrak{S}}([\![t_1]\!]^{\mathfrak{S}}, ..., [\![t_n]\!]^{\mathfrak{S}})$, while the interpretation $[\![R]\!]^{\mathfrak{S}}$ of a rule R in state $\mathfrak{S}$ corresponds to the update set produced. Note that we also introduce new notations for explicit parallel composition and sequential composition.

C.4.3.1 Parallel composition

We denote by $val(f)$ the value of a nullary dynamic function at a particular time during the execution of the ASM. For a rule R defined as **par** R_1 R_2 ... R_n **endpar** in state $\mathfrak{S}$

- For all R_i $(1 \leq i \leq n)$ and all $f \in upd(R_i)$, where R_i is of the form $f := g(t_1, ..., t_n)$, $val(f) = [\![f]\!]^{\mathfrak{S}}$ and $[\![R_i]\!]^{\mathfrak{S}} = (f, v)$ where $v = g^{\mathfrak{S}}([\![t_1]\!]^{\mathfrak{S}}, ..., [\![t_n]\!]^{\mathfrak{S}})$
- For all R_i $(1 \leq i \leq n)$ and all $f \in$ *c-use*(R_i), $val(f) = [\![f]\!]^{\mathfrak{S}}$
- For all R_i $(1 \leq i \leq n)$ and all $f \in$ *p-use*(R_i), $val(f) = [\![f]\!]^{\mathfrak{S}}$
- while $[\![R]\!]^{\mathfrak{S}} = \bigcup_{i=1}^{n} [\![R_i]\!]^{\mathfrak{S}}$

Intuitively, for all roles of a function f in a parallel composition, its value in all member rules R_i of the composition corresponds to its interpretation in the current state $\mathfrak{S}$. This current state is then changed by an update set generated from the union of all of the update sets of the member rules.

C.4.3.2 Sequential composition

For a rule R defined as **seq** R_1 R_2 ... R_n **endseq** in state $\mathfrak{S}$

- Consider the function $next(U)$ denoting the state produced by applying the update set U. For each R_i $(1 \leq i \leq n)$ and all $f \in upd(R_i)$, where R_i contains an update of the form $f := g(t_1, ..., t_n)$ and $\mathfrak{S}_i = next([\![R_{i-1}]\!]^{\mathfrak{S}_{-1}})$ with $\mathfrak{S}_o = \mathfrak{S}$, $val(f) = [\![f]\!]^{\mathfrak{S}_i}$ and $[\![R_i]\!]^{\mathfrak{S}_i} = (f, v)$ where $v = g^{\mathfrak{S}_i}([\![t_1]\!]^{\mathfrak{S}_i}, ..., [\![t_n]\!]^{\mathfrak{S}_i})$
- For each R_i $(1 \leq i \leq n)$ and all $f \in$ *c-use*(R_i), $val(f) = [\![f]\!]^{\mathfrak{S}_i}$
- For each R_i $(1 \leq i \leq n)$ and all $f \in$ *p-use*(R_i), $val(f) = [\![f]\!]^{\mathfrak{S}_i}$
- while $[\![R]\!]^{\mathfrak{S}} = \oplus_{i=1}^{n} [\![R_i]\!]^{\mathfrak{S}_i}$

Intuitively, all updates generated by a rule in the sequence become visible to its subsequent rule, for all of the roles in which the dynamic function takes part. The sequential composition generates an update set in which the updates made by the latter rules in the sequence take precedence.

ANNEX D
HYPATIA installation

Required software

- Java SE SDK 1.6.0 http://ant.apache.org/
- Apache Ant 1.7.1 http://ant.apache.org/
- Apache Tomcat 5.5.27 http://tomcat.apache.org/
- JAX-WS 2.1.5 RI http://jax-ws.java.net/
- ANTLR v3 http://www.antlr.org/
- ANTLR v3 Ant Task http://www.antlr.org/depot/antlr4/main/antlr-ant/main/antlr3-task/antlr3.jar
- JGraphLayout v1.4.3.4 http://www.jgraph.com/jgraph.html
- MySQL Connector/J 5.1.13 http://www.mysql.com/products/connector/
- MySQL Community Server 5.1.50 http://dev.mysql.com/downloads/mysql/5.1.html
- Apache CXF 2.2.10 http://cxf.apache.org/

Environment variables

Defined for Windows XP SP3 with `C:\` as the base directory

- `Path=...;C:\jdk1.6.0_21\bin;`
  `C:\apache-ant-1.7.1\bin;C:\apache-cxf-2.2.10\bin`
- `JAVA_HOME=C:\jdk1.6.0_21`
- `JAXWS_HOME=C:\jaxws-ri`
- `CATALINA_HOME=C:\apache-tomcat-5.5.27`
- `ANTLR_HOME=C:\antlrworks-1.2.3`
- `ANT_HOME=C:\apache-ant-1.7.1`
- `CXF_HOME=C:\apache-cxf-2.2.10`

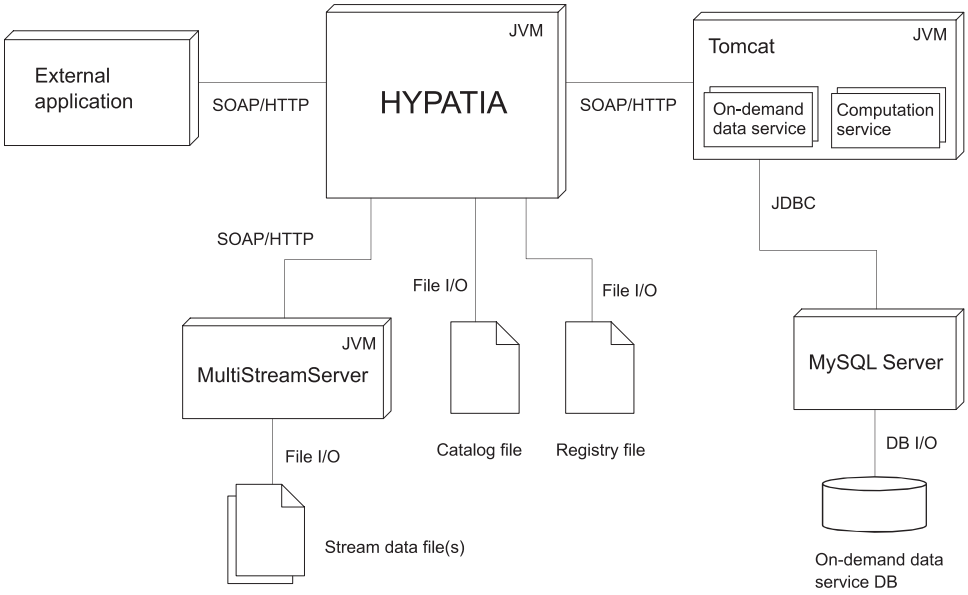


Figure D.1: Software components in our implementation

Deployment view - software components

Figure D.1 presents the deployment view (see Section 5.1) of the main software components in our implementation and validation of a service-based hybrid query processor presented in Chapter 5.

Deployment view - directories and files

We present in Figure D.2 the structure of the files and folders in the implementation of HYPATIA, we limit ourselves to the main application using jar file packaged subsystems.

Execution commands

The Tomcat Application Server and MySQL Sever should be running.

HYPATIA: `ant run` (input parameters are set in the `run` task in build.xml)
MultiStreamServer: `ant run -buildfile [buildfile].xml` (buildfile.xml has the input parameters for the desired stream data services)

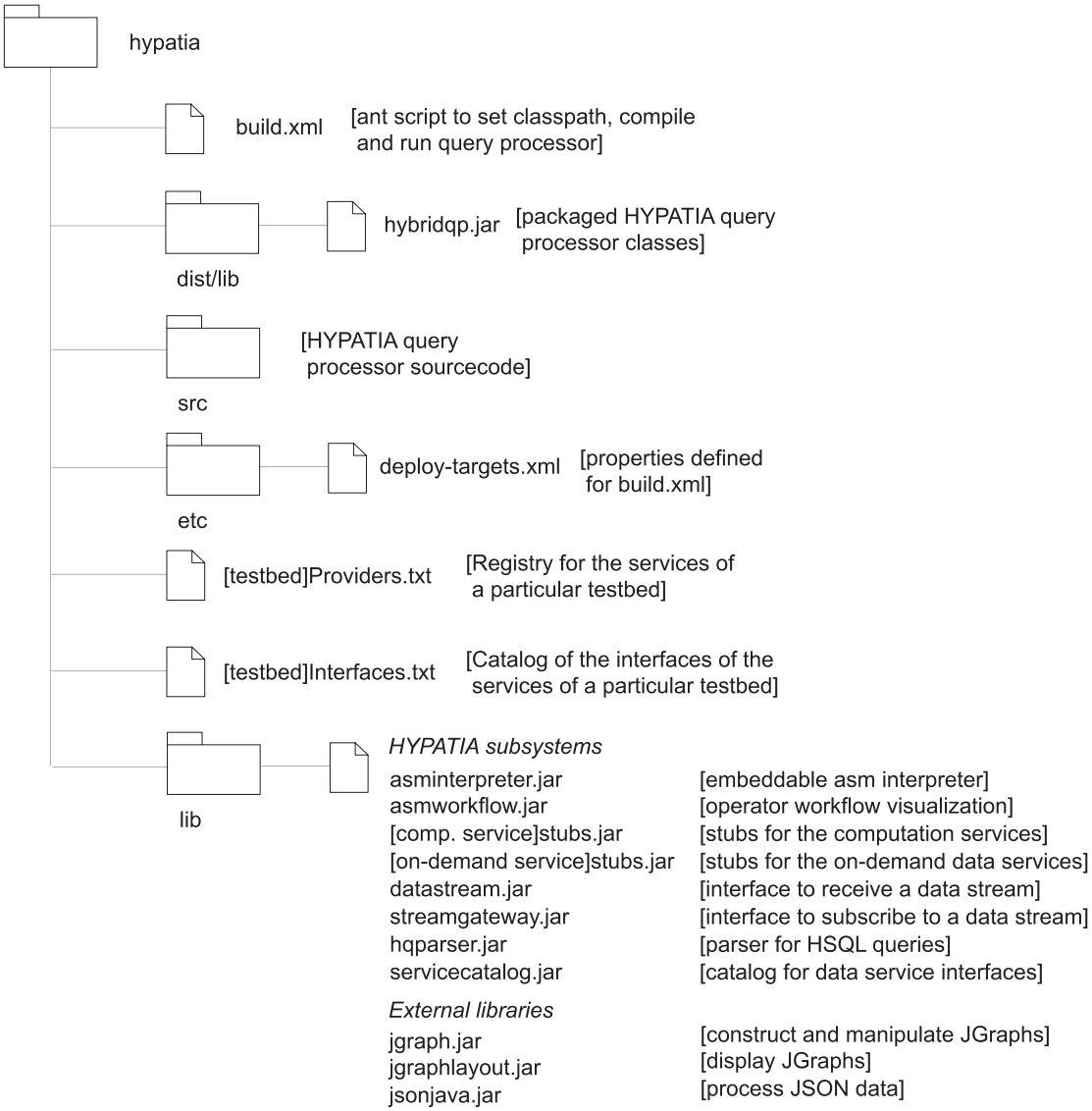


Figure D.2: File structure of the HYPATIA hybrid query processor

EVALUATION OF HYBRID QUERIES BASED ON SERVICE COORDINATION

Víctor Cuevas-Vicenttín