UNIVERSITÉ DE VERSAILLES SAINT-QUENTIN-EN-YVELINES

UFR DE SCIENCES

**THESE**

pour obtenir le grade de

DOCTEUR DE L’UNIVERSITE DE
VERSAILLES-SAINTE-QUENTIN-EN-YVELINES

*Discipline : Informatique*

présentée et soutenue publiquement

par

Ioana MANOLESCU-GOUJOT

le 19 Décembre 2001

Optimization techniques for querying heterogeneous distributed data sources

---

*Directeur de thèse : Patrick Valduriez*

---

**JURY**

M. Philippe Pucheral   Président
M. Serge Abiteboul   Rapporteur
Mme. Daniela Florescu   Examinateur
M. Tamer Ozu   Rapporteur
M. Eric Simon   Examinateur
M. Patrick Valduriez   Examinateur
Contents

1 Introduction ................................. 5
  1.1 The wrapper-mediator architecture ............... 6
  1.2 Query processing in an integration system .......... 7
  1.3 Contributions of this thesis ...................... 10
  1.4 Organization of this thesis .................... 12

2 Data and program publication in LeSelect.......... 14
  2.1 Overview of LeSelect .......................... 14
  2.2 Modeling heterogeneous resources using binding patterns .......... 16
    2.2.1 The concept of binding pattern .................. 17
    2.2.2 Modeling relational data ........................ 18
    2.2.3 Modeling programs ................................ 18
    2.2.4 Modeling tables with binary attributes .......... 18
    2.2.5 Modeling object data ............................ 20
    2.2.6 Modeling DOM-compatible data sources ............. 23
    2.2.7 Modeling a full-text XML index ................. 24
  2.3 Answering queries over tables with binding patterns .......... 25
    2.3.1 Queries over tables with binding patterns .......... 25
    2.3.2 Answering queries over a single table .......... 29
    2.3.3 Answering queries over several tables with binding patterns .......... 35
    2.3.4 Answering disjunctive queries .................. 39
  2.4 Publishing resources using binding patterns .......... 41
    2.4.1 Resource metadata ................................ 42
    2.4.2 Wrapper capabilities ............................. 43
  2.5 State of the art in modeling of source capabilities .......... 44
    2.5.1 Describing source capabilities using binding patterns .......... 45
    2.5.2 Describing source capabilities using an operator model .......... 48

3 Query optimization with binding patterns: principles and application in a distributed context .......... 50
  3.1 Introduction .................................. 50
  3.2 Running example ................................ 51
  3.3 Operators for accessing restricted resources ............ 54
3.3.1 The BindJoin operator ............................................. 54
3.3.2 The BindAccess operator ............................................. 55
3.4 Principles of query optimization using binding patterns ................. 57
3.5 Search space for query optimization with binding patterns ............... 61
3.5.1 Classes of query execution plans .................................. 61
3.5.2 Theoretical study of the size of the search space .................... 65
3.5.3 Experimental study of the size of the search space .................... 68
3.6 Optimization algorithms using binding patterns ........................ 73
3.6.1 Common principles ................................................. 73
3.6.2 Dynamic programming optimization algorithm ....................... 75
3.6.3 Best-first search optimization algorithm ............................ 75
3.6.4 Implementation .................................................... 79
3.6.5 Experimental comparison of dynamic programming and best-first algorithms .................................................... 82
3.7 Distributed query optimization algorithm in LeSelect ..................... 83
3.7.1 Optimization guidelines .......................................... 83
3.7.2 Cost model ....................................................... 87
3.7.3 Distributed query optimization algorithm .......................... 91
3.8 Related work ................................................................ 97
3.8.1 Query optimization with binding patterns .......................... 97
3.8.2 Query optimization algorithms for distributed query processing ..... 98

4 Efficient execution of queries over restricted resources .......................... 102
4.1 Introduction ..................................................................... 102
4.2 Motivating example ................................................................ 104
4.3 LeSelect infrastructure for querying restricted resources ................. 107
4.3.1 Iterators and the Exchange operator ................................ 108
4.3.2 Using the BindJoin and BindAccess operators for restricted resources ................................................. 109
4.3.3 BlobManager for storing and transferring blobs ................. 110
4.4 Algorithms for BindJoin and BindAccess ................................ 111
4.4.1 Physical operators for BindJoin and BindAccess .................... 112
4.4.2 Algorithms for the BindAccess operator ............................ 112
4.4.3 The CacheFIFO BindJoin algorithm ................................ 114
4.4.4 The CacheParallel BindJoin algorithm .............................. 115
4.4.5 Experimental evaluation of the BindJoin algorithms ............... 119
4.4.6 Introducing intra-operator parallelism in the BindJoin algorithm ...... 124
4.4.7 Adaptive tuning of the degree of parallelism ......................... 126
4.5 Query execution in the presence of space restrictions ...................... 128
4.5.1 Influence of space restrictions on query execution ................. 128
4.5.2 Recovery procedure for queries involving produced blobs .......... 130
4.5.3 Safety of concurrent recovery procedures .......................... 131
4.6 Optimization issues ....................................................... 133
4.7 Related work .................................................................. 136
5 **The Agora data integration system**

5.1 Introduction ................................................................. 140
5.2 XML data integration methodology ..................................... 142
  5.2.1 Problem definition ..................................................... 142
  5.2.2 Motivating example ................................................... 142
  5.2.3 The XQuery query language ......................................... 142
  5.2.4 Data integration methodology ..................................... 146
5.3 XQuery normalization ....................................................... 146
  5.3.1 Normalization rules ................................................... 148
  5.3.2 Difficulties of the normalization process ......................... 151
5.4 Translating normalized XQuery into SQL .......................... 151
  5.4.1 Virtual generic schema as support for translation ............... 152
  5.4.2 Translating simple path expressions .............................. 152
  5.4.3 Translating FLWR expressions .................................... 154
  5.4.4 Difficulties for the translation process .......................... 157
5.5 Relational query rewriting .............................................. 159
  5.5.1 View definitions for relational sources .......................... 160
  5.5.2 View definitions for DOM sources ................................. 161
  5.5.3 View definition for a XML full-text index ....................... 162
  5.5.4 Rewriting algorithm ................................................. 163
5.6 Translating queries with intermediate XML results ............. 167
5.7 Related work ............................................................. 168

6 **Conclusion** .............................................................. 173
6.1 Summary ................................................................. 173
6.2 Directions for future work ............................................. 174
Chapter 1

Introduction

In recent years, the development of information processing technology has led to a wide range of resources being connected to the Internet and thus available through standard general-purpose communication protocols (e.g., TCP/IP, HTTP, Corba). We consider a resource to be any data collection or application program that an owner agrees to share. Thus, accessing a resource means retrieving data, or making a program invocation and collecting its results.

While the existing communication infrastructures make it possible to access a resource in isolation, the purpose of a resource integration system is to produce high-quality integrated information using autonomous, distributed, heterogeneous resources [120]. Resources are autonomous, since they are frequently developed and maintained in isolation and independently of the applications that may use them; therefore, the design, communication paradigm, and processing power of a resource are decided by their owners. Often, we need to integrate resources that are physically distributed on several sites. Finally, they are heterogeneous with respect to their structure, or to their processing capabilities. Data collections can follow different data models, ranging from regular standard formats (relational, object-oriented) to more flexible data formats like OEM [89] and XML [123], to very loosely structured files (e.g., textual data or HTML pages). The implementation of an application program depends on the software and hardware environment, and the interfaces available for its invocation may also vary (e.g., CGI, Corba, RMI). High-quality information is produced from a set of resources by expressing in a high-level query language a combination of resource accesses and data-processing tasks that cannot be performed by the resources, and providing the execution mechanism needed in order to process this query.

Our work is placed in the context of virtual integration, in which resources are exploited from their original sites. The alternative warehouse approach consists of replicating the sources within a single repository, on which all queries are evaluated. The warehousing approach is likely to provide better performance, since query processing does not involve inter-site communication, and better reliability, since the replicated sources are within the control of a single site. However, to preserve the freshness of the data in a warehouse, a maintenance process is required, which is typically complex and costly. Also, it may be impossible to replicate all interesting resources in a single repository, for privacy, copyright or security reasons; scalability is also an issue when dealing with a large number of resources. Finally,
some resources, and in particular programs, may depend on the particular environment of the owner’s site and thus cannot be replicated elsewhere.

1.1 The wrapper-mediator architecture

Wrappers and mediators The classical architecture for virtual resource integration consists of two types of modules: mediators and wrappers [120]. A mediator module interacts with the distributed resources, and with applications requiring integrated information. It provides an unique and uniform access point to the heterogeneous distributed resources, and provides a query interface to the applications. Also, a mediator provides for the independence of the applications requiring integrated information from the autonomy and evolution of the resources. Finally, a mediator may add value to the data and programs it integrates, by using domain-dependent knowledge.

A wrapper module acts as an intermediary between the resources and the mediators. Wrappers hide the internal format and the implementation of their resources from the mediator, provide to the mediator descriptions of these resources in terms of the mediator’s schema, and may accept to process subqueries on their resources, at the request of mediators. Thus, query processing in a wrapper-mediator system is divided between wrappers and mediators. Other types of information exported by wrappers include: resource metadata (e.g., content description for their data sources), a description of their query processing capabilities, as well as data statistics and cost parameters characterizing their query processing power. In summary, wrappers are directly connected to their diverse resources via an interface adapted to the resource; on the other hand, the mediators communicate with all wrappers via the same standardized interface.

Local schema and global schema The uniform interface between wrappers and mediators is typically based on a common data model in terms of which all resources have been described. Thus, each wrapper corresponding to one or several resources presents to the mediator a local schema, reflecting its resources. The schema that the mediator presents to the applications using its services is called the global schema. In existing systems, the format of the global schema is object-oriented, or semistructured. One of the first applications of loosely structured data, and the motivation behind OEM, the first semi-structured data model, was precisely the need for a format flexible enough to describe data sources with diverse structures.

There are three important problems to be addressed when designing an integration system based on the wrapper-mediator architecture.

First, a common model has to be found for the heterogeneous resources. This model should be rich enough to capture the interesting features of the resources, and should allow for the optimization of queries formulated in the terms of this model. Then, a wrapper is needed for each type of resource, in order to present it under the common chosen model. Writing wrappers is usually a tedious task, and its difficulty is a major obstacle to large-scale
resource integration. In several existing projects, wrapper generation is partially automatic and is done from some high-level specification of the resource contents and capabilities.

Second, the global schema has to be designed, through a process of schema integration. This process is also needed in integration systems following the warehouse approach. Schema integration may involve resolving domain and ontology differences, interpolation or extrapolation of temporal data, abstraction, summarization, identification of common entities in different data sources according to common keys, pruning of low-quality data etc.

Finally, we need to address the aspects related to query processing, which is typically distributed among the mediator and the wrappers. Since query processing in a distributed resource integration system is the focus of this thesis, we address it separately in the following section.

1.2 Query processing in an integration system

In a wrapper-mediator architecture with a single mediator, the following steps are taken in order to process a query:

1. The query, which is expressed in terms of the global schema, has to be reformulated in terms of the wrappers’ local schemas.

2. The mediator decomposes the reformulated query into: several subqueries to be sent to the wrappers, and some operations that the mediator will execute on the results of the wrapper queries. Since there are several alternative decompositions, this step involves a distributed optimization algorithm.

3. The subqueries are sent to the wrappers, which carry on their execution.

4. The results of the wrapper subqueries are handed to the mediator, which applies the remaining query processing steps and returns the complete result.

These steps are the same in systems with several mediators; the only difference is that if many mediators are involved in processing a query, they may also be assigned subqueries to be processed, just like the wrappers. Even if there are several mediators, usually there is just one query processing coordinator among them for a given query.

The conceptual link between the local and global schemas is important for query processing, since it determines the complexity of the first step mentioned above. According to the way the local and global schemas are related, data integration systems can be classified in the two categories that we describe next.

Query reformulation

Following the local-as-view approach (LAV), the local schemas are defined as views (queries) formulated in terms of the global schema. In contrast, in a global-as-view (GAV) integration system, the global schema is described as a query over the local schemas.
The tradeoffs between LAV and GAV (as presented in [57]) are the following. In the GAV approach, translating the query on the global schema into queries on the local schemas is done by composing the query with the definition of the global schema; this process is known as view unfolding and is relatively simple. In the case of LAV, the query on the global schema needs to be reformulated in the terms of the local data sources’ schemas; this process is traditionally known as “rewriting queries using views” and is in general a hard problem [1, 57]. On the other hand, in a GAV architecture, to handle modifications in the local schemas, the new global schema needs to be re-designed. In a LAV architecture, a local change to a data source can be handled locally, by adding, removing or updating only the view definitions concerning this source; therefore, LAV scales much better. Another advantage of the LAV approach is that it takes advantage of redundant storage in the local sources. Indeed, if the contents of two data sources $S_1$ and $S_2$ overlap at some extent, this information may be exposed by the view definitions corresponding to these sources. Therefore, a query $Q$ over the overlapping subset of $S_1$ and $S_2$ will be rewritten in two equivalent manners, using either one source or the other; these equivalent rewritings provide more choices as to how to execute the query.

**Query optimization**

We present the distinguishing aspects of the query optimization problem in an integration system via the three dimensions of this problem: search space, cost model, and search strategy.

**Search space**  A first thing to note is that in a distributed system, the parallelism opportunities provided by the presence of several processing sites should be exploited as much as possible; therefore, an optimizer in an integration system should search for the best bushy query execution plan, since bushy QEPs allow for parallel execution.

Second, in an integration system the optimizer has to take into account the limited processing capabilities of the wrappers. For example, the wrapper of a Yellow Pages Web site may only perform a limited class of queries, retrieving the phone number corresponding to a given person name and address. So, the optimizer’s search space is not only determined by the query and the localization of the resources involved in the query, but also by the query processing capabilities of each participant to the execution of the query; it is important that the optimizer understands these capabilities in order to take maximal advantage of them.

We distinguish between positive query processing capabilities, i.e. operations that a wrapper may execute besides providing free access to its resource, and negative query processing capabilities, i.e. restrictions that the wrapper imposes on accessing the resources it manages. Thus, in a distributed DBMS setting, all sites have full positive query processing capabilities, while in an integration context wrappers may have positive or negative capabilities. Note that due to the presence of negative query capabilities, it may be the case that a query is rejected, since the wrappers do not accept it, and the mediator can’t execute it because it does not have access to the data sources. For example, if the Yellow Pages server
requires the name and the zip code of a person in order to provide its phone number, if the
zip code is not provided, the query cannot be answered.

Following our Yellow Pages example, it has recently become possible to learn the name
of a person, if the telephone number is known. In such cases, there may be more than one
access method to a given data source. In a (distributed) database setting, alternative access
method to a data source provided the same functionality, albeit at different cost. In the case
of a heterogeneous resource integration system, alternative restricted access methods have to
be investigated in order to be sure that if an execution plan exists, the optimizer has found
it.

**Cost model** Traditionally, the query optimization process is based on the presence of
some data statistics; in a distributed DBMS, detailed statistics can be easily computed
and stored for the optimizer to consult them. In an integration system, however, it is
more difficult to get data statistics, due to the nature of the data sources involved, to
the restrictions on accessing these sources, or to the frequent changes in the contents of the
source. Also, the optimizer needs cost formulas for the operations that the wrappers are able
to perform. Several approaches have been taken to obtain these formulas. Each wrapper
writer may provide the cost formulas for its wrapper, or a generic cost formula may be used
for all wrappers, and calibrated by executing a set of test queries [66]; in some cases, these
two approaches are used together, where the wrapper-specific formulas override the generic
calibrated ones. A different method consists of monitoring query execution and learning to
guess correctly the cost of subsequent queries [66].

**Search strategy** In the presence of limited source capabilities, the optimizer’s search
strategy should be modified to explore only the space of plans that are feasible, i.e., the
localization of operators among the actors involved in query processing respects their query
processing capabilities.

**Query execution**

The challenges in executing queries in an integration system come from three sources: the
distribution, the heterogeneity of the query capabilities, and the lack of control over distant
execution sites. To diminish the communication overhead entailed by the distribution of
resources, execution techniques validated in the context of distributed database systems,
like **row blocking**, or **fetch-as-needed** join strategies [76] can also be applied in this context.
To cope with the diverse query processing capabilities, after the optimizer has constructed
a QEP, the sub-query assigned to a wrapper has to be translated in a language that the
wrapper understands. Finally, due to the autonomy of the execution sites, parameters of
query execution may change abruptly during the execution, e.g., the output rate of an
operator may decrease abruptly or a source may stop sending data. One approach taken
in order to reduce the impact of such unpredictable events on the execution of the query
relies on **adaptive operators**, that change their evaluation strategy in order to adapt to the
changing environment. A different approach is to have an **adaptive execution strategy**, that
may change during query execution: fragments of the QEP being executed are rescheduled, or a new QEP is constructed, or memory resources are dynamically re-allocated to reflect the new configuration.

When integrating heterogeneous and distributed data and programs, it is important to note that while data may in general be transferred from its original site to a distant site for processing, programs do not always allow this. It often happens that we need to integrate programs that were not meant originally to be used outside the environment where they were created, and were thus developed depending on a specific software or hardware environment, hidden resources etc. Thus, we are forced to ship the program arguments to the program’s site, since it is the only place where it can be executed. The per-tuple cost of calling a program is often much higher than the per-tuple processing cost of a regular operator; thus, program invocation costs are an important component of the cost of a QEP, and special care needs to be taken when optimizing and executing such queries. Also, data sources that can be published and shared often include images or other large data objects; the transfer costs associated to queries involving such objects need to be reduced.

1.3 Contributions of this thesis

In this work, we investigate several aspects of query processing on distributed heterogeneous resources. The negative query capabilities that we consider are expressed using the formalism of binding patterns, originally introduced in [93]. The work presented in this thesis took place in the general framework of the LeSelect relational data integration system [68]. This thesis makes the following contributions:

- We provide necessary and sufficient query feasibility criteria, given the restricted capabilities of the data sources, modeled as tables with binding patterns. This problem has received a lot of attention in the case of Datalog queries, using the universal relation assumption. In LeSelect, where queries are expressed in SQL and have bag semantics, some restrictions have to be posed on the way the optimizer combines the accesses to several tables with binding patterns, in order to make sure that the answer returned is correct. Our interest in the binding pattern formalism comes from the fact that it provides an uniform way of modeling both data and programs. We show how several classes of resources, in particular relational, object and tree-structured data, and programs, can be described as tables with binding patterns; this modeling approach is used by LeSelect.

- We study the problem of cost-based query optimization in the presence of binding patterns. Query feasibility had been investigated prior to our work, and comprehensive results exist concerning how to decide whether a query over resources with binding patterns has at least one execution plan. Our study [39] was the first to show how to integrate binding pattern limitations in cost-based query optimization. We show that the presence of binding patterns leads to a new search space, that needs to be explored by the optimizer in order not to miss the optimal plans, and characterize the
size of this search space both analytically and through experiments. We show that depending on the source binding patterns, the size of the search space can be much smaller, or significantly larger than the one considered in the absence of binding pattern limitations. Thus, we propose two optimization algorithms. The first one is based on dynamic programming and requires minimal modifications to a regular query optimizer; its drawback is that complete query execution plans are only found towards the end of the search, which may be unacceptable for very large search spaces. A second algorithm, based on best-first search, is designed to find some query execution plans fast even if the search space is important; we studied the tradeoffs between these two algorithms through several experiments. In the presence of binding pattern limitations, selection and join operators in a QEP interact in a new manner; we provide pruning conditions that consider these interactions, and allow for optimal selection placement. These algorithms only consider the negative resource capabilities encoded by binding patterns. Finally, as an extension, we present the distributed query optimization algorithm of LeSelect, that takes into account the negative and positive resource query capabilities, as well as the distribution of the resources.

- Based on the observation that most time-consuming queries in LeSelect involve calls to expensive programs and/or large data transfers, we designed an efficient physical BindJoin operator that eliminates useless computations (data transfer as well as program invocations). This operator incorporates caching, is non-blocking, distributes well over different sites and may take advantage of the presence of duplicates in its input to provide a steady output rate in the early stages of query execution. This feature decreases the chances that the operators further up in the QEP get idle, and may be very useful in a system designed for human users, as is the case of LeSelect, since it increases the probability that the users get some answers fast. Furthermore, our BindJoin operator allows for intra-operator parallelism, and adapts its parallelism degree to the restricted resource response time [77].

- In many applications, structured data sources, and in particular relational data, need to be exploited together with document data, presented in the XML format. Furthermore, integration applications with a specific domain (e.g., health care, electronic commerce etc.) deal with a well-defined set of concepts and entities, that should be captured by the global schema; recently, standardization efforts have lead to the synthesis of several domain-specific XML dialects\(^1\). We designed and implemented the Agora system on top of LeSelect, integrating relational and XML data sources under an XML global schema. All data sources are managed by wrappers connected to LeSelect servers, and they are all presented to LeSelect’s query execution engine as tables with binding patterns; thus, a query over these heterogeneous data sources is optimized and executed within a single process. Query processing in Agora consists of: translating the query formulated in an XML query language into a SQL query on the data sources; optimizing and executing this query, which is done by LeSelect; and

\(^1\)See [www.XML.org](http://www.XML.org) for a listing.
constructing the XML result from the resulting tuples (this last step is done following a method proposed in [107]). To relate the data sources to the XML global schema, we define them as views over an intermediate relational virtual generic schema, that describes the generic structure of an XML document. An XML query over the XML global schema is first, simplified by applying simple equivalence rules, then translated into a SQL query over this virtual generic schema, and finally re-written into a SQL query over the real data sources. As explained before, the local-as-view data integration approach allows us to exploit the redundancy of the data sources; in particular, it allows us to exploit a materialized XML full-text index, as an alternative access path to XML elements within a document; we had argued the interest of extending XML query language with a text search predicate in [38].

The Agora system was demonstrated recently [79] and its query translation methodology was detailed in [78].

1.4 Organization of this thesis

This document is organized as follows. The next chapter details the way heterogeneous resources are described and modeled in the LeSelect data integration system. We present the concept of tables with binding patterns, and determine which binding patterns may be combined in order to correctly compute the answer to a query.

Chapter 3 presents the principles of cost-based query optimization on resources modeled as tables with binding patterns, and studies the associated search space. We also introduce two basic optimization algorithms, and compare their performances in terms of total optimization time and time to the first QEP produced. In this chapter, we also describe the distributed query optimization algorithm we designed and implemented for LeSelect. This algorithm considers a restricted form of the query optimization problem when dealing with binding pattern access restrictions; at the same time, it deals with the distribution of resources and with their positive query processing capabilities.

Chapter 4 considers the execution of queries involving costly programs and large data objects in the LeSelect system. In this chapter, we propose several improvements to its current query execution architecture, that would allow it first, to deal efficiently with large data objects, and second, to parallelize operator execution and data transfers. We designed a BindJoin operator that we use for calling programs and transferring large objects between sites; the interest of merging processing and transfer in a single operator is that both may use a cache to avoid redundant effort, and in both cases a resource is accessed following a restricted binding pattern.

In chapter 5, we present Agora, a system integrating relational and XML data sources under an XML global schema. In this chapter, we describe our approach for describing the local sources as views over the XML global schema. Also, we explain the mechanism of translating queries posed in an XML query language into SQL queries over the data sources, that LeSelect can execute.

Finally, chapter 6 summarizes the work presented in this thesis and the results obtained.
We also present interesting directions for future work.

Due to the different scope of each chapter, we split the comparison of our work with existing research results over the chapters 2, 3, 4 and 5.
Chapter 2

Data and program publication in LeSelect

In this chapter, we describe the process of data and program publication in the LeSelect data integration system. LeSelect is a fully distributed mediator system developed in the Caravel project, at INRIA Rocquencourt [68]. It has been specifically designed to facilitate the publication of resources within a virtual community, and enable users to formulate high-level queries over published resources.

2.1 Overview of LeSelect

The design of LeSelect is based on the wrapper-mediator generic architecture, as shown in figure 2.1. Several LeSelect mediators (also called servers) may run on a set of hosts. On every server, resource owners publish their resources using wrappers. The internal data model of LeSelect is relational; therefore, wrappers present their data sources to the server as a set of tables with binding patterns. A table has an universally unique name, consisting of: the table name, the name of the wrapper publishing the table, and the identifier of the LeSelect server to which the wrapper is attached (host name and port). We will discuss the details of the modeling shortly.

LeSelect offers the framework for publishing data and programs. Sample categories of data that can be exploited through LeSelect are: relational data stored within an DBMS, tabular text files, DOM-compliant data sources etc. A program, for the purpose of this thesis, is defined as any application taking as input a tuple of arguments and returning, for each input tuple, zero or several output tuples.\(^1\) Examples of programs that can be published and used within LeSelect are: a scientific image processing program transforming an image into another one by applying some numerical methods, a Yellow Pages service returning the phone number of a given person, or a search engine returning relevant URLs for a given keyword.

\(^1\)Programs taking as input one or several tables and returning tables in the output are modeled differently in LeSelect [68].
From the point of view of the integration system, modeling and query processing are very similar for data and programs, and therefore we denote either of these by the generic term of resource. We illustrate the similarities between data and programs with the following example. Consider a bibliographic server providing complete information about a book, when given the book's ISBN number. The data is retrieved from a database, which would make us consider this bibliographic server to be a data source; at the same time, the data retrieval functionality supposes the existence of an active software component, making the case for this service to be considered a program. Thus, the difference between data and programs appears only at the conceptual level, while their modeling is done in an uniform manner, as we show in this chapter.

The query language supported by LeSelect is the select-project-join subset of SQL (no updates). SQL queries can be posed on any server, and may involve resources published on several servers. By connecting to one LeSelect server, a user may learn what resources are published on that server; however, there is no centralized catalog of all the resources being published\(^2\). Therefore, it is the responsibility of the user to identify which resources she is interested in, and formulate her query using the universally unique table names.

Any server contains a set of modules that enable it to monitor and participate to the processing of the queries it receives. The interactions between these modules are represented in figure 2.1, where solid lines represent data flow, and dashed lines trace command or statistic information flow. The query analyzer parses the query, unfolds view definitions, and constructs an internal representation of the query that is handled to the optimizer. The optimizer gathers statistic information about the resources involved in the query, and constructs a distributed query execution plan (QEP). In general, different servers run independently and ignore each other’s existence, but are always willing to cooperate in order to answer a query. Therefore, the distributed QEP may delegate some processing to the wrappers where the queried resources were published, to their respective servers, or to the server where the query has been received. Any LeSelect server contains a query execution module, implementing a set of physical relational operators. Also, the resource wrappers may have some query processing capabilities; using descriptions of these capabilities, the optimizer may delegate some work to the resource wrappers, too.

When processing a query, the query execution unit uses the services of a communication module to transmit and gather data to and from other LeSelect servers. Several communication modules exist, using different data exchange standards (TCP, HTTP, Corba etc.).

We start by describing tables with binding patterns, the basic abstraction used for the modeling of heterogeneous resources in LeSelect. In section 2.2, we provide a syntax for specifying binding patterns, and the semantics of an accessing a table following one of its binding patterns. We then illustrate the modeling of several classes of resources using the notion of binding patterns. In section 2.3, we study the problem of answering queries over tables with binding patterns, and give an algorithm for deciding when a query is feasible. Section 2.4 describes the resource publication process in LeSelect. Finally, in section 2.5, we compare the resource modeling of LeSelect with other related proposals.

\(^{2}\)However, such a catalog could easily be added, e.g., using a registry standard like WSDL or UDDI
2.2 Modeling heterogeneous resources using binding patterns

LeSelect is a virtual data integration system, in which resources remain within the administration of the resource owner, and may only be queried within the limits that the owner allows. The most complete access to a table that the owner may allow is a full scan; however, in many cases, more restricted access methods are used, replacing or complementing the possibility to scan the table. The access to a given resource may be restricted for two main reasons: (1) the underlying data may actually be stored in a structured file or legacy system hence the interface to the data is naturally limited, and (2) even if the data is stored in a traditional database system, the source may provide only limited access capabilities for reasons of security or performance. We note that data access restrictions are also encountered in the case of graph-structured data models (e.g., OEM [89]), where the logical model of the data is a graph, while the particular way of storing the data poses natural limitations on the access patterns to the data.

In LeSelect, we describe these negative data source capabilities using the concept of binding pattern introduced in [93]. In this section, we describe the basic concept of binding patterns, and show how to use them for modeling a large family of interesting resources.

Notations We denote by capital letters like \( R, S, T, \ldots \), sometimes with subscripts, relational tables. Capital letters like \( X, Y, Z, \ldots \), sometimes with subscripts, denote individual variables of an atomic type, ranging over the values of some column in a table. Sets of variable are denoted using upper bars, as in \( \overline{X} \); sets of constants are denoted by lowercase letters with upper bars, as in \( \overline{c} \). We use lowercase greek letters like \( \rho, \tau, \phi \) to represent type variables, and calligraphic capital letters for sets of types. A special set of types that we use, \( \mathcal{A} \), is the set of atomic types such as integer, string, double, boolean etc. We also include in \( \mathcal{A} \) a special atomic type binary which stands for large data objects (as an intuition, an object of type blob is usually an image, or a large unstructured chunk of text).
Bag semantics It is important to note that in this chapter, we use a bag-oriented data model, and we assume bag semantics for the query language and for the regular relational operators. Our interest for bag semantics comes from the popularity of the SQL query language, that is often used in the context of bags. As we will see in section 2.3, the distinction between bags and sets is important when answering queries over tables with binding patterns.

2.2.1 The concept of binding pattern

A binding pattern $bp$ for a table $R(\overline{X})$, $\overline{X} = X_1, X_2, \ldots, X_n$, is a mapping from $\overline{X}$ to the alphabet $\{b, f\}$. The meaning of the binding pattern is the following: those $X_i$ that $bp$ maps to $b$ are bound, i.e., their values must be supplied in order to obtain information from $R$, while values of attributes mapped to $f$ are free, i.e., they can be obtained from the data source as soon as values for all $b$ attributes are supplied. Note that $bp$ may be incomplete, i.e. there may be no associated value for several $X_i, 1 \leq i \leq n$. An attribute $X_i$ that is absent from a binding pattern $bp$ cannot be queried using this binding pattern, i.e. it can be neither supplied nor obtained (hidden attribute). A value assignment for a bound attribute of the form $X_i = c$, where $c$ is a constant, is referred to as a binding.

As a notation, we represent a binding pattern by indicating next to each $X_i$, in upper script, the $b$ or $f$ value that the mapping assigns it; hidden attributes are omitted. A given table may have several binding patterns, corresponding to several restricted access patterns allowed; we denote by $bp(R)$ the set of binding patterns for the table $R$.

Finally, we use the notation $\bowtie_{\text{nat}}$ to denote the natural join of two tables $R(\overline{X})$ and $S(\overline{Y})$, such that there is a common subset $\overline{Z} = \overline{X} \cap \overline{Y}$: in this case, $R \bowtie_{\text{nat}} S$ has the column set $\overline{X} \cup \overline{Y} \setminus \overline{Z}$, and one row for each pair of tuples $t_1 \in R, t_2 \in S$ such that $t_1.Z = t_2.Z$.

Binding patterns and semantics of a table It is important to note that in general, the fact that a table is accessible by a binding pattern of the form $R(\overline{U}^b \overline{V}^f)$ is completely independent of the semantic relationship among the attributes $\overline{U}$ and $\overline{V}$. However, semantic information about $R$ is necessary in order to decide whether several binding patterns of $R$ may be used together in order to answer a given query $Q$, as we explain in the next section.

Binding pattern semantics in terms of relational algebra Let $bp$ be a binding pattern of table $R$, having the form $R(\overline{U}^b \overline{V}^f)$, $\overline{U}, \overline{V} \subseteq \overline{X}, \overline{U} \cap \overline{V} = \emptyset$. Let $C(\overline{U})$ be a conjunction of predicates of the form $X_i = c_i$, where $X_i \in \overline{U}$ (we also call $C(\overline{U})$ a set of bindings). Let us denote by $\text{acc}(R, bp, C(\overline{U}))$ the tuples obtained by accessing $R$ using $bp$, with a set of bindings $C(\overline{U})$; then, we have:

$$\text{acc}(R, bp, C(\overline{U})) = \pi_{\overline{U} \cup \overline{V}}( \sigma_{C(\overline{U})}(R))$$

(2.1)

Example 2.2.1 Consider a table $R(X, Y, Z)$, containing the tuples $\{(1, 1, 2), (2, 1, 4), (1, 2, 4)\}$, having a binding pattern $bp_1 = R(X^b Y^f)$. Let $C(X)$ be the predicate $X = 1$. Then, $\text{acc}(R, bp_1, C(X)) = \{(1, 1), (1, 2)\}$. 

17
In particular, if $\overline{U} = \emptyset$, the tuples obtained by accessing $R$ following a binding pattern of the form $R(\overline{V}^f)$ are those returned by the projection $\pi_{\overline{V}}(R)$.

### 2.2.2 Modeling relational data

The modeling of a relational table is quite straightforward: we only need to determine the binding patterns for the table.

**Example 2.2.2** Consider a table $R(X, Y; Z)$, stored within an RDBMS. Typically, a full scan of such a table is allowed, therefore its binding pattern set may include $R(X^f Y^f Z^f)$. However, if an index on $R.X$ exists within the RDBMS, a second binding pattern of the form $R(X^b Y^f Z^f)$ is used to model the alternative access path, that requires knowledge of the values of $X$.

### 2.2.3 Modeling programs

In general, consider a program $p$ having the signature

$$p(X_1 : \tau_1, X_2 : \tau_2, \ldots, X_n : \tau_n) \rightarrow Y_1 : \rho_1, Y_2 : \rho_2, \ldots, Y_m : \rho_m, \tau_i \in \mathcal{A}, \rho_j \in \mathcal{A}.$$  

The program takes as input a tuple of $n$ arguments $X_1, X_2, \ldots, X_n$ and returns zero, one or several tuples of results, of the form $Y_1, Y_2, \ldots, Y_m$. In LeSelect, such a program is modeled as a relational table with the following binding pattern:

$$p(X_1^b X_2^b \ldots X_n^b Y_1^f Y_2^f \ldots Y_m^f)$$

If the arguments $X_j, X_{j+1}, \ldots, X_n$ are optional, a second binding pattern

$$p(X_1^b X_2^b \ldots X_{j-1}^b Y_1^f Y_2^f \ldots Y_j^f)$$

may be used to describe this.

**Example 2.2.3** A currency converter, available via a form on a Web site, translates an amount from a given currency into another one. The converter can be thought of as a function $convert(inputCurrency, inputAmount, outputCurrency) \rightarrow outputAmount$. In our framework, the converter can be described as a table with the following binding pattern: $convert(inputCurrency^b inputAmount^b outputCurrency^b outputAmount^f)$.

### 2.2.4 Modeling tables with binary attributes

In chapter 4, we propose special execution and optimization techniques, to speed up the execution of queries involving large data objects, of type binary, as used in LeSelect. In order to apply these techniques, we set some requirements on the modeling of a table with binary objects. (If these requirements are not met, we are still able to execute the query, although with poor performance.) We call those binary attributes satisfying our publication requirements *blobs*.
Definition (Blob) An attribute $A$ of table $R$ is a blob iff:

- $A$ is declared to be of the LeSelect data type binary;
- if $A$ is mapped to $f$ by some binding pattern of $R$, then $R$ contains a small-sized attribute $blobID$ which identifies $A$ (i.e., such that the functional dependency $blobID \rightarrow A$ holds);
- $R$ has at least the binding pattern $R(blobID^kA^l)$.

The choice of declaring an attribute as binary is left to the publisher; as in commercial DBMSs, attributes with a size greater than a given threshold (e.g., a few Kb) should be declared as binary.

For any blob that is published, or created by a program, if the blob may be obtained in a query (i.e., is mapped to $f$ in at least one binding pattern), we require a blobID in order to minimize the number of times the blob is transferred among several sites when executing a query, and the number of times a program is invoked on the same blob. BlobIDs are system-generated in the case of published data residing in an DBMS. In a simpler setting, a large data object is usually stored in a separate file, whose complete name (i.e. host/filename) can be used as a blobID.

Example 2.2.4 A data source contains table satellite images of several regions of the Earth; physically, each image is stored in a separate file. The data is modeled as a table SatelliteImg(taken: date, region: int, filename: string, img: blob) in which each image is described by one tuple. This table allows either a “catalog” access of the summary attributes, excluding the image, or retrieving an image by its filename: \{Satellite(taken, region, filename), Satellite(filename, img)\}.

Consider a table $R$ publishing a blob attribute $A$, following the requirements above. Usually, besides the required binding pattern $R(blobID^kA^l)$, $R$ has several other binding patterns, that allow accessing the other attributes of the table. We impose the functional dependency $blobID \rightarrow A$ in order to be able to use $R(blobID^kA^l)$ after having accessed $R$ following another binding pattern; thus, we make sure that this binding pattern sequence is a sound sequence, as defined by proposition 2.3.3.

The wrapper of a resource that creates blobs must assign to each produced blob a system-wide unique identifier, e.g., by appending the host name and a local reference, as shown in the following example.

Example 2.2.5 An ImgConvert program translates between several image encoding formats. ImgConvert takes as input the blob $img_1$, the format codes of the input and output images, and produces a $img_2$ blob result. ImgConvert must have among its binding patterns (Img2ID$^k$Img2$^l$): the Img2ID attribute is required since the program exports a binding pattern where Img2 is free. Such an attribute is not necessary for Img1 since there is no real data to be extracted - on the contrary, only Img1 has to be supplied. Therefore, the binding pattern set associated to ImgConvert is \{ImgConvert(Format$^k$Format2$^l$Img1$^l$Img2$^l$Img2ID$^l$), ImgConvert(Img2ID$^k$Img2$^l$))\}. 
2.2.5 Modeling object data

Tables with binding patterns may be used to describe collections of objects; we explain the principles behind the modeling as well as its limitations.

The idea is the following. The data members of a class can be captured by a simple table; the methods of a class can be thought of as programs, if we consider the object receiving the method to be an extra program argument. Thus, methods are modeled as tables with restricted binding patterns.

We assume a type system consisting of:

- atomic types \( A \)
- list, set and bag types derived from \( A \) and denoted \([\tau], \{\tau\}\) and respectively \(|\tau|\), with \( \tau \in A \)
- class types \( C \). Let \( \varphi \) be a class having a set of data members and a set of method members. Each data member \( a \) has a type \( \tau \) that is either: an atomic type from \( A \), or a collection type derived from \( A \), or an object type \( \phi \), or a collection type derived from \( C \). Each method \( m \) takes as arguments a tuple of arguments \((x_1 : \tau_1, x_2 : \tau_2, \ldots x_n : \tau_n)\), where each \( \tau_i \in A \) or \( \tau_i \in C \), and returns a result \( y : \rho \), where

\[
\rho \in A \cup C \cup \{[\tau], \{\tau\}, |\tau|, \{(string, \tau)\} \mid \tau \in A \cup C \}
\]

- list, set and bag types derived from \( C \) and denoted \([\varphi], \{\varphi\}\) and respectively \(|\varphi|\), with \( \varphi \in C \)

For our modeling, we assume the existence, for each class \( \varphi \), of a Skolem function \( \varphi ID \) that computes an integer that uniquely determines each object within the class. The following rules describe how to model a class \( \varphi \).

1. To each class \( \varphi \) we associate a relational table \( R_\varphi \), including an integer \( ID \) attribute that stores the identifier computed by \( \varphi ID \). The attributes of \( R_\varphi \) are obtained from the data members of class \( \varphi \) by applying the three following rules.

2. A data member \( a : \tau \) of the class \( \varphi \), such that \( \tau \in A \) appears as an attribute in the table \( R_\varphi \).

3. A data member \( a : \tau \) of the class \( \varphi \), such that \( \tau = [\nu] \) or \( \tau = \{\nu\} \) or \( \tau = |\nu| \), \( \nu \in A \), is modeled by an auxiliary table \( R_{\varphi,a}(ownerID : int, value : \nu, pos : int) \). The pos attribute appears only if \( \tau = [\nu] \), and it reflects the order of each item within the list.

4. A data member \( a : \tau \) of the class \( \varphi \), such that \( \tau \in C \) is represented in \( R_\varphi \) by a foreign key into \( R_\tau \).

The following five rules explain how to model methods of the form \( m(x_1 : \tau_1, x_2 : \tau_2, \ldots x_n : \tau_n) \rightarrow y : \rho \) in which \( \tau_i \in A \cup C \).
5. We represent every method \( m \) of the form above as a separate table \( R_{\varphi, m} \). This table contains an attribute \( ID : \text{int} \), corresponding to the ID of the object receiving the method call. Furthermore, \( R_{\varphi, m} \) contains, for every \( x_i \) of an atomic type, an attribute \( z_i : \tau_i \), and for every \( x_i \) of a class type \( \tau_i \), an integer attribute \( z_i : \text{int} \), where \( z_i \) is a foreign key into \( R_{\tau_i} \). The other attributes of \( R_{\varphi, m} \) are determined by the exact type of \( \rho \), the method result, as follows.

6. If \( \rho \in \mathcal{A} \), then \( R_{\varphi, m} \) contains an attribute \( y : \rho \) and \( bp(R_{\varphi, m}) = \{ R_{\varphi, m}(ID^b z_1^b, \ldots, z_n^b y^f) \} \)

7. If \( \rho \in \mathcal{C} \), then \( R_{\varphi, m} \) contains an attribute \( yID : \text{int} \), which is a foreign key into \( R_{\rho} \), and \( bp(R_{\varphi, m}) = \{ R_{\varphi, m}(ID^b z_1^b, \ldots, z_n^b y^f) \} \)

8. If \( \rho = \{ \nu \} \) or \( \rho = [\nu] \) or \( \rho = \{ [\nu] \} \), \( \nu \in \mathcal{A} \cup \mathcal{C} \) then \( R_{\varphi, m} \) contains the attributes \( y : \theta \) and \( pos : \text{int} \), with \( bp(R_{\varphi, m}) = \{ R_{\varphi, m}(ID^b z_1^b, \ldots, z_n^b y^f pos^f) \} \)

In this case, \( pos \) is used only if \( \rho = [\nu] \), and it reflects the order of each item within the list. If \( \nu \in \mathcal{A} \), then \( \theta = \nu \) and the result itself is an attribute of \( R_{\varphi, m} \); if \( \nu \in \mathcal{C} \), then \( \theta = \text{int} \) and \( y \) is a foreign key into \( R_{\rho} \). In both cases, \( u \) is the name assigned to each item in the method result.

9. If \( \rho = \{(\text{string}, \nu)\} \mid \nu \in \mathcal{A} \cup \mathcal{C} \), then \( R_{\varphi, m} \) contains the attributes \( u : \text{string} \) and \( y : \theta \), with \( bp(R_{\varphi, m}) = \{ R_{\varphi, m}(ID^b z_1^b, \ldots, z_n^b y^f u^f) \} \)

If \( \nu \in \mathcal{A} \), then \( \theta = \nu \) and the result itself is an attribute of \( R_{\varphi, m} \); if \( \nu \in \mathcal{C} \), then \( \theta = \text{int} \) and \( y \) is a foreign key into \( R_{\rho} \). In both cases, \( u \) is the name assigned to each item in the method result.

10. Assuming the extent of the class \( \varphi \) can be obtained, \( R_{\varphi} \) has one binding pattern of the form \( R_{\varphi}(ID^f) \). Let \( \{a_1, a_2, \ldots, a_k\} \) be the public data members of class \( \varphi \), of an atomic, respectively class type. Then, \( R_{\varphi} \) has one binding pattern of the form \( R_{\varphi}(ID^f a_1^e a_2^e \ldots a_k^e c_1^e c_2^e \ldots c_j^e) \). For any atomic, or object data member \( a \) of class \( \varphi \), that can be obtained only via accessor methods, and that has been modeled using one or two attributes \( a_1, a_2 \) (following the rules 2, 3 and 4), \( R_{\varphi} \) has one binding pattern of the type \( T_{\varphi}(ID^b a_1^f a_2^f) \).

By applying these rules, we break the object schema into tables in a quite straightforward manner. It can be seen that for representing data members of a class, the choice of our relational tables is very similar to a 3NF schema resulting from an Entity-Relationship analysis of the class. The difference is that this schema is not used for storage, but for modeling. In representing method members of a class, we use binding patterns to describe the inherent restrictions when calling a method: the object that receives the method call, and the method arguments, must be known before the call is made. Finally, we model accessor methods as restricted access tables.
Example 2.2.6 Consider the classes Item and Buyer, used in an e-commerce application. An Item has an ID, a description, a price, and a photo. A Buyer is identified by an ID, has a name, several shipping addresses, and several credit cards registered with the site. The getItems method returns the list of items bought by a Buyer. Note that in both cases, there already exists an unique ID for objects in each class, thus, we do not need to use Skolem functions to that purpose. Also, we assume that each Item photo is unique, thus the Item.ID attribute uniquely determines the photo.

The modeling of these classes yields the following tables with binding patterns:

\[\text{Item}(\text{ID}: \text{int}, \text{description}: \text{string}, \text{price}: \text{int}, \text{photo}: \text{blob}), \text{with} \]
\[bp(\text{Item}) = \{\text{Item}(\text{ID}^\prime, \text{description}^\prime, \text{price}^\prime), \text{Item}(\text{ID}^\prime, \text{photo}^\prime)\}\]

\[\text{Buyer}(\text{ID}: \text{int}, \text{name}: \text{string}), \text{with} \]
\[bp(\text{Buyer}) = \{\text{Buyer}(\text{ID}^\prime, \text{name}^\prime)\}\]

\[\text{BuyerAddress}(\text{ID}: \text{int}, \text{shippingAddress}: \text{string}), \text{with} \]
\[bp(\text{BuyerAddress}) = \{\text{BuyerAddress}(\text{ID}^\prime, \text{shippingAddress}^\prime)\}\]

\[\text{BuyerCard}(\text{ID}: \text{int}, \text{cardNo}: \text{string}), \text{with} \]
\[bp(\text{BuyerCard}) = \{\text{BuyerCard}(\text{ID}^\prime, \text{cardNo}^\prime)\}\]

\[\text{BuyerGetItems}(\text{ID}: \text{int}, \text{itemID}: \text{int}, \text{pos}: \text{string}), \text{with} \]
\[bp(\text{BuyerGetItems}) = \{\text{BuyerGetItems}(\text{ID}^\prime, \text{itemID}^\prime, \text{pos}^\prime)\}.\]

Limitations of the modeling Some interesting features of the object-oriented model [111] are not captured by these relational tables. First, the inheritance mechanism is not reflected; therefore, each class is modeled in isolation, and no link unites subclasses with superclasses. Second, the updates that methods may perform, and that may change the internal state of the object, are ignored; in some sense, we assume that all methods are “read-only”; i.e., the values of an object’s data members are not altered after a method call. Third, the type system we allow has some restrictions; for example, method arguments cannot have collection types etc.

In our framework, queries over object, as well as relational data come under the form of SQL queries over the tables. However, object bases are usually queried using object-oriented query language, such as OQL [111]. Thus, in order to fully exploit an object base, one would have to provide a translation from queries posed in an object-oriented query language like OQL, into SQL queries over the relational tables modeling the data. We do not study query translation from OQL to SQL (of course, due to the above limitations, and to many others, some OQL features cannot be properly dealt with in this relational framework). Our goal is not to “simulate” an object-oriented database by using a relational system; it is to integrate data that is mostly relational, with other repositories, using SQL as a query language and relational query processing techniques. However, we do study an interesting related problem of query translation, from an XML query language into SQL, in chapter 5.
2.2.6 Modeling DOM-compatible data sources

As an important application of object data modeling, tables with binding patterns can be used to describe any data sources that are compliant with the DOM interface [125]. The Document Object Model is a generic API, standardized by the World Wide Web Consortium (W3C), that enables the manipulation of hierarchically structured data sources.

Figure 2.2 shows the DOM representation (the tree drawn in solid lines) of the XML fragment shown on the left. A DOM representation of a hierarchically structured document is a tree of nodes. To each document corresponds a Document node (represented with a triangle in figure 2.2). Any document has a distinguished root Element node, in our case person; element nodes are represented as black dots. Element nodes have a set of children; the node children are ordered. Elements can also have attributes, which are name-value pairs; in our example, SSno is an attribute of person, and the attribute node is shown as a black rectangle. Elements may have PCDATA (text) children nodes; these are the leaves of the DOM tree, and are represented with hollow rectangles in figure 2.2. A DOM representation of a data source can be obtained in different ways; in particular, to transform an XML or HTML file into such a tree-structured representation is to use a DOM parser, as for example those provided by IBM [127].

The DOM API defines a set of classes, like Document, Element, Attribute etc.; all inherit from the Node abstract class. We distinguish two types of methods belonging to DOM classes: navigation methods, that enable the retrieval of related node in the tree, and update methods, for inserting, removing, or modifying data in the document. As explained before, our modeling of object data does not take into account update methods.

Figure 2.2 illustrates several navigation methods, shown in dashed curved arrows. For example, the getDocumentRoot() method of the Document node returns the person element. From this element, the getAttribute("SSno") method returns the identifier of the SSno attribute. Calling getElementsByTag("hobby") on the same element yields the two hobby descendants of the person node. Other examples of navigation methods are: given a Node, retrieve the list of its children, its parent node, the set of its attributes, its name, its siblings (children of the same parent node) etc. It is important to note that these methods allow upward, downward, and lateral navigation, thus, there are several alternative ways to reach a given node in the tree.
Following the generic methodology for representing object data, we model the node classes as tables with binding patterns; in particular, we use one table per possible navigational DOM API method. The set of class types that we consider, \( \mathcal{N} \), denotes the set of all specialized node types identified by the DOM API:

\[
\mathcal{N} = \{ \text{Attribute, CDATASection, CharacterData, Comment, Document, DocumentFragment, Element, Entity, EntityReference, ProcessingInstruction} \}
\]

In the case of DOM objects, the extents of the classes in \( \mathcal{N} \) cannot be freely obtained. In order to access some data item from a document, one has to obtain the Document node corresponding to that document. The Document class is modelled by a table \( \text{Document}(\text{ID}: \text{int}, \text{URI}: \text{string}) \), with the binding pattern \( \text{Document}(\text{ID}^b: \text{URI}^b) \), where URI stands for “Uniform Resource Identifier”; broadly speaking, an URI is a complete address of the document [119]. The URI is sufficient to retrieve the document; by parsing it, we obtain the Document node that is the root of the document. Once the Document node is obtained, navigation methods can be used to reach some Element nodes, their attributes etc.

**Example 2.2.7** Consider the following method of the \( \text{Document} \) class:

\[
\text{getElementsByName}(\text{name:}\text{string}) \rightarrow \{\text{Element}\}
\]

This method takes as argument a string and returns the list of all elements occurring in the document, whose tag is equal to that string, in the order in which they appear in the document. We model this call by a table

\[
\text{getElementsByName}(\text{docID:}\text{int}, \text{name:}\text{string}, \text{eID}: \text{int}, \text{pos}: \text{int})
\]

having the binding pattern \( \text{getElementsByName}(\text{docID}^b: \text{name}^b: \text{eID}^f: \text{pos}^f) \). In a similar manner, the method \( \text{getChildNodes}() \rightarrow \{\text{Element}\} \) present in the \( \text{Element} \) class is modeled as a table

\[
\text{getChildNodes}(\text{eID}^b: \text{eID}^c: \text{pos}^f)
\]

where \( \text{eID}_1 \) denotes the parent and \( \text{eID}_2 \) the children elements.

### 2.2.7 Modeling a full-text XML index

Suppose a materialized full-text index (FTI) of one or several XML documents is available. This is an important particular case when tables with restricted binding patterns model several access paths to a data collection. As we have seen in the previous section, the content of XML documents may be obtained by using the tables with binding patterns corresponding to their DOM representation. Alternatively, a widely used access method for textual documents consists of using a full-text index (FTI).

Full-text indexes for XML are usually built at the granularity of XML elements, and they offer the following basic functionality: given a keyword, retrieve the elements (from several documents) whose text content contains that keyword. Such an index is easily modeled as the following table:
FTI (word,docID,elID), with \( bp(FTI) = \{ FTI(word^i,docID^j,elID^j) \} \)

Variants of FTI like the ones described in [38] and [5] expose several additional informations:

- whether the word appears as an element or attribute name, or as element content;
- whether it appears directly within the element identified by the element ID or in one of its children/descendents;
- the path in the DOM tree where the element containing the given word is situated

Note that in some index proposals, the encoding of the element ID plays a very important role in facilitating the use of the FTI together other access methods to the XML data, and is more complex than an integer attribute (see, e.g., [5] and [29]). Therefore, to use such an index in conjunction with the tables representing DOM-compliant data sources, some translation between the two element identifiers is necessary.

2.3 Answering queries over tables with binding patterns

2.3.1 Queries over tables with binding patterns

Let us consider the class of select-project-join queries, also known as conjunctive queries. We denote a conjunctive query \( Q \) as:

\[
\text{select } \overline{X}_{\text{proj}} \text{ from } R_1(\overline{X}_1), R_2(\overline{X}_2), \ldots, R_n(\overline{X}_n) \text{ where } C(\overline{X}) \text{ and } J(\overline{X})
\]  

(2.2)

In this notation, \( \overline{X}_i \) denotes the set of attributes of the table \( R_i \), \( \overline{X} = \overline{X}_1 \cup \ldots \cup \overline{X}_n \), and the set of attributes returned by the query satisfies \( \overline{X}_{\text{proj}} \subseteq \overline{X} \). The conditions in the where clause are grouped in two categories. The set of selection conditions, \( C(\overline{X}) \), is a conjunction of simple selection predicates of the form \( U = c \), where \( U \in \overline{X} \) and \( c \) is a constant. \( J(\overline{X}) \) is a conjunction of predicates of the form \( U = V \), where for some \( i, j \), equal or distinct, \( U \in \overline{X}_i \) and \( V \in \overline{X}_j \).

On a conjunctive query \( Q \), we make the following notations. Let \( \text{Var}(C(\overline{X})) \) be the variables from \( \overline{X} \) that actually appear in \( C(\overline{X}) \), and \( \text{Var}(J(\overline{X})) \) the variables appearing in \( J(\overline{X}) \). Then, we denote by \( \overline{X}_{\text{res}} \) the following set of variables:

\[
\overline{X}_{\text{res}} = \overline{X}_{\text{proj}} \cup \text{Var}(C(\overline{X})) \cup \text{Var}(J(\overline{X}))
\]  

(2.3)

Furthermore, for any \( i \), we denote by \( \overline{X}_{i,\text{res}} \) the intersection of \( \overline{X}_i \) and \( \overline{X}_{\text{res}} \).

It is important to note that throughout this chapter, and in particular in the above notations, we assume bag semantics for the tables (which may contain duplicate tuples) and for the relational operators.

For some queries \( Q \) on a given set of tables, with given sets of binding patterns, there may be no way to compute the answer to \( Q \). The intuition is that the query does not supply enough information to allow us to use some restricted table accesses in order to obtain data.
Example 2.3.1 Consider the table \( R(X,Y) \) with \( bp(R) = \{(X^bY^f)\} \), and the query \( Q_1: \text{“select } R.Y \text{ from } R' \). This query cannot be answered, since a value for \( X \) was required in order to obtain \( Y \) values. If we replace it by \( Q_2 : \text{“select } R.Y \text{ from } R \text{ where } R.X = 5' \), we are able to use \( R \)'s binding pattern to obtain an answer.

In some sense, answering a query using restricted access tables is a very simple variant of the general problem of answering a query using only a set of materialized views. Indeed, we can think of a binding pattern over the table \( R \) as a "materialized view" over this table. Two important distinctions need to be made between our problem and the general one. First, our "materialized views" are parameterized, i.e. some information has to be supplied in order to use them; and second, the "view definitions" are restricted to select-project queries, as shown in equation 2.1.

Since there are cases when no particular binding pattern of a table \( R \) is sufficient in order to answer a query \( Q \), we are entitled to ask: can we combine several binding patterns of \( R \) to augment the limited knowledge we have about \( R \)? The answer is not always positive, as the following three examples show.

Example 2.3.2 Consider a table \( R_0(X,Y,Z) \) such that \( bp(R_0) = \{R_0(X^fY^f), R_0(Y^bZ^f)\} \), and \( R_0 \) consists of the tuples \{\((1,2,3),(4,5,6)\)\}. An access following \( R_0(X^fY^f) \) yields the tuples \{\((1,2),(4,5)\)\}. If we provide the \( Y \) values from this tuple set as inputs to the second pattern \( R_1(Y^bZ^f) \), we obtain the tuple set: \{\((1,2,3),(4,5,6)\)\}, which is exactly the content of \( R_0 \). In this case, combining the two binding pattern proved useful.

Example 2.3.3 Now consider a table \( R_1(X,Y,Z) \) with the same signature and binding patterns as \( R_0 \), but consisting of the tuples \{\((1,2,3),(2,2,5)\)\}. An access following \( R_1(X^fY^f) \) yields the tuples \{\((1,2),(2,2)\)\}. If we provide the \( Y \) values from this tuple bag as inputs to the second pattern \( R_1(Y^bZ^f) \), we obtain the tuples: \{\((1,2,3),(1,2,5),(2,2,3),(2,2,5)\)\}. This tuple bag is a superset of \( R_1 \): it contains the tuples \((1,2,5)\) and \((2,2,3)\), which were not present in \( R_1 \).

Example 2.3.4 Consider the table \( R_2(X,Y) \), with the binding pattern set \( bp(R_2) = \{R_2(X^f), R_2(X^bY^f)\} \), consisting of the tuples \{\((1,3),(1,3),(2,2)\)\}. Using the binding pattern \( R_2(X^f) \), we obtain the bag of tuples \{\((1),(1),(2)\)\}. Accessing \( R_2 \) following the binding pattern \( R_2(X^bY^f) \) with \( X = 1 \) yields two tuples, \{\((1,3),(1,3)\)\}. Accessing \( R_2(X^bY^f) \) successively, providing it with the values obtained from \( R_2(X^f) \), will thus produce the following tuple bag: \{\((1,3),(1,3),(1,3),(1,3),(2,2)\)\}; this tuple set is a superset of \( R_2 \), since it contains false duplicate tuples, which were not present in \( R_2 \).
The above examples show that there are circumstances in which “chaining” accesses to a table following several of its binding patterns generates false tuples or false duplicates. Since binding patterns are parameterized select-project queries over $R$, chaining several accesses to $R$ in order to “reconstruct” it is similar to the reconstruction of a table that has been split into several tables for the purpose of normalization. Consider the case when we accessing $R$ first, following $R(\overline{U}_1 \cup \overline{V}_1)$ with a given binding set $C(\overline{U}_1)$, and second, following $R(\overline{U}_2 \cup \overline{V}_2)$, where $\overline{U}_2$ is a subset of $\overline{U}_1 \cup \overline{V}_1$. This amounts to taking the natural join of $\pi_{\overline{U}_1 \cup \overline{V}_1}(\sigma_{C(\overline{U}_1)}(R))$ and $\pi_{\overline{U}_2 \cup \overline{V}_2}(R)$. Our goal when chaining these accesses was to reconstruct a bigger part of $R$ than the one provided by each binding pattern; namely, we aim at obtaining $\pi_{\overline{U}_1 \cup \overline{V}_1 \cup \overline{V}_2}(\sigma_{C(\overline{U}_1)}(R))$.

It would thus seem that we are faced with a particular instance of the well-known lossless join decomposition problem (described, e.g., in [2]). Recall that in the relational model without duplicates, decomposing $T(\overline{X})$ in $T_1(\overline{Y})$ et $T_2(\overline{Z})$ is lossless if and only if $\overline{Y} \cap \overline{Z} \rightarrow \overline{Y}$ or $\overline{Y} \cap \overline{Z} \rightarrow \overline{Z}$. In our case, ignoring the selection, the question is whether $\pi_{\overline{U}_1 \cup \overline{V}_1}(R)$ and $\pi_{\overline{U}_2 \cup \overline{V}_2}(R)$ are a lossless join decomposition for $\pi_{\overline{U}_1 \cup \overline{V}_1 \cup \overline{V}_2}(R)$. The intersection of the columns present in the two tables contains at least $\overline{U}_2$. Applying the lossless join decomposition criterion, if, for example, $\overline{U}_2 \rightarrow \overline{U}_2 \cup \overline{V}_2$, that is, $\overline{U}_2 \rightarrow \overline{V}_2$, then composing the two binding accesses would only produce correct tuples. Let us see how this carries on, using the examples 2.3.4 and 2.3.3 above, modified to verify the required functional dependencies.

**Example 2.3.5** As in example 2.3.3, consider the table $R_1(X, Y, Z)$, such that $b_p(R_1) = \{R_1(X^fY^f), R_1(Y^bZ^f)\}$. We modify the instance of $R_1$ into one that verifies $Y \rightarrow Z$: $\{(1, 2, 3), (2, 3, 5)\}$. Now, accessing $R_1$ following $R_1(X^fY^f)$ yields the tuples $\{(1, 2), (2, 3)\}$. If we use the obtained $Y$ values as inputs to the second binding pattern $R_1(Y^bZ^f)$, we obtain the tuples $\{(1, 2, 3), (2, 3, 5)\}$, that is, the original tuples in $R_1$. For this instance of $R_1$, indeed, accessing it through the two binding patterns yields only correct results.

**Example 2.3.6** Consider the table $R_2$, its binding pattern set, and its instance exactly as in example 2.3.4. Note that the instance verifies $X \rightarrow Y$; therefore, one would expect that combining the binding patterns $R_2(X^f)$ and $R_2(X^bY^f)$ correctly reconstructs $R_2(X, Y)$. However, as seen in example 2.3.4, this is not the case: spurious duplicates appear in the result.

Now consider a modified instance of $R_2$, consisting of $\{(1, 3), (2, 2)\}$; it is easy to check that accessing this instance following $R_2(X^f)$ and $R_2(X^bY^f)$ correctly reconstructs $R_2$.

These examples highlight the difference between the duplicate-free relational model, which is the framework in which the lossless join decomposition was previously studied, and the model including duplicates that we use. If a functional dependency is sufficient to make sure that combining two binding patterns does not yield false tuples, it is not sufficient in order to guarantee that no false duplicates are produced.
Consider again a table $R$ and two binding patterns $R(\overline{U}_1^b \overline{V}_1^f)$ and $R(\overline{U}_2^b \overline{V}_2^f)$, where $\overline{U}_2 \subseteq \overline{U}_1 \cup \overline{V}_1$. In order to combine these binding patterns and obtain information from $R$, excluding false duplicates or false tuples, we have the following choice:

1. Enforce that $\overline{U}_2$ be a unique key for $\pi_{\overline{U}_2 \cup \overline{V}_2}(R)$ (that is, enforce that for every two distinct tuples $t_1, t_2$ of $R$, $t_1 \overline{U}_2 \neq t_2 \overline{U}_2$). Indeed, in this case, $\overline{U}_2 \rightarrow \overline{U}_2 \cup \overline{V}_2$, which forbids obtaining false tuples by chaining the two accesses; furthermore, $\overline{U}_2$ values are unique, and therefore no false duplicates can be produced.

2. Modify the semantics of accessing a table following a binding pattern, so that it takes into account the order in which binding patterns are used. For any given $bp_1 = R(\overline{U}_1^b \overline{V}_1^f)$ that is the first to be used to access $R(\overline{X})$, with a set of bindings $C(\overline{U}_1)$, when answering a query $Q$, we return all tuples that match the selection condition, duplicates included. For a binding pattern $bp_2 = R(\overline{U}_2^b \overline{V}_2^f)$ that we use afterwards, for each tuple of inputs $\overline{U}_2$, we return the duplicate-free set of $(\overline{U}_2, \overline{V}_2)$ matches. With this modified semantics for table access, and if $\overline{U}_2 \rightarrow \overline{V}_2$ (functional dependency), combining the two binding patterns can be shown to be sound.

We consider the constraint for $\overline{U}_2$ to be an unique key in $\overline{U}_2 \cup \overline{V}_2$ as too restrictive. Indeed, this would entail that $\overline{U}_2$ is an unique key to $R$, and such a condition is likely to be more often false than true, thus stopping us from using more than one binding pattern to access $R$.

Therefore, we opt for the second choice above, and modify the semantic of accessing a table $R$ following a binding pattern so that it includes duplicates only for the first access to $R$. The remaining part of section 2.2 answers the following question: given a conjunctive query $Q$ over $\{R_1, R_2, \ldots, R_n\}$, which binding patterns from $bp(R_i)$ are we allowed to use in order to answer $Q$? We start by introducing a simple notation that will be useful in our discussion.

**Fragments** Let $R(\overline{X})$ be an instance of a relational table. A fragment of $R$ is a table $T(\overline{X}_1)$ such that $\overline{X}_1 \subseteq \overline{X}$ and there exists a logical expression $C(\overline{X})$ (not necessarily a conjunctive expression) such that

$$T = \pi_{\overline{X}_1} \sigma_{C(\overline{X})}(R)$$

(2.4)

The intuition is that a fragment of $R$ contains only the kind of partial information that can be derived by selections and projections from $R$. Note that, as stated before, we consider bag semantics for all relational operators, and as a consequence, $T$, as well and $R$, include duplicates.

**Example 2.3.7** Let $R(X, Y, Z)$ consist of the tuples $\{(1, 2, 3), (1, 2, 5), (2, 2, 4)\}$. Let $T_1(X, Y)$, $T_2(X, Y)$, $T_3(X, Y)$, $T_4(Z)$, $T_5(X, Y, Z)$ and $T_6(Y, Z)$ be six relational tables, such that: $T_1 = \{(1, 2), (1, 2)\}$, $T_2 = \{(1, 2), (2, 2)\}$, $T_3 = \{(1, 2, 3)\}$, $T_4 = \{(3), (5), (4)\}$, $T_5 = \{(1, 2, 4)\}$, $T_6 = \{(2, 3), (2, 3)\}$. It is easy to verify that
$T_1$, $T_2$, $T_3$, and $T_4$ are fragments of $R$. In contrast, $T_5$ and $T_6$ are not fragments of $R$.

**Notations** For each sequence $\text{seq} = [bp_1, bp_2, \ldots, bp_k]$ of binding patterns of a table $R$, where $bp_i = R(U_i, V_i^f)$, let $\text{Var}(\text{seq})$ denote the set of variables attained by the binding patterns in $\text{seq}$: $\text{Var}(\text{seq}) = U_1 \cup V_1 \cup U_2 \cup V_2 \cup \ldots \cup U_k \cup V_k$ (the union is to be considered duplicate-free). Also, let $\text{seq}_i$ denote the prefix of length $i$ of $\text{seq}$.

Let $\bar{X}$ be a set of variables, and $\bar{Y}_1, \bar{Y}_2$ two subsets of $\bar{X}$. Whenever $C(\bar{X})$ is a conjunction of selection predicates of the form $X = c$, we denote by $C(\bar{Y}_1)$ the restriction of $C(\bar{X})$ to the variables in $\bar{Y}_1$. Whenever $J(\bar{X})$ is a conjunction of predicates of the form $U = V$, with $U, V \in \bar{X}$, we denote by $J(\bar{Y}_1)$ the restriction of $J(\bar{X})$ to $\bar{Y}_1$; finally, $J(\bar{Y}_1, \bar{Y}_2)$ denotes the conjunction of all predicates $U = V$ in $J(\bar{X})$ such that $U \in \bar{Y}_1$ and $V \in \bar{Y}_2$.

In the next section, we address the problem of answering conjunctive queries over a single table with binding patterns; in section 2.3.3, we extend the discussion to the general case of conjunctive queries over $n$ tables.

### 2.3.2 Answering queries over a single table

Let us determine the conditions under which a sequence of binding patterns of a table $R$ is usable when answering a query over $R$. In general, a query over $R$ is of the form “select $\bar{X}_{\text{proj}}$ from $R(\bar{X})$ where $C(\bar{X})$ and $J(\bar{X})$”. Let us first focus on answering the simpler query “select $\bar{X}_{\text{proj}}$ from $R(\bar{X})$ where $C(\bar{X})$”.

**Usable sequence of binding patterns** In general, we say a sequence of binding patterns of a table $R$ is **usable** in order to answer a query $Q$ if the query provides enough information to enable successive accesses to $R$ following this sequence.

**Proposition 2.3.1 (Usability condition)** Let $Q$ be the query: “select $\bar{X}_{\text{proj}}$ from $R(\bar{X})$ where $C(\bar{X})$”. Consider the sequence of binding patterns $\text{seq} = [bp_1, bp_2, \ldots, bp_k]$, $bp_i = R(U_i, V_i^f)$. The binding pattern sequence $\text{seq}$ may be used to answer $Q$ only if

- for every $X \in U_1$, there is a predicate of the form $X = c$ in $C(\bar{X})$

- for every $i = 2, 3, \ldots, k$ and every $X \in U_i$, either
  - there is a predicate of the form $X = c$ in $C(\bar{X})$, or
  - $X \in \text{Var}(\text{seq}_{i-1})$

**Proof:** The proof follows directly from the semantics of bound and free attributes in a binding pattern. 

[End of proof]
Accessing a table using a sequence of binding patterns

Let \( Q \) be the query: “select \( \overline{X}_{\text{proj}} \) from \( R(\overline{X}) \) where \( C(\overline{X}) \)”. Let \( \text{seq} = [bp_1, bp_2, \ldots, bp_k] \) be a usable sequence of binding patterns of \( R \). We define the result of accessing the table \( R \) using the binding pattern sequence \( \text{seq} \) and the bindings \( C(\overline{X}) \) recursively as follows (\( \pi^0 \) stands for duplicate-free projection):

- \( \text{acc}(R, [bp_1], C(\overline{X})) = \text{acc}(R, bp_1, C(\overline{U}_1)) \)
- for any \( i = 2, 3, \ldots, k \):

\[
\text{acc}(R, \text{seq}_i, C(\overline{X})) = \text{acc}(R, \text{seq}_{i-1}, C(\overline{X})) \bowtie_{\text{nat}} \pi^0_{\overline{U}_i \cup \overline{V}_i}(\sigma_{C(\overline{U}_i)}(R)) \quad (2.5)
\]

In the last formula above, the access to \( R \) using \( R(\overline{U}_i \overline{V}_i) \) eliminates the duplicate tuples in \( \overline{U}_i \cup \overline{V}_i \). It follows from the usability condition in proposition 2.3.1 that all columns in \( \overline{U}_i \) are either (a) covered by a selection in \( C(\overline{U}_i) \), or (b) included in the columns of \( \text{acc}(R, \text{seq}_{i-1}, C(\overline{X})) \). The last selection on \( R \) enforces the conditions from \( C(\overline{U}_i) \), which contains bindings necessary in order to access \( R \). The natural join condition provides the rest of the bindings needed for \( \overline{U}_i \).

Selection conditions vs. join conditions Let us now explain why we cannot use join conditions of the form \( J(\overline{X}) \) to link several binding patterns of \( R \), in order to get more information from it. The potential pitfalls in doing so are illustrated by the following example.

Example 2.3.8 Let \( R(Y, Z, T, U) \) be a relational table consisting of two tuples \( R(1, 1, 2) \) and \( R(2, 2, 2, 3) \). Assume that \( R \) has the binding patterns \( R(Y^I Z^I) \), and \( R(T^I U^b) \), and consider the query “select \(*\) from \( R \) where \( R.Z = R.U \). We can start by using \( R(Y^I Z^I) \), which yields two tuples \( \{(1, 1), (2, 2)\} \). Now, with the values we have for \( Z \), and using the predicate \( R.Z = R.U \), we might try to get information through \( R(T^I U^b) \). For \( Z = 1 \), there is no corresponding \( U \) value, and thus no new tuple. For \( Z = 2 \), however, there is a match, since there was a tuple in \( R \) with \( Z = 2 \). We obtain thus the tuple \( (2, 2, 2, 1) \), which was not part of the original instance of \( R \) - obviously, a false answer.

What happened? Using an equality predicate among variables in \( R \) is wrong because it would (incorrectly) induce a self-join on \( R \), while the semantics of an SQL query containing \( R \) only once in the from clause is a selection-projection of \( R \). Therefore, it makes no sense to join \( R \) with itself, otherwise than by natural join (note that the join recomposition of a fragmented relation is also done by natural join).

Sound access We define accessing \( R(\overline{X}) \) by a sequence of binding patterns to be sound if the resulting tuples are a fragment of \( R \).
Soundness conditions for using a sequence of binding patterns  Having defined the notions of fragment and sound access, we aim at determining which sequences of binding patterns of a table \( R \) can be soundly used to answer a conjunctive query \( Q \) over \( R \). We first establish an useful result.

**Proposition 2.3.2** Let \( R(\overline{X}) \) be a relational table, and \( \overline{U}, \overline{V} \) be two disjoint subsets of \( \overline{X} \). Let \( T(\overline{Y}) \) be a fragment of \( R \), such that \( \overline{U} \subseteq \overline{Y} \). If \( \overline{U} \rightarrow \overline{V} \), then \( T \Join_{\text{nat}} \pi_{\overline{U}\setminus\overline{V}}^{0}(R) \) is a fragment of \( R \).

**Proof:** If \( T \) is a fragment of \( R \), then \( T \) can be written as \( \pi_{\overline{V}} \sigma_{C(\overline{X})}(R) \). Then,

\[
T \Join_{\text{nat}} \pi_{\overline{U}\setminus\overline{V}}^{0}(R) = (\pi_{\overline{V}} \sigma_{C(\overline{X})}(R)) \Join_{\text{nat}} \pi_{\overline{U}\setminus\overline{V}}^{0}(R).
\]

If \( \overline{U} \rightarrow \overline{V} \), then to each value of \( \overline{U} \) is associated exactly one value of \( \overline{V} \), and therefore in \( \pi_{\overline{U}\setminus\overline{V}}^{0}(R) \) there is exactly one tuple for each value of \( \overline{U} \) present in \( R \). Thus, by joining \( \pi_{\overline{U}\setminus\overline{V}}^{0}(R) \) with a fragment of \( R \), we only add the \( \overline{V} \) columns to the tuples from the fragment; the result is a fragment, too.

From the above proposition, we derive a sufficient condition for soundness when using a sequence of binding patterns \( \text{seq} \) to answer a query \( Q \).

**Proposition 2.3.3** (*Soundness condition*) Let \( Q \) be the query: “select \( \overline{X}_{\text{proj}} \) from \( R(\overline{X}) \)” where \( C(\overline{X}) \). Let \( \text{seq} = [b_{p_{1}}, b_{p_{2}}, \ldots, b_{p_{k}}] \) be a usable binding pattern sequence for \( Q \), \( b_{p_{i}} = R(\overline{U}_{i}\setminus\overline{V}_{i}) \). If, for every \( i = 2, 3, \ldots, k \), the functional dependency \( \overline{U}_{i} \rightarrow \overline{V}_{i} \) holds, then \( \sigma_{C(\overline{X})}\text{acc}(R, \text{seq}, C(\overline{X})) \) is a fragment of \( Q \).

**Proof:** We proceed by induction on \( k \). When \( k = 1 \), we have \( \text{acc}(R, [b_{p_{1}}], C(\overline{X})) \) is the same as \( \text{acc}(R, b_{p_{1}}, C(\overline{X})) \), which, by equation 2.1, is a fragment of \( R \).

Now let \( \text{seq}_{i} \) be the prefix of \( \text{seq} \) of length \( i \), and assume \( \text{acc}(R, \text{seq}_{i}, C(\overline{X})) \) is a fragment of \( R \). We have \( \text{acc}(R, \text{seq}_{i+1}, C(\overline{X})) = \text{acc}(R, \text{seq}_{i}, C(\overline{X})) \Join_{\text{nat}} \pi_{\overline{U}_{i+1}\setminus\overline{V}_{i+1}}^{0}(R) \). Using proposition 2.3.2, it follows that \( \text{acc}(R, \text{seq}_{i+1}, C(\overline{X})) \) is a fragment of \( R \).

**Remark 1** If \( \text{seq} = [b_{p_{1}}, b_{p_{2}}, \ldots, b_{p_{k}}] \) is usable (proposition 2.3.1) and sound (2.3.3), the number of tuples in \( \text{acc}(R, \text{seq}, C(\overline{X})) \) is exactly the number of tuples in \( \text{acc}(R, b_{p_{1}}, C(\overline{X})) \). Indeed, due to the functional dependency \( \overline{U}_{i} \rightarrow \overline{V}_{i} \), and because of the duplicate-free semantic of the chained access, no tuple is erased or multiplied by using a binding pattern \( b_{p_{i}} \), \( i = 2, 3, \ldots, k \).

**Remark 2** Consider two sequences \( \text{seq}_{i} = [b_{p_{1}}, b_{p_{2}}, \ldots, b_{p_{i}}] \) and \( \text{seq}_{i+1} = [b_{p_{1}}, b_{p_{2}}, \ldots, b_{p_{i}}, b_{p_{i+1}}] \). Assume \( \text{seq}_{i} \) and \( \text{seq}_{i+1} \) are usable and sound for answering a given query \( Q \). Then, following the previous remark, using a binding pattern \( b_{p_{i+1}} \) after having used \( b_{p_{1}}, b_{p_{2}}, \ldots, b_{p_{i}} \) will not bring any new tuples; it may at most give values for some columns for which we did not have any values previously. These variables are: \( \text{Var}((\text{seq}_{i})) = \overline{U}_{i} \cup \overline{V}_{i} \cup \ldots \cup \overline{U}_{i} \cup \overline{V}_{i} \). So \( b_{p_{i+1}} \) may bring some new information only if \( \overline{U}_{i+1} \cup \overline{V}_{i+1} \) is not a subset of \( \text{Var}((\text{seq}_{i})) \).
Redundant sequences of binding patterns  We say a usable sequence of binding patterns \( seq = [b_{p_1}, b_{p_2}, \ldots, b_{p_k}] \), where \( b_{p_i} = R(U_i \overline{V}_i) \), is redundant when answering a conjunctive query \( Q \) over \( R \) if, for some \( i \), \( \overline{U}_i \cup \overline{V}_i \subseteq U_1 \cup \overline{V}_1 \cup \ldots \cup U_{i-1} \cup \overline{V}_{i-1} \).

Example 2.3.9  Consider the table \( R(X, Y, Z) \) such that \( b_{p}(R) = \{R(X^b Y^b), R(Y^b Z^b), R(X^b Z^b)\} \). The sequence \( [R(X^b Y^b), R(Y^b Z^b)] \) is not redundant. At the same time, the binding pattern sequence \( [R(X^b Y^b), R(Y^b Z^b), R(X^b Z^b)] \) is redundant, since \( X \) and \( Z \), present in the last pattern, were already attained by the previous two.

Remark 3  If \( seq = [b_{p_1}, b_{p_2}, \ldots, b_{p_k}] \) is usable and sound when answering the query \( Q \), then any prefix \( seq_i \) of \( seq \) satisfies these conditions, too. The intuition is that \( acc(R, seq_1, C(\overline{X})) \) is a projection of \( acc(R, seq, C(\overline{X})) \), and therefore is a fragment of \( R \). Furthermore, there may be several usable and sound sequences \( seq' \) consisting of elements from \( seq \), not necessarily in the order derived from \( seq \).

Example 2.3.10  Consider the table \( R(X, Y, Z, U) \), where \( b_{p}(R) = \{b_{p_1}, b_{p_2}, b_{p_3}\} \), with \( b_{p_1} = R(X^b Y^b), b_{p_2} = R(X^b Z^b), b_{p_3} = R(Z^b U^b) \). Assume that \( X \rightarrow Z \) and \( Z \rightarrow U \), and let \( Q \) be: “select \( R.Y, R.Z, R.U \) from \( R \) where \( R.X = 5 \)”. Then, the sequence \( seq_1 = [b_{p_1}, b_{p_2}, b_{p_3}] \) is usable and sound in order to answer \( Q \), but so are \( seq_2 = [b_{p_1}, b_{p_2}] \) and \( seq_3 = [b_{p_2}, b_{p_1}, b_{p_3}] \).

As a consequence of this remark, we need to characterize when a binding pattern sequence \( seq \) is sufficient for completeness in order to answer a query \( Q \). This is the goal of the following proposition.

Proposition 2.3.4  Consider the query \( Q \): “select \( \overline{X}_{proj} \) from \( R(\overline{X}) \) where \( C(\overline{X}) \)”. Let \( seq \) be a sequence of binding patterns of \( T \) that is usable with respect to \( Q \) and sound.

1. (Completeness condition) With the notation given in equation 2.3, it is possible to obtain an answer for \( Q \) using only the binding patterns in \( seq \) if and only if \( \overline{X}_{res} \subseteq Var(seq) \).

2. (Query feasibility) \( Q \) is feasible if and only if there exists some binding pattern sequence \( seq \) over \( bp(T) \) that is usable for \( Q \), sound, and such that \( \overline{X}_{proj} \subseteq Var(seq) \).

Proof:

1. We need \( \overline{X}_{proj} \) to be part of \( Var(seq) \) so that we can perform the final projection. For some of the variables \( X \in Var(C(\overline{X})) \), the values \( c \) taken from the selections \( X = c \) will be provided as inputs to some binding patterns in \( seq \), and therefore such \( X \) variables are part of \( Var(seq) \). Values for the rest of the variables from \( Var(C(\overline{X})) \) need to be obtained, so that we can enforce the selection conditions specified on them; therefore, \( Var(C(\overline{X})) \subseteq Var(seq) \). Finally, values for the variables appearing in \( Var(J(\overline{X})) \) have to be obtained, in order to apply the \( J(\overline{X}) \) predicates.
We illustrate sequence completeness on two examples.

**Example 2.3.11** Let \( R(X, Y, Z) \) be a table, with \( bp(R) = \{ R(X^bY^f), R(Y^bZ^f) \} \), and let \( Q_1 \) be the query “select \( R.X, R.Y \) from \( R \) where \( R.X = 5 \)”. In this case, \( \overline{X}_{\text{proj}} = \{ X, Y \}, \var{C(\overline{X})} = \{ X \}, \var{J(\overline{X})} = \emptyset \). Assume that there is no functional dependency on \( R \). Then, the only sequence that is sound and usable for \( Q \) is \( seq_1 = [bp_1] \); we have \( \var{seq_1} = \{ X, Y \} \) and thus \( seq_1 \) is complete to answer our query, since \( \overline{X}_{\text{res}} \subseteq \var{seq_1} \). Indeed, \( acc(R, bp_1, C(\overline{X})) \) yields \( \pi_{X,Y} \sigma_{X=5}(R) \), which is exactly the query result. However, if we change the query to \( Q_2 \): “select \( R.X, R.Y \) from \( R \) where \( R.X = 5 \) and \( R.Y = R.Z \)”, now \( \var{C(\overline{X})} = \{ X, Z \} \), and the query cannot be answered, since \( \var{seq_1} \) does not include \( \var{C(\overline{X})} \). Indeed, after using \( bp_1 \), we only obtained the \( X, Y \) columns, \( bp_2 \) cannot be soundly used, and we cannot check whether \( Y = Z \). Now assume \( Y \rightarrow Z \); then, \( seq_1 = [bp_1, bp_2] \), \( \var{seq_1} = \{ X, Y, Z \} \), therefore \( Q_1 \) and \( Q_2 \) can be answered using the binding patterns in \( seq_1 \).

**Example 2.3.12** Let \( R(X, Y, Z, U) \) be a table, with

\[
bp(R_1) = \{ bp_i, 1 \leq i \leq 5 \} = \{ R(X^bY^f), R(Y^bZ^f), R(Y^fZ^b), R(Z^bU^f), R(Z^fU^b) \}
\]

Assume \( Y \rightarrow Z, Z \rightarrow Y \), and \( Z \rightarrow U \). Consider the query \( Q \): “select \( R.Z \) from \( R \) where \( R.Y = 5 \) and \( R.U = 3 \)”. There are two sound, usable and complete sequences for answering \( Q \): \( seq_1 = [bp_2, bp_4] \), and \( seq_2 = [bp_5, bp_3] \). In this case, \( \var{seq_1} = \{ Y, Z, U \} \), \( \var{seq_2} = \{ Y, Z, U \} \). Using \( seq_1 \), respectively \( seq_2 \) we obtain the same result: \( \pi_{Z} \sigma_{Y=5 \land U=3}(R) \).

**The BPSeq algorithm**

From Proposition 2.3.3 and the preceding remarks, we derive the BPSeq algorithm for computing all sound, non-redundant, and complete binding pattern sequences over \( bp(R) \) that are usable for answering the conjunctive query \( Q \) over \( R \). The algorithm is shown in figure 2.3.

For each binding pattern \( bp_i \) whose bound variables are covered by the selections in \( C(\overline{X}) \), BPSeq invokes a recursive algorithm \( \text{recBPSeq} \) on the binding pattern sequence consisting of \( bp_i \). The role of \( \text{recBPSeq} \) is to augment a given sequence of binding patterns, that is sound, usable and non redundant, in all possible ways, such that the resulting sequence has the same properties.

RecBPSeq receives as parameters the sequence to be enhanced, \( seq \), and the given query \( Q \). If the sequence is already complete, recBPSeq exits, since there is no need to augment a complete sequence. Otherwise, the algorithm attempts to identify binding patterns \( bp_j \)
Algorithm BPSeq

INPUT: $bp(R), Q : \text{select } \overline{X}_{\text{proj}} \text{ from } R(\overline{X}) \text{ where } C(\overline{X}) \text{ and } J(\overline{X})$

OUTPUT: \{ $seq^j = [bp^1_j, bp^2_j, \ldots, bp^n_j], bp^i_j \in bp(R) \mid seq^j \text{ usable for } Q, \text{ sound, non-redundant and complete} \}$

let $S = \emptyset$

foreach $bp_i \in bp(R)$
    if $\overline{U}_i \subseteq Var(C(\overline{X}))$
        then let $S = S \cup \text{recBPSeq}([bp_i], Q)$

return $S$

Algorithm recBPSeq

INPUT: $seq$ sequence of binding patterns, $Q$

OUTPUT: \{ $seq^j = [bp^1_j, bp^2_j, \ldots, bp^n_j], bp^i_j \in bp(T) \mid seq^j \text{ usable for } Q, \text{ sound, non-redundant and complete, } seq \text{ is a prefix of } seq^j \}$

if $\overline{X}_{\text{res}} \subseteq Var(seq)$
    then return \{ $seq$ \}
else let $S_{aux} = \emptyset$
    foreach $bp_j \in bp(T), bp_j \notin seq$
        if $\overline{U}_j \rightarrow \overline{V}_j$ and $Var(bp_j) \not\subseteq Var(seq)$
            then if $\forall X \in \overline{U}_j, (X \in Var(C(\overline{X})) \text{ or } X \in Var(seq))$
                then let $S_{aux} = S_{aux} \cup \text{recBPSeq}(seq||[bp_j], Q)$

return $S_{aux}$

Figure 2.3: Algorithm BPSeq for computing the binding patterns sequences over $bp(R)$ that may be used to answer query $Q$. 
that can be appended to seq such that seq || [bp_j] is usable, sound and non-redundant (here, || stands for list concatenation).

First, we check that \( \text{\textit{U}}_j \rightarrow \text{\textit{V}}_j \) for soundness, and we verify that bp_j is not redundant through the condition \( \text{Var}(bp_j) \not\subseteq \text{Var}(seq) \). To make sure bp_j satisfies the usability condition given by proposition 2.3.1, we verify that all variables in \( \text{\textit{U}}_j \) are covered by selections in \( C(\overline{X}) \), or are included in \( \text{Var}(seq) \). If bp_j satisfies these conditions, it is appended to seq and the algorithm is re-invoked on seq || [bp_j].

Due to the way \textsc{BPSeq} constructs its sequences, no returned sequence is a prefix of another. Furthermore, the resulting sequences, together with all their prefixes, are all the usable, sound, non-redundant and complete binding pattern sequences over \( R \).

**Example 2.3.13** Consider the table \( R \) and the binding patterns given in the example 2.3.12. By inspecting the conditions in \( C(\overline{X}) \), the \textsc{BPSeq} algorithm identifies two starting sequences: \([bp_2] \) and \([bp_3] \). Launching rec\textsc{BPSeq} on \([bp_2] \) produces \([bp_2, bp_4] \), which is complete, therefore it is no longer augmented, and is returned to \textsc{BPSeq}. Launching rec\textsc{BPSeq} on \([bp_5] \) yields \([bp_5, bp_3] \) which is complete, and cannot be augmented (adding \( bp_2 \) or \( bp_4 \) would be redundant). Therefore, \textsc{BPSeq} exits with the two binding pattern sequences that can be used ways of solving \( Q \): \([bp_2, bp_4] \) and \([bp_5, bp_3] \). The answer to \( Q \) can be computed, alternatively, as:

\[ \pi_Z \sigma_{U=3} \text{acc}(R, [bp_2, bp_4], C(\overline{X})) \text{ or } \pi_Z \sigma_{Y=5} \text{acc}(R, [bp_5, bp_3], C(\overline{X})) \]

Note that using any of the binding pattern sequences computed by the algorithm \textsc{BPSeq}, we obtain a superset of the answer to query \( Q \); a final selection-projection step may be necessary in order to compute the exact answer.

2.3.3 Answering queries over several tables with binding patterns

In this section, we aim at generalizing Proposition 2.3.4 into necessary and sufficient conditions for a conjunctive query \( Q \) over \( n \) tables to be feasible.

To answer \( Q \), we need to use binding patterns of all the \( n \) tables, and perhaps several binding patterns for a single table. Therefore, we define, in a manner similar to equation 2.5, the semantics of accessing \( n \) tables, using a sequence of binding patterns seq = \([bp_1, bp_2, \ldots, bp_k] \), in which each \( bp_k \in \{bp(R_1) \cup bp(R_2) \cup \ldots \cup bp(R_k)\} \).

- If \( bp_1 \) is a binding pattern of \( R_j \),
\[ \text{acc}\{R_1, R_2, \ldots, R_n\}, [bp_1], C(\overline{X}), J(\overline{X}) \} = \text{acc}(R_j, bp_1, C(\overline{X})) \]

- If \( bp_{i+1} = R_k(\overline{U}^j \overline{V}^j) \), and \( bp_{i+1} \) is the first occurrence in \( seq \) of a binding pattern of \( R_k \),
\[
\text{acc}\{R_1, R_2, \ldots, R_n\}, seq_{i+1}, C(\overline{X}), J(\overline{X}) \} =
\text{acc}\{R_1, R_2, \ldots, R_n\}, seq_i, C(\overline{X}), J(\overline{X}) \} \bowtie_{J(\text{Var}(seq_i), \overline{V})} \pi_{\overline{V} \cup \overline{\sigma}} \sigma_{C(\overline{U})}(R_k)
\] (2.6)
• If \( bp_{i+1} = R_k(\overline{U}{\overline{V}}^f) \), and \( bp_{i+1} \) is not the first occurrence in \( \text{seq} \) of a binding pattern of \( R_k \),

\[
\begin{align*}
\text{acc}(\{R_1, R_2, \ldots, R_n\}, \text{seq}_{i+1}, C(\overline{X}), J(\overline{X})) = \\
\text{acc}(\{R_1, R_2, \ldots, R_n\}, \text{seq}_i, C(\overline{X}), J(\overline{X})) \land_{\text{nat}} \pi^U_{U \cup V} \sigma_{C(\overline{U})}(R_k)
\end{align*}
\]

(2.7)

The difference between the two last items above consists only of using a regular versus a duplicate-free projection on \( R_k \), and using arbitrary join conditions versus natural join.

**Proposition 2.3.5** Let \( Q \) be “select \( \overline{X}_{\text{proj}} \) from \( R_1(\overline{X}_1), R_2(\overline{X}_2), \ldots, R_n(\overline{X}_n) \) where \( C(\overline{X}) \) and \( J(\overline{X}) \)”.

Then \( Q \) is feasible if and only if there exists a sequence \( \text{seq} = [bp_1, bp_2, \ldots, bp_k] \) of \( k \) binding patterns, \( k \geq n \), where every \( bp_j \) from \( \text{seq} \) belongs to some \( bp(R_i) \), \( i = 1, 2, \ldots, n \), such that:

1. (Usability) \( bp_1 \) is usable, given the bindings in \( C(\overline{X}) \).

2. (Usability) for every \( i = 1, 2, \ldots, k-1 \), such that \( bp_{i+1} \in R_j \), if \( bp_{i+1} \) is the first binding pattern from \( bp(R_j) \) to appear seq, the bound variables of \( bp_{i+1} \) are either covered by selections in \( C(\overline{X}) \), or appear in \( \text{Var}(\text{seq}_i) \), or are covered by joins with variables from \( \text{Var}(\text{seq}_i) \).

3. (Usability) for every \( i = 1, 2, \ldots, k-1 \), such that \( bp_{i+1} \in R_j \), if \( bp_{i+1} \) is not the first binding pattern from \( bp(R_j) \) to appear seq, the bound variables of \( bp_{i+1} \) are either covered by selections in \( C(\overline{X}) \), or appear in \( \text{Var}(\text{seq}_i) \).

4. (Soundness) For every \( i = 1, 2, \ldots, n \), let \( \text{seq}_{R_i} \) be the sub-sequence of \( \text{seq} \) consisting only of binding patterns from \( bp(R_i) \). Then, for all binding pattern \( bp = R_i(\overline{U}{\overline{V}}^f) \) in \( \text{seq}_{R_i} \) but the first, \( U \rightarrow V \).

5. (Completeness) \( \overline{X}_{\text{res}} \subseteq \text{Var}(\text{seq}) \), where \( \overline{X}_{\text{res}} \) is defined with respect to \( Q \) as in equation 2.3.

**Proof:**

\( \Rightarrow \)\:

The condition 1 has to be true, otherwise there is no way to start answering the query: no table can be accessed, given their binding pattern sets, and the binding patterns supplied by the query. Furthermore, in order for the binding pattern \( bp_{j+1} \) to be usable, bindings for its values have to be provided, by inclusion, selection or join. We have distinguished among the two cases, whether \( bp_{j+1} \) is the first binding pattern used to access \( R_j \) or not, in order to avoid erroneous self-joins of \( R_j \) with itself, as illustrated in example 2.3.8.

Note that we have not included the possibility that bindings for \( bp_{j+1} \) be obtained by extending \( \text{seq}_j \) with some binding patterns that do not appear in \( \text{seq} \). (If the query can only be answered this way, let \( \text{seq}' \) be the extension of \( \text{seq} \) with these hidden binding patterns, and the condition 2 above will hold for \( \text{seq}' \).

The soundness condition has to be verified, otherwise, by chaining the accesses in \( \text{seq} \), we obtain false results. To see why, let \( \text{seq}_j = [bp_1, bp_2, \ldots, bp_i, \ldots, bp_j] \) be a prefix of \( \text{seq} \),
such that $bp_i$ and $bp_j$ are the first and the second binding patterns of a given $R_k$ appearing in $seq$. By equation 2.6, using $seq_{i-1}$ yields a result of the form $\pi_{\overline{Y}} \sigma_{E(X)}(S \bowtie R_k)$, for some $\overline{Y} \subseteq \overline{X}$, and selection conditions $E(X)$ (we used the join commutativity to isolate $R_k$ from the rest of the joined tables, represented by $S$). When using $bp_j$ after this sequence, by equation 2.7, the result is of the form
\[
\pi_{\overline{Y}} \sigma_{C(\overline{Z})}(S \bowtie R_k) \bowtie \pi^{\ast}_{U \cup \overline{Y}} \sigma_{C(\overline{U})}(R_k)
\]

This expression is a fragment of $S \bowtie R_k$ if and only if $\overline{U} \rightarrow \overline{V}$.

Let us show that the completeness condition is necessary. The variables in $\overline{X}_{proj}$ need to be covered by $seq$ in order to be returned. Part of the predicates in $C(\overline{X})$ have been used to provide bindings for some restricted binding pattern in $seq$, while the other selection predicates need to be verified on the result, therefore we have $Var(C(\overline{X})) \subseteq Var(seq)$. By a similar reasoning, $Var(J(\overline{X})) \subseteq Var(seq)$, therefore $\overline{X}_{res} \subseteq Var(seq)$.

"≤":

From the usability conditions it follows directly that we are able to compute $acc(\{R_1, R_2, \ldots, R_n\}, seq, C(\overline{X}), J(\overline{X}))$. Using the equations 2.6 and 2.7, and the soundness condition, the result of $acc(\{R_1, R_2, \ldots, R_n\}, seq, C(\overline{X}), J(\overline{X}))$ is a fragment of $(R_1 \times R_2 \times \ldots \times R_n)$. By the completeness condition, we are able to apply all the predicates in $C(\overline{X})$ and $J(\overline{X})$, and also to project the variables in $\overline{X}_{proj}$, thus computing the exact answer for $Q$.

We illustrate the feasibility of conjunctive queries with some examples.

**Example 2.3.14** Consider the tables $R_1(X, Y), R_2(Z)$, and let $bp(R_1) = \{R_1(X^bY^f)\}$ and $bp(R_2) = \{R_2(Z^f)\}$. The following query cannot be answered, since there is no way of accessing the information in $R_1$:

$$Q_1 : \text{select } R_1.X, R_1.Y, R_1.Z \text{ from } R_1, R_2 \text{ where } R_1.Y = R_2.Z$$

If we modify the where clause into “where $R_1.X = R_2.Z$”, the query becomes feasible, since the values of $R_2.Z$ are available, and may be used to extract data from $R_1$. In a similar manner, if the where clause contains a selection predicate of the form $R_1.X = c_1$, where $c_1$ is some constant, the query is also feasible.

Consider the tables $R_1(X, Y), R_2(U)$, and let $bp(R_1) = \{R_1(X^f), R_1(X^bY^f)\}$, $bp(R_2) = \{R_2(U^f)\}$. Also, consider the query $Q$ : “select $R_1.Y$ from $R_1, R_2$ where $R_1.X = R_2.U$”. If $X \rightarrow Y$, then $Q$ is feasible: access $R_1(X^f)$, join with $R_2(U^f)$, and retrieve the corresponding $Y$ values by $R_1(X^bY^f)$. Otherwise, $Q$ is unfeasible.

**Remark 4** The proposition 2.3.5 requires only some binding pattern sequence $seq$. Several such sequences might exist; they do not necessarily have the same length.
Algorithm `solveBP`

**INPUT**
- `Q`: “select \( X_{\text{proj}} \) from \( R_1(\overline{X}_1), R_2(\overline{X}_2), \ldots, R_n(\overline{X}_n) \) where \( C(\overline{X}) \) and \( J(\overline{X}) \)”

**OUTPUT**
- \( \{ \text{seq binding pattern sequence over } bp(R_1) \cup bp(R_2) \cup \ldots \cup bp(R_n) \mid \text{seq usable, sound, non-redundant and complete for } Q \} \)
- let \( S = \emptyset, \text{bps} = bp(R_1) \cup bp(R_2) \cup \ldots \cup bp(R_n) \)
- foreach \( bp = R_k(\overline{U}^{h}\overline{V}^{f}) \in \text{bps} \) such that \( \overline{U} \subseteq \text{Var}(C(\overline{X})) \)
  - let \( S = S \cup \text{recSolveBP}(Q, \text{bps}, [bp]) \)
- return \( S \)

Algorithm `recSolveBP`

**INPUT**
- `seq`: binding pattern sequence, `bps`, `Q`

**OUTPUT**
- \( \{ \text{seq’ binding pattern sequence over } \text{bps} \mid \text{seq’ usable for } Q, \text{ sound, non-redundant and complete, seq is a prefix of seq’} \} \)
- if \( \overline{X}_{\text{res}} \subseteq \text{Var}(seq) \)
  - then return \( \{seq\} \)
- else let \( S_{\text{aux}} = \emptyset \)
  - foreach \( bp = R_k(\overline{U}^{h}\overline{V}^{f}) \in \text{bps}, bp \notin seq, \overline{U} \cup \overline{V} \not\subseteq \text{Var}(seq) \)
    - if `seq` already contains a binding pattern of \( R_k \)
      - then let `isFirst`=false
    - else let `isFirst`=true
    - if (isFirst and \( \overline{U} \not\supseteq \overline{V} \))
      - then continue
    - if \( \forall X \in \overline{U}, (X \in \text{Var}(C(\overline{X})) \text{ or } X \in \text{Var}(\text{seq}) \text{ or } (\text{isFirst and } J(\overline{X}) \text{ contains } X = Y, \text{ where } hY \in \text{Var}(\text{seq})) \)
      - then let \( S_{\text{aux}} = S_{\text{aux}} \cup \text{recSolveBP}(Q, \text{bps}, \text{seq}[[bp]]) \)
- return \( S_{\text{aux}} \)

---

**Example 2.3.15** Consider the tables \( R(X, Y, Z) \) and \( S(U, V) \), such that \( bp(R) = \{(X^{f}Z^{f}), (X^{b}Y^{f}), (Y^{b}Z^{f})\} \) and \( bp(S) = \{S(U^{h}V^{f})\} \). Let \( Q \) be the query “select \( R, Z, S, V \) from \( R, S \) where \( R.X = 5 \) and \( R.X = S.U \) and \( R.Z = S.U \)”. One binding pattern sequence for answering \( Q \) is \([R(X^{f}Z^{f}), S(U^{h}V^{f})]\); in this case, \( R.X = 5 \) is applied as a selection after having accessed \( R \). Alternatively, the sequence \([R(X^{f}Y^{f}), S(U^{h}V^{f}), R(Y^{b}Z^{f})]\) may also be used to answer \( Q \), and in this case, \( R.X = 5 \) is used as binding for the first binding pattern.

**The solveBP algorithm**

From the last proposition, we derive the `solveBP` algorithm, shown in figure 2.4. Given a conjunctive query \( Q \), this algorithm computes all usable, sound, non redundant and complete (with respect to \( Q \)) binding pattern sequences over \( bps = bp(R_1) \cup bp(R_2) \cup \ldots \cup bp(R_k) \).
This algorithm is very similar to $\text{bpSeq}$, that we had used in the particular case of queries over a single table. It constructs a new sequence for all binding patterns that satisfy the first usability condition in proposition 2.3.5, and calls the recursive $\text{recSolveBP}$ algorithm to augment these core sequences in all possible usable, sound and complete ways.

The $\text{recSolveBP}$ algorithm returns the input sequence as-is if it is already complete. Otherwise, the algorithm tests, for each binding pattern not already included in its input sequence, the usability and soundness conditions provided in proposition 2.3.5. If these conditions are satisfied, $\text{recSolveBP}$ is recursively invoked on a sequence one step longer than the input one.

The $\text{solveBP}$ algorithm is the extension, for bag semantics, of the query feasibility check we have presented in [39].

**Complexity** The worst-case complexity of $\text{solveBP}$ is $m!$, where $m$ is the number of binding patterns in $\text{bps}$, $m \geq n$; in practice, since the algorithm builds its sequences incrementally and drops a prefix as soon as it cannot be continued, it is likely to do significantly less work than blindly enumerating all possible sequences. Note that $\text{solveBP}$ tries to find an ordering for accessing $m$ tables, and solves a problem very similar to the search for left-deep QEPs for a query over $m$ tables, whose complexity is $m!$ [84]. In our case, it is the number of binding patterns, not the number of tables, that determines the complexity, since, under the conditions we specified, we are able to access a table following several binding patterns.

### 2.3.4 Answering disjunctive queries

Until now, we were only concerned with conjunctive queries. Let us now examine the feasibility conditions for disjunctive queries. A query with disjunctions, in disjunctive normal form, has the following general form:

$$\text{select } X_{\text{proj}} \text{ from } R_1(\overline{X}_1), \ldots, R_n(\overline{X}_n) \text{ where } C_1(\overline{X}) \lor C_2(\overline{X}) \lor \ldots \lor C_m(\overline{X}) \quad (2.8)$$

where each $C_i(\overline{X})$ is a conjunction of predicates $C_{i,j}(\overline{X})$ of one of the following forms: (i) $X_k = X_l$, (ii) $X_k = c$, where $X_k, X_l \in \overline{X}$.

We are interested in the conditions on $\text{bp}(R_1), \text{bp}(R_2), \ldots, \text{bp}(R_n)$ and on the query $Q$ that guarantee $Q$ is feasible.

**Proposition 2.3.6** Let $Q$ be the query “select $X_{\text{proj}}$ from $R_1(\overline{X}_1), R_2(\overline{X}_2), \ldots, R_n(\overline{X}_n)$ where $C_1(\overline{X}) \lor C_2(\overline{X}) \lor \ldots \lor C_m(\overline{X})$”. For every $i = 1, 2, \ldots, m$, let $Q_i$ be the query

$$\text{select } X_{\text{proj}} \text{ from } R_1(\overline{X}_1), R_2(\overline{X}_2), \ldots, R_n(\overline{X}_n) \text{ where } C_i(\overline{X})$$

Then, $Q$ is feasible if and only if all $Q_i$s are feasible.

**Proof:**

“$\Rightarrow$”

If all $Q_i$ are feasible, then the answer to $Q$ can be computed as follows. Let $S = \{Q_1, Q_2, \ldots, Q_m\}$, and $P(S)$ be the power set of $S$. Then,
\[ Q = \bigcup_{s \in P(S)} \left( (\bigcap_{Q_i \in s} Q_i) \setminus (\bigcup_{Q_j \in S \setminus s} Q_j) \right) \]

Thus, \( Q \) can be computed as the union of \( 2^m - 1 \) terms. We have to compute all these terms in order to make sure that a tuple satisfying more than one condition \( C_i \) appears the correct number of times in the result; taking the simple union of \( Q_i \)'s would introduce spurious duplicates. As before, all operations are to be considered with bag semantics.

\( \Rightarrow \)

Assume that for some \( i, Q_i \) is not feasible. Our goal is that the tuples in the answer to \( Q_i \) appear exactly once in the result, whether they are also in the answer of another query \( Q_j \), with \( i \neq j \), or not.

Since \( Q_i \) is not feasible, there is no direct way of obtaining exactly the tuples in its answer. There are two cases:

1. The answer to \( Q_i \) is included in (the union of) the answers to other feasible queries. Then, we are able to attain all tuples needed for \( Q_i \) by taking the union of these other queries, but since we do not know the exact answer to \( Q_i \), we do not know how many times each tuple appears. Note that this problem would not exist if we were using set semantics.

2. Some tuples in the answer to \( Q_i \) are not included in the answers to other queries \( Q_j \); in this case, part of the answer to \( Q \) cannot be computed.

\[ \blacksquare \]

**Example 2.3.16** Consider a table \( R(X, Y; Z) \), with the binding pattern set \( \{R(X^iY^jZ^k), R(X^iY^jZ^k)\} \), and the query \( Q \): “select \( X, Y, Z \) from \( R \) where \( Y = 2 \) or \( Z = 4 \).” Let \( Q_1 \) be the query “select \( X, Y, Z \) from \( R \) where \( Y = 2 \)”, and \( Q_2 \) be “select \( X, Y, Z \) from \( R \) where \( Z = 4 \).” Then, we can compute \( Q = (Q_1 \cap Q_2) \cup (Q_1 \triangle Q_2) \), where \( \triangle \) stands for the symmetrical difference.

The above proposition describes the cases when \( Q \) is feasible; however, answering \( Q \) by using the formula provided in the proof is in general prohibitively expensive. Two important particular cases merit some attention. We use the notations from proposition 2.3.6. First, if we know that all \( C_i \) are mutually exclusive, then the contents of \( Q_i \) are mutually disjoint, and \( Q \) can be computed by taking the union of all \( Q_i \).

Now consider the conjunctive normal form of the query \( Q \):

select \( X_{\text{proj}} \) from \( R_1(X_1), R_2(X_2), \ldots, R_n(X_n) \) where \( D_1(X) \land D_2(X) \land \ldots \land D_k(X) \)

in which each \( D_l(X) \) is a disjunction of atomic predicates. Furthermore, assume that for some \( l, 0 \leq l \leq k \), the first \( l \) terms of the form \( D_i(X) \) consist of only one predicate, and let \( E(X) \) be the conjunction of these \( l \) terms. Thus, we can write \( Q \) as

\[ Q: \text{select } X_{\text{proj}} \text{ from } R_1(X_1), R_2(X_2), \ldots, R_n(X_n) \text{ where } E(X) \land D_{l+1}(X) \land \ldots \land D_k(X) \]

Let \( Q' \) be the following query:
\( Q : \text{select } \overline{X}_{proj} \cup \text{Var}(D_{i+1}(X)) \cup \ldots \cup \text{Var}(D_k(X)) \)

from \( R_1(\overline{X}_1), R_2(\overline{X}_2), \ldots, R_n(\overline{X}_n) \) where \( E(X) \)

If we are able to answer \( Q' \), then all the disjunctive predicates in \( D_{i+1}, \ldots, D_k \) can be applied as final selections on the result of \( Q' \). The feasibility of \( Q' \) implies that of \( Q \), but it is a stronger condition, that is, it may be the case that \( Q' \) is not feasible, while \( Q \) is.

**Example 2.3.17** Consider a table \( R(X,Y,Z) \), with the binding pattern set \( \{R(X^bY^fZ^f), R(X^fY^bZ^f), R(X^fY^fZ^b)\} \), and the query \( Q \)

select \( X, Y, Z \) from \( R \) where \( (X = 5) \) and \( (Y = 2 \text{ or } Z = 4) \)

Now \( Q_1 \) is the “select \( X, Y, Z \) from \( R \) where \( X = 5 \text{ and } Y = 2 \)”, and \( Q_2 \) is “select \( X, Y, Z \) from \( R \) where \( X = 5 \text{ and } Z = 4 \)”. Both \( Q_1 \) and \( Q_2 \) are feasible, and we can compute \( Q \) using \( Q_1 \) and \( Q_2 \) as in the previous example. Also, the query \( Q' \) below is feasible, and can alternatively be used to answer \( Q \):

select \( X, Y, Z \) from \( R \) where \( X = 5 \)

Now, if we eliminate the first binding pattern from \( bp(R) \), \( Q' \) is no longer feasible. However, \( Q_1 \) and \( Q_2 \) are feasible, and can still be used to compute the answer to \( Q \).

We have thus shown that the feasibility of disjunctive queries can be decided by investigating the feasibility of several disjunctive queries, although at the expense of some costly computations. Therefore, in the rest of this thesis, unless otherwise specified, we focus on conjunctive queries only.

### 2.4 Publishing resources using binding patterns

In a data integration system, a wrapper has two important functions. First, it provides a translation layer between the resource it wraps and the mediator's data model, enabling the execution of queries over its resource. Second, the wrapper provides a set of informations regarding the published resource, and the wrapper's query processing capabilities. Metadata concerning the published resources is useful for the optimizer, as well as for the human users browsing a catalog of available resources. If the wrapper is capable of processing subqueries, a characterization of its query processing capabilities is important for the query optimizer, since it allows it to divide the query processing tasks, in an optimal manner, among the wrappers and the mediator(s).

In the case of a relational data integration system, the resource metadata may include the table and attribute names, attribute types, and semantic information about the table. Also, even if the wrapper has no query processing capabilities, it has to supply some way of accessing its resource; estimates of these access costs are also important, since resource accesses are the basic building blocks with which the optimizer may construct its QEPs.
There is no general consensus as to how to specify wrapper capabilities. They may be specified in terms of algebraic operators, with more or less details as to which subset of operations are supported; as a list of parameterized queries that the wrapper is able to process; or they may not be specified at all, in which case, the query optimizer "learns" what the wrapper is capable of doing, every time a new query is optimized.

In the following, we describe the resource metadata and wrapper processing capabilities exported by wrappers in LeSelect. We then compare the approach used in LeSelect for resource modeling and wrapper capabilities description with alternative solutions to these problems, presented in other works.

The following steps are required in order to publish a resource (data collection or program) in LeSelect: (1) describe it as one or more relational tables, (2) obtain a wrapper for it and (3) register the wrapper with a LeSelect server. Thus, on any publishing site, a LeSelect server has to be running.

In LeSelect, we aim at keeping the effort required for publishing a resource as low as possible. In this section we describe the simple metadata and minimal query processing capabilities required from a LeSelect wrapper to enable the efficient execution of queries over tables with binding patterns.

### 2.4.1 Resource metadata

A wrapper $W$ manages a set of tables $\{R_1, R_2, \ldots, R_n\}$, where each $R_i$ corresponds to a resource. For every table $R$, $W$ exports two kinds of metadata.

The first one concerns the definition of the table: columns number, names and types, as well as a set of binding patterns over $R$’s attributes $\{bp_1, \ldots, bp_R\}$. Formally, for each $R(\overline{X})$, the wrapper declares:

- the expected cardinality (number of tuples) in $R$, $n_R$
- the set $bp(R)$ of binding patterns available for $R$,
- the closure set $F^+$ of functional dependencies that $R$ verifies.

Furthermore, for each attribute $X \in \overline{X}$, the wrapper declares

- the atomic type $\tau_X$ of $X$,
- optionally, the average size in bytes $s_X$ of $X$,
- optionally, the percentage of distinct values in the column $R.X$, $dv_X$

The size of an attribute defaults to a value determined by its type. The last parameter, $dv_X$, allows us to estimate the number of duplicates in the result of a partial query execution plan (QEP). The default value is 1.00, i.e., unless specified otherwise, we assume all values are distinct. The number of distinct values in $R.X$ is estimated as $n_R \cdot dv_X$. 

42
The second kind of metadata consists of statistics on the data published via $R$, which are intensively exploited by the query optimizer. The wrapper exports a set of parameters for each binding pattern $bp = \overline{R(U^lV^l)}$:

- the average startup cost (in milliseconds), $c^0_{bp}$. We consider $c^0_{bp}$ to be the time elapsed between the moment when the restricted resource is accessed, and the moment when the first resulting tuple is returned.

- the average per-tuple access cost (in milliseconds), $c_{bp}$. When tuples are returned as the result of an access, $c_{bp}$ represents the average time elapsed between two tuples being received. $c_{bp}$ is to be interpreted as the total time between the return of the first and last tuple, divided by the total number of tuples returned.

- if $bp$ is restricted, the wrapper exports the average number of returned tuples per tuple of arguments, $s_{bp}$. The $s_{bp}$ value reflects the selectivity of the access following $bp$, since there may be no answer for a given argument set.

- the optimal number of parallel accesses to the resource ($opt_\|_p$). In some cases, this number may be statically determined; for example, if a program owner knows that a batch of five computers is available to run the program in parallel, then $opt_\|_p = 5$. In other cases, $opt_\|_p$ is very difficult to predict since it may depend on parameters like machine load, network speed, etc. In section 4.4.6 we will discuss how to estimate $opt_\|_p$ experimentally.

### 2.4.2 Wrapper capabilities

The query capabilities of a wrapper are provided as the following set of boolean methods:

- canDoEqualityTest()
- canDoAnyTest()
- canDoJoin()
- canDoUnion()
- canDoProject()
- canDoEvaluate()

In LeSelect, a wrapper can only apply an operator to resources that it publishes, or that it has produced by applying some operators. The methods canDoEqualityTest() and canDoAnyTest() describe the ability of the wrapper to perform a selection; we only distinguish between simple equality tests, and any other more complicated predicates. By definition, canDoAnyTest()=true implies canDoEqualityTest()=true.

The methods canDoJoin() specifies whether the wrapper is capable of evaluating a Cartesian product. The method canDoProject() returns true iff the wrapper is capable of performing a simple projection, i.e., discarding some columns of the input. The canDoUnion() method specifies whether the wrapper has some implementation of a logical $n$-ary Union operator. The canDoEvaluate() method returns true if the wrapper is capable of evaluating an arithmetic expression defined over the columns of a set of tuples, and append to each tuple the result of the evaluation (we will discuss Evaluate in more details when describing LeSelect’s query optimization algorithm, in section 3.7.1).
Note that `canDoJoin()` and `canDoEqualityTest()` are orthogonal: both are necessary in order for the wrapper to be able to perform an equi-join. If only `canDoJoin()` is true, then the wrapper may only perform Cartesian products.

The optimizer applies a set of simple rules to decide whether the capabilities of a wrapper $W$ make it capable of executing a specific operation $op$; these rules are provided in section 3.7.3, when we discuss the query optimization process.

At query execution time, a wrapper agrees to execute a sub-plan $sQEP$ of a global query plan iff (a) it publishes all the resources in the $sQEP$’s leaf nodes and (b) the wrapper has declared itself as being capable of executing all operators found in the internal $sQEP$’s nodes.

**Required query processing capabilities** To keep the publication process simple, we require only minimal query processing capabilities from a wrapper. First, it has to execute “leaf” operators providing access to its resources:

- For each binding pattern $bp$ of the form $R_i(\bar{v}^f)$, the wrapper must accept $\text{Scan}(R_i, bp)$ queries, i.e. it must be able to provide the tuples corresponding to $acc(R, bp, \emptyset)$.

- For each binding pattern $bp$ of the form $bp = R_i(\bar{u}^\bar{v}^f)$, we require that the publisher provides a call-based interface, of the form $\text{callResource}(\bar{c})$, where $\bar{c}$ is a binding set for the required $\bar{U}$. Once this method is provided by the wrapper, we have devised a simple configurable wrapper that uses this interface to access the resource. If we denote by $(\bar{U} = \bar{c})$ the conjunction of the predicates $X = c_i$, for all $X \in \bar{U}$, then, using the equation (2.1), we have:

$$\text{callResource}(\bar{c}) = \pi_{\bar{c}}\text{acc}(R_i, bp, (\bar{U} = \bar{c}))$$

(2.9)

Second, for any table containing a blob, the wrapper must provide a method

\begin{verbatim}
readBlob(blobID, startPos, endPos, buffer)
\end{verbatim}

that allows reading the contents of the blob in data chunks. We require this method in order to ensure that the blob is physically accessible.

### 2.5 State of the art in modeling of source capabilities

**Access restrictions and wrapper capabilities** In this section, we describe several methods that have been proposed for describing the capabilities of a data source in data integration systems. We use the term “source capabilities” to designate:

- access restrictions to the data source
- query processing capabilities of the data sources
In a data integration system, data sources’ query processing capabilities are also called \textit{wrapper} capabilities. In a wrapper-mediator architecture, when some query processing can be pushed into a data source, from the mediator’s point of view it is the wrapper that accepts the task (even if the wrapper may delegate it to the underlying data source, e.g., when that source is a full-fledged DBMS).

The difference between access restrictions and wrapper capabilities is the following. When discussing wrapper capabilities, even if the wrapper cannot execute any algebraic operator, it is assumed that it is always able to provide the full contents of its data sources; therefore, as a fall back option for query processing, all the query processing may be performed by the mediator(s). Thus, wrapper capabilities are important in order to find an \textit{optimal} plan, but a plan (in which the mediator executes all the query) always exists. In contrast, in the presence of restricted access to a resource, that fall back option is not always available: there may be no plan for a given query and access restrictions to the data sources.

\subsection{2.5.1 Describing source capabilities using binding patterns}

\textbf{Original TSIMMIS approach} The usage of binding patterns for describing heterogeneous data source capabilities is pioneered in the TSIMMIS system [93]. A data source is described by a conjunctive Datalog rule, whose head relation is adorned with a binding pattern. The meaning of the source description is that the data it contains can only be accessed if bindings for the variables bound in the view head are provided. A query is expressed using the same formalism: in particular, the head of the query also has a binding pattern. An answer to the query is a semantically equivalent Datalog program, expressed only in terms of the views. The authors show that an answer to the query can be found in non-deterministic polynomial time (in the size of the query and of the set of views) if the query, the views and the answer are restricted to conjunctive Datalog programs. If arithmetic comparisons are allowed in views, queries, and query answers, finding a solution can be done in non-deterministic exponential time.

The approach we took in modeling restricted data sources is a simplified version of the one presented in [93], since we considered a source description to consist of one atom (one table with a binding pattern), instead of an annotated join query over several tables. The complexity of \texttt{solveBP} agrees with the one determined in [93], in our particular case when all views have a size of 1.

Finally, we took some extra precautions to generate sound answers in our context that includes duplicates. The soundness conditions restrict the search for complete binding pattern sequences, since they forbid the use of some binding patterns.

\textbf{Complex binding patterns} Extending the standard binding patterns proposed in [93], a more complex system of annotations is presented in [130]. Besides the \textit{b, f} and $-$ adornments, they propose using two extra annotations for a given attribute in a binding pattern, producing thus five-values adornments. An attribute mapped to $c[s]$, where $s$ is a fixed set of constants, is called \textit{constant}; its value must be specified in the query, but the value can only be chosen
from among the values in \( s \). In a similar manner, an attribute mapped to \( o[s] \) is called optional; its value may or may not be specified, but if it is, it must be set to one of the values in \( s \). A mediator is specified by a set of views, each of which consists of a specific QEP involving data sources, and relational operators. The focus of this paper is on constructing the query capabilities of the mediator, that is, deriving all possible access patterns on the mediator’s view. Thus, in order to answer a query, one has only to check if the query provides enough information to use one of the pre-computed mediator binding patterns. Deriving the mediator capabilities, unsurprisingly, is exponential in the number of data sources; the authors propose a binding pattern containment test that eliminates some of the redundant mediator binding patterns.

In contrast, in our approach, to answer a query, we dynamically combine the source capabilities, trying to gather the information that the query requested. This approach makes sense, especially since we combine the feasibility test with a cost-based query optimization process, as we will describe in chapter 3.

**MSL-based capabilities description in TSIMMIS** While [93] poses the theoretical basis for query answering using binding pattern source descriptions in annotated Datalog, the data model of TSIMMIS is the semi-structured model OEM [89]. Data sources are presented by the wrappers as queries specified in MSL [88], which is an OEM query language [43]. In [59, 74], data source descriptions are enriched with sets of data items for which values have to be supplied, in order to extract data from the sources; thus, they become parameterized MSL queries. Given an MSL query and a set of such source description, a Matcher module identifies the data sources descriptions that may be used to answer the query, and passes them to a Sequencer module that finds all valid orders for accessing the sources. In the absence of a detailed description of the Matcher and the Sequencer, it can be assumed that they fulfill, together, the role of our solveBP algorithm, i.e., finding all sequences of data source accesses that completely answer the query.

**Template-based capabilities description in the Information Manifold** In the Information Manifold, the contents of a data source is specified in terms of a hierarchy of classes [65, 69]. Also, to each data source is associated a capability description consisting of: the attribute set \( S_{in} \) for which values must be provided (similar to the bound attributes used in binding patterns), the attribute set \( S_{out} \) for which values can be obtained by feeding these inputs (corresponding to the free attributes), and the set of attributes \( S_{sel} \subseteq S_{in} \cup S_{out} \) on which the source can apply selections. The specification of the \( S_{in} \) set is more flexible than that of bound attributes in a binding pattern, since a capability description also contains a lower and an upper bound, \( n_{min} \) and \( n_{max} \), of parameters from \( S_{in} \) that have to be specified in order to obtain some information from the data source. Using \( n_{min} \) and \( n_{max} \) allows one template to describe the information that would be otherwise specified by \( \binom{k}{n_{min}} + \binom{k}{n_{min}+1} + \ldots + \binom{k}{n_{max}} \), where \( k \) is the size of \( S_{in} \). On the other hand, if not all subsets of \( S_{in} \) with at least \( n_{min} \) and at most \( n_{max} \) elements are valid inputs, then several more restricted templates have to be used.
It is interesting to note that, besides the capabilities specification, the Information Manifold uses contents description of the data sources. Since many sources may contain objects from the same class, a semantic specification of one data source’s contents is useful in order to decide whether or not that source can be used in order to answer a query. For example, if several data sources contain Cars, source $S_1$ may only contain used cars, while source $S_2$ may only contain Toyotas. To answer a query asking for a Renault, source $S_2$ is not consulted, since it cannot provide useful data. Thus, contents descriptions are likely to reduce the effort required in answering a query.

Contents descriptions in the Information Manifold are used in a context very similar to horizontal fragmentation [18]: several sources contain (possibly overlapping) fragments of one class extent. In contrast, our framework assumes that the contents of all tables are unrelated, and therefore, the data sets obtained by using binding patterns of two different tables cannot be compared. The information made available by several bindings patterns of a single table $T$, as shown by equation 2.1, are rather vertical fragments, that can be accessed providing values for the bound variables present in the fragment. This is why the soundness conditions we proposed for the correct recomposition of the table are similar to the lossless join decomposition problem.

Based on the capabilities and contents descriptions, a bucket algorithm is used to find an execution plan for a given query. The algorithm starts by collecting, for each class extent appearing in the query, a “bucket” of source descriptions that contain useful information for the query; all combinations of one element per bucket are semantically correct plans (this step has exponential complexity). A final polynomial-time goal ordering step is applied on each such plan. Our solveBP algorithm provides the same functionality as the bucket algorithm, although it does not consider content descriptions, and it supports bag semantics for accessing a binding pattern and for the query.

Finally, it is important to note that the result returned by the bucket algorithm is maximally contained in the query result, that is, it is the biggest subset of the query result that could be computed with the available source descriptions. In contrast, by the semantics we give to the access function in equations 2.1 and 2.7, we assume that, in the presence of the proper bindings, all relevant data from a table can be obtained. Thus, solveBP finds a complete answer to the query, or fails and declares that no plan exists.

Answering queries using sources with binding patterns Recent studies [72, 73] address the following problem. For a given conjunctive query over a set of sources with binding patterns, there are situations when accessing more data sources than those present in the query can yield extra ways of answering the query. However, in some cases, these accesses do not bring any new information, and even if they are possible, they are not useful. The authors characterize the situations where accessing off-query sources is feasible and useful. That work is done using set semantics, and taking the universal relation assumptions. When using bag semantics, however, accessing more tables than appear in the query cannot provide correct results, since it has been shown [24] that two SQL queries for bag semantics are equivalent only if they are identical up to renaming and reordering.
In [70], the feasibility of a wider class of queries over sources with binding patterns is addressed. Namely, the author distinguishes between stable queries, that are feasible for any instance of the relations appearing in the query, and unstable ones, whose feasibility depends on the content of these relations. With respect to this classification, our work focuses on stable queries only. Using Datalog queries with set semantics, it is shown that sometimes removing redundant subgoals from an unfeasible query may make it feasible; using bag semantics, for the reason outlined above, this technique no longer applies. Also, among the alternatives envisioned in order to solve a query, a table not appearing in the query may be used, if its columns are the same as those in the query (again, the universal relation assumption holds). In our context, we assume that distinct tables have different contents; we only have the alternative of using one or another binding pattern for the same table.

Finding the minimal view set in order to answer a query Finally, [71] addresses the following problem: given a set of conjunctive views \( \mathcal{V} \) and a conjunctive query \( Q \), what is the minimal subset of \( \mathcal{V} \) that is sufficient to answer \( Q \)? The authors introduce the notion of \( p \)-containment, denoted \( \preceq_p \), on sets of views, that captures the relation between the query answering power of those sets. Thus, if \( \mathcal{V}_\infty \preceq_p \mathcal{V}_c \), then by using \( \mathcal{V}_c \) one can answer all queries answerable by using \( \mathcal{V}_\infty \), and perhaps some more. The notion of \( p \)-containment has also a relative variant, with respect to a (finite or infinite) set of queries: one view set may allow to answer at least all queries from the set that the other one can answer, and perhaps some more. With respect to this terminology, our solveBP algorithm computes all minimal view subsets, with respect to \( p \)-containment, and to the given query \( Q \) (if we identify every binding pattern to a view). Also, the views considered in [71] are not restricted to select-project expressions, but they do not have access restrictions.

2.5.2 Describing source capabilities using an operator model

The Garlic approach Garlic is a heterogeneous data integration project developed at IBM [100, 54]. Garlic’s data model is object-oriented, and is based on ODMG [111]. Data collections integrated in Garlic are described in terms of class extents. Also, for each object in each class extent, a unique object ID is available; for each object, this ID is computed by the wrapper exporting the collection to which the object belongs, and encapsulates the name of the wrapper. Furthermore, the wrapper must provide fetch methods that retrieve the attributes of an object, given its OID. Thus, the minimal query processing capabilities required from a wrapper are: access to the OIDs of its data collections, and fetch methods for all attributes of the objects it publishes. Comparing this approach to the use of binding patterns, it amounts to enforcing (1) unique keys for all exported data items, with a binding pattern of the form \( \text{Obj}(\text{OID}_f) \), and (2) fetch methods for every attribute \( a \), that we could represent as \( \text{Obj}(\text{OID}_a^a) \). In our framework, we did not rely on the presence of unique keys, as we considered this to be a constraint too hard on the data publishers. Also, we pose no restrictions on the source binding patterns, and we attempt to use them as they are (however, for correctness when combining several binding patterns for a given source, we
pose some semantic constraints).

The reader may remember that we did require unique identifiers in the case of blobs, in section 2.2.4, and when modeling object data as described in section 2.2.5. In the context of object data, we felt it was more likely that unique IDs are available than in a relational context. Furthermore, in the case of blobs, the blobIDs are needed to enable an efficient execution technique, only as an optimization.

There are two main differences between the Garlic approach and the one taken in LeSelect, with respect to the modeling of source capabilities.

First, at a modeling level, we separate binding patterns from query processing capabilities, and explicit all access methods to a resource under the form of binding patterns. In contrast, the fetch methods used in Garlic are not described at the same level as, say, Scan plan that the wrappers may support.

Second, the capability descriptions used in LeSelect have a coarser grain of detail than what is allowed in Garlic. For example, we assume that if a wrapper can apply a selection predicate, then it can apply any selection predicate. We found this assumption reasonable when considering the classes of resources we integrated using LeSelect so far, since we had roughly two very different classes of wrappers: powerful wrappers, backed by DBMSs, that are able of executing all relational operators; and weak wrappers, typically used for publishing textual files or a restricted-access resource, that provide some (perhaps restricted) access capabilities and cannot apply relational operators.

**The Disco approach** An interesting approach for handling heterogeneous source capabilities was taken in the Disco data integration project [113]. In Disco, data is structured in class extents, as in the Information Manifold and in Garlic. A wrapper declares, for each logical operator (e.g., select, project, join) the classes of objects on which the wrapper can perform the operation, the attributes that can be given as input to these operators, and the minimal number of attributes for which values are required. Also, the classes of predicates that can be applied in a selection or a join, e.g., =, $\neq$, $\leq$, $\geq$ are specified.

The approach we have chosen in LeSelect in order to model wrapper capabilities is very similar to the one used in Disco. An important difference is that in Disco, it was assumed that if a wrapper has no query processing capabilities, it is always able to provide a Scan operator. Thus, the mediator may extract all the data from a source, and apply the remaining operators. In contrast, as explained in section 2.2, in LeSelect we do not always have the fall-back option of scanning the data source, and have to deal with restricted access patterns.
Chapter 3

Query optimization with binding patterns: principles and application in a distributed context

3.1 Introduction

So far, we have introduced the concept of table with binding patterns, and we have provided an algorithm for checking query feasibility in chapter 2. In this chapter, we show how to incorporate binding pattern access restriction in a cost-based query optimization. The idea is to explore a search space of annotated query plans, in which the annotation of a sub-plan describes which variables of the query must be given as inputs to the sub-plan.

The idea of adding binding patterns as annotations to subqueries is not new. Such annotations were used in magic-set transformations [104, 95] and for exploring sideways information passing strategies. The focus of this chapter is on incorporating such annotations into cost-based optimization and studying the effects of such annotations on the size of the space of query execution plans. Furthermore, traditional dynamic-programming based query optimization algorithms also annotate query execution plans with the set of constants necessary for executing the plan. However, the set of necessary constants is always a subset of those given in the query, while our annotations may include variable values passed from other sub-plans. As we show in our discussion, considering a larger set of annotations may lead to finding better plans.

The contributions that we make in this chapter are the following:

- We show how the presence of binding pattern limitations affects several fundamental properties of the search space, such as the need to consider different binding-pattern annotations on query execution plans, the need to explore the space of bushy trees as opposed to left-linear trees, and the specialized handling of placing selections.

To address these issues, we show how to extend the traditional cost-based query optimization to search through space of valid annotated query execution plans. In particular, a query execution plan is annotated with a set of bindings that it receives as
inputs, in addition to the classical annotations (that include subgoals that the plan
covers and whether or not the plan produces an interesting order).

- We provide an analytical and empirical study of the effect of adding annotations on
the size of the search space, in a centralized database context. The study considers
different shapes of queries, bushy vs. left-linear trees, plans with or without Cartesian
products, and different numbers of binding patterns associated with each database
relation. While the study shows that in some important cases the number of valid query
execution plans is actually considerably smaller than the corresponding case without
annotations, there are still important cases in which search spaces grows significantly
compared to traditional System-R optimization.

- We describe two query optimization algorithms taking into account binding patterns.
Both algorithms consider only valid execution plans, and prune early plans that are not
viable, i.e., cannot be part of any valid execution plan. The first algorithm is a simple
extension to the System-R style dynamic programming algorithm. The disadvantage
of System-R bottom-up optimization is that it only produces a complete QEP towards
the end of the optimization process. Such behavior would not be acceptable in cases
in which the search space is significantly larger than in the traditional case. Therefore,
we propose a best-first search optimization algorithm, in order to produce the first
complete query execution plan relatively fast. The algorithms use a novel method to
combine the join enumeration and selection placement.

- Finally, we present the distributed query optimization algorithm employed in the LeS-
elect system, based on the System-R algorithm mentioned above. This algorithm uses
the source capability descriptions provided by LeSelect wrappers, described in sec-
tion 2.4, to distribute query execution among wrappers and LeSelect servers.

This chapter is organized as follows. In section 3.2, we introduce an example to illustrate
the issues involved in query optimization in the presence of binding patterns. We present
the basic principles of query optimization in the presence of binding patterns in section 3.4.
Then, in section 3.5, we discuss the influence of the access restrictions on the size of the query
optimizer search space. Section 3.6 presents the two variants of optimization algorithms
taking into account binding patterns, based on dynamic programming, respectively on best-
first search, and compares their relative performance. Section 3.7 describes the dynamic-
programming based distributed query optimization algorithm, using binding patterns, that is
employed in LeSelect. Finally, in section 3.8, we compare the algorithms presented with the
state of the art in query optimization in the presence of access restrictions, and heterogeneous
source capabilities.

3.2 Running example

In this section, we illustrate the problems involved in query optimization using binding
patterns with an example, drawn from a real-life application of scientific data integration.
The goal of the application, conducted by the Marine Institute of Crete, is to study the levels of water pollution in the Mediterranean Sea. The application includes two sources of data, each containing the results of sets of experiments. The first source stores the results of some experiments concerning water circulation, and the second source reports the results of experiments concerning the level of pollution in the water. Integrating the data from the two sources enables the scientists to predict water pollution levels for a wide range of times and locations, by combining the results of the pollution experiments and those of the water circulation.

In both sources, the location represents the geographical coordinates of a rectangle. However, the sharing of the sea surface in rectangles is not done in the same fashion across the two sources, i.e. the sources refer to different sets of rectangles. In order to facilitate the integration, the scientists use a Coincides program that answers queries about such rectangles. Given two input locations the source returns a number between 0 and 1 describing the similarity between the sources (0 represents disjointness and 1 represents that they’re identical).

The data resulting from the experiments, and the Concide programs, are modeled by relational tables with the following schema (note that each experiment is defined by a unique key):

Source 1: Experim1(key,date,depth) Location1(key,location) Result1(key,circulation)  
Source 2: Experim2(key,date,depth) Location2(key,location) Result2(key,emission)  
Source 3: Coincides(location1, location2, similarity)  

In the experiment data sources, the data is stored in a proprietary store rather than a relational database. Accordingly, the possible operations on the data are limited. In Source 1, it is possible to ask for the keys of all the experiments that have been done at a certain date (i.e., to select on a given date) or at a certain depth. Given a experiment key, it is possible to retrieve its location from relation Location1, which is a complex value encoding the geographical coordinates of a rectangle, or to retrieve the result of the experiment (from relation Result1) which is a complex object (picture plus a list of vectors) describing the speed and direction of the water circulation. The situation is similar for Source 2. Given a date or a depth we can retrieve the key of the corresponding experiments. Using the key, we can retrieve the location and the pollution emission, which is an complex image. Selections on the location or the emission are not possible. The binding patterns resulting for these relations are the following:

Source 1: Experim1(keyb,dateb,depthb) Experim1(keyb,dateb,depthb)  
Experim1(keyb,dateb,depthb) Experim2(keyb,dateb,depthb)  
Location1(keyb,locationb) Location2(keyb,locationb)  
Result1(keyb,circulationb) Result2(keyb,circulationb)  
Source 2: Experim2(keyb,dateb,depthb) Experim2(keyb,dateb,depthb)  
Source 3: Coincides(location1b, location2b, similarityb)
The typical operation that scientists need to perform on this data is to answer queries of the form: “retrieve the water circulation and the pollution emission on the 1/10/98 for locations matching with high degree of similarity (= 0.9)”. The query can be written as the following conjunctive query:

\[
\begin{align*}
Q: & \quad \text{select } r1.\text{circulation}, r2.\text{circulation} \\
& \quad \text{from } \text{Experim1 } e1, \text{Location1 } l1, \text{Result1 } r1, \text{Experim2 } e2, \text{Location2 } l2, \text{Result2 } r2, \\
& \quad \text{Coincides } c \\
& \quad \text{where } e1.\text{key}=l1.\text{key} \text{ and } l1.\text{key}=r1.\text{key} \text{ and } e2.\text{key}=l2.\text{key} \text{ and } l2.\text{key}=r2.\text{key} \text{ and } \\
& \quad e1.\text{date}="1/10/98" \text{ and } c.\text{location2}=l2.\text{location} \text{ and } c.\text{similarity}=0.9
\end{align*}
\]

Our goal is to find the optimal way to evaluate this query, giving the binding patterns of the resources concerned. Figure 3.1 describes two possible ways to evaluate Q, which are valid according to the source binding patterns, and one relational query execution plan which is not valid. The plans depicted use standard join (\(\bowtie\)) as well as BindJoin (\(\vec{\bowtie}\)) operators. Both types of joins are binary operators and apply recursively on subtrees corresponding to query execution sub-plans. The difference between regular joins and BindJoins resides in the dependency of the right-hand child on the left-hand child in the case of a BindJoin. The two children plans of a regular join can be executed independently of each other, resulting in two tables that can be joined using any of the traditional join algorithms (e.g., hash-join, sort-merge join). In the case of a BindJoin, the right input subtree cannot be executed independently, because it requires bindings that are obtained from the result of the left subtree. We formalize BindJoins in the following section; let us now consider the differences between the valid plans, and the invalid one, in figure 3.1.

The evaluation strategy followed by the plan in figure 3.1(a) is the following. First, use the selection condition on date on both sources in order to retrieve the keys of the experiments
performed on this date. Then, in each source, the keys are used to obtain the corresponding experiment locations. This operation is performed by a BindJoin. The join (on depth) of those two temporary relations is sent to Source 3 which computes the similarity degree for each pair of locations and selects those that satisfy the similarity predicate. In the last step of the evaluation, the keys of the selected experiments are used again to retrieve the desired images (water circulation and pollution emission).

A different plan to evaluate this query, depicted in figure 3.1(b), is to start by retrieving from Source 1 the keys of the experiments performed on the “1/10/98”, together with their corresponding depth. For each tuple in the result, the values of the depth and date could be used for the following complex computation: using the depth, we can retrieve from the relation Experiments the keys of the experiments performed in Source 2 on this depth, then the result is filtered using the desired date; the key of the resulting experiments are sent to Source 2 in order to retrieve their respective location, and then they are sent once more in order to retrieve the corresponding image. Finally, the location of the experiments of Source 1 is retrieved, the locations of experiments from Source 3 are tested for the similarity and, in the last step, the image from source 1 is retrieved.

These two QEPs are valid, in the sense that they can be executed, given the available binding patterns of the resources involved. However, the plan depicted in figure 3.1(c) cannot be executed, given the available binding patterns. The reason is that it is impossible to retrieve all the tuples of the relation Location1 or Location2.

3.3 Operators for accessing restricted resources

3.3.1 The BindJoin operator

Consider two arbitrary relations $T$ having a restricted binding pattern $bp_T = T(\bar{U}' \bar{V}')$, and $R$, with an unrestricted binding pattern $bp_R = R(\bar{W}' \bar{Z}')$. Assume $|\bar{U}'| = |\bar{Z}'|$, and that we are given the conjunction of equality predicates $\bar{U}' = \bar{Z}'$. Thus, we are able to use the information available in $bp_R$ to extract information about $T$, via $bp_T$.

With the same notations as in the previous chapter, the BindJoin of $R$ and $T$ is defined by the following formula:

$$ R(\bar{W}' \bar{Z}') \bowtie_{\bar{Z}'=\bar{U}'} T(\bar{U}' \bar{V}') = \{(w, z, \bar{v}) | (w, z) \in R \land \bar{v} \in S.callResource(z)\} \quad (3.1) $$

Using the equation 2.9, we obtain:

$$ R(\bar{W}' \bar{Z}') \bowtie_{\bar{Z}'=\bar{U}'} T(\bar{U}' \bar{V}') = \{(w, z, \bar{v}) | (w, z) \in R \land (z, \bar{v}) \in acc(T, bp_T, (\bar{U}' = \bar{Z}'))\} \quad (3.2) $$

It follows from the definition of $acc$ that $R(\bar{W}' \bar{Z}') \bowtie_{\bar{Z}'=\bar{U}'} T(\bar{U}' \bar{V}') = R \bowtie_{\bar{Z}'=\bar{U}'} T$. The bag of tuples returned by the BindJoin is simply the result of joining $R$ and $T$ on the $\bar{U}$ variables. The difference between a BindJoin and a regular join resides only in the access restrictions on the right-hand side of the operator: the information passed from $R$ to $T$ is required in
order to get information from $T$. As an interesting observation, a regular join between two tables is completely specified by designing the tables and the join predicates; to specify a BindJoin, one must also provide a binding pattern for each table.

### 3.3.2 The BindAccess operator

As we have shown in chapter 2, any resource is modeled as a table with several binding patterns. When optimizing a query over such tables, the basic building blocks for the optimization process are the access plans to the tables appearing in the query. Therefore, we have to show which operators we use to construct access plans. In the traditional context, optimizers like System R [103] keep one plan per table involved in the query (neglecting the various data orders). In our case, we have to construct one access plan per binding pattern of a table $R_i$.

In the case of unrestricted binding patterns, the access plan consists simply of a Scan operator. However, for restricted access patterns, we need a new logical operator; indeed, tuples are output by such an operator only when bindings for its required inputs are available. This is why we introduce the logical BindAccess operator, that we present in this section. A BindAccess may be thought of as a parameterized Scan; its closest equivalent, among the known access operators, is an index lookup. Indeed, in the traditional query optimization, an index lookup may prove to be an efficient alternative to a full scan, and can only be used if the values of the index fields are known [94]. We use the term BindAccess, rather than index lookup, because a materialized index is just one of the restricted access resources that we model.

Formally, let $T$ be a relational table, having the binding pattern $bp_T = T(\overline{U}^{\bar{d}}{V}^{\bar{f}})$. Let $R(\overline{Z})$ be a table such that $|\overline{Z}| = |\overline{U}|$. For every tuple $\overline{z} = (z_1, z_2, \ldots, z_k)$ of $R$, we denote by $\overline{U} = \overline{z}$ the conjunction of predicates of the form $X_i = z_i$, for every $X_i \in \overline{U}$.

We define the BindAccess to $T$, following the binding pattern $bp_T = (\overline{U}^{\bar{d}}{V}^{\bar{f}})$, and with a table of bindings $R(\overline{Z})$ for the $\overline{U}$ variables, as follows:

$$BA(T, bp_T, R(\overline{Z})) = \bigcup_{\overline{z} \in R(\overline{Z})} acc(T, bp_T, (\overline{U} = \overline{z})) = R(\overline{Z}) \pi_{\overline{z} = \overline{U}} \pi_{\overline{U} \cup \overline{V}}(T) \quad (3.3)$$

Remember that, in section 2.2, we had introduced, for the explanation, the access function $acc$ that, given a table $T$, a binding pattern $bp_T \in bp(T)$, and a set of bindings for the bound variables of $bp_T$, returns the tuples from $T$ that match the given bindings. Furthermore, when describing the required wrapper capabilities in LeSelect, we had described an API that the wrapper has to implement for every restricted resource: the $callResource$ method, that returns tuples of values for the free variables in $\overline{V}$. While $acc$ is an ad-hoc notation, and $callResource$ is a method required from the resource wrapper, the BindAccess is an operator that wraps the access to the restricted resource in an object with the common properties of a partial query plan.

Let us come back to the example introduced in section 3.3.1: consider the table $R(\overline{W}'{\overline{Z}}')$ and the predicates $\overline{Z} = \overline{U}$. We use the bindings provided by $R$ to extract information from $T$. We have:

55
\[ R(\overline{W'}, \overline{Z'}) \Join^{\alpha}_{\overline{Z'-U}} T(\overline{U_i'} \overline{V_i'}) = \{ (\overline{w}, \overline{z}, \overline{v}) | (\overline{w}, \overline{z}) \in R \land (\overline{z}, \overline{v}) \in B(A(T, bp_T, \pi_{\overline{Z}}(R))) \} \quad (3.4) \]

**The BindAccess\(^0\) operator** The BindAccess and BindJoin operator defined above give access to the table \( T \) following a binding pattern \( bp_1 = T(\overline{U_1'} \overline{V_1'}) \), using several bindings for \( \overline{U_1} \) provided by a table \( R \). Thus, we gained access to

\[ \pi_{\overline{U_1'} \cup \overline{W_1} \cup R} R \Join^{\alpha}_{\overline{Z'-U}} T \]

However, assume that after this access, we want to use other binding patterns \( bp_2, bp_2, \ldots, bp_k \) of \( T \), \( bp_i = T(\overline{U_i'} \overline{V_i'}) \), in order to obtain a larger projection of \( R \Join^{\alpha}_{\overline{Z'-U}} T \).

If for every \( i = 2, 3, \ldots, k \), we have \( \overline{U_i} \to \overline{V_i} \), and \( \overline{U_i} \subseteq \overline{U_1} \cup \overline{V_1} \cup \ldots \cup \overline{V_{i-1}} \), the sequence \( seq = [bp_1, bp_2, \ldots, bp_k] \) is sound, usable and complete with respect to the query

\[ Q : \text{select} \ R, \overline{Z}, \ R, \overline{W}, \ T.\overline{V_1} \cup T.\overline{V_2} \cup \ldots \cup T.\overline{V_k} \text{ from } S, \ T \text{ where } R.\overline{Z} = T.\overline{U_1} \]

the result of this query may be written as:

\[ R \Join^{\alpha}_{\overline{Z'-U_1}} \pi_{\overline{U_1'} \cup \overline{V_1}}(T) \Join^{\alpha}_{\overline{U_2'} \cup \overline{V_2}} \pi_{\overline{U_2'} \cup \overline{V_2}}(T) \Join^{\alpha}_{\overline{U_3'} \cup \overline{V_3}} \ldots \Join^{\alpha}_{\overline{U_k'} \cup \overline{V_k}} \pi_{\overline{U_k'} \cup \overline{V_k}}(T) \quad (3.5) \]

The duplicate-free projections, together with the functional dependencies \( \overline{U_i} \to \overline{V_i} \), ensure that no false tuple are generated when reconstructing \( T \) using the binding pattern sequence \( \text{seq} \).

We notice that this query result cannot be expressed using only the BindJoin and BindAccess operators, due to the duplicate-free projection that the BindAccess operator does not perform. We will define, therefore, a variant of the BindAccess operator, BindAccess\(^0\), that will allow us to express the above query by a combination of BindJoin, BindAccess and BindAccess\(^0\) operators.

We start by defining, by analogy with the access function \( acc \) introduced in equation 2.1, a second access function \( accUniq \). Just like \( acc \), this function allows accessing \( T \), using a binding pattern \( T(\overline{U_i'} \overline{V_i'}) \), and a set of bindings \( (\overline{U_i} = \overline{v}) \), i.e. a conjunction of predicates of the form \( X = c, \forall X \in \overline{U_i} \):

\[ accUniq(T, bp_i, (\overline{U_i} = \overline{v})) = \pi_{\overline{U_i'} \cup \overline{V_i}}(\pi_{\overline{U_i} = \overline{v}}(T)) \quad (3.6) \]

The difference between the \( acc \) and \( accUniq \) functions is that for a given set of bindings \( X = c \), \( acc \) returns all tuples that match the bindings, while \( accUniq \) returns the duplicate-free set of matching tuples. Using the \( accUniq \) access function, we define the BindAccess\(^0\) operator, very similar to the BindAccess, where we replace \( acc \) by \( accUniq \):

\[ BA^0(T, bp_i, R(\overline{Z})) = \bigcup_{\overline{z} \in R(\overline{Z})} accUniq(T, bp_i, (\overline{U_i} = \overline{z})) = R(\overline{Z}) \Join^{\alpha}_{\overline{Z'-U_i}} \pi_{\overline{U_i'} \cup \overline{V_i}}(T) \quad (3.7) \]
From this last equation, it can be seen why we introduce a new operator, BindAccess, instead of applying a duplicate-free projection on the result of a simple BindAccess. First, we cannot eliminate duplicates from \( \pi_{\overline{U}_1 \cup \overline{V}_1}(T) \) directly, since we are not capable of computing this expression; given the restricted binding pattern \( bp_T \), we can only get parts of this expression, corresponding to a given set of bindings for \( \overline{U}_1 \). Also, if we eliminate duplicates after the join, we incorrectly erase some correct duplicates. If a tuple \((z_0, \overline{v}_0)\) appears twice in \( R(\overline{Z}) \), and \((z_0, \overline{v}_0)\) occurs twice in \( T(\overline{U}, \overline{V}) \), then the only way to obtain \((z_0, \overline{v}_0)\) twice in \( BA^0(T, bp_i, R(\overline{Z})) \) (not once, and not four times) is to eliminate duplicates in the result of every access made to \( T \); this is exactly the job done by accUniq.

Now, using BindAccess, BindAccess, and BindJoin, we can compute the result of the query \( Q \), provided in equation 3.5, as \( res(T, seq, R) \), where \( res(T, seq, R) \) is defined by induction over \( seq \) as follows:

- \( res(T, [bp_1], R) = R(\overline{Z}^f \overline{W}^f) \bowtie_{\overline{Z} = \overline{U}_1} BA(T, bp_1, \pi_{\overline{U}_1}(T)) \)
- For any \( i, 2 \leq i \leq k \), if we denote by \( seq_i \) the prefix \([bp_1, bp_2, \ldots, bp_i]\) of \( seq \),
  \( res(T, seq_i, R) = res(T, seq_{i-1}, R) \bowtie_{\overline{U}_i} BA^0(T, bp_i, \pi_{\overline{U}_i}(res(T, seq_{i-1}, R))) \).

From now on, when the difference between BindAccess and BindAccess is not important, we refer to any of them using the term BindAccess. When necessary, we will distinguish regular BindAccess operators (that return all duplicates) from BindAccess.

### 3.4 Principles of query optimization using binding patterns

Given a conjunctive query over several tables with binding patterns, our goal is to find the least expensive valid QEP for the query. In order to generate valid QEPs over tables with restricted access, the optimizer must search through a space of annotated query plans where the annotation of a sub-plan describes which variables of the query must be given as input to the sub-plan. We study the effect of adding annotations on the size of the resulting search space, and describe an efficient algorithm for searching the space.

**Annotated QEPs** A query execution plan for a query \( Q \) is a tree whose leaves are labeled with relations in the query and whose internal nodes are algebraic operators. We refer to the leaves of a query execution plan as atomic plans: as atomic plan is a pair \((T, bp_T)\) where \( bp_T \in bp(T) \). In this chapter, we consider plans with join and selection operators. We consider only selections on the simple form \( X_i = c_i \). To simplify our discussion we do not consider plans with projections, and assume they are introduced at a later stage.

**Simplifying assumption: each table accessed only once** In sections 3.4, 3.5 and 3.6, we assume that for each table present in the query, we only use one of its binding patterns. We make this assumption in order to simplify the exposition; it is equivalent to considering
that we are aware of no functional dependencies on the query tables, and therefore, by the soundness condition provided in the proposition 2.3.5, we are not allowed to use more than one binding pattern for each table. We will ease this constraint in section 3.7.

In the sequel, whenever we do not specify the leaf operator in a QEP, it is to be understood that if a table $R$ is accessed following an unrestricted binding pattern, the operator is a Scan, otherwise, it is a BindAccess.

**Extended binding patterns**

As mentioned earlier, an optimization algorithm will search the space of annotated query execution plans. To explain the annotations of a QEP, we first extend the notion of binding patterns to cover partial QEPs, not only atomic plans. The intuition is the following. The binding pattern of a QEP constructed in order to answer $Q$ is defined over all the variables of all the tables present in $Q$. As in the case of table binding patterns, a QEP binding pattern partitions $Q$'s variables into three disjoint sets: bound variables, free variables, and hidden variables:

- The bound variables are those for which bindings are required in order for the QEP to be valid.
- The free variables are those for which we can obtain values, provided that we supply bindings for the bound variables.
- The hidden variables are all the variables of $Q$, except for the bound and free variables.

The binding pattern of a partial QEP is recursively derived from the binding pattern of its leaves, as follows:

1. The binding pattern of an atomic plan $(T, bp_T)$ is obtained by extending $bp_T$ in the following simple manner: all variables of $Q$ that are not covered by $bp_T$ are hidden.
2. The binding pattern of a plan of the form $\sigma_{X = c}(p)$ is obtained from the binding pattern of $p$, in which we mapped all bound variables present in $X$ to $f$.
3. The binding pattern of a projection plan of the form $\pi_{\pi}(p)$ is obtained from the binding pattern of $p$, by mapping all variables not present in $X$ to $-$ (hidden).
4. The binding pattern of a regular join plan of the form $p \bowtie q$ is obtained by merging the binding patterns of $p$ and $q$. Note that in a regular join, there are no bindings passed from $p$ to $q$, i.e., there is no variable $X$ mapped to $b$ in one binding pattern and to $f$ in the other. Therefore, the only conflicts that may arise between the binding patterns of $p$ and $q$ are of one of the following kinds: either some variable $X$ is mapped to $b$ in $bp_p$ and to $-$ in $bp_q$, or $X$ is mapped to $f$ in $bp_p$ and to $-$ in $bp_q$ (or the symmetric situations). In these cases, $X$ appears in the binding pattern of the join plan as $b$, respectively $f$.
5. The binding pattern of a BindJoin of the form $p \Join_X q$, where \( X \) is the variable set for which bindings are passed from \( p \) to \( q \), is obtained by mapping all the variables in \( X \) to \( f \), and merging the rest of the binding patterns of \( p \) and \( q \) as for a regular join.

Having extended the notion of binding pattern from tables to partial QEPs, we may now describe the QEP annotations we use. Formally, each node \( n \) in a query execution plan tree is labeled by:

- the set \( \text{conj}(n) \) of tables present in \( Q \), each with a choice of a binding pattern, that \( n \) covers
- the subset of all equality predicates from \( Q \)’s where clause, between variables belonging to distinct tables in \( \text{conj}(n) \)
- the set of variables belonging to tables from \( \text{conj}(n) \) for which bindings are required in order for the subtree \( n \) to be executable.

The last item in the annotation is exactly the set of bound variables in \( n \)’s QEP.

**Notation** The set of bound variables appears as an index to the pair (conjunct set, predicate set). For our convenience, we do not represent the free variables (they are all the variables in \( \text{conj}(n) \), minus the bound variables), nor the hidden variables (all the variables from \( Q \) except the bound and free variables).

**Example 3.4.1** The following algebraic expression represents a fragment of the query execution plan depicted in figure 3.1(a):

\[
(\text{Experim}_1(\text{key}^b \text{date}^b \text{depth}^f) \Join \text{Location}_1(\text{key}^b \text{location}^f)) \Join \\
(\text{Experim}_2(\text{key}^b \text{date}^b \text{depth}^f) \Join \text{Location}_2(\text{key}^b \text{location}^f))
\]

This partial plan is annotated by:
- the conjunct set \{ \text{Experim}_1(\text{key}^b \text{date}^b \text{depth}^f), \text{Location}_1(\text{key}^b \text{location}^f), \text{Experim}_2(\text{key}^b \text{date}^b \text{depth}^f), \text{Location}_2(\text{key}^b \text{location}^f) \};
- the predicate set \{ \text{Experim}_1.\text{key}=\text{Location}_1.\text{key}, \text{Experim}_2.\text{key}=\text{Location}_2.\text{key}, \text{Experim}_1.\text{date}=\text{Experim}_2.\text{date} \};
- the bound variable \text{Experim}_1.\text{date}. Indeed, if a binding for this variable is provided, we can extend it into a binding for \text{Experim}_2.\text{date}, too, since we know from the conjunct’s predicates that the experiments have the same date. With these bindings, both BindJoins become valid plans, and so is the QEP.

Each of the query execution plans \( p \) has an associated cost, denoted by \( \text{cost}(p) \). A query optimization algorithm (any of those described in sections 3.6 and 3.7) includes a component which takes a query execution plan, as described above, and chooses a physical implementation for each of the relational operators. Therefore, the cost of a query execution plan is the cost of the best physical query execution plan implementing it. The cost of atomic query
execution plans are directly deduced from the cost parameters associated to the binding patterns of the tables involved, as we explained in section 2.4, and the cost of the non-atomic query execution plans is an estimate based on the cost of the operator and the cost of the sub-plans. The particular cost functions we use is orthogonal to the search strategy that the algorithm employs, though it can, in some cases, influence the effectiveness of our pruning methods. In section 3.7, we will detail a particular cost model we use in one variant of cost-based query optimization using binding patterns. For the purposes of this section, we only make the monotonicity assumption about our cost model which is a reasonable one in practice: if the plan \( p' \) is obtained from the plan \( p \) by replacing a sub-plan \( p_1 \) of \( p \) by a cheaper and equivalent sub-plan \( p_2 \), then \( p' \) is cheaper than \( p \).

**Complete query execution plan** A query execution plan that covers all the conjuncts in the query, and whose adornment maps precisely the bound variables in the query to \( b \) is called a complete query execution plan.

**Equivalent query execution plans** Two query execution plans are considered equivalent if they are labeled with the same set of conjuncts from the query and have identical binding patterns.

Intuitively, two equivalent QEPs solve the same query (to simplify, we ignore the issue of interesting orders produced by query execution plans). The equivalence relation partitions the set of query execution plans into classes. Note that if two plans \( p_1 \) and \( p_2 \) are equivalent, \( p_1 \) is valid if and only if \( p_2 \) is valid. This follows from the equality of their conjunct sets and binding patterns.

**Viable query execution plan** A query execution plan is viable if it can be part of a complete query execution plan.

As before, if \( p_1 \) is equivalent to \( p_2 \), then plan \( p_1 \) is viable if and only if \( p_2 \) is viable. Therefore, in the sequel, we will assess validity and viability of equivalence classes of QEPs.

**Validity and query feasibility** In section 2.3.3, we had defined feasible queries as those for which there is a way of chaining table accesses such that the result contains the desired query result. Having introduced the notion of valid plans, the relation between validity and feasibility is the following: a query is feasible if and only if it has at least a complete valid QEP. Note that validity is a property of query plans, while feasibility is a property of a query.

**Weaker QEP binding patterns** A binding pattern \( bp_1 \) is said to be weaker than a binding pattern \( bp_2 \) if every variable that is mapped to \( b \) in \( bp_1 \) is also mapped to \( b \) in \( bp_2 \), and the two binding patterns map the same set of variables to \(-\).

**Covering QEPs** A query execution plan \( p_1 \) covers a query execution plan \( p_2 \) if the tables appearing in their annotations are the same, and the binding pattern of \( p_1 \) is weaker than the binding pattern of \( p_2 \). Note that this definition involves the tables, not the binding patterns,
considered in the annotation of a plan; two plans may be compared to find if one is weaker than the other even if in their set of conjuncts, different binding patterns are considered for a given table.

Also, if a plan $p_1$ covers a plan $p_2$, then any plan $p'_1$ equivalent to $p_1$ will cover any plan $p'_2$ equivalent to $p_2$. Intuitively, if the equivalence class $P_1$ covers $P_2$, then by applying selections to any of the plans in $P_1$ we can obtain plans in $P_2$.

**Example 3.4.2** Consider the following query: “select * from $R_1(X_1, X_2)$, $R_2(X_3, X_4)$, $R_3(X_5, X_6)$, $R_4(X_7, X_8)$ where $X_2 = X_3$ and $X_4 = X_5$ and $X_6 = X_7$ and $X_1 = c_1$ and $X_8 = c_2$, and assume that each table $R_i$ has two binding patterns: \{fb, bf\}.

The following two equivalence classes are valid and viable:
\[
[R_1(X_1, X_2), R_2(X_3, X_4), X_2 = X_3]_{\{X_1\}} \text{, } [R_1(X_1, X_2), R_2(X_3, X_4), X_2 = X_3]_{\{X_1, X_4\}}
\]

The equivalence class \([R_1(X_1, X_2), R_3(X_5, X_6), R_4(X_7, X_8), X_6 = X_7]_{\{X_2, X_3\}}\] is valid but is not viable. The reason is that in order to construct a complete QEP using this equivalence class, we can only access $R_2$, the only table left, and only once. Now, no plan that includes only $R_2$ produces bindings for both required inputs $X_2$ and $X_5$, since we have to choose one of its binding patterns.

The equivalence class \([R_1(X_1, X_2)]\) is not valid.

The equivalence class \([R_1(X_1, X_2), R_2(X_3, X_4), X_2 = X_3]_{\{X_1\}}\] covers the equivalence class \([R_1(X_1, X_2), R_2(X_3, X_4), X_2 = X_3]_{\{X_1, X_4\}}\).

### 3.5 Search space for query optimization with binding patterns

In this section we show through a set of examples how some of the basic properties underlying System-R style optimization need to be reconsidered in the presence of binding-pattern limitations. In fact, we will compare not with System-R, but rather with the Garlic data integration system [54], which partially handles binding pattern limitations within the framework of System-R. We begin by arguing that it is essential to consider query execution plans annotated by their input variables. This will be the key feature distinguishing our algorithm from that in Garlic.

#### 3.5.1 Classes of query execution plans

"Open" partial query execution plans In order to compare our approach with that of Garlic, we distinguish two classes of partial query execution plans. An open partial query execution plan is a non-atomic plan (i.e., with more than one relation), which cannot be executed only with the constants available in the query. Of course, in order for an open sub-plan to be part of a complete plan, it must receive bindings from some other parts of the plan. A closed sub-plan is one that can be executed given the bindings from the query.
Garlic’s search strategy only considers closed partial query execution plans.\footnote{For atomic plans (plans on a single relation) Garlic considers one plan for every viable binding pattern.} We now argue that optimal plans may include open sub-plans, and therefore, it is important to consider a larger space of plans. Subsequently we show that looking at this larger search space has several important ramifications to the properties of our search space.

**Example 3.5.1** Consider the following query \([R(X, Y), S(Z, U), T(V, W), Y = Z, U = V]\) and suppose that \(bp(R) = \{R(X^f Y^f)\}, bp(S) = \{S(Z^b U^b), S(Z^f U^b)\}, \) and \(bp(T) = \{T(V^b W^f), T(V^f W^b)\}.\) That is, it is possible to retrieve all the tuples of the relation \(R\), but for the relations \(S\) and \(T\) we need to give either the value of the first or the second attribute.

If we build the set of partial query execution plans according to the Garlic strategy, we will construct in a bottom-up fashion, for each equivalence class, all the corresponding query execution plans. For example, in the first iteration we will keep the plans for \(R(X^f Y^f), S(Z^b U^f)\) (because \(Z\) could be obtained from the equality with \(Y\)), \(S(Z^f U^b)\) (because \(U\) could be obtained from the equality with \(V\)), and \(T(V^b W^f)\) (because \(V\) could be obtained from the equality with \(U\)). In the second iteration, only a single plan will be produced: \(R(X^f Y^f) \bowtie_{Y = Z} S(Z^b U^f)\), since this is the only plan of size two which is executable given the constants in the query. Finally, the third iteration will yield a single plan: \(R(X^f Y^f) \bowtie_{Y = Z} S(Z^b U^f) \bowtie_{U = V} T(V^b W^f)\).

If we do consider in the generation phase plans which are temporarily open, we would construct in the iteration 2 also the plan \(S(Z^b U^f) \bowtie_{U = V} T(V^b W^f)\), which would have lead to another final plan \(R(X^f Y^f) \bowtie_{Y = Z} S(Z^b U^f) \bowtie_{U = V} T(V^b W^f)\).

The search strategy employed by Garlic has two important properties. First, it can be shown that if there exists a plan for answering the query, then Garlic will find one, even if it is not the optimal plan. Second, if only limited implementations of the BindJoin operator are available, then it turns out that for any plan that has an open sub-plan, there exists a closed sub-plan that model the same execution. For example, if we consider that the BindJoins are implemented using only a nested loop-based algorithm and that the execution follows the iterator model [51], then the two trees shown previously actually model the same execution, and hence it suffices to consider only the closed plans.

The above two points also highlight the limitations of the Garlic approach, which are of special importance in the context of data integration where some of the sources are on a wide-area network. Nested loop join is far from being the most efficient way of implementing a dependent join. For example, in [21, 62] the authors propose several variants of the known join algorithms in order to implement the dependent join algorithm. In the case of such complicated implementation techniques, the two query execution plans above do not result in the same execution. For example, the first query execution plan above cannot model an execution using caching techniques where every time that the expression \(S(Z^b U^f) \bowtie_{U = V} T(V^b W^f)\) is executed for a given value of \(Z\), the output tuples are cached (with the input \(Z\)). This
execution cannot be modeled by the first tree, since \( S(Z^b U^f) \bowtie T(V^b W^f) \) does not appear as a subexpression. By considering only closed partial query execution plans Garlic will miss a more efficient plan. In the context of data integration it is essential to consider more sophisticated implementations of dependent joins, and hence we need to develop an algorithm that efficiently explores open sub-plans.

In what follows we describe several other important properties of the search space in the presence of binding pattern limitations.

**Refined equivalence classes** As an immediate consequence of the above discussion is that we need to refine our notion of equivalence classes during our join-order enumeration. System-R style optimizers keep the cheapest query execution plan for every equivalence class, where two plans belong to the same equivalence class if the cover the same set of base relations in the query. In our new context, it is necessary to annotate every plan with the set of variables that are required as inputs in addition to the set of base relations that are covered. From now on, the equivalence class of a plan is determined by the combination of the sub-query solved by a plan, and its required input variables.

**Valid and viable plans** An immediate consequence of the fact that we must consider refined annotations for our plans is that the number of equivalence classes grows significantly, hence leading to a more expensive search problem. Fortunately, two classes of plans can be pruned early in the search: plans belonging to invalid or non-viable equivalence classes. An equivalence class is invalid if there is no query execution plan for that class that can be executed given the limitations on access patterns. For example, in the previous query, the equivalence class \([S(Z, U), T(V, W), U = V]|_{W}\) is invalid. An equivalence class is not viable if none of the plans in the class can be part of a complete query execution plan. In section 3.5.2 we show that pruning these two classes of plans has a dramatic effect on the size of the search space, and in section 3.6 we show that validity can be checked as part of the enumeration algorithm, and viability can be checked efficiently.

**The need for bushy trees** As the following example shows, in the presence of limited access patterns, there are cases where the set of left-linear trees includes only plans with Cartesian products, while the set of bushy trees does contain a query execution plan without Cartesian products. Hence, if we want to avoid plans with Cartesian products, we must search the space of bushy trees.

**Example 3.5.2** Consider the following query:

\[ [P(X_1, Y_1), R(X_2, Y_2), S(X_3, Y_3), T(X_4, Y_4), Y_1 = X_2, Y_3 = X_4, Y_2 = Y_4]|_{W} \]

Suppose that the only access patterns allowed to the relations \( P, R \) and \( T \) are: \( P(X_1^f Y_1^f), R(X_2^f Y_2^f), S(X_3^f Y_3^f), \) and \( T(X_4^f Y_4^f). \) It is easy to note that all the linear query execution plans will include a Cartesian product. However, the following bushy-tree does not include a Cartesian product: \( (P(X_1^f Y_1^f) \bowtie_{Y_1 = X_2} R(X_2^f Y_2^f)) \bowtie_{Y_2 = Y_4} (S(X_3^f Y_3^f) \bowtie_{Y_3 = X_4} T(X_4^f Y_4^f)). \)
Recall that in the traditional System-R setting, if the query graph is connected, then the space of left-linear trees necessarily includes a plan without Cartesian products. Hence, in that context, the query optimizer can limit its search to left-linear trees without having to use Cartesian products.

**Placement of selections** In the context of System-R it is possible to decouple the decision on join ordering from the decision on placement of selections. The placement could either be made heuristically by pushing selections as far down as possible in the query tree, in a cost-based fashion in a separate phase [63], or mixing the decision on the join order with the decision on the placement of expensive predicates in a dynamic programming style optimization like in [22]. In our context, since we are considering query execution plans that are annotated by variables that must be given as inputs to the plan, the interaction between the placement of the selection and the join ordering is much more subtle. Specifically, in our search space, the join ordering decision may result in plans that differ only w.r.t. placement of selections. This is illustrated by the following example.

**Example 3.5.3** Consider the query \([R(X, Y), S(Z, U), T(V, W), Y = Z, U = V]_0\), and suppose \(bp(R) = \{R(X^1Y^1)\}, bp(S) = \{S(Z^bU^b), S(Z^fU^f)\}, bp(T) = \{T(V^1W^1)\}\).

By combining the pattern \(R(X^1Y^1)\) with the pattern \(S(Z^bU^b)\) via a BindJoin operator, we obtain a plan \(p_1\) for the equivalence class \([R(X, Y), S(Z, U), Y = Z]_0\). By combining the pattern \(R(X^1Y^1)\) with the pattern \(S(Z^fU^f)\) via a join operator, we obtain a plan \(p_2\) for the equivalence class \([R(X, Y), S(Z, U), Y = Z]_0\). It is easy to see that the plan \(p_1\) covers the plan \(p_2\) i.e., by applying a selection on the variable \(U\) in the plan \(p_1\), we obtain a plan \(p_3\) which is equivalent to \(p_2\). Hence, a plan without any selections turns out to be equivalent to a plan with selection, and in our case, \(p_3\) is probably less expensive than \(p_2\).

In standard System-R optimization the equivalence class \([R(X, Y), S(Z, U), Y = Z]_0\) would have not be considered at all, since \(U\) is not bound in the original query. In our setting, as shown previously, we have to keep one plan per set of conjuncts and set of bound variables (even if they are not bound in the original query). As a consequence, if we ignore selections, we will be in the situation where we do consider a plan for the class \([R(X, Y), S(Z, U), Y = Z]_0\), but the plan we think is optimal for this class (i.e., \(p_2\)) is actually not the real optimal one.

To conclude this section, we note that in order to perform optimization in the presence of access pattern limitations, the optimizer must search the space of annotated query plans. The algorithm should avoid invalid plans and prune non-viable plans as early as possible. In order to avoid Cartesian products, the algorithm needs to consider bushy trees and not only left-deep trees. Furthermore, special care must be given to placing selections and to detect multiple query plans that result in identical executions. In the two following sections, we analyze the size of the search space sanctioned by the conclusions of this section.
The size of the space that needs to be searched by a query optimizer employing dynamic programming is relatively well understood [86, 84, 122]. We study the effect on the size of the search space in the presence of access pattern limitations, and the associated need to search the space of annotated query execution plans. We present both an analytical and empirical study. The results of this study also justify some of the choices we made in designing the query optimization algorithm described in section 3.6.3.

3.5.2 Theoretical study of the size of the search space

Our study examines the size of two measures: the number of valid complete query execution plans and the number of valid partial query execution plans considered by a dynamic-programming style optimizer. Knowing the number of partial plans computed by the dynamic programming algorithm is important, since many of the current commercial optimizers implement some form of dynamic programming, hence our analysis represents a prediction of their complexity if they were extended to deal with binding patterns. Of course, while the number of query execution plans is usually very large (and that of partial plans even larger), dynamic programming only considers a small fraction of the partial plans. For example, for the case of chain queries with \( n \) relations, the number of plans without Cartesian products is known to be \( \left( \frac{2(n-1)}{n-1} \right)^{\frac{1}{n}} \), while the number of bushy partial plans without Cartesian products considered by the dynamic programming algorithm is known to be only \( \frac{n^3-n}{6} \) [86]. We note that even though we consider the number of plans explored by a dynamic programming optimizer, the results are of interest even if we were to employ a different paradigm, since dynamic programming is sometimes used as a yardstick for the others. For example, [9] shows that a classical rule-based optimizer considers, in general, a strictly higher sized search space than dynamic programming, and present an improved rule-based optimization algorithm whose complexity matches that of dynamic programming.

The results of our analysis are shown in Table 3.1. We focus our study on chain queries \( Q \) over binary relations:

```plaintext
select * from R_1(X_1, Y_1), R_2(X_2, Y_2), \ldots, R_n(X_n, Y_n)
where Y_1 = X_2 and Y_2 = X_3 and \ldots and Y_{n-1} = X_n and C(\overline{X})
```

In this query, \( \overline{X} = \{X_1, X_2, \ldots, X_n, Y_1, Y_2, \ldots, Y_n\}, C(\overline{X}) \) is a conjunction of simple selection predicates on the variables from \( \overline{X} \); we refer to the variables on which there are selections in \( C(\overline{X}) \) as bound(\( Q \)). We consider different combinations of access patterns and different sets of bound variables.

Thus, in the first line of the table, for \( i = 1, 2, \ldots, n \), \( bp(R_i) = \{R_i(X_i^j Y_i^j)\} \), and bound(\( Q \)) = \( \emptyset \). This line represents the classical case, with no access patterns and no selections, and is for comparison purposes (all entries are taken from the references). Note that since we are counting plans with Cartesian products, the numbers in the first row apply to any query shape, not just chain queries. In the second line, for \( i = 1, 2, \ldots, n \), \( bp(R_i) = \{R_i(X_i^j Y_i^j)\} \), and bound(\( Q \)) = \( \{X_1\} \). The third line analyses the transition from line 1 to line 2, by letting the number \( m \) of relations with binding pattern \( R_i(X_i^j Y_i^j) \) vary
<table>
<thead>
<tr>
<th>Graph</th>
<th>QEP’s</th>
<th>LL QEP’s</th>
<th>PQE’s, dp</th>
<th>LL PQE’s, dp</th>
</tr>
</thead>
</table>
| $R_i(X_i^1, Y_i^f)$, $i = 1, n$  
bound$(Q) = \emptyset$ | $(\binom{2(n-1)}{n-1}) (n-1)!$  
[84] | $n!$  
[84] | $3^n - 2^{n+1} + 1$  
[86, 122] | $n(2^{n-1} - 1)$  
[86, 122] |
| $R_i(X_i^1, X_i^f)$, $i = 1, n$  
bound$(Q) = \{X_0\}$ | $(\binom{2(n-1)}{n-1}) \frac{1}{n}$ | 1 | $\frac{n^3 - n}{6}$  
[86, 122] | $n - 1$ |
| $R_i(X_i^1, Y_i^f)$, for  
i $\in \{i_1, \ldots, i_m\}$  
bound$(Q) = \emptyset$  
i $\notin \{i, \ldots, i_m\}$  
$R_i(X_i^1, X_i^1)$, $i = 1, n$  
bound$(Q) = \{X_0, X_n\}$ | $\leq (\binom{2(n-1)}{n-1}) \frac{(n-1)!}{((\frac{n}{m})!)^m}$  
$\leq \frac{n!}{((\frac{n}{m})!)^m}$ | $\leq \frac{(\frac{n}{m})^3 + 6(\frac{n}{m})^2 + 5(\frac{n}{m}) + 6^m}{6^m}$ | $\leq m(\frac{n}{m} + 1)^m$ |

Table 3.1: Bounds on the size of the search space for a dynamic-programming optimizer in the presence of binding patterns, for a chain query $Q$. 

66
from 1 to \( n \) (the other \( n - m \) relations have binding pattern \( R_j(X_j'Y_j') \)). We assume here that \( R_i \) is always among these \( m \), i.e. we have the access pattern \( R_i(X_i'Y_i') \): this guarantees that there always exists a query execution plan, although \( \text{bound}(Q) = \emptyset \). In line four each of the relations has two binding patterns, \( \text{bp}(R_i) = \{R_i(X_i'Y_i'), R_i(X_i'Y_i')\} \), and there are two bound variables: \( \text{bound}(Q) = \{X_i, Y_i\} \) (i.e. we can start either from the left or from the right). Finally, in the last line each relation \( R_i \) has both binding patterns \( R_i(X_i'Y_i') \) and \( R_i(X'_iY'_i) \).

The table shows the maximal number of valid bushy query execution plans (denoted QEP’s in column 1), left-linear QEP’s (column 2), valid partial query execution plans generated by the dynamic programming algorithm for the cases in which all bushy trees are considered (column 3) and for the case in which only left-linear trees are considered (column 4). The formulas include query execution plans that have Cartesian products.

Lines two and three illustrate an example where the complexity of join ordering decreases because of the limited access patterns. Line two represents an extreme case, with a single left-linear solution (namely \( \ldots ((R_1 \bowtie R_2) \bowtie R_3) \bowtie \ldots \bowtie R_n) \). There are several solutions with bushy trees, basically all ways to parenthesize this expression, but still less than in the classical case (line 1) where, in addition, one could take all permutations. The number of plans considered by the dynamic programming algorithm also decreases dramatically from line one to line two. We remark that the number of bushy trees considered here is the same as that considered in the classical case for plans without Cartesian products [86, 122]. The next line refines the analysis by allowing a number \( m \) of relations \( R_i \) to be \( ff \), the rest being \( bf \). Of course, the exact formulas in each entry depend on which \( m \) relation one chooses: the table only shows their maximum values, obtained precisely when the \( m \) are chosen equidistantly (i.e. \( R_1, R_{n+1}, R_{2n+1}, \ldots \)). It is interesting to observe that the formulas in this line coincide with those in line 1 for \( m = n \), and with those in line 2 for \( m = 1 \).

Line four considers an interesting particular case when we can “start at both ends”. All left linear plans are obtained by shuffling a join from the left \( R_1 \bowtie R_2 \bowtie R_3 \bowtie \ldots \bowtie R_k \) with one from the right, \( R_n \bowtie R_{n-1} \bowtie R_{n-2} \bowtie \ldots \bowtie R_{k+1} \), e.g., \( R_1 \bowtie R_2 \bowtie R_n \bowtie R_{n-1} \bowtie R_4 \bowtie \ldots \) there are \( 2^n \) ways to do that. The complexity here is higher than in line two, but still far less than the case without limited access patterns.

Finally, line five illustrates a case when the complexity increases because of the additional access patterns (both \( ff \) and \( bf \)). The increase however is still within the same general complexity: it increases from one exponential to a higher exponential, and not to, say, a double exponential. For example (comparing lines 1 and 5) the partial query execution plans considered by the dynamic programming algorithm increased from \( O(3^n) \) to \( O(5.36^n) \). The proofs for the formulas presented in table 3.1 (except those that we took from existing references) can be found in [40].

The key conclusion we draw from the table is that for some query shapes the presence of limited access patterns significantly reduces the number of valid plans; since the dynamic programming algorithm will discard invalid plans, it will be able to explore a significantly smaller space in these cases. At the same time, in other cases, the complexity actually increases, but the increase stays within the general complexity of join ordering (i.e. expo-
3.5.3 Experimental study of the size of the search space

The analytical study provides only upper bounds on the size of the search space, in cases where mathematical analysis is possible. In this section we describe a series of experiments designed to measure the impact of the presence of binding patterns on the size of the search space.

We study the effects of several factors on the size of the space: size of the query, number of variables, shape of the query graph and the type and structure of binding patterns. In our study we consider three measures: (1) the number of complete query execution plans, (2) the number of viable partial query execution plans, and (3) the number of valid but possibly non-viable query execution plans. The real size of the search space is (1), but the complexity of our algorithm is not proportional to this number, but to (2). The complexity of our algorithm without the viability test would be proportional to (3). We measure these numbers for both left-linear trees, as well as for bushy trees. Finally, we measure the effect of considering plans with Cartesian products.

Random query generator For our experiments we implemented a random query generator which takes as input: (1) the number of relations, (b) the number of variables and (c) the desired shape of the query graph, and produces as output a conjunctive query with the required properties. We support four kinds of query shapes: chain queries, star queries, complete queries and random. The first three types of queries are well known in the literature [84]. Queries in the last category are constructed as follows: we fix two parameters, $k_2$ and $k_3$, which represent the percent of variables that appear in two-way, respectively three-way joins. The rest of the variables do not appear in join predicates. Given the values $k_2$ and $k_3$, the associations between variables and relations is done randomly. In the rest of the experiments we took the values $k_2 = 1/3$ and $k_3 = 1/12$. Cardinalities of the relations are generated randomly from 1000 to 10000 tuples. A certain number of simple selection predicates are randomly distributed over the variables present in the query. Note that if a selection $X_i = c_i$ is imposed on variable $X_i$, we use it as a binding for all variables that are linked to $X_i$ via equi-join predicates. The selectivities were randomly chosen between 0.00001 to 1.0. Finally, the cost per tuple associated with each binding pattern was selected randomly between 1 to 1000.

Perturbations on the binding patterns A simple analysis would show that the size of the search space can be affected by two contradictory factors: (1) when binding pattern limitations become more restrictive, the size of the space decreases, and (2) when new binding patterns are added, the size of the space increases. In order to analyze those two contradictory factors and their respective effect, we apply the following strategy. In both cases, we start from the simple case when all the relations have only a $fff \ldots f$ binding pattern. We iteratively apply two transformations on this set of original input binding patterns:
Figure 3.2: The evolution of the valid partial query execution plans and the complete query execution plans depending on the number of bind transformations.
Figure 3.3: The evolution of the valid partial query execution plans and the complete query execution plans depending on the number of addBind transformations.

- **bind**: take a binding pattern from the current set, transform one of the $f$’s into a $b$, and put it back to the current set.

- **addBind**: do as in bind, but do not remove the original binding pattern.

Intuitively, the first transformation would decrease the size of the search space, since the access patterns are more restricted. The second transformation will increase the size of the search space, because there are more ways of accessing the data.

In the figures each point has been obtained from the results of 30 queries generated randomly with the same parameters. We show the average ratios between the number of plans after the transformations and the number of plans for the $ffff$ binding patterns.

**The effect of the bind transformation** In figure 3.2 we show how the number of complete query execution plans and the number of partial viable query execution plans vary
with the number of applications of the bind transformation. The queries have 6 relations, 50 variables, and 12 variables are bound in the query. We show the results for bushy trees, with Cartesian products, and for the four types of queries shapes. We can observe that the size of the search space is decreasing very quickly for all types of queries, as soon as binding patterns are introduced. For example, after 15 applications of the transformation, none of the queries have plans. The number of viable partial query execution plans is globally decreasing, even if sometimes, for star queries, it first slowly increases.

We ran the same tests when varying the other parameters of interest. In particular, we observed that the number of complete query execution plans depends on the shape of the query (it decreases faster for complete queries and much slower for star queries). On the other hand, the size of the query, the shape of the query execution plans (i.e., bushy vs. left-linear) and the consideration of plans with Cartesian products does not seem to have a strong effect on the relative average.

**The effect of the addBind transformation** In figure 3.3 we show how the number of complete query execution plan and the number of partial viable query execution plans vary with the number of applications of the addBind transformation. Here too the queries have 6 relations, 50 variables and 12 variables are bound. We show the results for bushy trees with Cartesian products. The figures show the growth of the search space, on a logarithmic scale.

For the four types of queries, we notice an exponential growth of the search space depending on the number of the addBind transformations. The number of viable partial query execution plans grows accordingly.

**The number of non-viable plans** We claimed that a viability test is essential. In figure 3.4(top) we show how the number of partial plans that can be obtained by a generative algorithm grows when non-viable plans are also considered. As shown by the two bottom curves in the figure, the number of viable partial plans and the number of complete plans are rapidly decreasing. However, the top curve shows that the total number of partial query execution plans increases before it decreases. Hence, this underscores the importance of checking viable plans.

**Cartesian products** Finally, we tested the effects of allowing plans with Cartesian products. In figure 3.4(bottom) we consider the ratio of the size of the search space (i.e., complete query execution plans) with Cartesian product versus the size without Cartesian products. We show how this ratio varies with the number of bind transformations. The ratio is rather constant or slowly growing when we look at star, complete or normal queries. For chain queries the ratio grows drastically.

**Consequences of the size of the search space**

The analysis of the size of the search space clearly illustrates the need to develop an algorithm that considers only valid and viable plans. The problem that the analysis raises is that it is
Figure 3.4: The top graph shows how the number of plans changes with the number of bind transformations. Even though the number of viable plans decreases when we perform more bind transformations, the number of total valid plans can increase. The bottom graph shows the effect of including Cartesian products in the plans. The growth of the number of plans depends heavily on the shape of the query.
hard to predict how the size of the space will be affected: in some cases, it may be smaller than the traditional case, while in others it can be significantly larger. Hence, in order for an optimization algorithm to be effective in all cases, it must be able to handle large search spaces.

3.6 Optimization algorithms using binding patterns

In this section, we describe two query optimization algorithms using binding patterns. First, we propose an extension to the traditional System-R optimization algorithm [103], based on dynamic programming, that handles binding pattern restrictions. The main problem with System-R style optimization when the search space is large is that the first plan is produced only towards the end of the optimization.

Hence, the approach that we pursue in the next section is to employ a best-first search strategy whose main advantage is to produce a first plan relatively quickly, and improve it as the optimization proceeds. We focus on some of the important aspects of this algorithm: on particular, we present new pruning rules that allow us combine selections, joins and BindJoins at every step in the optimization process, in order to try all interesting placements of selections in the QEP.

3.6.1 Common principles

Both the dynamic programming and best-first search optimization algorithms choose the optimal plan in the search space characterized by the following properties: (a) bushy trees, (b) plans that include Cartesian products. The algorithms adopt the following principles:

1. At every point in the optimization, the algorithm maintains a set of partial query execution plans, $S$. Each plan $p \in S$ is labeled with the equivalence class to which it belongs and its cost. The equivalence class is specified by the set of conjuncts covered by $p$, the query predicates enforced in $p$, and its binding pattern.

2. Initially, the set $S$ contains atomic plans, i.e., plans for accessing a single relation. For a relation $R$, $S$ contains an atomic query execution plan for every binding pattern describing an access pattern to the tuples of $R$.

3. In the iterative step of the algorithm, we add new plans to $S$ by combining existing plans in $S$. We create one resulting plan for every adornment that satisfies the conditions on adornments described in section 3.4, hence, creating only valid plans.

4. At every point, $S$ contains at most one plan for every equivalence class of query execution plans, which is the cheapest one found thus far.

5. In the combination step, we prune non-viable plans. While this pruning is not handled by the basic System R algorithm, we add it in order to have a fair comparison between the two classes of algorithms.
Algorithm dynProg
Input \( Q: \) "select \( X_{res} \) from \( R_1(\overline{X}_1), R_2(\overline{X}_2), \ldots, R_n(\overline{X}_n) \) where \( J(\overline{X}) \) and \( C(\overline{X})" 
for each \( R_i, \) \( bp(R_i) = \{bp_i\} \)
Output optimal QEP for \( Q \)
1 if \( Q \) is not feasible 
2 then stop 
3 let \( S = \emptyset \)
4 foreach \( R_i, \) \( i = 1, 2, \ldots, n \)
5 let \( p_i \) be the optimal access plan to \( R_i \) following \( bq_i \) (Scan or BindAccess) 
6 let \( q_i \) be the result of applying all possible selections on \( p_i \)
7 let \( S = S \cup q_i \)
8 foreach \( i = 2, 3, \ldots, n \)
9 foreach \( j = 1, 2, \ldots, i \)
10 let \( k = i - j \)
11 foreach \( p_j \in S \) such that \( p_j \) covers \( j \) conjuncts 
12 let \( S_k = \{q \in S, \) \( q \) covers \( k \) conjuncts, disjoint from \( p_j \)’s conjuncts\} 
13 let \( S_{res} = \{p_j \Join p_k, p_k \in S_k \} \cup \{p_j \Join p_k, p_k \in S_k \} \}
14 eliminate non-viable plans from \( S_{res} \)
15 apply all possible selections on the plans in \( S_{res} \)
16 prune(S_{res}, S)
17 let \( S = S \cup S_{res} \)
18 let \( S_{cov} \) be the set of plans in \( Q \)’s equivalence class, or covering this class
19 let \( S_{sol} = \emptyset \)
20 foreach \( r \in S_{cov} \)
21 let \( s \) be the result of the remaining selections and projections on \( r \)
22 let \( S_{sol} = S_{sol} \cup s \)
23 return cheapest plan from \( S_{sol} \)

Figure 3.5: Dynamic programming algorithm for query optimization using binding patterns.

It is important to note that we only create valid plans; this is achieved by enforcing some correctness conditions when generating a partial QEP. Also, we check the viability of all generated partial QEPs.

**Testing plan viability** To test if a sub-plan \( p \) with a binding pattern \( bp_p \) is viable, we proceed as follows. We construct a query \( Q' \) that is derived from \( Q \), in which we replaced all occurrences of \( p \)’s conjuncts with a single table \( R_p \); we consider this table to have a single binding pattern, which is \( bp_p \). It is easy to see that \( Q' \) is feasible if and only if \( p \) is viable. Thus, by using the solveBP algorithm we described in section 2.3.5, we can decide on the viability of a plan.
3.6.2 Dynamic programming optimization algorithm

The first query optimization algorithm using binding patterns that we present is based on dynamic programming; the algorithm is described in figure 3.5. The algorithm takes as input a conjunctive query $Q$, and one binding pattern per table appearing in the query. Also, this algorithm applies projections only at the end of the optimization process.

The algorithm starts by checking if $Q$ is feasible (the algorithm $\texttt{solveBP}$ has been given in figure 2.4). If the test is successful, the optimizer starts by constructing access plans to every $R_i$; if $bp_i$ is unrestricted, a Scan plan is constructed, otherwise, it is a BindAccess plan. The cheapest implementation is chosen for each of these access plans. Following the System R heuristic, all relevant selections are applied as soon as possible the resulting QEPs are inserted into $S$, the current set of partial QEPs (lines 2-7 of the algorithm).

In the sequel, the optimization algorithm starts building “generations” of plans, in the order of the number of conjuncts they cover (the $i$ parameter in line 8). In order to construct plans over $i$ conjuncts, we have to combine (via joins or dependent joins) plans that cover $j$ conjuncts with plans over $k = i - j$ conjuncts. The variable $p_j$ iterates over all plans in $S$; following the general principles we described, $S$ contains only the optimal plan per equivalence class constructed so far. We gather in $S_k$ all the join candidates for $p_j$ (line 13). The set $S_{res}$ contains all join and BindJoin products of $p_j$ with plans from $S_k$; we eliminate from $S_{res}$ the plans that are not viable, and we eagerly apply all possible selections on the resulting plans (lines 14-16). The strength of the dynamic programming paradigm for query optimization is that it allows pruning all suboptimal plans as soon as they are generated. In our case, for every plan $q$ (including the selections) from $S_{res}$ we check to see if $S$ did not already include a cheaper plan from the same equivalence class. Thus, $q$ is kept in $S_{res}$ only if (a) there was no plan in $S$ from the same equivalence class or (b) there was such a plan in $S$, but it was more expensive than $q$. Finally, the plans that survived pruning and are still in $S_{res}$ are added to $S$.

When all plans covering the $n$ query tables have been generated, we check to see if among them, there are plans covering the query’s equivalence class: these are plans that allow, by some selections and projections, to compute the desired query result. The algorithm returns the cheapest plan among the complete plans obtained, if there are any. Note that complete QEPs are only obtained at the end of the algorithm. (Actually, complete plans may be identified a little bit earlier, while constructing the plans over $n$ conjuncts; for readability, we isolated the solution identification phase from the plan generation phase, since the former follows closely the latter.)

The extensions that this algorithm brings over the classical System R paradigm are: the query feasibility test (lines 1-2), the generation of plans with binding patterns (line 14), and the viability test on partial generated QEPs (line 15).

3.6.3 Best-first search optimization algorithm

System-R builds query execution plans by considering one equivalence class at a time. The equivalence classes are considered in order of the number of conjuncts they cover. Therefore,
the best query execution plan of a class and its cost are determined at one point and are not changed later. The disadvantage of this strategy is that the first complete query execution plan is obtained only at the last phase of the optimization. As the analysis in section 3.5.3 showed, such behavior will not be acceptable in our context.

To address the problem of large search spaces, we employ a best-first search algorithm which interleaves the exploration of different equivalence classes. Specifically, we associate a utility measure with each partial execution plan we produce. The utility function depends on the number of conjuncts that are covered by the plan and the number of input variables. At each step of the search we choose the partial execution plan with the greatest utility measure, and try to combine it with plans that cover a disjoint set of conjuncts in the query.

The advantage of the best-first search algorithm is that we can tune the utility function to produce a complete plan relatively fast. The main disadvantage of the algorithm is a consequence of the fact that we do not consider each equivalence class in isolation. Therefore, the cost of the best plan for an equivalence class may decrease over time, and hence the cost of the plans using it has to be changed accordingly. The extra bookkeeping resulting from tracking the changes to the costs incurs additional cost. As we show in section 3.6.5 the tradeoff between the two factors is in our favor.

The best-first search optimization algorithm differs from dynProg in the following respects:

1. The choice of the partial query execution plans to be combined is based on a utility measure. This is significantly different from System-R, where equivalence classes are considered strictly in order of the number of conjuncts they cover.

2. A novel method for selection placement is used, that guarantees we do not miss optimal plans, while avoiding to exhaustively generate all join and selection combination.

Placement of selections

Considering all the possible join orderings and the possible placement of selections blows up the size of the search space. Previous works have used the powerful heuristic of decoupling the decision about join orders and the placement of selections. In System-R the optimizer heuristically introduces selections as soon as possible in query execution plans (pushing them down as far as possible). In the presence of expensive predicates [63, 60] the placement of the selections is done in a cost-based fashion in a separate phase, after the join ordering decision has been made. That method has been proved sub-optimal, and [22] provides an extension to a dynamic programming-based algorithm, that is guaranteed to find the optimal placement.

As shown in the example 3.5.3, it is possible that a plan with a selection as the top operator may be in the same equivalence class as a plan that does not contain any selection. Hence, if we completely ignore selections during the generation phase, we could miss the optimal plan. The goal of our algorithm is to consider selections in the combination phase only to the extent that it is required in order not to miss optimal plans.
Algorithm \text{bestFirst}

Input

\begin{itemize}
    \item $Q$: "select $\overline{X}_{res}$ from $R_1(\overline{X}_1), R_2(\overline{X}_2), \ldots, R_n(\overline{X}_n)$ where $J(\overline{X})$ and $C(\overline{X})$"
    \item for each $R_i$, $bp(R_i) = \{b_{p_i}\}$
\end{itemize}

Output

optimal QEP for $Q$

1. if $Q$ is not feasible
2. then stop
3. let $S = \emptyset$
4. foreach $R_i$, $i = 1, 2, \ldots, n$
5. \hspace{1em} let $p_i$ be the optimal access plan to $R_i$ following $bp_i$ (Scan or BindAccess)
6. \hspace{1em} let $S = S \cup p_i$
7. while new plans are generated
8. \hspace{1em} let $p$ be the plan from $S$ maximizing the utility measure
9. \hspace{1em} let $S_{cand} = \{q \in S$ such that $q$'s conjuncts are disjoint from $p$'s$\}$
10. \hspace{1em} foreach $q \in S_{cand}$, in the decreasing order of their utility measure
11. \hspace{2em} let $r = p \bowtie q$ or $r = p \overline{\bowtie} q$
12. \hspace{2em} if there exists $r' \in S$, such that $r'$ covers $r$, and $cost(\sigma(r')) < cost(r)$
13. \hspace{2em} then let $r = \sigma(r')$
14. \hspace{2em} if there exists $r'' \in S$, $r'' \equiv r$
15. \hspace{2em} then if $cost(r'') < cost(r)$
16. \hspace{3em} then discard $r$
17. \hspace{3em} else foreach $s \in S$, $s$ uses $r''$
18. \hspace{3em} replace $r''$ by $r$ in $s$ and recompute the cost of $s$
19. \hspace{2em} if $r$ is not viable
20. \hspace{3em} then discard $r$; continue
21. \hspace{2em} let $S = S \cup \{r\}$
22. \hspace{2em} foreach $t \in S$ such that $t$ covers $r$ and $cost(\sigma(t)) < cost(t)$
23. \hspace{3em} replace $t$, in every $v \in S$ using $t$, by $\sigma(t)$ and recompute $cost(v)$
24. let $S_{cov}$ be the set of plans in $Q$'s equivalence class, or covering this class
25. let $S_{sol} = \emptyset$
26. foreach $r \in S_{cov}$
27. \hspace{1em} insert in $S_{sol}$ all possible placements for the remaining selections
28. \hspace{1em} and projections on $r$
29. return cheapest plan from $S_{sol}$

Figure 3.6: The best-first search query optimization algorithm.
The best-first search algorithm considers selections in the following fashion. Suppose we have created a new plan \( p \) which is the cheapest one found so far for its equivalence class. Before proceeding, the algorithm checks if it is possible to obtain an equivalent plan to \( p \) by applying a selection to a plan that already exists in \( S \). Specifically, the algorithm checks if there exists a plan \( p' \in S \), such that \( p' \) covers \( p \) (i.e., \( p \) is equivalent to a selection applied to \( p' \)), and the cost of applying the selection to \( p' \) is less than the cost of \( p \). In this case, the plan with a selection on \( p' \) is added to \( S \) instead of \( p \).

Furthermore, the algorithm checks whether applying a selection on \( p \) enables to improve the best plan of another existing equivalent class. Specifically, the algorithm checks whether there exists \( p' \in S \) such that a selection on \( p \) is equivalent to \( p' \), and the cost of the selection on \( p \) is less than the cost of \( p' \). In this case, the algorithm replaces the plan \( p' \) in \( S \) by the plan with a selection on \( p \).

The effect of the two steps described above is that the set of equivalence classes maintained in \( S \) can be characterized as follows: if a class \( C \) is in \( S \), then there exists at least one query execution plan in \( C \) that uses only the join operator and no selections. The classes for which all plans contain at least one selection are not maintained in \( S \). In a sense, this property entails that the algorithm maintains a minimal number of equivalence classes. Furthermore, for these equivalence classes, the algorithm will find the optimal plan with selections.

As a result of the above property, it may be the case that at the end of the generation phase, the equivalence class corresponding to the original query is not in \( S \). Assuming the query has at least one query execution plan, this can only happen when all the query execution plans for the query contain selections. In this case, it is easy to check that the optimal plan for the entire query can be obtained by introducing selections in the optimal plans of the equivalence classes covering the query.

Hence, only in the case when the equivalence class of the query does not belong to \( S \) at the end of the generation phase, the algorithm applies a second phase, which exhaustively enumerates all the possible placement of selections, but only in the optimal plans of the equivalence classes covering the query. Hence, the cost of this phase is relatively small.

The bestFirst optimization algorithm is depicted in figure 3.6. At the beginning, the algorithm constructs access plans, just like dynProg. However, in the main plan generation loop (lines 8-24), partial QEPs are considered in the increasing order of their utility, estimated by the optimizer using some utility measure. The pruning conditions explained above are more complex; in figure 3.6, we used the \( \equiv \) notation for plan equivalence. The extra complexity of the pruning criteria is due to the fact that, unlike the dynamic programming algorithm, bestFirst does not explore one equivalence class within well-defined interval. (This interval is included in the iteration for the \( i \) value corresponding to the plan in the dynProg algorithm.) Using best-first search, we may insert a plan \( p \) for a given equivalence class, construct more complex plans using \( p \) as a building block, discover later a cheaper plan \( q \equiv p \), and replace \( p \) with \( q \) in all such plans. This replacement also entails recomputing the costs of these plans.
3.6.4 Implementation

We implemented both dynProg and bestFirst optimization algorithms. We used the same data structures (as described shortly) for the two algorithms, and were careful to ensure that the optimizations made in the data structures to efficiently support best-first search do not bias the running times against dynamic programming. The implementation was performed in Java, using JDK 1.0. We briefly explain here some of the choices made when implementing the algorithm we already described.

Data structure for the set of partial query execution plans A crucial issue that was considered in the implementation is developing a data structure for storing the set of partial plans that have been constructed (denoted by $S$). An optimal such structure would need to efficiently support the following accesses to the set of plans:

- For a plan $p$, find all plans $q \in S$, such that $p$ and $q$ have disjoint sets of conjuncts (i.e., the join candidates for $p$).
- For a plan $p$, find an equivalent plan $p' \in S$.
- For a plan $p$, find all the plans $q$ that cover $p$.
- For a plan $p$, find all the plans $q$ that are covered by $p$

Given these requirements and the observed frequencies of the different accesses, we decided to adapt the following indexing structure for $S$. Plans are clustered by the set of conjuncts that compose them; note that the join candidates are the same for all the elements of a cluster. In order to avoid repetitive computation of the joinable clusters, the link between joinable clusters is established and materialized when the cluster is given its first member. In addition, the plans in each cluster are indexed by their adornments. It should be emphasized that since equivalent and covering plans belong to the same cluster, and the size of the clusters is relatively small, optimal performance was achieved by not adding structures for indexing equivalent and covering plans. Finally, in order to support best-first search, every plan contains a link to the plans using it.

Cost Model In our experiments we considered a relatively simple cost model. The cost is derived from the cost of the leaf data accesses and standard formulas for computing the cost of joins (hash-join and nested-loop dependent joins). Costs of selections are assumed to be negligible, even though they affect the cardinality of the results. As long as the cost model respects the monotonicity property, the choice of the model is irrelevant to the results we show in the experiments.
Figure 3.7: The top graph shows the absolute time taken to find the first solution. The bottom graph shows the ratio between the time to first solution and the time for exhaustive search.

**Utility Measure** The bestFirst algorithm is based on a utility function for choosing the next plan to expand. In our experiments we considered several measures, including (1) the number of conjuncts covered by a plan (2) cost of the plans, (3) number of free variables, and several combinations of the 1–3. Considering only measure (1) resulted in better performance (e.g., up to a factor of 4) in terms of total time and time to first solution, even though the quality of the plans produced early on were not as good as in several of the more complex measures. Considering complex utility measures produces better plans early on in the search but the overhead of the search is significant.
Figure 3.8: The graphs show the time for exhaustive search in the case of varying the number of bind transformations (top) and varying the number of addBind transformations (bottom).
3.6.5 Experimental comparison of dynamic programming and best-first algorithms

Experiments were run on a SUN 4 SPARC, under Solaris, using JDK with 100Mb of memory. The main limitation of the algorithm comes from the memory consumption, which is probably increased by the fact that we used Java. Clearly, the use of Java also affects absolute running times of both algorithms, which is why our study focuses on relative numbers. Every point in the graphs is obtained by averaging over 20 queries generated randomly with the same parameters (as described in Section 3.5.3). All the experiments are done with queries including 10 relations and 70 variables.

The main motivation for developing bestFirst is that dynamic programming produces the first solution relatively late in the optimization, which yields unacceptable performance in the context of large search spaces. The advantage of best-first search is that it produces the first solutions relatively quickly, but with the added expense in total optimization. Hence, in the following, we quantify the gain in terms of finding the first solution and the price for total optimization time.

Time to first solutions Figure 3.7(top) shows the time taken to obtain the first solution for the bestFirst and dynProg algorithms. The figure varies the number of addBind transformations from 0 to 8. Recall that this variation has the effect of significantly increasing the size of the search space. We observe that the time to first solution for bestFirst is almost constant as the size of the search space increases, while dynamic programming degrades considerably.

Figure 3.7(bottom) shows the ratio between the time to first solution and the total optimization time for both algorithms. Since, as we show below, the total optimization time is in favor of dynamic programming, this graph underscores the superiority of bestFirst w.r.t. finding the first solution. Finally, we observe that the ratio for the best-first search algorithm is relatively constant, while it grows for dynamic programming.

It is important to emphasize that bestFirst produces solutions in a relatively steady pace. Hence, we are more likely to obtain a good solution even before dynamic programming produces its first.

Time for exhaustive search Figure 3.8 compares the running times for exhaustive search for the two algorithms, as we vary the number of bind transformations (top) and as we vary the number of addBind transformations (right). We observe that in both cases dynamic programming has a better running time. In the case of bind transformations (when the size of the search space decreases), the best-first search algorithm takes double time than dynamic programming in the worst case. As the number of bind transformation increases, and hence the size of the space decreases, the differences between the running times are negligible. In the case of addBind transformations, the running time of both algorithms grows exponentially (note that the Y axis is on a logarithmic scale). Even though bestFirst performs worse, the general growth tendency is the same as for dynamic programming.

82
3.7 Distributed query optimization algorithm in LeSelect

In this section, we describe the query optimization algorithm that we designed and implemented in the context of the LeSelect data integration system. This algorithm is designed for queries over tables with binding patterns, resulting from the modeling of heterogeneous resources described in chapter 2. In some respects, as we will show, LeSelect’s optimizer considers a more restricted search space than the one we described in section 3.5-2; on the other hand, besides binding pattern restrictions, it handles heterogeneous query execution capabilities, and takes a pragmatic approach in order to allow several binding patterns of the same table to be used when answering a query.

We start by describing the query execution environment of LeSelect and its consequences on the query optimization algorithm, in section 3.7.1. We then outline the optimizer’s cost model in section 3.7.2, and describe the actual algorithm in section 3.7.3.

3.7.1 Optimization guidelines

In this section, we explain the basic principles that guided our choices when designing a distributed query optimizer for LeSelect. These principles follow from the general architecture of LeSelect, and from the context of its frequent applications.

Language and model The query language that LeSelect supports is the select-project-join subset of SQL, enriched at top-level with the union operator. A query is received by the optimizer as a union of query terms in disjunctive normal form (DNF): $Q = Q_1 \cup Q_2 \cup \ldots \cup Q_m$, where each $Q_i$ is of the following form:

$$Q_i: \text{select } E(\overline{X}_{\text{proj}}) \text{ from } R_1(\overline{X}_1), R_2(\overline{X}_2), \ldots, R_n(\overline{X}_n)$$

$$\text{ where } C(\overline{X}) \land D_1(\overline{X}) \land D_2(\overline{X}) \land \ldots D_k(\overline{X})$$

In this formula, $E(\overline{X}_{\text{proj}})$ is a set of arithmetic expressions over the set of variables $\overline{X}_{\text{proj}} \subseteq \overline{X}$. Each $D_i(\overline{X})$ is a disjunction of several comparison predicates of the form $e_1(\overline{X}) \alpha e_2(\overline{X})$, where $e_1(\overline{X})$, $e_2(\overline{X})$ are arithmetic expressions defined over variables from $\overline{X}$, and $\alpha \in \{=, \neq, <, \leq, >, \geq\}$, while $C(\overline{X})$ is a conjunction of comparison predicates. That is, we group for our convenience in $C(\overline{X})$ all disjunctions consisting of a single predicate.

Each table appearing in the from clause of an union term is identified by its universal name, containing the table name, the name of the wrapper on which the table is published, and the name of the LeSelect server to which the wrapper is attached. Each union term is optimized in isolation, and the optimal QEP is the union of the optimal plans found for the terms.

We assume usual bag semantics for the query; also, the top-level union includes all duplicates.
Minimizing total work  The goal of the query optimization process in LeSelect is to find a QEP minimizing the total work (TW). Reducing the total work necessitated by a given query is more important that reducing its response time (RT), since we aim at providing an overall good throughput with respect to all the servers involved. Some interesting issues linked to reducing the query response time are discussed in the next chapter, in section 4.6.

Restriction on QEPs involving BindJoins  In general, a BindJoin can have arbitrary operator trees as left-hand and right-hand children. However, supporting QEPs where a BindJoin has an arbitrary QEPs as a right-hand child is quite complicated. The reason is that all operators involved in the right-hand child of the BindJoin must be able to receive bindings from the BindJoin, and pass them on to their children, lower in the QEP. This top-down information propagation is different from the normal bottom-up information flow in a QEP composed of iterators [51]. LeSelect operators, on the other hand, are implemented as iterators. Therefore, the current BindJoin implementation accepts only BindAccess operators as right-hand child. Conversely, a BindAccess operator only appears as the right-hand child of a BindJoin.

In section 3.5.1, we have shown that supporting arbitrary QEPs as right-hand child of a BindJoin may lead to more efficient query execution. On the other hand, if the right-hand child of a BindJoin is limited to a BindAccess operator, we do not restrict the set of feasible queries.

As a consequence, when optimizing a query in LeSelect, all partial QEPs, except for BindAccess plans, are closed (in the sense defined in section 3.5.1). Indeed, consider an open partial QEP \( q \), i.e, one that needs bindings from another part of a QEP in order to be executed. Then, we would need to have \( q \) as a right-hand child of a BindJoin; then, \( q \) needs to be a BindAccess plan.

The Evaluate operator  As a consequence of the above restriction on QEPs involving BindJoins, we face the following problem: our operator set is not rich enough to answer an important subset of feasible queries, as the following example shows.

Example 3.7.1 Consider the tables \( R(X, Y, Z) \), \( S(W) \), with \( bp(R) = \{ R(X^bY^bZ^j) \} \), \( bp(S) = \{ S(W^j) \} \). Let \( Q \) be the query

\[
\text{select } * \text{ from } R, S \text{ where } R.X = 5 \text{ and } R.Y = S.W
\]

One of the bindings required by \( R \) is provided by the join predicate, while the other one comes from a selection condition. However, we are not able to construct a plan of the following form, since we forbid any operator between a BindJoin and a BindAccess:

\[
\text{Scan}(S) \Join \sigma_{X=5}BA(R(X^bY^bZ^j))
\]

We cannot apply the selection after the BindJoin, either, because the value for \( X \) needs to be known before executing the BindJoin.

84
To solve such problems, we introduce a very simple logical operator that we call Evaluate. An Evaluate operator takes as input a QEP and a set of arithmetic expressions involving its columns. The output of the Evaluate operator is the set of input tuples, to each of which has been appended the result of evaluating the input expressions on that tuple. Formally, if we denote by $E(\overline{x}) = (e_1(\overline{x}), e_2(\overline{x}), \ldots, e_k(\overline{x}))$ the input expressions, we have:

$$Eval(p(\overline{x}), E(\overline{x})) = \{(\overline{x}, \overline{v}) | \overline{x} \in p(\overline{x}) \land \overline{v} = E(\overline{x})\}$$  \hspace{1cm} (3.8)

Using Eval, the query from the example 3.7.2 can be answered by the following QEP:

$$Eval(Scan(S), 5) \bowtie BA(R(X^b Y^b Z^f))$$

The Evaluate operator is also useful to compute the result of some arithmetic expressions in one of the following two cases: (a) to provide it as a binding to a BindAccess operator, or (b) to return it to the user.

**Example 3.7.2** Consider the tables $R(X, Y), S(Z, W)$, having $bp(R) = \{R(X^b Y^b)\}$ and $bp(S) = \{S(Z^b W^b)\}$. Let $Q_1$ be the query select * from $R, S$ where $R.X = S.Z + 2 * S.W$

The only QEP for $Q_1$ is:

$$Eval(Scan(S), (Z + 2 * W)) \bowtie BA(R(X^b Y^b))$$

Now consider the table $R(X, Y)$, with $bp(R) = \{R(X^f Y^f)\}$, and the query $Q_2$:

select $R.X + R.Y$ from $R$ where $R.X = 5$” This query is answered by the QEP $Eval((\sigma_{X=5} Scan(R)), (X + Y))$

In the case when the select clause contains the arithmetic expressions $E(\overline{x}_{proj})$, we optimize the query returning $\overline{x}_{proj}$ and apply the final Evaluate step.

**Bushy join trees** A query posed to LeSelect involving distributed resources is very likely to be executed in a distributed fashion. Thus, the distinct parts of a QEP may be processed in parallel, at different locations. To take full advantage of these parallelism opportunities, and to examine a bigger search space, the query optimizer inspects the space of bushy join trees. Of course, this only concerns regular joins, and excludes BindJoins as explained above. Finally, whether the optimizer running on a given LeSelect server uses bushy trees or not is controlled by a switch, whose default value is true.

**Algorithm based on dynamic programming** LeSelect’s query optimization algorithm is based on dynamic programming. While in section 3.6 we had described the advantages of a best-first search algorithm over dynamic programming enhanced with binding patterns, in the case of LeSelect we chose dynamic programming for the following reasons. First, since we do not consider open partial QEPs, the size of the search space is close to the one to which is confronted a “regular” query optimizer (i.e. one that does not consider binding patterns). Second, in the queries we encountered in the existing applications of LeSelect, few queries involved more than a relatively small number of tables. Therefore, we can afford to wait until the optimization completes, even if we might apply some heuristics (e.g., disallow bushy join trees) for particularly large queries.
Use of Cartesian products We generate plans including Cartesian products. The main reason is that some queries become simply unfeasible if we rule out Cartesian products, due to restricted access patterns.

Example 3.7.3 Consider the tables $R(X)$, $S(Y)$ and $T(U, V, W)$, such that $bp(R) = \{R(X')\}$, $bp(S) = \{S(Y')\}$, and $bp(T) = \{T(U^b, V^b, W')\}$. Furthermore, let $Q$ be the query

$$\text{select } * \text{ from } R, S, T \text{ where } R.X = T.U \text{ and } S.Y = T.V$$

Although the query provides enough bindings to access $T$, combining those bindings requires a Cartesian product of $R$ and $S$. Therefore, such queries can only be answered by using Cartesian products.

If we allow arbitrary QEPs as right-hand children of a BindJoin, i.e. unrestricted bushy trees, then, as we shown in section 3.5.1, we can avoid Cartesian products. In LeSelect, with our constraint on QEPs involving BindJoins, we are forced to investigate Cartesian products.

Partial usage of disjunctive predicates The optimization algorithms described in the previous section deal with conjunctive queries only. In section 2.3.4, we have shown that the result of a disjunctive query over tables with binding patterns can be obtained only by a computation on the results of a set of conjunctive queries. The number of these conjunctive queries is exponential in the number of disjunctions appearing in the query, and the final computation, described in proposition 2.3.6, involves difference and intersection operations on these results; this approach is potentially very expensive. Therefore, we adopt the following partial solution. We answer a disjunctive query $Q$ only if all its disjunctive predicates can be applied as selections or regular join predicates, i.e. we do not use these predicates to provide bindings for restricted access resources. Formally, we are able to optimize the disjunctive query

$$Q: \text{select } \overrightarrow{X}_{\text{proj}} \text{ from } R_1(\overrightarrow{X}_1), \ldots, R_n(\overrightarrow{X}_n) \text{ where } C(\overrightarrow{X}) \land D_1(\overrightarrow{X}) \land D_2(\overrightarrow{X}) \land \cdots \land D_k(\overrightarrow{X})$$

only if the following conjunctive query $Q'$ is feasible:

$$Q': \text{select } \overrightarrow{X}_{\text{proj}} \cup (\cup_{i=1,2,\ldots,k} \text{Var}(D_i(\overrightarrow{X}))) \text{ from } R_1(\overrightarrow{X}_1), R_2(\overrightarrow{X}_2), \ldots, R_n(\overrightarrow{X}_n) \text{ where } C(\overrightarrow{X})$$

Work distribution among wrappers and LeSelect servers A query posed to a LeSelect server is in general executed by sharing the work among:

- the wrappers of the resources appearing in the query;
- the servers to which these wrappers are attached;
- the server where the query has been asked.
We are restricted to this set of participants during query execution, because the LeSelect server receiving the query has no way of knowing which other LeSelect servers exist. Indeed, for the time being, there is no central registry listing the LeSelect servers that are up and available.

We explained how the wrappers specify their query processing capabilities in section 2.4. In particular, access plans are always executed by wrappers. A LeSelect server is capable of executing any relational operator, on resources published via wrappers on the same site, or on data shipped from a distant site. Note that restricted resources cannot be shipped; they are only accessed through local BindAccess operators, and it is the resulting data that can be sent to a distant server.

In a QEP constructed by our optimizer, each operator is labeled with its location, that can be either a wrapper, or a LeSelect server. During the execution of a distributed query, a LeSelect server $S_1$ may accept to process a partial QEP $p$. The leaf operators in $p$ are executed, as we said, by wrappers. However, it might be the case that a subtree of $p$ is labeled with a location that is neither $S_1$ nor any of the wrappers attached to it. In that case, $S_1$ delegates part of the work to the corresponding location. Thus, the work that $S_1$ does takes as input the results of one or several subqueries; in some sense, $S_1$ performs the operations around the root of the tree, while the leaves and maybe some subtrees are delegated to other parties.

\subsection*{3.7.2 Cost model}

We now describe the cost model used by LeSelect’s query optimizer. This cost model is admittedly very simple; more detailed cost informations could be directly incorporated into the optimization algorithm, yielding better quality plans.

\textbf{Limited statistic information} When we delegate the execution of a QEP fragment to wrapper or a LeSelect server, they will not provide their own cost estimation for the QEP fragment. Instead, the optimizer on the query site assigns the same cost to a given operator, independently of the location where the operator will be executed. In reality, this assumption is likely to be false, but it is difficult to gather more precise information:

\begin{itemize}
  \item Detailed costs statistics for wrappers would require specific skills from the resource publishers, and since we do not expect LeSelect servers to be installed and used by computer scientists, it is very unlikely that the users have these skills.
  
  \item It is difficult to get precise statistics concerning the execution times on a LeSelect server, since we know very little about the hardware and software environment where the server is running, that machine may be solicited by other tasks that we cannot foresee, monitor or control.
\end{itemize}

The only location-dependent cost informations that the optimizer has are the cost estimates provided by wrappers for accessing their resources, as explained in section 2.4.
The current architecture of LeSelect could be enriched by requiring cost estimates from wrappers and servers. Also, a statistic-gathering component may be added, to supervise query execution and collect observations concerning the execution times, construct data histograms etc., as it is currently done in commercial DBMSs. The optimizer would be able to profit from these extensions.

**Basic cost parameters**

Besides the metadata exported by the wrappers, as described in section 2.4, the optimizer of a given LeSelect server uses the following elementary parameters of the cost model:

- size of a buffer page inside a LeSelect query engine, in Kb;
- number of buffer pages available to the execution engine;
- for every two sites $S_1$ and $S_2$, an estimate of the bandwidth from $S_1$ to $S_2$, denoted $BWS_{S_1 \rightarrow S_2}$.

Of course, one constant value is a very crude estimate of the bandwidth between two sites, since the time of the day, the network congestion etc. may have an important influence on the available bandwidth. A learning-based tool for detailed estimation of the network transfer time between two sites has been proposed in [53].

**Cost formulas**

The cost of any distributed QEP $p$ is computed as a weighted sum of its *execution cost* and *transfer cost*:

$$cost(p) = exec(p) + \gamma \cdot transfer(p)$$

(3.9)

The execution cost is computed recursively as the sum of the execution costs of all operators in the QEP rooted in $p$. Similarly, the transfer cost is the sum of the transfer costs on all the edges of the QEP rooted in $p$.

A direct consequence of this simplistic cost model is that *selections and projections should be applied as early as possible*. Indeed, a QEP in which a selection has been delayed after the moment when it first became applicable has the same execution cost, and a greater transfer cost, than the same QEP where the selection has been pushed down as much as possible. Pushing selections and projections to diminish data transfers is a common heuristic when executing distributed queries; we review and discuss existing approaches for placing selections in section 3.8.

We now provide formulas used in LeSelect for estimating the cost of a QEP. In the case of usual relational operators (join, selection, projection), these formulas can be found in, e.g., [103]. We show only the formulas for BindAccess and BindJoins, since we devote special attention to these operators, and Scan plans, to allow a comparison with the BindAccess.

Using the notations from section 2.4, the operators’ *execution costs* are estimated as follows:
Scan  Let $p$ be a Scan plan for relation $R$, using a binding pattern $bp$ of the form $R(V^f)$. The execution cost of $p$ is: $c_{bp}^0 + n_R * c_{bp}$.

BindAccess  The execution cost of a BindAccess plan for relation $R$, using a binding pattern $bp$ of the form $R(U^b V^f)$, is: $c_{bp}^0 + s_{bp} * c_{bp}$. The interpretation is that a BindAccess plan returns the set of tuples corresponding to an access with one binding set for the bound attributes $U$. Indeed, we cannot estimate more precisely the set of tuples returned by the BindAccess, before we know what its bindings are.

BindJoin  Let $p = r \Join_\overline{X} s$ be a BindJoin plan, where $\overline{X}$ are the binding arguments passed to the BindAccess $s$. The BindJoin execution cost is: $n_r * cost(s)$. Indeed, $s$ is accessed once per tuple in $r$.

The fanout of a QEP $p$ is derived from the fanout of its child operators as follows:

Scan  If $p$ is a Scan on $R$, i.e. it accesses $R$ following $R(V^f)$, then $n_p = n_R$.

BindAccess  If $p$ is a BindAccess plan for table $R$, using the binding pattern $bp = R(U^b V^f)$, then the fanout of $p$ is $n_p = s_{bp}$. If we consider the BindAccess$^0$ variant, the fanout becomes $s_{bp} * dv_{V,R}$, where $dv_{V,X}$ is the ratio of distinct values in the $V$ columns of $R$ (in other words, $dv_{V,X}$ is the ratio between the fanout of $\pi_0^X(R)$ and $n_R$). We will explain shortly how to estimate $dv_{V,X}$, when we discuss the estimation of distinct value ratios.

BindJoin  Although semantically equivalent to a join, the fanout of a BindJoin plan is computed differently. The reason is that the BindAccess plan that has to be the right-hand child of the join has a fanout corresponding to one argument tuple. Therefore, if $p = r \Join_\overline{X} s$, we have $n_p = n_r * n_s$.

During optimization, we also estimate, for every partial QEP $p(X)$, the ratio of distinct values in each column $X \in \overline{X}$, denoted by $dv_{X,p}$. This estimate is necessary in order to compute the selectivity of selections, joins etc. With our simple data statistics, the formulas we use are quite standard and are inspired from [103]. Recall that wrappers export for each table $R$ its fanout $n_R$, and for each column $X$ the number of distinct values of $X$ in $R$, $dv_{X,R}$. We assume the values in each column are uniformly distributed, and that except for functional dependencies, value distributions for any two different columns are independent.

We show the formulas for computing the distinct value ratio only for Scan, BindJoin and BindAccess.

Scan  If $p$ is a Scan plan over the table $R$, following $R(V^f)$, then, for every $X \in V$, $dv_{X,p} = dv_{X,R}$.

BindAccess  If $p$ is a BindAccess to table $R$, following a binding pattern $R(U^b V^f)$, then, for every $X \in U$, $dv_{X,p} = 1.0$, and for every $Y \in V$, $dv_{Y,p} = dv_{Y,R}$, where $dv_{Y,R}$ is the distinct value ratio published by the wrapper. These estimations correspond to
the case where we access $R$ with one set of bindings for $\overline{U}$. Although in practice, the BindAccess is used with several successive bindings for $\overline{U}$, we cannot determine more precise parameters for the BindAccess before knowing where do the bindings come from. These estimation are made on the BindJoin operator(s) that use this BindAccess.

In the case of BindAccess$^{0}$, for every $X \in \overline{U} \cup \overline{V}$, we have $dv_{X,p} = 1.0$.

**BindJoin** Let $p = r \bowtie_{\overline{T}} s$, where $s$ is a BindAccess plan of the form $BA(R(\overline{U}^{b} \overline{V}^{f}))$. Remember that the BindJoin pairs each tuple from $r$ with $n_s$ tuples from $s$. Then, for each $X \in \overline{U}$, $dv_{X,p} = dv_{X,r}/n_s$. For every $X \in \overline{V}$, the number of distinct $X$ values for a given set of bindings for $\overline{U}$ is $dv_{X,s}$. In $p = r \bowtie s$, there are $dv_{\overline{T},r} \times n_p$ distinct values in the $\overline{U}$ columns, and therefore up to $dv_{\overline{T},r} \times n_p \times dv_{X,s}$ distinct values for $X$. Thus, the distinct value ratio for $X \in \overline{V}$ is $p$ is $dv_{X,p} = dv_{\overline{T},r} \times dv_{X,s}$.

The data volume that results from a QEP $p$, $size(p)$, is computed trivially as $n_p \times width(p)$, where $width(p)$ is the estimated tuple size for any tuple in $p$’s result. We estimate width as follows:

**Scan** If $p$ is a Scan plan on $R(\overline{V}^{f})$, $width(p) = \sum_{X \in \overline{V}} s_X$, where we recall from section 2.4 that $s_X$ is the average size, in bytes, of attribute $X$.

**BindAccess** If $p$ is a BindAccess plan on $R(\overline{U}^{b} \overline{V}^{f})$, then $width(p) = \sum_{X \in \overline{U} \cup \overline{V}} s_X$.

**BindJoin** If $p = r \bowtie s$, then $width(p) = width(r) + width(s)$.

The cost of transferring the result of $n_1$, executed at location $l_1$, to $l_2$, is computed as follows:

- if $l_1$ is a wrapper and $l_2$ is the site to which the wrapper is attached (or vice-versa), the transfer cost is 0.

- if $l_1$ is a wrapper on site $S_1$ and $l_2$ is a site $S_2$, the transfer cost is the same as the transfer cost from $S_1$ to $S_2$.

- the transfer cost of the result of $n_1$ from a site $S_1$ to a site $S_2$ is obtained by dividing the size of the result in $n_1$, $size(n_1)$, by the bandwidth from $S_1$ to $S_2$, $BW_{S_1 \rightarrow S_2}$.

This is an optimistic estimate, that assumes the whole available bandwidth from $S_1$ to $S_2$ is available for the needs of the query we optimize, which may be false in general. However, there is no mechanism in LeSelect to inquire which operations are currently being executed, within different queries, that involve data transfers between $S_1$ and $S_2$.

**Transfer costs between a BindJoin and a BindAccess** If $p = r \bowtie q$, the computation of the transfer costs involved in $p$ is slightly different. The reason is that in this case, two different types of data are exchanged by $p$ and $q$. First, the bindings for $q$ need to be transferred from $p$’s site to the site of $q$; second, the results of the restricted access are brought from $q$ to $p$. Both these two partial costs need to be reflected in the transfer cost of $p$.  

90
The triangle rule When constructing distributed QEPs, we assume the following the triangle rule holds. Let $loc_1$, $loc_2$, and $loc_3$ be three locations (wrappers or LeSelect servers). Then, it is always the case that

$$1/BW_{loc_1 \rightarrow loc_2} < 1/BW_{loc_1 \rightarrow loc_3} + 1/BW_{loc_3 \rightarrow loc_2}$$

(3.10)

This rule states that it takes less effort to send data directly from $loc_1$ to $loc_2$ than have the data transit from $loc_1$ to $loc_3$ and from $loc_3$ to $loc_2$. Indeed, whenever this assumption becomes false, we assume a “routing” component corrects the path from $loc_1$ to $loc_2$, indicating that it must go through $loc_3$.

### 3.7.3 Distributed query optimization algorithm

The goal of the optimization process is to construct a distributed QEP $p$, whose nodes are annotated with their execution location, that minimizes $cost(p)$. Following the work distribution principle we provided in section 3.7.1, the interesting localizations that we consider for the nodes of $p$ when solving $Q$ are:

$$loc(Q) = \bigcup_{R_i \text{ published on } S_i} W_i \{W_i, S_i\} \cup \{Q\}$$

(3.11)

In this formula, $S_Q$ is the LeSelect server where the query $Q$ has been posed. Our query optimization algorithm has to search the space of all distributed plans $p$ computing the answer to $Q$, such that (a) all leaf operators in $p$ are executed on some $W_i$, (b) the root operator in $p$ is executed on $S_Q$, and (c) internal nodes are executed on some locations from $loc(Q)$.

### General rules for operator localization

To decide whether an operator can be executed at a given location, the optimizer applies the following decision rules.

1. $Scan(R(\overline{V}^f))$ or $BindAccess(R(\overline{U}^b\overline{V}^f))$ can only be executed by the wrapper where $R$ is published.

2. Any other operator can be executed by any LeSelect server.

3. A wrapper $W$ can apply a relational operator if and only if (a) the children operators are executed on $W$, and (b) the wrapper’s capabilities are sufficient for the wrapper.

---

\(^{2}\)The topmost relational operator in $p$ might be executed elsewhere. However, in our QEPs, we insert special $Submit$ operators between nodes $n_1$ and $n_2$ whenever their locations are different. The $Submit$ nodes are relational no-ops, and their purpose is to mark the fact that $n_1$ submits a sub-query to $n_2$ and uses its output. Thus, if the topmost logical relational operator is executed on $S \neq S_Q$, the topmost physical operator will be a $Submit$ node located on $S_Q$, delegating $p$ to $S$. The placement of $Submit$ operators is completely determined by the locations of operators in $p$, therefore we will focus on choosing these locations and ignore $Submit$ nodes.
to execute the topmost operator. This last condition is decided, using the wrapper capabilities description we discussed in section 2.4, as follows:

(a) \( W \) can apply a projection only if \( W \cdot \text{canDoProject}() = \text{true} \).

(b) \( W \) can apply an Evaluate operator only if \( W \cdot \text{canDoEvaluate}() = \text{true} \).

(c) \( W \) can apply a \( n \)-ary union only if \( W \cdot \text{canDoUnion}() = \text{true} \).

(d) \( W \) can apply a selection on a logical condition in which all comparison operators are \( \text{“} = \text{”} \) only if \( W \cdot \text{canDoEqualityTest}() = \text{true} \). \( W \) can apply a selection on a condition involving (also) other comparators only if \( W \cdot \text{canDoAnyTest}() = \text{true} \).

(e) \( W \) can apply a HashJoin (on a conjunction of equality predicates) only if \( W \cdot \text{canJoin}() = \text{true} \) and \( W \cdot \text{canDoEqualityTest}() = \text{true} \).

(f) \( W \) can apply a NestedLoopJoin (on a disjunctive condition, or a conjunction of predicates involving other comparators than \( \text{“} = \text{”} \)) only if \( W \cdot \text{canJoin}() = \text{true} \) and \( W \cdot \text{canDoAnyTest}() = \text{true} \).

(g) \( W \) can apply a Cartesian product if and only if \( W \cdot \text{canJoin}() = \text{true} \).

**Localization of unary operators**  In LeSelect, selections are applied eagerly, projections are pushed, and Evaluate operators are only inserted where they are necessary: between a plan \( p \) and a BindJoin operator, or as the topmost operator in a complete QEP. The triangle rule expressed by equation 3.10 has some interesting consequences on the localization of unary operators.

Assume we have to place an unary operator \( op \) (selection, projection, or evaluate) between two QEP nodes \( n_1 \) and \( n_2 \), localized at \( \text{loc}_1 \) and respectively \( \text{loc}_2 \). We denote by \( c_x \) the cost of the QEP \( n_2^{\text{loc}_2}(op^{\pi}(n_1^{\text{loc}_1})) \), where \( x \in \text{loc}(Q) \) and superscripts indicate the localization of operators. We have:

\[
c_x = \text{cost}(n_2^{\text{loc}_2}(op^{\pi}(n_1^{\text{loc}_1})) = \\
= \text{cost}(n_1) + \text{size}(n_1)/BW_{n_1 \rightarrow x} + \text{cost}(op) + \text{size}(op(n_1))/BW_{x \rightarrow n_2} + \text{cost}(n_2)
\]

In the particular cases when \( x = \text{loc}_1 \), respectively \( x = \text{loc}_2 \), we obtain:

\[
c_{\text{loc}_1} = \text{cost}(n_2^{\text{loc}_2}(op^{\pi}(n_1^{\text{loc}_1})) = \text{cost}(n_1) + \text{cost}(op) + \text{size}(op(n_1))/BW_{n_1 \rightarrow n_2} + \text{cost}(n_2)
\]

\[
c_{\text{loc}_2} = \text{cost}(n_2^{\text{loc}_2}(op^{\pi}(n_1^{\text{loc}_1})) = \text{cost}(n_1) + \text{size}(n_1)/BW_{n_1 \rightarrow n_2} + \text{cost}(op) + \text{cost}(n_2)
\]

If \( op \) is data reducing, i.e., \( \text{size}(n_1) > \text{size}(op(n_1)) \), then, for any \( x \in \text{loc}(Q) \),

\[
c_x - c_{\text{loc}_1} = \frac{\text{size}(op(n_1))(1/BW_{n_1 \rightarrow x} + 1/BW_{x \rightarrow n_2} - 1/BW_{n_1 \rightarrow n_2}) + (\text{size}(n_1) - \text{size}(op(n_1)))/BW_{n_1 \rightarrow x}
\]

By the triangle rule, the first term of the above sum is positive; also, since \( op \) is data-reducing, the second term is positive, too. Thus, \( c_x \geq c_{\text{loc}_1} \).

If \( op \) is data-inflating, i.e., \( \text{size}(n_1) < \text{size}(op(n_1)) \), then, for any \( x \in \text{loc}(Q) \),

\[
c_x - c_{\text{loc}_2} = \frac{\text{size}(n_1)(1/BW_{n_1 \rightarrow x} + 1/BW_{x \rightarrow n_2} - 1/BW_{n_1 \rightarrow n_2}) + (\text{size}(op(n_1)) - \text{size}(n_1))/BW_{x \rightarrow n_2}
\]
By a similar reasoning, the above sum is greater than 0, thus \( c_x \geq c_{loc_2} \).
We draw the following conclusions:

1. The optimal localization for a selection applied on a partial QEP \( p \) is next to \( p \).
2. The optimal localization for a projection pushed between an operators \( n_2 \) and its parent \( n_1 \) is next to the location of \( n_2 \).
3. The optimal localization for an Evaluate operator inserted under a \( R_{loc} \) is next to \( loc \); finally, for an Evaluate inserted as the topmost operator in a QEP, the optimal localization is \( S_Q \), the query site.

Note that whenever \( op \) cannot be executed by a given wrapper, we can always place \( op \) on the server to which the wrapper is attached, and the above formulas still hold. By applying these three rules, the localization of an unary operator is completely determined by the localizations of its parent and child operator; thus, these operators do not increase the complexity of the optimization.

**Generating access plans - the genAccessPlans algorithm**

In chapter 2, we have shown that using several binding patterns for a given table we enlarge the subset of queries that we are able to answer; also, we have provided in proposition 2.3.5 semantic conditions that guarantee that accessing a table using a sequence of its binding patterns can only generate correct results. For simplicity, when we described the dynProg algorithm in section 3.6.2, we only considered one binding pattern per table. We now relax this constraint.

To make the optimizer search through the space of all possible plans for a given query \( Q \), a simple approach is the following. For a given query \( Q \), with the notations from the beginning of section 3.7.1, let \( Q' \) be the corresponding conjunctive query. We run the solveBP algorithm described in section 2.3.3 to obtain all sequences of binding patterns of \( R_1, R_2, \ldots, R_n \) that can be used to answer \( Q' \). For each binding pattern sequence obtained this way, we construct a set of access plans, and we optimize the original \( Q \) starting from these access plans. Remember that \( Q \) and \( Q' \) differ only by some disjunctive selection predicates. The returned optimal plan for \( Q \) will be the best among the plans found this way.

This simple approach may generate a lot of redundant work. To see why, let us assume that solveBP returns a number of \( n_{BP} \) binding pattern sequences. Let \( l \) be the number of tables among \( R_1, R_2, \ldots, R_n \) having only one binding pattern. Then, the binding patterns of these \( l \) tables will be part of all the \( n_{BP} \) binding pattern sequences. All partial QEPs accessing a subset of these \( l \) tables will have to be developed \( n_{BP} \) times, once for every binding pattern sequence produced by solveBP.

Therefore, we adopt the following pragmatic approach: we construct one set of access plans, albeit containing more plans than are used in a complete QEP, and only optimize \( Q \) once, starting from these access plans; thus, we avoid doing redundant work.

The genAccessPlans algorithm constructs the access plans and prunes them as follows.
Let $Seq$ be the set of binding pattern sequences returned by $\texttt{solveBP}$. Let $S_{bp}$ be the set of all binding patterns appearing at least in one sequence from $Seq$. For each unrestricted binding pattern from $S_{bp}$, we construct a Scan plan; for each restricted binding pattern, a BindAccess plans is generated. Note that at this stage, we do not distinguish between BindAccess and BindAccess\textsuperscript{$\mu$} plans, since we do not know yet if such a plan will be combined with a QEP that has already accessed the same table or not. This distinction is made when constructing BindJoin plans, as we explain further.

We illustrate the construction phase of $\texttt{genAccessPlans}$ with an example:

**Example 3.7.4** Consider the table $R(X, Y, Z)$, with $bp(R) = \{ R(X^1), R(X^bY^1), R(X^bZ^f) \}$ and $S(U, V)$, with $bp(S) = \{ S(U^fV^1), S(U^bV^f) \}$, and assume $R.X \rightarrow R.Y$. Let $Q$ be the query

$$\text{select } R.Y, S.V \text{ from } R, S \text{ where } R.X = S.U \text{ and } S.V = 2$$

The binding pattern sequences returned by $\texttt{bpSort}$ are: $[R(X^1), R(X^bY^1), S(U^fV^1)], [R(X^1), S(U^bV^f), R(X^bY^1)], [R(X^1), S(U^fV^b), R(X^bY^f)], [S(U^fV^1), R(X^bY^f)]$. Thus, the set of binding patterns used in this set of sequences is $S_{bp} = \{ R(X^1), R(X^bY^1), S(U^bV^f), S(U^fV^b) \}$. Therefore, we construct four access plans: $\text{Scan}(R(X^1))$, $\text{BA}(R(X^bY^1))$, $\text{BA}(S(U^fV^1))$, $\text{BA}(S(U^bV^f))$.

Since access plans to a table $R_i$ are selection-projection views over $R_i$, in theory we could compare and prune these plans. Whenever $p_1$ can be expressed as a $\sigma - \pi$ query over $p_2$, at a lower cost than $p_2$, we could discard $p_1$. However, as stated before, the current execution model of LeSelect does not allow placing selections or projections just above a BindAccess operators. Therefore, we can only partially prune plans, as follows:

1. Let $p_1$ and $p_2$ be two access plans of the form $\text{Scan}(R_i(\overline{V}_1^f))$ and $\text{Scan}(R_i(\overline{V}_2^f))$ for some table $R_i$; they can be compared only if $\overline{V}_1 \subseteq \overline{V}_2$. In this case, if $\pi(p_2)$ is cheaper than $p_1$, then $p_1$ can be discarded. Otherwise, both $p_1$ and $p_2$ are kept.

2. Let $p_1$ be an access plan of the form $\text{BA}(R_i(\overline{U}^b\overline{V}^f))$, such that $Q$’s where clause contains the selection conditions $\overline{U} = \overline{c}$, where $\overline{c}$ is a vector of constants. Let $p_2$ be an access plan of the form $\text{Scan}(R_i(\overline{W}^f))$, where $\overline{W}$ is a superset of $\overline{U} \cup \overline{V}$. If $p_1$ is more expensive than $\sigma_{\overline{U} = \overline{c}} \pi_{\overline{U} \cup \overline{V}}(p_2)$, then we can discard $p_1$. Otherwise, both plans are kept.

At the end of the pruning phase, $\texttt{genAccessPlans}$ applies all possible selections on the resulting Scan plans.

Finally, note that while the restricted access plans for a single table cannot be compared, the plans constructed using these resources (and BindJoins) are compared and pruned without any difficulty, since these plans can be obtained one from another by applying selections and projections.
Algorithm optimize
INPUT \( Q: \) select \( \overline{X}_{proj} \) from \( R_1(\overline{X}_1), R_2(\overline{X}_2), \ldots, R_n(\overline{X}_n) \)
where \( C(\overline{X}) \land D_1(\overline{X}) \land D_2(\overline{X}) \land \ldots D_k(\overline{X}) \)
OUTPUT optimal QEP for \( Q \)
let \( S=\)genAccessPlans\( (Q) \)
let \( m=\)size\( (S) \)
foreach \( k=2, 3, \ldots, m \)
   .foreach \( i=1, 2, \ldots, k-1 \)
        foreach \( p_i \in S \) such that \( p_i \) covers \( i \) conjuncts
            let \( S_{cand} = \{ q_j \in S \mid q_j \text{ covers } j \text{ conjuncts, disjoint from } p_i \text{'s conjuncts} \} \)
            let \( S_{join} = \)joinProds\( (p_i, S_{cand}, C(\overline{X}), D_1(\overline{X}), D_2(\overline{X}), \ldots, D_k(\overline{X})) \)
            applySelections\( (S_{join}, C(\overline{X}), D_1(\overline{X}), D_2(\overline{X}), \ldots, D_k(\overline{X})) \)
            prune\( (S_{join}, S) \)
            let \( S = S \cup S_{join} \)
        let \( S_{res} = \{ p \in S \mid p \text{ covers } \overline{X}_{res} \} \)
        pushProjections\( (S_{res}) \)
return cheapest plan from \( S_{res} \)

Algorithm joinProds
INPUT plan \( p(\overline{Y}) \), candidate plan set \( S_{cand} \), predicates \( C(\overline{X}), D_1(\overline{X}), D_2(\overline{X}), \ldots, D_k(\overline{X}) \)
OUTPUT join products of \( p \) and \( S_{cand} \)
if \( p \) is a BindAccess plan
    then return \( \emptyset \)
    let \( S_{join} = \emptyset \)
foreach \( q(\overline{U}^\overline{Y}^f) \in S_{cand} \)
    if \( \overline{U} \neq \emptyset \), i.e. \( q \) is a BindAccess plan
        then if \( C(\overline{Y}, \overline{U}) \) contains bindings for all variables in \( \overline{U} \)
            then let \( S_{join} = S_{join} \cup \{ p \overline{U}^S \mid S \text{ server } \in \text{loc}_Q \} \)
        else if \( C(\overline{Y}, \overline{V}) \land D_1(\overline{Y} \cup \overline{U} \cup \overline{V}) \land D_2(\overline{Y} \cup \overline{U} \cup \overline{V}) \land \ldots \land D_k(\overline{Y} \cup \overline{U} \cup \overline{V}) = \emptyset \)
            then let \( S_{join} = S_{join} \cup \{ p \times \text{loc } \mid \text{loc } \in \{ \text{loc}_p, \text{loc}_q, \text{server } \in \text{loc}_Q \}, \text{loc can execute } \times \} \)
        else let \( S_{join} = S_{join} \cup \{ p \times \text{loc } \mid \text{loc } \in \{ \text{loc}_p, \text{loc}_q, \text{server } \in \text{loc}_Q \}, \text{loc can execute } \times \} \)
return \( S_{prod} \)

Figure 3.9: Query optimization algorithm in LeSelect.
Join plan generation

In this phase, we combine partial QEPs through regular join and BindJoin operations, following the general principles of dynamic programming. The algorithm, outlined in figure 3.9, is quite similar to the dynProg algorithm we have described in section 3.6.2. Therefore, we only discuss its distinguishing features.

First, the join generation phase considers the set of access plans computed by genAccessPlans; for the reasons explained above, the number of these plans, \( m \), may be bigger than \( n \), the number of tables appearing in the query.

Second, when combining a plan \( p \) with a plan \( q \), we construct either BindJoin plans, if \( q \) is a BindAccess operator, regular join plans if there are some join conditions on the variables of \( p \) and \( q \), or Cartesian products otherwise. A BindJoin is generated using the plans \( p(\bar{W}) \) and \( q = BA(R(\bar{U}^W V^f)) \) if there exist a partition of \( \bar{U} \) in three disjoint subsets, \( \bar{U}_1 \cup \bar{U}_2 \cup \bar{U}_3 \), such that \( C(\bar{X}) \) contains:

- for every variable \( X \in \bar{U}_1 \), a selection predicate of the form \( X = c \)
- for every variable \( X \in \bar{U}_2 \), an equality predicate of the form \( X = Y \), where \( Y \in \bar{W} \)
- for every variable \( X \in \bar{U}_3 \), an equality predicate of the form \( X = e(\bar{W}) \), where \( e \) is an arithmetic expression.

If \( \bar{U}_2 = \emptyset \) and \( \bar{U}_3 = \emptyset \), \( p \) is not used, and we generate a plan where all bindings for \( q \) come from the selections in \( Q \): \( Eval(null, \bar{e}) \supseteq q \). Otherwise, we generate a plan of the form \( Eval(p, \bar{e}, e(\bar{W})) \supseteq q \), that is, we append to \( p \)'s columns all the columns that are missing in order to provide bindings for all columns in \( \bar{U} \). Of course, all predicates used for the BindJoin, including simple selection predicates, are considered applied by the BindJoin plan, even if some of them are not explicitly reflected in selections.

Choosing between BindAccess and BindAccess^0 With the notations above, if \( p \) contains, among its leaf operators, a Scan or a BindAccess plan to \( R \), then in the BindJoin plan constructed, replace \( q \) by \( BA^0(R(\bar{U}^W V^f)) \). Otherwise, we use \( q \) as it is, i.e. BindAccess including duplicates. This way, we enforce the desired semantics of accessing a table several times, as described in section 3.3.2.

When combining \( p \) and \( q \), the algorithm joinProds generates between \( serv_Q \) and \( serv_Q + 2 \) distinct plans, where \( serv_Q \) is the number of LeSelect servers involved in the query \( Q \). These plans differ only in the choice of the top operator location. We are sure to generate \( serv_Q \) plans, since the join / Cartesian product can be executed by any server; we might generate one or two extra plans, if \( p \) and/or \( q \) are executed by wrappers, which are capable of performing the top operator.

As soon as a join plan is generated, we add all the selections that become applicable after the join. The resulting plans are pruned as follows. We compare two plans \( p_1 \) and \( p_2 \), located at \( loc_1 \) and respectively \( loc_2 \) only if:
• $p_1$ and $p_2$ have the same binding pattern
• $p_1$ and $p_2$ apply the same set of selection predicates

If, under these conditions, $\text{cost}(p_1) + \text{size}(p_1)/\text{BW}_{\text{loc}_1\rightarrow\text{loc}_2} < \text{cost}(p_2)$, then we prune $p_1$. Otherwise, we keep both plans. This simple pruning condition is mentioned in [66].

**Implementation**

As all the rest of LeSelect, the query optimization algorithm is implemented in Java (8,000 lines of Java code). The whole LeSelect server implementation consists of 36,000 lines of code (45,000 if we count also the code for pre-configured data and program wrappers that come with every LeSelect server).

### 3.8 Related work

#### 3.8.1 Query optimization with binding patterns

Several recent query processing applications share the fact that it is not always possible to perform complete scans on the data. One such application is optimization in the presence of foreign and table functions [19, 22, 97]. In most cases, such functions require a set of given inputs in order to return a set of tuples. The search space we described in section 3.5 is much larger than the search space for a query with user-defined predicates; indeed, restricting the search to the space described in [23] amounts to using only closed query execution plans. The same restricted search space is considered by the query optimization algorithms of LDL [25] and Garlic [54].

One of the reasons for using restricted access data collections is the existence of a mismatch between the logical model of the data (which is used for posing queries) and the actual physical storage of the data (used for evaluating queries). In the presence of this mismatch, the query optimizer requires an explicit description of the available access patterns to the data. GMAPs [114, 115] have been proposed as a method for describing storage structures. Using GMAPs, one can describe storage structures in which the stored data is a result of projections, selections and joins on the logical schema of the data. For example, using GMAPs it is possible to describe secondary indexes, path indexes and field replication. Ignoring the data access restrictions, the storage descriptions expressed by GMAPs are more powerful than those expressed by (parameterized) projection views, as is the case with binding patterns.

GMAPs and binding patterns characterize disjoint sets of mismatches between the logical and physical views of the data. To combine the two families of mismatches, we need to extend the algorithm in [115] in several ways. First, as we did in section 3.4, we need to consider annotated query execution plans. Note that in [115] the execution plans manipulate GMAPs (which can be thought of as materialized views) rather than database relations. Second, the join enumeration algorithm needs to consider plans of larger size. It follows from [93] that in
the combined context of binding patterns and GMAPs the query execution plan may require more joins than the number of relations in the query, and hence a relation mentioned once in the query may appear in more than one leaf in the query execution plan. In particular, given a query with $n$ subgoals and $m$ variables, the optimization algorithm needs to consider all the plans with $n + m - 1$ joins in order to be guaranteed to find a plan if one exists.

The problem of building query execution plans when only limited access patterns are available has been considered in work on data integration [83, 93, 69]. However, in these works they addressed the question of whether there exists some ordering of accesses to the data sources such that an answer to the query can be obtained. The question of finding an optimal order was not considered.

Stocker and Kossmann [109] propose iterative dynamic programming, as an alternative method for dealing with unpredictably larger search spaces (which may be due to causes other than binding patterns). In contrast to our method, they begin with pure dynamic programming, and adapt it later if the search space proves to be too large. The challenge in their approach is to detect when to switch from pure dynamic programming to their more directed method.

Constructing an optimal QEP for a query over sources with binding patterns combines two problems: (i) a simple instance of a query rewriting problem, if we consider restricted data sources to be views, and (ii) a particular type of query optimization problem, in which we may consider a larger search space than the one considered in System-R style optimizers, as shown in section 3.5. Traditionally, these two classes of problems - query rewriting and query optimization - have been studied in isolation. However, a recent study [4] combines the two problems, and provides an algorithm that searches for the optimal equivalent rewriting of a conjunctive query, where the optimality is evaluated in terms of the following simple cost models. First, the cost of a rewriting is computed as its number of subgoals. Then, a more refined cost model estimates the cost of a rewriting by summing the sizes of the intermediate results in a left-deep join tree constructed from the goals of the rewriting.

There are several differences between the context of this work and ours. First, they use Datalog semantics and the universal relation assumption; under bag semantics, there are fewer possible rewritings, and the problem is somehow easier. Second, the view definitions they use are general conjunctive queries, significantly more complex than select-project queries; however, they do not consider access restrictions. Finally, the cost model they use does not capture the bushy QEPs that may be constructed from a given rewriting, while, as we explained, we search the space of bushy trees.

### 3.8.2 Query optimization algorithms for distributed query processing

A significant body of research addressed the problem of query optimization for distributed query processing architecture. There are two major classes of applications: distributed databases, and data integration systems. The difference among these two contexts resides in the fact that in a typical data integration systems, data owners have limited query processing
capabilities, while in a distributed database system, all participating sites are full-fledged DBMSs. LeSelect shares common features with both these classes of systems. On one hand, resource owners keep their autonomy, the resource publication process is flexible and dynamic, and wrappers may have only very limited query processing capabilities; on the other hand, query execution and optimization modules are installed on all participating query sites, guaranteeing that we can always locate a given operation on a given processor. The comparisons between the existing distributed query optimization algorithms and the algorithm used in LeSelect have to be considered in the perspective of these hybrid features.

**System R** A distributed query optimization algorithm based on dynamic programming was proposed in System R* [102], the distributed extension of System R. Query optimization is performed by a master site, and its decision are followed by the other participating sites. First, the master site decides of a join tree, taking into account data statistics but ignoring transfer costs. Then, for a given join ordering, the master site selects the sites of intermediate join results. The other participating sites have the choice of the local ordering of the operations they perform, and of the join method. The query optimization algorithm in LeSelect is quite close to System R*, except for two features. First, LeSelect explores a larger search space, since localizations of operators are chosen for all join trees, not only for one. Second, in LeSelect we divide query processing not only among servers, but also among wrappers, within the limits of their query processing capabilities.

**IRO-DB** The IRO-DB federated database system integrates object-oriented data sources; on top of the data sources, a local layer is in charge of adapting the data sources to the ODMG standard, thus, the source wrappers support OQL. We mention IRO-DB for its innovative technique of *calibration* in order to obtain an accurate cost model [45]. First, cost formulas for several operators are established. A tuning application is then run on each site, and its performances measured, in order to obtain values for all the parameters of the generic cost formulas. It would be interesting to adopt this approach in LeSelect, in order to obtain more detailed cost parameters concerning the execution costs on each site.

A development of the calibration approach was proposed in [85]. Cost estimates are computed by default for all data wrappers, but the wrapper developers may overwrite these estimates if they feel they are inaccurate, providing thus more precise values.

**Garlic** Cost-based query optimization in the context of sources with restricted capabilities has also been studied in the Garlic system [100, 54]. Wrapper query processing capabilities in Garlic are expressed in terms of generic *STrategy Alternative Rules*, or STARs [75]. One such rule produces, for a given set of *plan properties*, the set of plans that the wrapper is able to execute, having *some* of the desired properties. The most important properties of a plan are: the tables it involves, the predicates it has enforced, and the attributes it projects. The plans returned by the wrapper can lack some of the required plan properties; the missing ones are returned to the mediator together with the plans. Thus, the mediator understands what operations are still missing, and may perform them. Garlic’s optimizer
also uses a dynamic programming algorithm; LeSelect’s optimizer explores the same search space as Garlic. There are significant differences in the resource modeling process between Garlic and LeSelect, as we explained in section 2.5. With respect to query optimization, the most important difference is that the query optimizer of LeSelect knows in advance the query processing capabilities of each wrapper, while in Garlic, the optimizer learns these capabilities by trying, as explained above. Finally, we note that a very comprehensive cost model for a wrapper-mediator architecture has been proposed in the context of the Garlic project in [99].

**Disco** The distributed query optimization algorithm used in the DISCO project [113] proceeds in two steps. First, a *preliminary logical plan* is constructed, neglecting for the time being the localization of each operator, and data transfer costs. In a second phase, the operators are assigned to wrappers or to the mediator. The goal of the second phase is to push as much work to the wrappers as possible, given their limited query processing capabilities. This two-steps approach cannot, however, guarantee optimality, since the preliminary logical plan chose by the first step may not be the one yielding the best distributed execution.

**ObjectGlobe** The ObjectGlobe data integration system [14] distinguishes three classes of actors that may be involved in query execution: data providers, function providers, and cycle providers, where this last category contains the sites that are available for executing query operators. It is assumed that the data and function providers specify their *preferred* cycle providers for processing their data or executing their functions. Thus, at query parsing time, the preferred cycle providers of all data and functions involved in the query is constructed; only these providers may contribute to process the query. If we consider that for every data and function published in LeSelect, the publishing site is the preferred site, the sites involved in executing a query in ObjectGlobe and LeSelect are the same; however, the ObjectGlobe policy is more general. It should be noted, however, that if no preferred sites are specified by the data and function providers, a query may be rejected by ObjectGlobe. LeSelect, on the contrary, makes the assumptions that the publishing sites will always cooperate.

An interesting feature of ObjectGlobe is taking into account data confidentiality when optimizing a query. Following the specification of the preferred cycle providers, as well as security and statistic information, a compatibility matrix is attached to each QEP during query optimization, specifying which cycle providers may execute which operator on the result of the QEP.

**Query optimization techniques in the presence of expensive functions** In the restricted search space considered by LeSelect’s optimizer, a legitimate question is whether the solution to the “expensive predicate” ordering problem, proposed in [22], could be used to order the BindJoin operations. Indeed, if BindJoin and BindAccess operators can only occur one next to the other, then this pair could have been modeled as a costly selection operator. An expensive function is more general than an expensive predicate - for example, a BindJoin-BindAccess pair can greatly increase the number of tuples and the size of each tuples, instead
of just eliminating tuples as a predicate does. But the obstacle in adopting the predicate ranking technique in the case of LeSelect comes from the distribution of restricted resources in LeSelect. If we take into consideration data transfer costs, then the predicate ranking method does no longer apply. Therefore, we model restricted resources as tables, and we access them via binary BindJoin operators; the resulting search space is exponential in the number of unrestricted and restricted access tables, while the predicate ranking technique is polynomial in the number of expensive predicates. However, as demonstrated by the complexity results shown in table 3.1, the presence of restricted binding patterns acts as a significant reducer of the search space.

When discussing the localization of unary operators in LeSelect, we used the terms data-reducing and data-inflating operator; this distinction has been done in the Mocha project [98]. They study the execution of costly operators (e.g. aggregates or user-defined expensive functions), and face the alternatives of shipping the function to the data site, or the data to the function site. They propose to ship the data if the operator is data-inflating, and ship the operator otherwise. In contrast, in our architecture, we cannot ship programs; we decide the localization of an operator based on its data reduction factor only for simple unary operators. Also, they make this decision as a heuristic; using the triangle rule, we shown it to be sound. Finally, a query in Mocha is executed by one mediator, and several data access/function providers; in contrast, we involve several LeSelect servers, achieving thus a better work distribution.

Pushing selections and projections to diminish data transfers is a common heuristic when executing distributed queries. It is not a good idea to push selections only in two cases: when the selection uses an index that will no longer be available for a costly join method; and when applying the selection itself is very expensive. In LeSelect, an index is modeled as a restricted binding pattern: to access the data source following this restriction, we use a BindAccess and a BindJoin operator, and will explore many possible placements for this BindJoin. Thus, we do not sacrifice the optimized access method by pushing selections. Finally, the simple selection operator we use in LeSelect is relatively inexpensive, of the kind considered in [22] to have 0 cost. Therefore, we can safely push them down as much as possible.
Chapter 4

Efficient execution of queries over restricted resources

4.1 Introduction

We have described in the previous chapters how heterogeneous resources are published via wrappers in LeSelect, using the flexible model of tables with binding patterns. We also presented the distributed query optimization algorithm of LeSelect, that deals with the restrictions to resource access, and with the limited capabilities of the wrappers.

In this chapter, we turn to the query execution process in the LeSelect framework. In particular, we notice that in many applications, the most expensive computations to be performed for a given query involve either costly program calls, or blob transfers. Due to the modeling constraint that we imposed in section 2.2.4, a blob is retrieved by accessing its table following a restricted binding pattern. Thus, both expensive program calls and blob transfers are handled in LeSelect by the BindAccess and BindJoin operators. Therefore, the main focus of this chapter is on efficient algorithms for these operators.

The current version of the LeSelect system provides functional implementation of the BindJoin and BindAccess operators. However, these operators can be improved in several ways. First, the information passing between the BindJoin and BindAccess operators is handled in a quite ad-hoc manner, making the BindJoin not compliant to the general iterator interface. Second, the BindJoin operator makes a call to the restricted resource for every tuple it receives from its left child. In the case of very expensive resource access, involving significant computation and/or blob transfers, the current BindJoin is quite inefficient. Third, even if from the metadata exported by table wrappers it can be inferred which attribute of a table are blobs, no special measures are taken to ensure the transfer of such large data items is reduced as much as possible. Reducing data transfers is always interesting in a network environment; however, it is of particular importance when blobs are involved. Finally, many opportunities for intra- and inter-operator parallelism go unexploited by the current version of LeSelect. By the implementation of the physical relational operators, query execution is sequential, and therefore does not take advantage of the inherent intra-site parallelism. Also,
the BindJoin operator only makes one access to its restricted resource at a given time, even if in some cases, that resource might support several parallel accesses.

In this chapter, we propose both algorithmic and architectural solutions to the shortcomings listed above.

We present an improved design of the physical BindJoin algorithm, including an internal cache that avoids useless program calls and data transfers; we also re-designed the BindAccess operator. Our algorithms distribute well over distant sites, follow the general iterator interface, and, under some conditions, use the presence of duplicates in their inputs to provide a steady tuple output rate in the early stages of the execution. Also, as an extra way to speed up the BindJoin execution, we design our operator so that it can monitor several parallel calls to the restricted resource (intra-operator parallelism), and provide a simple adaptive algorithm to determine the optimal number of parallel calls.

To improve on the current architecture, we propose introducing in every LeSelect server a BlobManager module that is specially designed to handle blobs, whether they are published, copied or produced on that server. This module is needed for our BindJoin algorithms to handle efficiently blob transfers. Also, to take advantage of the BindJoin algorithms that we propose, we recommend in this chapter to modify the paradigm of communication in LeSelect, by introducing Exchange operators between two operators, as soon as they do not run on the same LeSelect servers. Exchange operators were originally proposed by Graefe [50] as a means to encapsulate intra-operator parallelism in the Volcano parallel query processor; in our context, they allow the de-synchronization of operators running on distinct sites, and thus their parallelization.

With respect to LeSelect, therefore, the purpose of this chapter is: to propose an efficient operator for what we have perceived as being the performance bottleneck in most encountered queries, and to make all necessary modifications to the general query execution framework in order to fully exploit parallelism in general, and the good properties of the operators we proposed in particular. The implementation of the operator and infrastructure changes proposed in this chapter is currently on-going. The propositions we make in this chapter have been described in [77].

The chapter is organized as follows. In section 4.2, we introduce a sample query that illustrates the performance issues involved in executing a query involving expensive programs and blobs, we list the most relevant techniques proposed for solving these problems, and highlight their shortcomings. In section 4.3, we describe the proposed architectural enhancements of LeSelect. We then discuss work distribution among remote BindJoin and BindAccess, and present two cache-aware algorithms for implementing the BindJoin operator, in section 4.4. When describing these algorithms, we assume enough space is available for a perfect cache, avoiding any redundant computation. In section 4.5, we study the behavior of our BindJoin algorithm in the presence of space limitations. Unsurprisingly, such a situation leads to re-computation. We also show that in queries involving produced blobs (i.e. that result from the execution of a program), pipelined-parallel query execution may lead to deadlock or runtime errors. We provide safety conditions on the operators in a QEP involving cache-aware BindJoins, under which execution proceeds correctly even if there are space limitations. In
section 4.6, we show how we would need to modify the query optimization algorithm presented in the previous chapter, to make it consider inter- and intra-operator parallelism, as well as the presence of cache. Finally, in section 4.7 we compare our approach with other similar results.

4.2 Motivating example

To illustrate, consider a scientific application that involves two databases. On site $S_1$, satellite images have been processed into a map of the ozone cover of the French territory. On site $S_2$, a survey of the traffic in the same area resulted in a set of records corresponding to the days when traffic was particularly intense (e.g., holidays). On site $S_3$, an application OzoneLevels:img → \{level\} computes the set of distinct ozone density levels found in a given ozone cover image. Now, suppose that a user wants to retrieve on site $S_4$ the following information: for the days when the ozone levels were low, occurring just after days of heavy traffic, return the ozone cover image, its date, the correlated date of heavy traffic, and the ozone level values.

Consider the execution of this query in a distributed relational data integration engine (in our case, LeSelect). To publish resources needed for our example problem, we install a LeSelect server on each site; each resource is published via a wrapper, provided by the publisher. Data collections and programs are published as relational tables through these wrappers, following the methodology described in section 2.2, yielding the tables whose schemas are displayed in figure 4.1(a). We expressed the query in a SQL-like notation, prefixing data collections and the function by their sites.

The most expensive operations in the execution of such a distributed query are:

- function executions, since the per-tuple cost of a function execution may be significantly higher than the per-tuple cost of a relational operator;
- large image transfers on a wide area network.

As an illustration of the inefficient QEPs that may be generated in such cases, consider the admittedly simplistic QEP in figure 4.1(b). We have assigned a location to each node in the QEP, and circled together operations executed successively on the same location.

In this QEP, the Image-HighTraffic join is performed before the OzoneLevels selection. The reason is that since we assume the function is very costly, we use the join to reduce the number of images we send it. After the join, due to the join condition, one tuple from Image is paired with several tuples from HighTraffic. Sending this intermediary result from $S_1$ to $S_3$ may entail sending several times the same image, once per tuple it appears in. Furthermore, if OzoneLevels is invoked once per tuple received, it will be uselessly computed several times on the same image.

Considering these performance issues, it is easy to list the ideal characteristics of a QEP for our example query.
Figure 4.1: Sample query on distributed data and programs (a), basic QEP (b), QEP using semi-joins (c).

1. No costly function should be called twice with the same arguments.

2. No large object should be transferred twice within the same source and destination sites.

3. The total time spent in accessing restricted resources and transferring large objects should be minimal.

4. The function calls and the large object transfers should be parallelized as much as possible; the execution should proceed in pipeline (inter-operator parallelism).

5. When possible, within a single operator or data transfer step, several parallel function calls or data transfers should be done in parallel (intra-operator parallelism).

The last two items in this list do not derive from a limitation of our simple QEP; rather, they are reasonable principles that, in general, may lead to lower execution times and that we would like to see applied. Finally, whatever solution is used for a QEP to meet the above criteria, this solution should mix well with existing standards in distributed query optimization (dynamic programming-based) and execution; also, it should not lead to suboptimal choices in solving the remaining parts of the query (those that do not involve costly functions or large data transfers).

Several query execution or query optimization techniques previously proposed provide some of our desired characteristics. Caching the results of expensive methods has been studied in [62]; the authors compare hash- and sort-based techniques for avoiding useless function calls. In our context, sorting is not interesting since it is blocking, and thus contradicts our desirable property 4; therefore, hash-based caching solves issue 1 in our list, but it does not provide a solution for issue 2. Query optimization techniques for expensive functions have been proposed in [63, 61, 21, 23]; these techniques seem to provide a solution to our
items 2 and 3 above. However, as we explained in the previous chapter, the optimization
method proposed in [22] based on predicate ranking is not applicable in a distributed context.
Furthermore, predicate ranking assumes that expensive functions have a constant per-tuple
execution cost, which is false in the presence of caching, since the cost depends only on the
number of distinct values.

Another standard technique that aims at achieving our goals 1 to 3 consists of using
semi-joins, initially proposed in [11], as we did on our sample query in figure 4.1(c). In this
QEP, we start by a semi-join between Image and HighTraffic; only those Image tuples that
have at least a match in HighTraffic will be sent to $S_3$, and only on those tuples will the
function be called. Thus, we accomplished our goals 1 and 2. Now, once the ozone levels
have been computed, two things need to be done: (i) join again with Image and HighTraffic,
to get the full join result, and (ii) transfer exactly once the useful images from $S_1$ to $S_4$. To
solve (i), without transferring the images from $S_3$ to $S_2$ and to $S_4$, we project out the image
and retain only the image ID and ozone levels of those images that passed the selection;
then, we apply the two joins. To solve (ii), we apply a duplicate-free projection on the left
branch, denoted $\Pi^0_{\text{id}}$, and by a final join send the images to $S_4$. Note that (i) and (ii) could
not be performed one after another; therefore, we are forced to materialize the results of the
projection after OzoneLevels, thus the “Y”-shape QEP. (An alternative to materialization is
to execute the sub query twice, which is quite suboptimal.)

This semi-join solution has also several drawbacks. First, it has a significant execution
overhead, due to the repeated operations: for example, the Image-HighTraffic join is applied
twice. Also, it yields extra data transfers: HighTraffic has to be shipped twice to $S_1$. A recent
study on using semi-join motivates their interest by the presence of replicas: if HighTraffic
was replicated on $S_1$, there is no transfer overhead [110]. However, in a loosely coupled
integration system, data owners typically do not interact and do not maintain replicas.
Furthermore, the presence of semi-joins significantly complicates the optimization process [110];
we might not want to pay the price of a complicated optimizer that tries join and semi-join
combinations at all steps if we knew which are the few hot spots in a query (function calls
and large data transfers). Finally, the materialization in the “Y” point hinders the pipelined
execution that we aimed at.

To achieve inter-operator parallelism, which is our goal number 6, one would be tempted
to install on each site a parallel query processor, in the spirit of the Gamma [34] or Vol-
cano [52] parallel database machines. However, note that we do not need to parallelize the
complete query evaluation; we only want to exploit the parallel processing capabilities of the
published resources, when they are available. For example, if we knew that a batch of five
computers was available to run OzoneLevel, we would ideally issue at any time five parallel
function calls; this amounts to some parallelism within the selection shown in figure 4.1,
while we do not parallelize the remaining query execution.

A very similar problem is addressed in the WSQ/DSQ project: the goal is to combine
querying Web sources, modeled as virtual tables, and querying database sources [48]. The
solution relies on parallelizing a single operator, that is responsible for making calls to Web
sources. The calls are issued asynchronously, and a single supervisor operator matches call
arguments with their results, for all web calls made in a query. We may thus generalize the WSQ/DSQ solution, replacing web calls with calls to arbitrary functions; however, the single supervisor operator is not practical in a de-centralized, many-mediators architecture like LeSelect has. Indeed, consider a query in which we call both the OzoneLevels function on site $S_3$, and an ImageConvert program on site $S_2$ that converts the resulting images into a different format. One supervisor module would generate communications between the supervisor and the function site for any call that is made. Also, supervisors are inserted in a QEP after the plan was constructed by a query optimizer, making a post-processing step necessary.

The rest of this chapter provides the elements of the solution we propose to execute efficiently queries over expensive restricted resources. The guidelines of our solution are the following:

- From an operator point of view, we target all optimizations to the BindJoin-BindAccess operator pairs (cache and parallelism). It was also our goal to be able to distribute these operators on different sites with good performance.

- Architecturally, we suggest adding to each server a module dedicated to blob handling, and using Exchange operators to allow inter-site parallelism (and also, parallelism between computations and data transfers).

In the proposed framework, query optimization using BindJoins can proceed following a dynamic programming algorithm, enhanced with the usage of binding patterns.

4.3 LeSelect infrastructure for querying restricted resources

In this section, we describe the proposed query processing framework of LeSelect. We recall the general iterator model followed by all operators in LeSelect, and explain the usage of the Exchange operator for inter-site transfers of small data items. Next, we demonstrate the use of the logical BindJoin and BindAccess operators for queries involving expensive functions and/or blobs. Finally, we present the design of a Blob Manager that we use to efficiently handle blob transfers between sites.

Outline of suggested server architecture Figure 4.2 presents the outline of a LeSelect server, enhanced with the BlobManager module, and using a communication module that relies on distributed Exchange operators. Solid lines represent data flow, while dashed lines trace statistic or control flows. The BlobManager running on every LeSelect server is responsible of all the blob transfers that the execution of a query might entail. To that purpose, the BlobManager interacts with the physical operators implemented by the query execution engine.

All transfers of blob objects between two servers that the execution might entail are performed by those server’s Blob Managers; all other types of data are transferred via the two Distributed Communication Modules.
4.3.1 Iterators and the Exchange operator

In LeSelect, each operator of a QEP is implemented as an iterator, following [51]. Iterators are self-scheduling data processing units; their API consists of an initialization open() call, a next() method producing one tuple at a time, and a close() method to release resources and terminate. To answer a next() call, the top-level operator in a QEP may call the next() methods of one or more of its inputs, depending on its implementation. The basic iterator model is synchronous; if an operator p is the parent of an operator r, when p issues a next() call to r, p’s execution blocks until r returns a tuple.

The Exchange iterator was introduced to decouple the execution of several operators, allowing them to work in parallel [50]. Figure 4.3 shows an Exchange operator X inserted between an operator p and its child r. On X.open(), the communication queue Q is created, and X is split into two independent processes. Tp runs operator p and the Xp consumer part of the Exchange, while Tr executes Xr, the producer part of the Exchange. Tr is no longer driven by the upper part of the plan; it runs independently and iteratively issues next() calls to r until an eof is reached. An Exchange operator inserted between two operators r and p running on the same site acts as a synchronization buffer, absorbing bursty output from r and providing it to p at p’s required pace.

If operators p and r run on remote sites, Q is split into two queues Qp and Qr, each on the site of one operator, with Qr’s contents being transferred into Qp by some data communication mechanism; Tr and Tp run in parallel. In this case, the size of Q allows us to control the de-synchronization of Tr and Tp. The synchronous iterator model corresponds to a queue of size 0, where a tuple is processed by p immediately after being processed by r. By increasing Q’s size, we allow a slack among the tuples processed by r and p. The size of Q is a parameter of the Exchange operator, set at creation time by the optimizer.

In LeSelect, we propose to implement distributed Exchange operators by adding to each LeSelect server a distributed communication module (DCM) that creates the communication queues Qp and Qr, as in figure 4.3(c). Within each DCM, communication daemons are in charge of transferring tuples among the queues Qr and Qp. Thus, from now on, we assume
all transfers of small-size data between two LeSelect servers are done via the distributed Exchange operators implemented by the DCM.

4.3.2 Using the BindJoin and BindAccess operators for restricted resources

The access to a resource according to a restricted binding pattern is performed using a pair of BindJoin and BindAccess logical operators (defined in section 3.3.1, respectively in section 3.3.2). In this section, we illustrate the usage of these operators in queries involving expensive functions and retrieval of blobs.

Figure 4.4 shows two sample queries involving data and programs, and possible corresponding QEPs. The query in Figure 4.4(a) chains two BindJoin operators. The first BindJoin fetches a blob from the table $T$, using an equi-join condition $R.a = T.bID$, while the second BindJoin calls a function $f$ on this blob, where $f$ returns a small-size result. In this case, one access to $T$ and one access to $f$ are enough to retrieve all the necessary data, and we use regular BindAccess operators (that return all matching tuples, including duplicates).
Function $g$ in figure 4.4(b) takes as input a small attribute and returns a blob and an integer value $\text{intRes}$; its binding pattern set is \{$(\text{in}^{\text{outBlobID}^{\text{intRes}}}^{\text{out}}), g(\text{outBlobID}^{\text{outBlob}})$\}. A first BindJoin fetches the blob from $T$, while the second one calls the function $g$. The selection on $\text{intRes}$ may eliminate some of the blobs produced; the remaining ones are needed in the output. The last BindJoin-BindAccess\(^0\) pair fetches these blobs in the result. In order to correctly compute the answer to the query, we use a BindAccess\(^0\) when we access $t$ a second time. Retrieving a produced blob is a typical case when a BindAccess\(^0\) needs to be inserted. To understand why, consider what happens if instead, we use a regular BindAccess. If one pair $(\text{blob}_1, \text{blob}_2)$ appears several times in the table corresponding to $g$ (e.g. because the same blob was produced several times), when accessing $g$ with the $\text{blob}_1$ as binding, we get not one, but several result tuples, containing the same blob, which leads to erroneous duplicates in the result.

In LeSelect, the most important components of a query’s processing cost, namely program invocation and blob transfer, are performed using the BindJoin and BindAccess operators. Therefore, these operators are a good target for incorporating the optimizations described in section 4.2, without requiring any change to the remaining operators.

### 4.3.3 BlobManager for storing and transferring blobs

We now describe the BlobManager (BM) module. The main role of the BM of any site $S$ is to provide a uniform interface to any blob for all operators running on $S$, whether the blob has been published or produced by a program, on site $S$ or elsewhere. To achieve this, the BM is in charge of storing and managing the blobs produced by a program running on $S$, or transferred from other sites to $S$ in order to be consumed by programs running on $S$. Additionally, the BM of a site optimizes the transfer of blobs, by choosing where to transfer a blob from, if several copies the blob exist.

When a BindJoin operator running on $S_1$ indirectly invokes a program that produces a new blob, the BindJoin issues a $\text{store}$ call to the local BM to require the storage of this blob. The parameters of the $\text{store}$ call are the $\text{blobID}$ of the blob and the $id$ of the query within which the BindJoin is executed. The BM allocates a local blobID to any blob it stores which uniquely identifies a blob in BM’s site.

To access a blob, an operator running within a query $Q$ at site $S$ proceeds in two steps. First, the operator issues a $\text{getLocalID(blobID,Q)}$ call to the BM of $S$, where $\text{blobID}$ is the system-wide identifier of the blob. The BM obtains a copy of the required blob (if it was not already available on $S$), and returns a local blobID to the operator requiring the blob. In a second step, the operator uses the local blobID to read the blob. This way, the BM acts as an intermediary, hiding the original storage of a blob to the operator that requires it, as well as the possible communications involved in retrieving it.

As an optimization, we might allow copied or produced blobs to persist in the BM of $S$ not only for the needs of the BindJoin which has issued the corresponding $\text{getLocalID}$ or $\text{store}$ call, but also for other BindJoins, either part of $Q$’s QEP, or executed in a different query. Suppose that a blob has been copied from $S_1$ to $S_2$, and is needed later on $S_2$. When a BindJoin on $S_2$ issues a $\text{getLocalID}$ call to the BM on $S_2$, this BM has the choice of
transferring the blob to S₂ either from S or S₁. This choice may be interesting for several reasons: if getting a blob from its original repository costs money, and if site S allows us to exploit its copy for free, the benefit is obvious; also, at runtime, network bandwidth estimates may suggest that S → S₂ transfers are faster than S₁ → S₂, or that the transfers should be split in two, half from the original source and half from the temporary copy of the blob. To do this, we parameterize a BindJoin requiring a blob with the list of BMs in which the blob has been stored for the purposes of the current query. The BindJoin provides this list to its local BM, which may operate the choice.

**Lifespan of blobs in the Blob Manager** In general, the choice of whether and how long will the BM of S store a blob b it has acquired from another site for the needs of the query Q is determined statically by the query optimizer. This decision is guided by metadata published by the site S, specifying its storage policy. Several such policies may be envisioned, ranging from fully egotistic (S keeps a blob only as long as it uses the blob), to fully cooperative in single-query mode (all sites involved in a query Q keep any blob they copy until the end of Q) to fully cooperative in multi-query mode (blobs transferred for the needs of a query are kept to profit to other queries, too). The choice of a particular policy for a site S is mainly determined by the available space at S, but may also depend on other considerations, e.g., service or data subscriptions among users on different sites, data confidentiality, query priorities etc.

In the next section, we consider that all sites store all blobs they use to answer Q until the execution of Q ends. Note that all blob manipulations are done by BindJoin operators; following this line, the close method of a BindJoin executed within the QEP of Q sends a *mayDispose(Q)* message to the local BM, informing it that the blobs stored for the usage of Q are no longer needed. Since the close call is propagated from the top of the QEP, we know that all operators above the BindJoin have finished processing tuples when a BindJoin receives the close. In section 4.5, we remove this assumption and study the consequences of space limitations on the execution of queries involving blobs.

We make here the following remarks. First, keeping b until the end of Q’s execution is in some cases more than what is needed. For example, if a blob is produced by a BindJoin on a site S and is never used by another operator up in the query plan, the blob does not need to be stored on S after the BindJoin has produced it. For simplicity, we consider that blobs are kept until the end of Q’s execution, and make the assumption that enough storage space is available to this purpose. Second, while sharing transferred blobs among several queries is interesting and feasible, we do not investigate it in this thesis; this is an interesting direction for future work.

### 4.4 Algorithms for BindJoin and BindAccess

We now define algorithms for BindJoin and BindAccess that can take advantage of caching, independent parallelism, and adaptive intra-operator parallelism to improve the query response time.
Cache-based optimization The main justification for caching is to avoid redundant access to a restricted resource, when the left input of a BindJoin operator provides duplicate arguments. This can significantly improve query response time in the presence of duplicates, as long as the tuple input rate in the BindJoin operator is faster than the processing speed of the BindAccess. Also, under the same condition, the presence of duplicates allows us to significantly improve the BindJoin’s tuple output rate early in the execution.

4.4.1 Physical operators for BindJoin and BindAccess

The presence of a cache requires specific design decisions to suit our distributed context. The first decision concerns the cache localization: to reduce data transfer, it has to reside in the same site as the BindJoin. Indeed, suppose that the BindJoin and its left child operator run on a site $S_1$, while the BindAccess runs on another site $S_2$. Obviously, by caching results on $S_1$ we avoid sending to $S_2$ arguments for which the result was already computed. This decision impacts on the physical architecture of the BindAccess operator. Since the BindJoin controls the cache, it is the BindJoin that extracts tuples from its left child operator, while the BindAccess has to obtain binding arguments from these tuples, through the BindJoin. As a result, we have to provide the BindAccess with the capability of extracting argument data from a (possibly remote) data structure filled by the BindJoin. Therefore, we decompose the BindAccess into two physical operators named ComputeResult and GetBinding; ComputeResult will run on $S_2$, GetBinding on $S_1$. Following the general principles of the distributed iterator model, the arguments are transferred between these operators through a distributed Exchange.

Notations To explain the functioning of the physical operators for BindAccess and BindJoin, we use the following notation. The left child of the BindJoin is denoted as $r$, and its parent operator as $p$. In general, the tuples coming from $r$ are of the form $r(\overline{x}_i, \overline{z}_i)$; without loss of generality, let us assume that both $\overline{x}_i$ and $\overline{z}_i$ consist of a single attribute: $x_i$ is a binding for the restricted resource, and $z_i$ denotes the remaining attribute(s). The results of a resource access with the binding argument $x_i$ is a set of $(x_i, y_{i,j})$ tuples, where each $y_{i,j}$ represents one result returned for the arguments value tuple $x_i$. The outline of a physical QEP for a BindJoin-BindAccess pair is given in figure 4.5(a).

4.4.2 Algorithms for the BindAccess operator

In figure 4.5(b), we illustrate the functioning of the ComputeResult operator, whose role is to encapsulate the access to the restricted resource in an iterator envelope. Let us first consider the case when ComputeResult runs as a component of a regular BindAccess operator.

On a next() call issued by the BindJoin, ComputeResult has to return a result tuple. To achieve that, it needs to (1) obtain a binding argument and (2) invoke the callResource method provided by the restricted resource publisher, as explained in Section 2.4. To obtain the argument, ComputeResult issues a next() command to the GetBinding operator. This
Figure 4.5: Physical operators for BindJoin and BindAccess (a); the ComputeResult operator (b).

operator, in turn, extracts the argument attributes from one of the tuples accumulated in the data structure by the BindJoin.

If the call to the restricted resource returns several tuples, the ComputeResult is in charge of managing those tuples, in order to return them one at a time, transparently to the BindJoin. In figure 4.5(b), ComputeResult simply returns a tuple resulted from a previous computation to answer the second next() call. Furthermore, for a given argument $x_i$, ComputeResult will return first all resulting tuples, and then a special end-of-call tuple, informing the BindJoin that there are no more results to be obtained for the $x_i$ argument value.

Now, assume we want to implement a BindAccess$^0$ operator. Remember that a BindAccess$^0$ on a table $T(X^0 \rightarrow Y^f)$ is used only when the functional dependency $X \rightarrow Y$ holds. Therefore, for a given argument $x_i$, callResource will return one or several identical $y_i$ values. To implement BindAccess$^0$ instead of a regular BindAccess, the ComputeResult operator only needs to discard all but the first tuple returned by callResource.

We interject at this point the following remarks. First, ComputeResult is the only physical operator that a wrapper must implement; GetBindings and BindJoin run within the LeSelect server. Second, a ComputeResult iterator is quite generic, and a pre-defined wrapper implementing it can easily be devised using the callResource interface provided by the publisher.

Before explaining our cache-aware BindJoin algorithms, let us consider the kinds of data that must be stored inside this operator.

**Internal data structures for the BindJoin operator** A physical BindJoin operator using a cache needs to contain the following data structures:

- a cache of argument-result pairs $(x_i, y_{i,j})$ for all tuples from $r$ that have already been processed by ComputeResult;
- a cache of tuples received from $r$ but whose processing is not finished yet;
an internal output tuple queue $Q_1$. When the BindJoin encounters an $(x_i, z_i)$ tuple for which several $(x_i, y_{i,j})$ are in the cache, it has several $(x_i, y_{i,j}, z_i)$ tuples to output. These tuples are inserted into $Q_1$, from where the BindJoin transparently returns them one at a time on $next()$ calls.

### 4.4.3 The CacheFIFO BindJoin algorithm

We start by presenting a simple synchronous BindJoin algorithm using cache. The executions of BindJoin, ComputeResult and GetBinding are serialized: each $next()$ call is blocking for the caller. In CacheFIFO, at any moment, the data structure contains at most one $(x_i, z_i)$ tuple, which has been received from $r$ and whose processing is not yet finished; we call this tuple the current tuple. The algorithm is depicted in figure 4.6. When execution begins, there is no current tuple, i.e. $t_c = null$, the cache and the result queue $Q_1$ are both empty.

In response to a $next()$ call, the BindJoin attempts to return a result tuple from $Q_1$. If $Q_1$ is empty, the algorithm loops over the steps shown in figure 4.6, with the goal of obtaining a result tuple, computed from the current tuple $t_c$, to be inserted in $Q_1$.

If we already have a current tuple $t_c$, then CacheFIFO tries to obtain result tuples corresponding to $t_c$ by calling the $next()$ method of ComputeResult. If ComputeResult returns a tuple of the form $(x_i, y_{i,j})$, the algorithm inserts it into the cache and in the output queue. If ComputeResult returns an end-of-call mark, the current tuple is removed from the structure.

If there is no current tuple to be processed, CacheFIFO retrieves a new tuple $(x_i, z_i)$ from $r$ via an $r.next()$ call. Two cases may arise, depending on whether the $x_i$ value is in the cache or not. If $x_i$ is in the cache, the algorithm constructs and inserts into $Q_1$ one $(x_i, y_{i,j}, z_i)$
tuple for each \((x_i, y_{i,j})\) tuple found in the cache, and outputs the first such \((x_i, y_{i,j}, z_i)\) tuple in response to \texttt{next}(). Otherwise, if \(x_i\) is not in cache, \((x_i, z_i)\) becomes the current tuple, and the loop resumes.

The \texttt{next}() call exits when some tuple is available in \(Q_1\).

Output rate of the CacheFIFO algorithm  When the execution of this simple BindJoin algorithm starts, the cache is empty. The processing of most of the tuples received from \(r\) will entail a call to ComputeResult; thus, the tuple output rate is close to the processing rate of this operator. As execution continues, the cache is progressively filled, and tuples are output directly from the cache. Since we assume that cache lookup time is negligible compared to ComputeResult’s processing time, the output rate towards the end of the execution is significantly higher than at the beginning. Such an uneven output rate is unavoidable in the presence of cache, but tuple bursts at the beginning of the execution are much better than a sudden burst at the end. (If the BindJoin is the top QEP operator, tuples are returned to the user faster; if its parent operator receives tuples fast, accelerating the early output of the BindJoin could propagate early tuples towards the output.) Because of this fact, we present a second algorithm that aims at a fast output rate in the early stages of the execution.

4.4.4 The CacheParallel BindJoin algorithm

If tuples can be obtained from \(r\) faster than the processing of ComputeResult, cache and duplicates provide three opportunities to improve the early output rate of the BindJoin,

1. First, for each \(t = (x_i, z_i)\) incoming tuple such that an \((x_i, y_{i,j})\) is already in the cache, \((x_i, y_{i,j}, z_i)\) may be output directly by the BindJoin, and in parallel with the on-going processing in ComputeResult. This improvement comes at the price of losing input order. Also, it requires the ability to temporarily store tuples within the BindJoin, since we cannot “directly” obtain from \(r\)’s output tuples for which the results are already in the cache.

2. Second, by allowing tuples coming from \(r\), for which the result was not already in the cache, to accumulate within the BindJoin’s data structure, we increase the chances that the BindJoin outputs several tuples together. Whenever the BindJoin receives an \((x_i, y_{i,j})\) from ComputeResult, it identifies all \((x_i, z_k)\) tuples in the data structure, and outputs all the \((x_i, y_{i,j}, z_k)\) at once.

3. Finally, GetBinding may choose which arguments to provide to ComputeResult in a way that maximizes the output tuple burst. To this purpose, GetBinding picks the most popular \(x_i\) value in the data structure, i.e., the one corresponding to the biggest tuple group in the data structure.

The cache buffer   The cornerstone of our algorithm that implements these three requirements is a cache buffer storing all the information needed at any given moment of the
execution (the cache and the tuples received from $r$ and not yet fully processed). The trick is to allow different entities to run in parallel synchronizing their actions through the cache buffer.

Formally, we specify a cache buffer by four access functions ($XSet$, $state$, $Y$, $Z$), and two rules governing tuple insertion, and respectively tuple removal from the cache buffer. We first present the access functions.

- At any moment, $XSet$ denotes the set of $x$ values that are currently present in the cache buffer.

- For any value $x$ among those returned by $XSet$, the $state(x)$ function returns the state of this value, which can be either done, running or waiting. An $x$ value is done if it has been fully processed, i.e., ComputeResult has returned to the BindJoin all the results for this value. The value is running if it has been chosen by GetBindings to be processed, but it is not yet done. Finally, a value of $argSet$ that is neither done nor running is waiting. GetBinding can only choose waiting values.

- For any $x_i$ element of $XSet$ such that $state(x_i)$ is done, the function $Y(x_i)$ returns the set of $(x_i, y_{i,j})$ result tuples obtained by accessing the restricted resource with the argument $x_i$.

- For any $x_i$ element of $XSet$, the function $Z(x_i)$ denotes the set of tuples of the form $(x_i, z_i)$ present in the cache buffer.

We now describe the conditions under which tuples are inserted into, or removed from, the cache buffer.

**Tuple insertion rule** An $(x_i, z_i)$ tuple obtained from $r$ can be inserted in the cache buffer only in one of the two following situations. If $x_i \notin XSet$, then this value has never been seen before in the BindJoin: the tuple is inserted, and $state(x_i)$ on the insertion of this tuple becomes waiting. Otherwise, if $x_i \in XSet$ but $state(x_i) \neq$ done, the tuple is also inserted in the cache buffer; this insertion does not influence the value of $state(x_i)$.

**Tuple removal rule** A tuple $t = (x_i, z_i)$ is removed from the cache buffer if the state of $x_i$ changes from running to done. This state change entails the following actions. First, for each $(x_i, z_i) \in Z(x_i)$ and for each $(x_i, y_{i,j}) \in Y(x_i)$, an $(x_i, y_{i,j}, z_i)$ tuple is sent to the output; second, all $(x_i, z_i) \in Z(x_i)$ are eliminated from the cache buffer.

To achieve a maximum of parallelism, we share the cache buffer management between three independent entities, running in parallel.

- The first entity, BindJoin$_1$ extracts tuples $t = (x_i, z_i)$ from $r$, and tries to insert $t$ into the cache buffer following the insertion rule. If the insertion is not possible, meaning that $x_i$ is in $XSet$, and its state is done, then for each element $y_{i,j} \in Y(x_i)$, BindJoin$_1$ sends a result tuple $(x_i, y_{i,j}, z_i)$ to the internal result queue $Q_1$. 

116
Figure 4.7: Architecture of the CacheParallel variant of BindJoin: local ComputeResult (a), remote ComputeResult (b), blob fetching (c).
- The second entity, BindJoin\textsubscript{2}, is in charge of extracting and handling the results produced by ComputeResult. The results are grouped by their \( x \) value. For a given \( x_i \) value, BindJoin\textsubscript{2} enters the corresponding \( y_{i,j} \) results one by one into the cache buffer, and \((x_i, y_{i,j}, z_i)\) tuples into Q\textsubscript{1}. When ComputeResult finally returns an end-of-call, BindJoin\textsubscript{2} changes the state of \( x_i \) from running to done and eliminates tuples from the cache buffer, following the tuple elimination rule.

- Finally, a third entity noted BindJoin\textsubscript{3} desynchronizes the BindJoin from its parent operator \( p \). It is BindJoin\textsubscript{3} that answers \texttt{next()} simply by returning the first tuple from Q\textsubscript{1}, when one is available.

Besides BindJoin\textsubscript{1} and BindJoin\textsubscript{2}, the cache buffer is also accessed and updated by GetBinding. When GetBinding chooses an \( x \) argument, it changes its state from waiting to running. As shown in figure 4.7(a), when the BindJoin and ComputeResult run on the same site, GetBinding, BindJoin\textsubscript{2}, and ComputeResult can run synchronously without degrading performances. When BindJoin and ComputeResult run on two different sites, GetBinding must be run in parallel with BindJoin\textsubscript{1} and BindJoin\textsubscript{2} (see figure 4.7(b)). Nevertheless, it is convenient to limit drastically the capacity of Exchange queues between GetBinding and ComputeResult. Indeed, allowing too many elements in the queues leads to choosing \( x \) values a long time before they are effectively processed, making the choice non optimal.

Figure 4.7(c) illustrates the usage of the algorithm for fetching blobs from remote sites. Remember that our goals were first never to transfer the same blob on the same path twice, and second to enable the retrieval of a blob from one or several sites where a copy exists. The first goal is trivially met by the cache-aware BindJoin: the \( x \) arguments in this case are global blobIDs, and the BindJoin sends only distinct values to ComputeResult. All operators run on the BindJoin site; the physical localization of the source blob is considered not at the operator level, but in the physical blob transfer layer as follows. ComputeResult issues \texttt{getLocalID} commands to the BM on site S\textsubscript{3}. As explained in section 4.3.3, this BM was informed, on BindJoin\texttt{open()} that a blob published on S\textsubscript{1} has already been copied, within the current query, on S\textsubscript{3}. Thus, it can decide, on a per-blob basis, using money and time cost considerations, where to fetch the blob from.

**Output rate of the CacheParallel algorithm** The algorithm improves the output rate early in the execution. Figure 4.8 illustrates the behavior of a QEP fragment consisting of two BindJoins: \( r \xrightarrow{r} BA(f(x^by^t)) \xrightarrow{r} BA(g(z^bt^t)) \). In this example, the per tuple costs of accessing restricted resources \( f \) and \( g \) are set to \( c_f = 5 \) and \( c_g = 3 \) respectively, and both resources return exactly one tuple per argument; different indices for \( x, y, z, t \) indicate different values. Arrows follow tuple flow among \( r, CR_f \) (the ComputeResult operator accessing \( f \)) and \( CR_g \); we assume all seven tuples can be extracted from \( r \) as fast as desired.

When the BindJoins are implemented by the CacheFIFO algorithm (see figure 4.8(a)), tuples from \( r \) are extracted only at the pace of the bottleneck restricted access (\( f \), then \( g \)). Figures 4.8(b) and (c) show two CacheParallel variants. Since the extraction of tuples from \( r \) is very fast, all seven tuples are accumulated within the first BindJoin before the execution
Figure 4.8: CacheFIFO (a) and CacheParallel: choose oldest value (b), choose most popular value (c).

starts. It can be seen that CacheParallel helps solve some synchronization problems (tends to eliminate g’s idle time in the first phase of the execution), and that unlike CacheFIFO, it produces tuple bursts early in execution. As figure 4.8 shows, choosing the most popular value provides a larger early output rate than choosing the oldest. But the biggest advantage of this choice policy is that it is a stable technique, capable to cope with various input orders, even if the duplicates come last, as in figure 4.8. Also, since in a chain of BindJ oins we allow each to choose its own processing order, the decisions of the first BindJoin do not imply a bad ordering for those that follow.

4.4.5 Experimental evaluation of the BindJoin algorithms

In this section, we evaluate the advantages of the CacheParallel BindJoin algorithm over CacheFIFO, in terms of early tuple output rate. We study a sample query where two functions f and g are applied on the columns of the table R(X, Z), where X and Z are integer attributes. We vary the data distributions in the X and Z columns, the order of tuples in R, and the tuple output rate of R. This last parameter is important, since CacheParallel can only accumulate tuples in its data structure if the tuple input rate is larger than the tuple consumption rate of CacheParallel; we will show how R’s output rate influences the performance of the CacheParallel BindJoin algorithm.

Experimental setting For the purpose of this section, we used simple Java iterator-based prototype implementations of the CacheFIFO and CacheParallel algorithms. We study the behavior of these operators in isolation, since their full implementation and integration in LeSelect is currently ongoing. To validate our algorithms, we simulate function execution by timed waits (i.e. the ComputeResult operator simulating the application of function f cannot answers next() calls more frequently than every c_f time units, where c_f is the cost of applying function f on one argument tuple); we took c_f = 10 time units and c_g = 100 time units. Also, we assume that f and g return a single tuple per call. We consider the following
Figure 4.9: Sample tuple output rates of CacheFIFO and CacheParallel, uniform data distribution.
query and corresponding logical QEPs:

select $f(R.X), g(R.Z)$ from $R$
scan($R$) $\triangleright$ BindAccess($f$) $\triangleright$ BindAccess($g$)

The difference between these query execution plans is the order of application of $f$ and $g$. In addition to the different orders on $f$ and $g$, we vary the implementation of the BindJoin algorithm: both are implemented either as CacheFIFO, or as a variant of CacheParallel (choose oldest value / choose most frequent value).

The relation $R(X, Z)$ used in our simulation consists of 10,000 tuples of synthetic data, following the distribution characteristics we are interested in; we vary the output rate of $R$ by the same mechanism of timed wait. Unless otherwise specified, we assume all tuples from $R$ are available before the execution of the expensive functions starts.

In all our measures, we plot the tuple output rate of the QEPs considered as a function of time.

**The effect of CacheParallel on the early tuple output rate** In figure 4.9, the data distributions of $R.X$ and $R.Z$ are uniform and independent. The difference between the top two graphs resides only in the distribution parameters.

In the graph at the top, the CacheFIFO output rate is dictated by $g$, until the moment corresponding to $t \approx 100,000$, when $g$ has finished evaluating all distinct $R.Z$ values; from this point on, tuples are output from the cache, at a steady rate. In contrast, the CacheParallel variants achieve a much better early output rate. Since $f$ runs ten times faster than $g$, it immediately outputs $(x, y, z)$ if $x$ is in the cache, and outputs tuple packets whenever possible. Thus, $g$’s data structure is filled, and it may choose the most frequent $z$ value for processing.

The advantages of CacheParallel are smaller in the center graph in figure 4.9. Since there are many $z$ values, the bottleneck in query execution is always $g$. However, the CacheParallel variants achieve some improvements by allowing parallel passage of duplicates and packets of tuples sent to the output. The slope of the CacheParallel “choose most popular” variant reflects the size of the tuple packets that $g$ is able to output (4, then 3, 2, 1); since $f$ is ten times faster, $g$ has at any point at least 10 tuples to choose from, but there are few very frequent values in $z$.

**Synchronization problems eliminated by CacheParallel** In the bottom graph in figure 4.9, the CacheFIFO BindJoin chain performs badly since $g$ is the bottleneck for a long time, keeping $f$ mostly idle; only when $g$ has processed all unique values of $Z$ does the output rate grow. In contrast, CacheParallel maintains a steady rate during the entire execution, and reduces the running time by 20% since $f$ is kept busy by the packets output from $g$. In the previous two graphs, $f$ and $g$ were well synchronized, and the total running time was the maximum of the running times of the two BindJoins considered in isolation; in this third graph, synchronization problems are eliminated and pipeline parallelism achieved by the CacheParallel’s optimizations.
The influence of data distribution  The graph in figure 4.10 shows very important gains of CacheParallel with respect to CacheFIFO. In this case, the integer $X$ and $Z$ columns follow a \textit{zipf} distribution, with $\alpha = 0.2$, when $X$ can take values between 0 and 7,000, while $Z$ takes values between 0 and 1,000. By picking the most popular values early, large tuple packets are sent very fast to the output by both BindJoins.

We explain the small difference between the two CacheParallel variants in figure 4.9 and 4.10. In the case of uniform distributions, in the absence of very popular values, picking the most popular value cannot improve a lot over picking the oldest one, since there are no big output bursts anyway. In the case of \textit{zipf} distributions, since the very popular values are scattered over the 10,000 tuples, the BindJoins are sure to process those frequent values quite early, even by picking the oldest value; afterwards, subsequent copies of popular values are output directly from the cache by both CacheParallel variants.

The effect of limited input rate  Until now, we assumed that tuples are output from $R$ arbitrarily fast; we now remove this assumption. At the top of figure 4.11, we evaluate the impact of a limited input rate on the CacheParallel output rate (the “choose most frequent”) variant. The data and function parameters are unchanged from the last graph in figure 4.10. We show two extra curves, corresponding to a tuple output rate from relation $R$ of 1 tuple every time unit, respectively one tuple every five time units. Limiting the tuple output rate from $R$ prevents $f$’s and $g$’s buffers for filling, and therefore limits the benefits that CacheParallel BindJoins may take of the presence of many duplicates.

When the curves follow a linear slope, all tuples have not been extracted from $R$ yet, and the input rate is the only limitation of the output rate. Note that the incoming tuple rate (e.g., 1/5) is in theory better than both function rates (1/10, and 1/100 respectively); however, CacheParallel is able to exploit the duplicates to produce tuples much faster than the “nominal” rate of the two BindJoins. When all tuples have been received (at $t = 10,000$
Figure 4.11: CacheParallel behavior in the presence of limited input rate (top) and bad data order (bottom).
or \( t = 50,000 \), the curves join the one without input rate limitations. We note that even if only two tuples can be received during one execution of \( f \), the CacheParallel output rate is still much better than CacheFIFO.

**The effect of the input data order**  The graph at the bottom of figure 4.11 compares the effect of the incoming data order on the tuple output rate from several BindJoin algorithms. The function parameters are the same as in figure 4.10; the most frequent value, for both \( X \) and \( Z \) attributes, is 0. In the curve labeled “worst-case order”, we sorted the tuple set in the increasing order of value frequencies, first on \( X \), then on \( Z \). The first tuples contain mostly infrequent, singleton values, both for \( X \) and for \( Z \); the last tuples contain the most frequent values in both columns. We show a single curve for CacheParallel choosing the most frequent value, labeled “any order”, since, not surprisingly, by choosing among the whole accumulated tuple set, this algorithm is not affected by data order. The CacheParallel algorithm choosing the oldest value is strongly affected by the disadvantageous data order, since we made the oldest values the most infrequent ones; it behaves almost as badly as the CacheFIFO algorithm with the worst-case order. The small difference in the total running time between the worst-case and random order is due to a slight synchronization problem among the two BindJoins.

From these experiments, we draw the following conclusions. First, for an arbitrary data order, the most important advantage of CacheParallel is its ability of processing *in parallel* tuples for which results already exist in the cache, and tuples that necessitate computation. Second, the CacheParallel variant choosing the most popular value is robust in the presence of bad data order, while the variant choosing the oldest value behaves badly in such situations. Finally, CacheParallel adapts gracefully to a limited input rate, if there are enough duplicates in the input to compensate. Therefore, we recommend using the CacheParallel BindJoin algorithm, whenever the input is likely to have some duplicates; and if the data distribution is skewed, the variant choosing the most popular value to be processed provides the best performance.

**4.4.6 Introducing intra-operator parallelism in the BindJoin algorithm**

In this section, we show how to adapt the operator architecture for BindJoin and BindAccess to allow several parallel accesses to a restricted resource. We then provide a simple algorithm for determining the degree of intra-operator parallelism that should be used within a BindJoin-BindAccess pair. Our algorithm is designed to exploit the full capacity of the resource, while sharing it equally with all other queries using the same resource at the same time; also, it dynamically adapts to changes in the per-call execution time, which may be due to unpredictable parameters such as the machine load on the site executing a program, the network congestion, etc.

To allow \( N \) parallel accesses to a resource, we simultaneously run \( N \) instances of GetBinding, ComputeResult, and BindJoin, as illustrated in figure 4.12 for \( N = 2 \). All GetBinding
and BindJoin operators consult and update the cache buffer; all BindJoin instances produce in Q1, multiplying the chances of large output packets. In the case of a remote ComputeResult, all instances consume their input from Q3 and produce in Q2. When running several instances of GetBinding, we use locking on the data structure’s entries to avoid race conditions, whether ComputeResult is local or remote. From an implementation point of view, the open() call of BindJoin creates one instance of BindJoin1, which supervises the execution and decides on the appropriate number of BindJoin2 instances to create. In turn, each created instance of BindJoin2 opens its ComputeResult and GetBinding operators.

Determining the right degree of parallelism  Consider the case when a single BindJoin operator uses the restricted resource: its goal is to determine the maximum degree of parallelism possible. We propose an algorithm which greedily runs at any moment as many instance as possible. The execution proceeds in stages; at the end of each stage, BindJoin1 decides on the number of instances to run in the next stage.

In the first stage, one instance is run. In the second stage, the BindJoin tries to execute two instances in parallel, and measures their average per-tuple execution time avg. Let $t_{min}$ be the minimum execution time, per instance and per tuple, of a call to the restricted resource: it can be declared by the publisher, or may be recorded over previous executions.

We use two parameters, $incThreshold$ and $decThreshold$, which must be greater than 1, and satisfy $decThreshold > incThreshold$. To decide whether to increase, decrease, or keep the same number of instances, the BindJoin compares $avg$ with $incThreshold \times t_{min}$ and $decThreshold \times t_{min}$:

1. If $avg < incThreshold \times t_{min}$, then the last recorded average time is quite close to the best measured one, and BindJoin1 decides to add an extra instance in the next stage.

2. If $avg > decThreshold \times t_{min}$, the last average time is too far from the optimum, and BindJoin1 eliminates an instance in the next stage, trying to attain again better average running times.

3. If $avg$ falls within the window $[t_{min} \times incThreshold, t_{min} \times decThreshold]$, we keep the same number of instances, since neither increasing nor decreasing it seem interesting choices.
Note that in the last case, if the window is too narrow, the number of instances oscillates between some value $n$ and $n + 1$; if the window is too wide, the algorithm reacts too slowly to correct a bad decision. This algorithm greedily runs as many instances as possible.

**Sharing the access to a restricted resource**  We now address the problem of fairness in sharing a parallel resource among several BindJoins. We can do this simply by modifying the algorithm above as follows. For every restricted access published on $S_1$, the execution engine stores the number of parallel instances running for each BindJoin at a given moment. When a BindJoin decides, following the window scheme presented above, that parallelism is profitable and that it needs to add an extra instance, it may do so only if its current number of instances is the lowest among all the BindJoins concurrently exploiting the same resource. Similarly, whenever a BindJoin decides to eliminate an instance, it has to verify, by inspecting everybody’s counters of running instances, that its number of instance is the highest. This simple “politeness” mechanism ensures both maximal exploitation of a restricted resource and equitable repartition of instances among all BindJoins running in parallel, as we show in the next section.

### 4.4.7 Adaptive tuning of the degree of parallelism

We now describe experiments that validate the adaptive algorithm for choosing the optimal degree of parallelism in exploiting a restricted resource. We consider that before execution we know (from having measured previous executions) that $t_{\text{min}} = 10$. We assume the following behavior from the restricted resource: up to the optimal (unknown) degree of parallelism $opt_{||}$, any instance processes one tuple in 10 time units. If $n$, the number of instances run, is greater than $opt_{||}$, the processing time of a tuple per instance becomes $t_{\text{min}} \times opt_{||}/n$. For $t_{\text{min}} = 10$, we obtained good results with $incTreshold = 1.2$ and $decTreshold = 1.5$. We assume that if the machine where the resource is located were idle, we could run 7 parallel instances with good performance; if the machine is loaded, this number diminishes.

The left column of graphs in figure 4.13 show how a single BindJoin operator adapts its behavior to a variable load on the machine where the restricted resource is located. The top curve plots the load on the machine, due to external factors (other than the BindJoin), measured in abstract load units. The second curve plots the load of the machine due to the BindJoin instances, measured using the same units. The third curve sums the total load on the machine; ideally, this should be close to the load generated by $opt_{||}$ BindJoin instances.

It can be seen that a single BindJoin (second curve from top, on the left in figure 4.13) adapts its number of running instances to the machine load. At the beginning of the execution, the load on the machine is 4 units; the BindJoin starts one, then two, then three instances, using the resource to a maximum. When the external load decreases (at $t \approx 200$), the BindJoin measures decreased $avg$ times, and adds two extra instances. When the machine load increases to 6, the BindJoin adapts and restricts itself to only one running instance. Thus, during the running time of the BindJoin, the total machine load is $opt_{||}$, except for short disruptions, during which the algorithm adjusts to some change in the occupancy of the
Figure 4.13: Sample behavior of the adaptive algorithm for determining the optimal parallelism degree.
machine. By adapting to the machine load, the BindJoin tends to exploit all the remaining processing capabilities.

At right in figure 4.13, we demonstrate how two BindJoins adapt to the machine load and to the presence of each other. Here, we plot the external load, the load due to instances of each BindJoin, and the total load. The machine load starts at 4, then varies successively to 2, 3 and 7. The first BindJoin, represented by the dotted line, starts at $t = 0$ and runs three, then five, parallel instances. At $t = 300$, the second BindJoin, shown in a dashed curve, starts running, and launches up to three parallel instances. From the moment when BindJoin$_2$ starts, and until $t = 700$, the two BindJoins share the resource evenly, with 4 and 3 instances running for each. At $t = 700$, the load on the machine jumps to the equivalent of 6 BindJoin instances, therefore BindJoin$_1$ and BindJoin$_2$ restrict their number of instances to 1; since neither will completely stop before finishing its task, there is a period when the total load of the machine is $6 + 1 + 1 = 8$, rather than 7 equivalent BindJoin instances. The first BindJoin finishes execution at $t = 840$, and from that moment the remaining one exploits the resource on its own, with one running instance, since it knows it no longer has a concurrent.

4.5 Query execution in the presence of space restrictions

In our discussion, we have not considered space limitations on any LeSelect site. In particular, we assumed that the results of any program execution, as well as the argument-result cache pairs, are kept on the program’s site at least until the end of the query that produced them. Also, we assumed that whenever a BM is asked to retrieve a blob from a different site, the necessary storage space is available.

At any moment, a given site has a certain amount of space available for a given query, which varies following the needs of the operators running on that site. We consider this space to consist of memory and disk taken together, and we do not investigate when is memory used rather than disk, or vice versa. We assume that when memory runs out on a given site, operations are continued using disk space; in this section, we investigate the consequences of disk space limitations on query execution.

4.5.1 Influence of space restrictions on query execution

Space limitations may affect query execution in several ways. First, if the space available for the data structure in a CacheParallel BindJoin is limited, the BindJoin will stop fetching new arguments from its left child until some space is freed; thus, no special measure is necessary. Second, in the case of a BindJoin (either CacheFIFO or CacheParallel) producing small results, when the argument-result cache is full, some cache entries are dropped, e.g., according to an LRU policy. In this case, the same result may be computed many times during the execution of the query.

Special care needs to be taken to ensure queries involving blobs are correctly executed in the presence of space limitations, as we explain next.
Blobs and space limitations. To understand the issues involved, consider the QEPs shown in Figure 4.14.

In the QEP at left, BJ\textsubscript{1} commands the BM of S\textsubscript{2} to retrieve blobs from S\textsubscript{1}. The second BindJoin BJ\textsubscript{2}, executed on S\textsubscript{3}, needs the same blobs; as described in section 4.3.3, we had allowed the BM of S\textsubscript{3} to choose where to retrieve blobs from. In our case, the choice is between the BMs of S\textsubscript{1}, respectively S\textsubscript{2}. However, if on S\textsubscript{2} the disk space is limited, the BM on S\textsubscript{2} may decide to discard some transferred blobs. Thus, the BM on S\textsubscript{3} may no longer be sure that all blobs can be found in both locations. Therefore, in the presence of space limitations for blob storage, this blob transfer optimization might be restricted, in the sense that some temporary blob sources are ignored, and blob retrievals are more likely to be made from the original location.

The QEP at right in Figure 4.14 exhibits another potentially dangerous case. BJ\textsubscript{1} applies the function \( f \) to some arguments; for each tuple of argument values, \( f \) produces a blob and an integer. The blobID and the integer are sent from BJ\textsubscript{1} upwards in the QEP, and upper in the query plan BJ\textsubscript{2} tries to fetch the produced blobs, using the blobIDs. If there are several BindJoins consuming the blob produced by BJ\textsubscript{1}, consider BJ\textsubscript{2} to be the highest one in the QEP.

There is one potential pitfall in this case. If on the site of BJ\textsubscript{1} there is limited space, the BM may have discarded a produced blob before BJ\textsubscript{2} has the time to ask for it; this would entail a run-time error.

In order to avoid such errors, we may forbid a BM to discard a blob before we are sure the blob will no longer be used for the needs of the present query. The question is, how can a BM be sure that a blob is no longer necessary? Some of the blobs produced by BJ\textsubscript{1} may never be retrieved by BJ\textsubscript{2}, if all tuples containing their blobIDs are eliminated by some intermediary operators between BJ\textsubscript{1} and BJ\textsubscript{2}. Without knowing this, the BM of BJ\textsubscript{1} may pin useless blobs in its cache, while discarding useful blobs. Eventually, BJ\textsubscript{1} may be filled with useless blobs, and its execution stops for lack of space for new results, thus leading to a deadlock: BJ\textsubscript{1} waits for BJ\textsubscript{2} to empty its BM so that BJ\textsubscript{1} may continue, while BJ\textsubscript{2} waits for BJ\textsubscript{1} to produce some new tuples.

To avoid these pitfalls, we enforce some synchronization among the operators in the QEP, from BJ\textsubscript{1} to BJ\textsubscript{2}, as explained in the next section. Obviously, some of the blobs must be
4.5.2 Recovery procedure for queries involving produced blobs

As above, consider a query in which a function called by a BindJoin operator BJ₁ produces blobs, that are later fetched on a different site via a second BindJoin BJ₂.

When the BM of BJ₁ has no more place for the query in which BJ₁ runs, the BM needs to discard some blobs. To avoid run-time errors, the BM can only discard blobs that have been read by BJ₂, or that are useless (BJ₂ never reads them, since all the tuples output from BJ₁ containing their blobID have been eliminated by some operators between BJ₁ and BJ₂). To that purpose, we initiate a recovery procedure, during which all tuples emitted by BJ₁ and not yet processed by BJ₂ must reach BJ₂, and BJ₂ must finish fetching all the blobs it needs. During this time, BJ₁ stops its processing and does not send any new blobID to its parent operator; thus, the BJ₁-BJ₂ pipeline is temporarily emptied. When BJ₂ has finished fetching the blobs, it signals to the BM of BJ₁ the fact that all the blobs produced by BJ₁ that it currently holds can be discarded, and thus free some space for BJ₁ to work.

Note that the recovery procedure is local, in the sense that it only affects the operators in the path from BJ₁ to BJ₂. The following actions are taken by these operators during recovery:

**Behavior of BJ₁** When the BM of BJ₁ signals that it has no more space for the query in which BJ₁ runs, BJ₁ stops its execution (i.e., it stops fetching arguments from its left-hand child, and makes no further calls to ComputeResult). The first next() call received by BJ₁ after the moment when the BM was full is answered with a special synchronization tuple. The following next() received by BJ₁ blocks until its BM signals that the space has been freed. Upon reception of this signal, BJ₁ resumes its execution and may answer the pending next() call.

**Behavior of all operators between BJ₁ and BJ₂** Upon receiving, as result to a next() call, a synchronization tuple, all operators in the chain from BJ₁’s parent to BJ₂ must first finish processing their current tuples, then forward the synchronization tuple to the above operator.

**Behavior of BJ₂** When BJ₂ receives the synchronization tuple, it must retrieve the blobs corresponding to all the blobIDs it has received before the synchronization tuple. Thus, the BM of BJ₂ sends to the BM of BJ₁ the necessary read() calls, followed by a special read(null,BJ₁). This last read call signals to the BM of BJ₁ that all blobs produced by BJ₁ are no longer necessary, and therefore the BM discards them. Since BJ₂ is the last operator in the QEP to need the blobs produced by BJ₁, upper operators do not need to receive the synchronization tuple; therefore, BJ₂ does not propagate it further. To this purpose, the synchronization tuple sent by BJ₁ must be marked as addressed to BJ₂; this information is available at query compile time.
Figure 4.15: Possible configuration of BindJoin pairs producing and consuming blobs.

Let us consider the required steps from the operators between BJ₁ and BJ₂. These operators must ensure that once a synchronization tuple is received in input, it is returned to the parent operator after all tuples received before it have been processed, and their results forwarded to their parent operators. This condition is weaker than requiring FIFO tuple processing; however, it forbids operators that are blocking on the branch situated in the path between BJ₁ and BJ₂. An obvious counter-example is the case when the synchronization tuple is received in a hash join, from the “build” branch: the hash join cannot return result tuples before having seen all the inputs, and this situation leads to a deadlock again. Therefore, when constructing such QEPs, special care needs to be taken to ensure that on the path between BJ₁ and BJ₂, non-blocking operators only are inserted; we describe how this is done in section 4.6.

4.5.3 Safety of concurrent recovery procedures

In this section, we investigate the safety of several recovery procedures taking place at the same time within one QEP. To that purpose, we list the possible QEP configurations involving two pairs of blob producer - blob consumer BindJoins, and verify whether these configurations can lead to deadlocks during recovery or not.

Let us assume a query in which BJ₁ calls function f that produces a blob, and the blob is recovered later in the plan by BJ₂; also, the function g is invoked in the same QEP by BJ₃, and the blob it produces is consumed by BJ₄. In figure 4.15, we show all distinct configurations for BJ₁, BJ₂, BJ₃ and BJ₄ within a single QEP. We omitted non-significant
operators in the QEP, except for the four BindJoins and a regular join operator. We assume that all intermediary operators on the blob producer-blob consumer path are non-blocking for the path.

At the top of figure 4.15, we illustrate the three possible configurations when the four BindJoins are on the same path in the QEP. We consider different scenarios, depending on which recovery procedure is initiated first, and when does the second one start.

- In figure 4.15(a), let us first assume that the recovery procedure BJ₃-BJ₄ is executed while BJ₁ has enough space to run. If, during the “inner” recovery, a recovery procedure is also launched on BJ₁-BJ₂, the synchronization tuple output by BJ₁ will have to wait for the BJ₃-BJ₄ recovery to end before it can cross BJ₃, so the BJ₁-BJ₂ recovery cannot interfere with the other one. On the other hand, as soon as BJ₃ is working again, the message sent from BJ₁ will be transmitted towards BJ₂, thus the second recovery procedure will advance, too.

Now assume that during a BJ₁-BJ₂ recovery, BJ₃ runs out of space. If this happens before BJ₃ receives the synchronization tuple from BJ₁, then the BJ₃-BJ₄ recovery will be initiated, and finished, before the BJ₁-BJ₂ recovery can proceed. Finally, if BJ₃ receives the synchronization tuple from BJ₁ and, while emptying its inner buffer, it runs out of space, then BJ₃ will propagate the message from BJ₁ only after its own recovery procedure with BJ₄ has been conducted. Therefore, this configuration is safe.

- In figure 4.15(b), assume BJ₁ runs out of space first. If BJ₃ runs out of space before receiving the synchronization tuple from BJ₁, then BJ₃ sends its own synchronization tuple. BJ₂, upon receiving the synchronization tuple from BJ₃, will attempt to empty its own buffer and thus retrieve as many blobs as possible from BJ₁; thus, it advances the recovery of both producer-consumer chains. When BJ₄ finally gets the synchronization tuple emitted by BJ₃, eventually this will restart BJ₃, which will propagate the synchronization tuple from BJ₁ and let the BJ₁-BJ₃ recovery proceed. If BJ₃ receives the synchronization tuple from BJ₁ and runs out of space while trying to empty its data structure, the interactions are quite similar.

If BJ₃ runs out of space first, and during its recovery, BJ₁ is saturated, too, then the BJ₃-BJ₄ recovery procedure will take place first, independently of the other; as a side effect, BJ₂ might take some blobs from the site of BJ₁. In any case, once the path BJ₃-BJ₄ is clear, the BJ₁-BJ₂ recovery procedure will also take place.

- The QEP in figure 4.15(c) is obviously safe, since the recovery paths do not intersect each other.

So, we have made sure that if operators in the recovery paths are not blocking, then the possible QEPs where all the BindJoins are on the same path are safe, with respect to concurrent recovery procedures. Now let us turn to the other configurations: we have to envision QEPs containing at least a regular join operator, and place BJ₁ and BJ₂, the blob producers, on the two join branches. Then, there are two possible QEP configurations, depending on the relative positions of the blob consumers.
The QEP in figure 4.15(d) is safe, provided that the join branch on the path from BJ₁ to BJ₂ is non-blocking (this is required independently of the QEP shape). Recovery procedures between BJ₃-BJ₄ will be seen by the regular join as a slow-down in accessing the right-hand child. Assume that a recovery procedure takes place between BJ₁ and BJ₂; in the meantime, BJ₃ may run out of space one or several times, and the first procedure will have to wait, but overall, the configuration is safe.

In figure 4.15(e), if the regular join has a blocking branch, the plan is not safe. For example, if the left branch is blocking, assume that during a recovery procedure for BJ₃-BJ₄, BJ₁ runs out of space. Then, the regular join cannot complete its build, therefore cannot output any tuple, and the synchronization tuple emitted by BJ₃ is not propagated. If the right branch is blocking, the symmetrical situation occurs. Thus, such a configuration can only be allowed if the regular join operator is non-blocking for both input branches, e.g., the double-pipelined hash join proposed in [64], or the ripple join described in [56].

The QEP in figure 4.15(f) is clearly safe, since recovery procedures for BJ₁-BJ₂ take place in disjoint segments of the QEP.

4.6 Optimization issues

In this section, we discuss the modifications that need to be brought to the query optimization algorithm currently used in LeSelect in order to (a) estimate the work performed by a BindJoin using cache, (b) estimate the degree of parallelism in every QEP and favor parallel plans, minimizing the response time for a given query, (c) ensuring that the physical operators inserted in a QEP guarantee the safety of the recovery procedure described in the previous section. Also, we make some remarks concerning the possibility to share a BindJoin’s cache among several resources, and make a slight modification to the QEP generation mechanism to enable several blob retrievals in a single query.

Cost formulas for BindJoins using cache

A direct modification that needs to be made to LeSelect’s cost model is to adjust the execution cost formula of the BindJoin operator in order to reflect the presence of cache.

Let $p = r \overset{\pi}{\rightarrow} s$ be a BindJoin plan, implemented using cache, either by the CacheFIFO or CacheParallel algorithms, described in section 4.4, where $\pi$ are the binding arguments passed to the BindAccess $s$. The execution cost of $p$ is: $\text{cost}(r) + \text{cost}(s) + n_r \cdot dv_{\pi,r} \cdot \text{cost}(s)$, where $dv_{\pi,r}$ denotes the ratio between the number of distinct tuples in $\pi(r)$ and $n_r$, the number of tuples in $r$.

If $s$ is a BindAccess used for retrieving a blob, of the form $BA(R(blobID^bblobID^f))$, the transfer cost among $s$ and $p$ is $n_r \cdot dv_{\pi,r} \cdot s_{\text{blob}} / BW_{loc_x \rightarrow loc_y}$. This formula takes into account the fact that each blob is transferred only once.
The decision on whether to use or not a BindJoin with cache should be determined by the estimated frequency of duplicates in the left-hand child of the BindJoin. The formulas we use to estimate this frequency in LeSelect are very simple. A vast body of existing research tackles the problem of accurately estimating the number of tuples or the number of distinct values in a relational expression, based on histograms, statistical methods, sampling etc. (see, e.g., [112], [46], [6], [55], [91]).

Traditionally, one difficulty in using detailed data statistics is maintaining them when the underlying data changes. Currently, LeSelect only performs read-only queries; we might therefore exploit this opportunity and add a statistic-gathering component in any LeSelect server, that would run a few queries when a resource is published, and extract from the result several data statistics. Two further issues have to be addressed:

1. First, in many cases, LeSelect has no control over the actual data content of the resource being published; this content may change without notice. Examples are: the data repository behind a Web form, or tables stored in a DBMS and integrated via LeSelect.

2. Second, statistic-gathering is more difficult in the case of a restricted access resource, since we have to provide valid input values to the resource before obtaining results from it. If the table has several binding patterns, then they may be used in combination in order to guess valid input values. For example, in a table containing experiment results, of the form $Exp(date, imgID, img)$, with $bp(Exp) = \{Exp(date^f), Exp(date^bimgID^f), Exp(imgID^bimg^f)\}$, one might use the date to obtain image IDs, and estimate the number of distinct images; furthermore, use those IDs to estimate the average image size etc.

If a table has only one restricted access pattern, and especially if a required input can only come from a different table (foreign key), then it becomes more difficult to find valid bindings.

**Optimizing for parallel execution**

The query optimizer of LeSelect searches for a distributed QEP minimizing the total work. However, it will distinguish between two plans requiring the same amount of work, one of which is executed on several LeSelect servers running in parallel, while the other one is sequential and loads only one server. One of the good properties of the parallel plan is that it is likely to provide a better response time (RT) than the sequential ones.

When optimizing in order to reduce the total work (TW), the cost of a plan can be assessed using a single metric, which cumulates all the resource consumptions that the plan entails. This metric satisfies the principle of optimality, that is fundamental for the process of query optimization: if $p$ and $p'$ are identical except for two distinct sub-plans $q$ and $q'$, such that $TW(q) < TW(q')$, then $TW(p) < TW(p')$. Also, the TW metric supports a total order.

It has been shown in [42] that there exists no metric that respects the principle of optimality, predicts correctly the response time, and supports a total order. The intuition is
that in order to estimate the response time of a plan, we need to assess its consumption of each resource (in our case, CPU and bandwidth) that it uses, and consider the dependencies between tasks executed on diverse resources.

The solution proposed in [42] is to annotate each QEP $p$ with a resource consumption vector $rv(p)$. In fact, to take into account inter-operator parallelism, and in particular pipeline parallelism, two resource vectors need to be assigned to each plan $p$: the consumption until $p$ has output its first tuple, and the consumption until it has output all its tuples. The resource consumption vectors of an access plan are easily computed. When $p$ is a complex plan, to compute $rv(p)$ from the resource consumption of $p$'s children, we need to consider: the localization of the topmost operator in $p$, its estimated consumptions until the first / the last tuple is output, and the synchronization between the topmost operator in $p$ and its children. This last factor captures the pipelined or blocking edges among two operators. Thus, adopting the metric of [42] to our context would allow optimizing for response time, and in particular, favoring pipelined-parallel execution of two operators, as well as data transfers taking place in parallel with operator execution. To this last purpose, it suffices to represent a data transfer as a simple unary operation, consuming a single resource, the bandwidth among the two sites concerned, and whose cost depends linearly of the volume of data transferred. Finally, we note that using the resource consumption vectors, it is also possible to account for intra-operator parallelism, and thus capture the parallel behavior of the BindJoin operator that we proposed in section 4.4.6.

The backdrop of the resource consumption vector is that this metric no longer supports a total order. Thus, it becomes even more difficult to compare two plans, and plan pruning is significantly reduced.

**Ensuring the safety of concurrent recovery procedures**

In section 4.5 we had proposed a recovery procedure for the QEPs involving BindJoins that produce blobs, and BindJoins that consume them. We have shown that the recovery procedure is guaranteed to be safe only if all operators on the path between the blob produced and blob consumer are non-blocking. More specifically, these operators need to be able, at any moment, to process and output all tuples received so far on the branch between the blob consumer and blob producer. Our physical selection and projection operators trivially satisfy this constraint (they are FIFO). With regard to joins, the constraint may be enforced by the query optimizer as follows.

By inspecting the query, we can determine the set of blobs produced that the query involves. For each blob produced, we can determine the number of potential consumers as follows. Whenever the blob produced is used as an input to a program, there will be a blob-consumer BindJoin that fetches the blob before the program is applied. Also, if the blob is required in the query output, there will be a BindJoin fetching the blob on the query site. By counting all the programs that use the blob and the query site, we get an upper bound of the number of times the blob will actually be fetched; in practice, one BindJoin may obtain the blob for more than one usage.

Let $p$ be a partial QEP for the query $Q$. Assume $p$ contains one or several BindJoins that
produced blobs. We denote by \( cons(p) \) the upper bound on the number of blob consumer BindJoins that appear in any complete QEP for \( Q \), having \( p \) as a sub-plan.

Assume we construct a plan \( p = q \bowtie r \), such that \( cons(q) > 0 \). Then, the join operator has to be chosen so that the branch between \( p \) and \( q \) is non-blocking. If \( cons(r) \) isn’t 0 either, then \( p \) can only be built if we have a join algorithm non-blocking on both branches. In any case, \( cons(p) \) is computed as \( cons(q) + cons(r) \).

Whenever we construct a plan \( p = q \bigstar r \), such that the top BindJoin fetches one of the blobs that have been produced in \( p \), \( cons(p) \) is obtained as \( cons(q) - 1 \).

On a plan \( p \) such that \( cons(p) = 0 \), any join algorithm can be used.

**Sharing a cache among several queries** In the framework that we described, we assumed that the cache built by a BindJoin operator is destroyed when the execution of the query that involved this operator is finished. As a consequence, when we estimate the cost of a query we consider that the expensive functions it involves have to be computed on all the distinct argument values that they will receive. If function caches are shared among several operators and among several queries, then the real query execution cost is likely to be smaller, since the query may profit from the previously computed results. However, in the context of LeSelect, where servers have a high degree of autonomy, it is difficult to know which results survive, and for how long, in a function cache, since this decision belongs to the server where the function is published.

**Several accesses to a blob table** A particularity of the optimization process with the BindJoin and BindAccess operators is that a binding pattern of the form \( T(blobID^{b}blob^{f}) \) may be accessed more than once, if the blob needs to be transferred to more than one site different from its publishing sites. This contrasts with the usual assumption that an access plans is used only once when building a QEP. We have to modify the optimization algorithm as follows. We may allow BindJoin of a plan \( p \) with a BindAccess operator of the form \( BA(T(blobID^{b}blob^{f}) \) as soon as the blob attribute is hidden by the binding pattern of \( p \).

### 4.7 Related work

**Execution of expensive functions** If [21] mainly deals with the optimization of queries with expensive predicates, it also considers the execution of such queries, in a centralized context. The authors propose a *foreign sort-merge join algorithm*, sorting the tuples that are provided as input to the function, and using a one-tuple cache. They distinguish between the first join step, that applies the function to am argument tuple at a time, (or gets the result from the cache), and the *residual* join, that appends to the argument-result tuple the remaining attributes from the tuple. It is interesting to note that the BindJoin entity in the CacheParallel operator, depicted in figure 4.7, has the same role as the residual join. We took this study further by proposing several algorithms using cache, and a general framework to increase parallelism and reduce query response time, by eliminating not only useless function calls, but also useless blob transfers.
A comparison of several caching algorithms for expensive functions is provided in [62]. The authors show that the basic memoization technique may generate a lot of I/Os if the cache grows larger than the available memory, and propose a hybrid hash algorithm that handles gracefully many distinct argument values (and thus larger caches). In our work, however, we still used a simple memoization scheme. We explain next why.

The function cache may get too large for two independent reasons: either the results or the arguments are large, or there are many distinct argument values (up to 2 millions in their experiments). In LeSelect, this is less likely to happen, since in a virtual data integration context, we do not expect queries with very large intermediate results; also, we attempt to alleviate the risk of a huge cache by separating blobs from the small data items, and delegating their storage to the BlobManager. Indeed, the BlobManager may flush them to disk; however, a blob produced is never transmitted twice by the operator, and thus will not be repeatedly loaded in memory and flushed to disk, due to the way we construct our QEPs. So, we (optimistically) assume that the data structure storing the small-values cache does not outgrow the available memory. However, if queries with very large intermediate results were to be executed by LeSelect, we would use the hybrid hash technique.

The starting point of the work presented in this chapter is [13]. That study identified the categories of costly queries in LeSelect, and proposed: using caching, reducing blob transfer, and parallelizing the execution of operations that consume distinct resources. While that work lists the principles of our approach in efficiently executing costly queries in LeSelect, the components of the solution presented in this chapter are new and improve over [13]. We proposed the modeling of tables with binding patterns to describe restricted access resources. To improve the architecture, we introduced the BlobManager and recommended modeling communications by Exchange operators, as an elegant and uniform mean to parallelize data transfers and computations. Furthermore, we showed how to handle in an uniform manner program calls and blob transfers, modeling both by BindJoin operators, and designed cache-aware BindJoin algorithms. Also, while [13] proposed guidelines for optimizing queries involving expensive resources, we showed how to optimize such queries within the general framework of LeSelect.

In [80], the execution of queries with expensive user-defined functions (UDFs) is discussed in the context of an asymmetric client-server configuration. As in our case, the UDFs cannot be shipped, for confidentiality or portability reasons; they need thus to be executed at the client. The authors compare two approaches for executing the UDFs: one based on semi-joins, when the server sends the duplicate-free set of arguments to the client UDF, and another one when all the input tuples are sent to the client, to take benefit from some selections and projections it may perform after the UDF. It is showed that, due to the asymmetric bandwidth between the client and the server, if several UDF calls are issued before receiving any result, the function is kept busy, thus avoiding synchronization delays.

The execution techniques presented in this chapter have a broader scope than UDFs, since they apply for any restricted resource access, and in particular for blob transfer. We did not use semi-joins; instead, as an equivalent but more targeted solution, we designed the BindJoin operator to speed up the execution of hot spots in a query. As in [80], we do
not rely on ranking methods to optimize queries with expensive functions, since they are no longer optimal in a distributed context, and we assimilate expensive functions to tables. Both our and their optimization algorithms explore thus all the alternatives in ordering joins and function invocations, and push data-reducing operators whenever possible; the difference is that we consider a distributed peer-to-peer architecture with many servers and thus many options for placing operators, allowing for more parallelism. Furthermore, they did not apply caching.

In the Mocha project [98], expensive operators, including UDFs, may be shipped to reduce data transfers. In contrast, we consider expensive restricted resource that cannot be shipped, since they depend on a particular language, software environment, hidden resource, or the owners do not want them to be copied elsewhere.

**Adaptive query processing** Many general techniques have been proposed for deciding on the best way to execute a query by monitoring execution parameters and changing partially or completely the execution strategy to obtain better performance (e.g., [8, 118, 12, 117]). In general, these techniques are complementary to the ones we proposed in this chapter, where our focus was on the BindJoin operator. Nevertheless, some adaptive query execution techniques are closely related to the functioning of our CacheParallel BindJoin operator. Since this is not central to the work presented here, we will be brief and not exhaustive.

The query execution techniques presented in this chapter, together with the query optimization algorithm of LeSelect, represent one approach for dealing with queries involving distributed expensive resources. The *eddy* operator was proposed in [10] as an alternative to static query optimization. An eddy acts as a tuple router, monitoring the current processing speed of all the join operators that the tuple must go through, and sending it first to the fastest join it has to traverse. Since it attempts to keep all operators busy, the eddy performs a per-tuple scheduling *within* a pipeline. It would thus seem an interesting approach in order to avoid idle times in the execution of a distributed query in LeSelect. However, eddies were studied in the context of a single-site query execution; in a distributed context, the communication overhead that results from the obligation that all tuples transit many times through the same scheduling site is likely to be quite important.

The CacheParallel BindJoin that we described is pipelined, and attempts to make the best use of the input data to maintain a steady output rate. Several pipelined adaptive join operators have been previously proposed [121, 64]; the difference is that these algorithms are essentially symmetric, in the sense that they can switch the outer and the inner relation to maximize the tuple output rate. In contrast, the BindJoin operator is essentially asymmetric, since one child will always depend, and be commended by, the other.

In [96], a *reordering* operator is presented, that changes the order of the tuples in its input to allow high priority tuples to be output before the others, in queries involving online aggregation, sorting and scrolling. There are similarities between this approach and the fact that the CacheParallel join algorithm treats the most popular argument values first. The difference is that in their work, tuple priority is determined by user feedback, and it may well be the case that the full query result is not computed, since the user is not interested
in it; in our context, tuple reordering is done in order to exploit the existing cache, and to reduce the total query response time.

We surveyed existing approaches for optimizing queries in the presence of expensive functions, and compared them with our approach, in section 3.8.
Chapter 5

The Agora data integration system

5.1 Introduction

So far, we have presented query execution and query optimization techniques in the context of LeSelect, whose data model is relational. While LeSelect provides a framework for querying distributed relational tables, it has no global schema. Nowadays, the popularity of XML as a data exchange format makes a strong case for data integration applications under a global XML schema. Using an XML-based schema at the interface level allows to hide the proprietary schemas that the data owners do not want to disclose, and to adhere to a newly-established standardized interface without having to migrate existing data. Furthermore, for many application domains, standardized, domain-specific XML global schemas have already been established. These standardized schemas, available as DTDs or XML Schemas, provide the basis for large-scale integration applications.

In this chapter, we present Agora, an extension of LeSelect, that enables seamless querying of DOM-compliant and relational data sources under a single XML global schema:

1. A DOM-compliant data source is any document accessible through the DOM API. We have shown in section 2.2.6 how to model DOM-compliant data sources as tables with binding patterns, so that they can be queried (using SQL) by LeSelect. Among the DOM-compliant data sources, we focus only on XML data sources (XML documents accessible through the DOM wrapper), since non-XML data is more difficult to present and classify in the global XML schema.

2. A relational data source is any data source published as a relational table, published by a wrapper (distinct from the DOM wrapper) on a LeSelect site.

3. The XML global schema presented to the user in Agora consists of a set of DTDs and, for each DTD, the names of the available documents conforming to that DTD are provided.

Agora adopts the local as view [116] approach for relating the relational and XML data sources to the XML global schema. To that end, the tables with binding patterns corresponding to the XML and relational data sources are defined as views over a generic...
virtual, relational schema, that closely models the generic structure of an XML document. The query language supported in Agora is XQuery [126], the query language being standardized by the World Wide Web Consortium’s working group on XML query languages. An XQuery query posed against the set of documents from the global schema is translated into one or several SQL queries over the virtual generic schema, which are in turn rewritten into SQL queries on the tables corresponding to the data sources. The optimization and execution of these SQL queries is delegated to LeSelect; thus, the XQuery query over relational tables and XML documents is optimized and executed in a single step. Also, using a relational execution model allows us to profit from the relational query capabilities that the relational or XML sources may have, by pushing computations to these sources. The tuples resulting from the SQL query execution are structured into the desired XML result format by a tagger module, outside the execution engine; the tagger’s functioning is inspired by work done in [107].

This chapter is organized as follows. In section 5.2, we describe our approach for query processing in Agora: we introduce an example, present XQuery, the XML query language we use, and outline our architecture. The following sections describe the three query processing steps that are taken in order to transform an XQuery user query into an SQL query on the relational data sources. Section 5.3 provides normalization rules that make the query easier to translate on the generic schema, or signal the fact that the translation is unfeasible, due to the expressive power mismatch between XQuery and SQL. Section 5.4 shows how to translate normalized XQuery queries into SQL queries on the generic schema, and section 5.5 discusses the rewriting of the SQL query on the generic schema into a SQL query on the real data sources. Section 5.6 explains how we can enlarge the translatable subset of XQuery by allowing intermediate XML query results. A comparison with related work is provided in section 5.7.
for $no in document("records.xml")//record/patientSSno/data()
let $recs:=document("records.xml")//record[@SSNo=data()]=[$no]
return <pollutionIncident> { $no,
  (for $e in $recs
    where $e/date/data() > "1/1/91" and contains($e/diagnosis, "pollution")
    return $e/diagnosis) }
</pollutionIncident>

Figure 5.2: Sample XQuery query.

5.2 XML data integration methodology

5.2.1 Problem definition

The goal of the Agora system is to integrate relational and XML data sources under a global XML schema. Our data integration methodology must allow for efficient query processing, in particular by exploiting as much as possible the query processing capabilities of the local data sources, be they relational or DOM-compliant.

5.2.2 Motivating example

Our sample data sources are inspired from the domain of health care. Figure 5.1 shows the data presented to the user under the form of two XML documents. The document “patients.xml” contains administrative information about patients in the patient elements, while “records.xml” contains medical files that physicians keep on patients; for each patient, there is one record element. The global schema consists of the DTDs of these two documents. Data is actually stored in two local sources. On the LeSelect site S1, the administrative information is stored as such in a “patients.xml” file; a DOM wrapper W1 exports a set of tables with binding patterns allowing this document to be queried. On site S2, the medical records are stored in a relational format as the following table Entry, published via a wrapper W2:

Entry(recordID, entID, patientSSno, date, symptoms, diagnosis, medication)

For every entry element in “records.xml”, there has to be a tuple in the Entry table, containing the entry ID, the ID of the record containing this entry, and the information associated with the entry. The rel_previous link among entries in a record is established via a separate table EntryLinks(eID1, eID2).

5.2.3 The XQuery query language

The query language that our mediator supports is XQuery, the standard XML query language being elaborated by the W3C [126]. The XQuery data model views an XML document as
a labeled tree with references; its type system follows that of XSchema [128]. Besides value and node types, the data model considers only ordered lists; a significant general feature of the algebra is the automatic list flattening - lists of lists are always unnested [124]. XQuery has static and dynamic semantics, according to the way type checking is performed; in this work, we always consider dynamic semantics.

A central notion in XQuery is that of expression: starting from constants and variables, expressions can be nested and combined, using the usual arithmetic, logical and set operators, navigation primitives, function calls, higher order operators like sort, conditional expressions, element constructors etc.

Path Expressions  In this respect, XQuery borrows the abbreviated syntax of XPath [129]; we illustrate path expressions with examples. The document("mySite.org/records.xml") expression retrieves the root of the XML document situated at the given URI. The XQuery path expression document("records.xml")/record is the ordered list of all record children of the document root. The expression document("records.xml")//record designates the list of record elements at any depth in the document, in document order. In XQuery, a path expression can start in a document root or in an implicit current root node that is deduced from the evaluation context; the expression //patient/@ssNo retrieves the collection of values of the ssNo attributes in all patient elements in the current document. A dereference operator is also provided: //entry/@rel_previous→entry returns all medical entries that are “pointed at” by some other entry descendant of the current node. Specifying the tag of the target element is not mandatory (“*” can be provided instead of the tag).

To extract the data content of a given node, the data() access function is used. Thus, if //patient/@ssNo returns the list of all ssNo attributes, by writing //patient/@ssNo/data() we obtain the list of values of these attributes. In the presence of an XML Schema type description of the document, the type of these values may be inferred precisely. Otherwise, the type associated to the result of data() is always string.

The string-value() function can be used to obtain the text image of a given expression. This text image is constructed following the rules defined in the XPath specification [129].

Path expression can contain path predicates that restrict the navigation to nodes satisfying certain logical conditions. For example, //entry/date/data() = "1/9/90"/ has the following meaning: all the entry elements having at least one date child, whose string value is “1/9/90”, are returned. A path predicate can also select a specific range; as an example, document("records.xml")/entry/range 2 to 5/ will only return the second to fifth entry elements, in document order. From this description of XQuery path expressions, it can be seen that automatic list flattening applies here, too. With respect to the requirements, XQuery path expressions provide for hierarchy and sequence querying and preservation, and for reference traversal.

We note that at the time of this writing, the precise semantics of path expressions is still under discussion; we consider a snapshot of the semantics for simple path expressions, as it was in February 2001.
Operators  XQuery provides the usual set of first-order operators (arithmetic, logical and set-oriented). The distinct-value and distinct-node operators are used to eliminate duplicates from a sequence of values, respectively from a sequence of nodes. The comma is a list concatenation operator; for example, (//entry, //name) returns the concatenation of the list of all entries, followed by all names, in document order.

Second order operators in XQuery are the logical quantifiers some and every, and sort. For example, the expression

\[
\text{some } p \text{ in document("patients.xml") satisfies } p/@ssNo/data()="123"
\]

evaluates to true on our sample XML database. Also, assume the date elements in the document “records.xml” are of type date, as standardized in the XML Schema. Then, the expression \(\text{document("records.xml")//entry sort by date descending}\) will return all entry elements, the most recent first. Since the semantics of arithmetic and boolean operators is also being currently discussed, in this thesis we interpret them following simple SQL semantics.

It is important to note that, while by default most XQuery constructs return their results in an order derived from the input order, an unordered() operator is provided, that allows to specify that the order of the result is not important. The order of elements in the result of an expression nested within unordered() is arbitrary.

Conditional expressions  These expressions correspond directly to an important feature of the algebra. The syntax is if \(C\) then \(E_1\) else \(E_2\); an important thing to note is that the type of this expression is the union of the types resulting from the two branches.

Element constructors  This class of expressions provides for construction of new XML structure; for example, \(<\text{alphabetist}>\{\text{document("patients.xml")//name}\}<\text{/alphabetist}>\) constructs an alphabetic list of all patient names. Syntactically, the element’s tag, attribute names and attribute values can be obtained from constants or variables, while the children are specified as a list of arbitrary expressions.

Element identity  In the XQuery data model, every instance of a node has a unique identity, that is preserved by most XQuery operators. To test whether two nodes are the same, the “=” operator is used. It is important to note that the result of an element constructor is a new element, to which a new identity is assigned. Also, if we insert elements from the input as children of the newly created element, they are copied into the output. For example, if we consider the element constructor above, all its name children have new identities and are different from the nodes in the input.

The type system of XQuery also contains a node reference type, and an operator ref for obtaining the reference to an existing element. For example, if in the element constructor above we want to insert references to the names in the input, instead of their copies, we would write

\(<\text{alphabetist}>\{\text{for }x\text{ in document("patients.xml")//name return ref($x))}\]<\text{/alphabetist}>\)
**FLWR expressions**  “FLWR” is an abbreviation for “for-let-where-return”. A FLWR (pronounced “flower”) expression consists of three parts: a for-let clause, that make variables iterate over the result of an expression, or binds variables to arbitrary expressions, a where clause that allows the specification of restrictions on the variables, and a return clause, that can construct new XML elements as output of the query.

As an example, consider the sample query shown in figure 5.2. This query constructs a summary of medical records of people with health problems related to pollution within the last ten years. Variables defined with a for are iterators: the $no variable iterates over all the SSno attributes of record elements. For each value of SSno, the let clause binds $recs to the list of record elements corresponding to this value. Such for-let combinations are the XQuery mechanism for grouping; note that, unlike XML-QL [31], XQuery does not use the fusion mechanism of Skolem functions for grouping. The return clause contains a nested FLWR expression, that is dependent on the externally bound variable $no; one new pollutionIncident element is constructed for each value of $no. FLWR expressions allow combining data from different documents, grouping, construction of new structure, and are natural candidates for query composition.

**Functions**  XQuery provides a syntax for defining functions; these can be called in XQuery queries. In particular, recursive queries can only be implemented in XQuery by calls to recursive functions. The syntax for a function definition is illustrated in figure 5.3; we show the function declaration and a sample query calling it. The function pastEntries returns the list of all past record entries linked to a given record. This example also features a function from its standard library, empty (other functions like count, max, avg etc. are included).

**General form of a query**  We can now describe the general structure of a query in XQuery: it consists of an optional list of namespace definitions, followed by a list of function definitions, followed by a single expression. It is important to notice that in XQuery there is no “first-class citizen” data type - document("patients.xml")/entry, as well as (5+1) and avg((2, 3)) are legal XQuery queries; also, by its functional nature, it allows arbitrary nesting of expressions within a query. Nested queries tend to be more difficult to translate because SQL does not allow the same freedom in nesting; this is why in section 5.3 we provide rules...
for unnesting XQuery queries. Redundant XQuery constructs are also eliminated in this phase.

5.2.4 Data integration methodology

The general methodology for data integration in Agora is the following. To execute XQuery queries via LeSelect's relational engine, we devised a query translation methodology that proceeds in three steps, shown in figure 5.4. First, the query is normalized, applying equivalent transformations that bring it to a syntactical form which can be directly translated to SQL, if this is possible. The normalized query is translated into a SQL query on a generic, virtual, relational schema. This schema, detailed in section 5.4, is used only as an intermediate layer; it is never materialized as such, and is invisible to the system's users. This translation step is completely independent of the relation between the virtual XML global schema and the real data sources; it only gets the query across the language gap. Finally, the SQL query on the generic schema is rewritten into a SQL query on the real data sources. In this relational query rewriting step, we use the definitions of the data sources as views over the virtual generic schema.

Not all features of XQuery can be translated to SQL; there are two distinct sources of difficulties. First, some of the language's features do not have SQL equivalents due to a semantic mismatch between the two models; such features are identified (and the translation fails) during the normalization phase. Second, for those XQuery queries that could be brought to a SQL form on the virtual generic schema, relational query rewriting might fail, because state-of-the-art query rewriting algorithms for SQL semantics do not handle well arbitrary levels of nesting, grouping etc. We stress the fact that these difficulties are not due to our translation methodology; we merely separated the language-dependent translation step, transforming an XQuery query into SQL on the generic schema, from the rewriting step reformulating the query in terms of relevant data sources. This separation allows us to provide independent solutions for the two steps, and to distinguish among the two sources of difficulties.

If the rewriting step succeeds, we obtain a SQL query referring to well-identified local data sources, either relational or DOM-compliant. Tuples resulting from the relational execution of this query are treated by a tagger module, that structures them into the desired XML format of the result. This structure information is produced during the translation step and is passed directly from the translator to the tagger under the form of a tagging template, as shown in figure 5.4. The tagger's functioning is inspired by work done in [107].

Our approach is implemented into the Agora research prototype, demonstrated at [79].

5.3 XQuery normalization

We now describe the steps that can be taken to simplify XQuery queries. An algebra for XQuery is currently being developed [124], partly in parallel with our work, and it provides
several equivalence rules; for brevity, we do not introduce the algebra formalism and provide our simplification rules in a bare XQuery syntax.

In this section we use the following notations. Lower case letter like $x, y, z$ correspond to individual XQuery query variables, while capital letters like $E, R, C$ denote XQuery expressions. We denote simple path expressions by $PE$, and element constructor expressions by $EC$. For brevity, we some times write a single for clause “for $\vec{x}$ in $E$” instead of “for $x_1$ in $E_1$, $x_2$ in $E_2(x_1)$, $\ldots$ $x_n$ in $E_n(x_1, \ldots x_{n-1})$”; in this case, $E$ is an expression of arity $n$, and $\vec{x}$ are consecutively bound to each tuple of values that result from $E$’s evaluation.

Using these notations, the classes of translatable queries can be informally described as follows:

- simple path expressions, starting with a document node or with an implicit context node, consisting of steps of the following kinds: child, descendant, attribute, and dereferencing, and eventually interspersed predicates.
- element constructors whose tags and data are either constants or come from simple path expressions as described above, or from translatable FLWR expressions;
- translatable FLWR expressions of the form for $\vec{x}$ in $E$ where $C(\vec{x})$ return $R(\vec{x})$, where: $E$ denotes a n-uple of simple path expressions, $C(\vec{x})$ is a logical expression constructed with simple path expressions depending on $\vec{x}$ and usual operators; $R(\vec{x})$ is a list of simple path expressions depending on $\vec{x}$, or a translatable element constructor;
- arithmetical and logical expressions on scalar types.

In the sequel, for simplicity, we always consider queries in which all path expressions start with a $document()$ node. If the incoming query refers to an implicit context node, we assume
a context resolution has taken place before the query is processed, so that all elements involved in the query can be related to a specific document.

5.3.1 Normalization rules

In this section, we provide several equivalence rules to simplify the user’s query and bring it to one of the translatable forms, when possible.

Let clauses are treated as temporary variable definitions. During normalization, they are eliminated as shown in rule NR1: the expression binding the variable \( y \) is substituted to all its occurrences. Non-recursive function definitions are eliminated; calls to such functions are replaced with the body of the function, applying the proper substitutions.

\[
\begin{align*}
\text{NR1} & \quad \text{for } \overline{x} \text{ in } E_1, \quad \text{for } \overline{x} \text{ in } E_1, \\
& \quad \text{let } y = E_2(\overline{x}) \quad \overline{x} \text{ in } E_3(\overline{x}, E_2(\overline{x})) \\
& \quad \text{for } \overline{z} \text{ in } E_3(\overline{x}, y) \Rightarrow \text{ where } C(\overline{x}, E_2(\overline{x}), \overline{z}) \\
& \quad \text{where } C(\overline{x}, y, \overline{z}) \Rightarrow \text{ return } R(\overline{x}, y, \overline{z}) \\
& \quad \text{return } R(\overline{x}, \overline{z}) \\
\end{align*}
\]

In XQuery, FLWR expressions can be used as building blocks for more complex expressions. Rule NR2 unnestes expressions of the form \( E_1(FLWR) \), in the case when expression \( E_1 \) distributes over list concatenation, e.g. \( E_1 \) is a child path step (illustrated under the rule). This rule is a consequence of the automatic list flattening feature of the XQuery algebra. Rule NR2 does not hold if \( E_1 \) is, for example, a range operator, or an aggregate function.

\[
\begin{align*}
\text{NR2} & \quad E_1(\overline{x}) \text{ in } E_2, \quad \text{for } \overline{x} \text{ in } E_2, \\
& \quad \text{where } C(\overline{x}) \Rightarrow \text{ where } C(\overline{x}) \\
& \quad \text{return } E_3(\overline{x}) \Rightarrow \text{ return } E_1(E_3(\overline{x})) \\
& \quad (\text{for } \overline{x} \text{ in } E_2) \Rightarrow \text{ where } C(\overline{x}) \\
& \quad \text{where } C(\overline{x}) \Rightarrow \text{ return } E_1(E_3(\overline{x}))/\text{nameTest} \\
& \quad \text{return } E_3(\overline{x})/\text{nameTest} \\
\end{align*}
\]

Element constructors nested within path expressions have the general form \( PE(EC(\overline{x})) \), where \( \overline{x} \) represent variables that may have been bound outside this expression. If \( PE \) consists of path steps without the range predicate, the path steps can be composed with the element constructor and the expression rewritten, so that the element constructor disappears. Rule NR3 shows how to push such steps into element constructors, when \( E(\overline{x}) \) evaluates to a list of XML elements; the comma represents sequence concatenation. If the element constructed by the expression \( EC(\overline{x}) \) has text children, they are erased by the translation. A simple similar rule holds for attribute steps.
for $x$ in document("records.xml")/entry 
where $x$/date/data()="1/9/90"
return 
(for $y$ in documents("patients.xml")//patient 
where $y$/@ssNo/data()=$x$/SSNo/data() return $y$)
⇒
for $x$ in document("records.xml")//entry, 
$z$ in (for $y$ in documents("patients.xml")//patient 
where $y$/@ssNo/data()=$x$/SSNo/data() 
return $y$)
where $x$/date/data()="1/9/90" return $z$

Figure 5.5: Example of unnesting return clauses.

<table>
<thead>
<tr>
<th>Rule NR$_3$</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;tag&gt;)</td>
<td>$E(\overrightarrow{x})$</td>
<td>if tag = nameTest</td>
</tr>
<tr>
<td></td>
<td>{&lt;tag&gt;}</td>
<td>⇒ then (&lt;tag&gt;$E(\overrightarrow{x})$&lt;/tag&gt;)</td>
</tr>
<tr>
<td>&lt;/tag&gt;/nameTest</td>
<td></td>
<td>$E(\overrightarrow{x})$/nameTest</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else $E(\overrightarrow{x})$/nameTest</td>
</tr>
<tr>
<td>(&lt;tag&gt;)</td>
<td>$E(\overrightarrow{x})$</td>
<td>for $y$ in $E(\overrightarrow{x})$</td>
</tr>
<tr>
<td>&lt;/tag&gt;/nameTest</td>
<td></td>
<td>⇒ where name($y$)= nameTest</td>
</tr>
<tr>
<td></td>
<td>return $y$</td>
<td></td>
</tr>
</tbody>
</table>

Rule NR$_4$ unnests FLWR expressions nested within the for clause of an outer FLWR expressions.

<table>
<thead>
<tr>
<th>Rule NR$_4$</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
</table>
|           | $\overrightarrow{x}$ in $E_1$, $y$ in (for $\overrightarrow{x}$ in $E_2(\overrightarrow{x})$ 
where $C_1(\overrightarrow{x}, \overrightarrow{y})$ return $E_3(\overrightarrow{x}, \overrightarrow{y})$) 
where $C_2(\overrightarrow{x}, y)$ return $E_4(\overrightarrow{x}, y)$ 
⇒
for $\overrightarrow{x}$ in $E_1$, $\overrightarrow{x}$ in $E_2(\overrightarrow{x})$ 
where $C_1(\overrightarrow{x}, \overrightarrow{y})$ and $C_2(\overrightarrow{x}, E_3(\overrightarrow{x}, \overrightarrow{y}))$ 
return $E_4(\overrightarrow{x}, E_3(\overrightarrow{x}, \overrightarrow{y}))$ |

Rule NR$_5$ unnests FLWR expressions nested in the return clause of another FLWR expression. This rule is valid because of the implicit list flattening of the algebra; such a rule would not hold in OQL.

<table>
<thead>
<tr>
<th>Rule NR$_5$</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
</table>
|           | $\overrightarrow{x}$ in $E_1$, $\overrightarrow{y}$ in $E_1$, 
where $C_1(\overrightarrow{x})$ return $E_3(\overrightarrow{x}, \overrightarrow{y})$ 
(for $\overrightarrow{y}$ in $E_3(\overrightarrow{x})$ return $E_3(\overrightarrow{x}, \overrightarrow{y})$) 
where $C_2(\overrightarrow{x}, \overrightarrow{y})$ return $E_3(\overrightarrow{x}, \overrightarrow{y})$ |

Rule NR$_6$ unnests complex expressions built on top of conditional expressions. NR$_6$(a)
is meant for cases when \( E \) is constructed only with simple path expression steps, element constructors, or arbitrary function calls.

\[
\begin{array}{|c|c|}
\hline
\text{NR}_6 (a) & E(\text{if } C(x) \text{ then } E_1(x) \text{ else } E_2(x)) \\
& \Rightarrow \\
& \text{if } C(x) \text{ then } E(E_1(x)) \text{ else } E(E_2(x)) \\
\hline
\text{NR}_6 (b) & \text{for } \vec{x} \text{ in } E_1, y \text{ in } (\text{if } C_1(\vec{x}) \text{ then } E_2(\vec{x}) \text{ else } E_3(\vec{x})) \\
& \text{where } C_2(\vec{x}, y) \text{ return } E_4(\vec{x}, y) \\
& \Rightarrow \\
& \text{for } \vec{x} \text{ in } E_1, y \text{ in } E_2(\vec{x}) \\
& \text{where } C_1(\vec{x}) \text{ and } C_2(\vec{x}, y) \text{ return } E_4(\vec{x}, y) \\
& \cup \\
& \text{for } \vec{x} \text{ in } E_1, y \text{ in } E_3(\vec{x}) \\
& \text{where } \neg C_1(\vec{x}) \text{ and } C_2(\vec{x}, y) \text{ return } E_4(\vec{x}, y) \\
\hline
\text{NR}_6 (c) & \text{for } \vec{x} \text{ in } E_1 \\
& \text{where } (\text{if } C_1(\vec{x}) \text{ then } E_2(\vec{x}) \text{ else } E_3(\vec{x})) \Rightarrow \\
& \text{return } E_4(\vec{x}) \\
\hline
\text{NR}_6 (d) & \text{for } \vec{x} \text{ in } E_1 \\
& \text{where } C_1(\vec{x}) \Rightarrow \\
& \text{return } (\text{if } C_2(\vec{x}) \text{ then } E_2(\vec{x}) \text{ else } E_3(\vec{x})) \\
\end{array}
\]

The rules \( \text{NR}_6 (b), (c) \) and \( (d) \) show how to eliminate conditional expressions directly nested within a \( \text{for}, \text{where} \), respectively \( \text{return} \) clause; note that these rules modify the order of the result, therefore they can be applied only if the order of the result is not important.

Rule \( \text{NR}_7 \) performs a simple syntactic transformation: if \( E_2 \) is a predicate restricting the result of \( E_1 \)'s evaluation, the path predicate notation can be replaced with a where clause in a FLWR expression, since the test has existential semantics in both cases. As an application, path predicates in the for clause of a FLWR expression can be moved to the where clause; we denoted by \( /PE \) the final part of the simple path expression \( x \) iterates over.

\[
\begin{array}{|c|c|}
\hline
\text{NR}_7 & E_1[E_2] \\
& \Rightarrow \\
& \text{for } x \text{ in } E_1 \\
& \text{where } E_2(x) \\
& \text{return } x \\
\hline
& \text{for } x \text{ in } E_1[E_2]/PE \\
& \text{where } C(x)/PE \\
& \text{return } R(x) \\
\hline
& \text{for } y \text{ in } E_1, x \text{ in } y/PE \\
& \text{where } C(x) \text{ and } E_2(y) \\
& \text{return } R(x) \\
\end{array}
\]
5.3.2 Difficulties of the normalization process

The purpose of the normalization rules shown so far was to bring the user query to one of the generic forms described at the beginning of section 5.3. It is interesting at this point to identify which XQuery expressions cannot be simplified into one of these forms. Since in our context, normalization means unnesting, the question is thus: which classes of XQuery expressions we are unable to unnest?

An important source of queries that we are unable to normalize comes is that of queries that call XQuery functions based on structural recursion, or external functions, i.e. functions whose definition is not known to the query processor. Such is the case, for instance, for user-defined functions, which may be written in a different language, and called in an XQuery query. As an example, consider a function \texttt{listHospitals}($\text{ELEMENT}\; \$proc, \text{ELEMENT}\; \$address) \rightarrow \{\text{ELEMENT}\}$: this function returns a list of hospitals capable of performing a given surgical procedure, sorted by their closeness to a given address. A path expression on the result of such a function call cannot be unnested, since we do not know the internal structure of the function result. To execute such queries in our framework, one needs to make several passes, materializing intermediate XML results and running a sequence of XQuery queries, as we show in section 5.6.

5.4 Translating normalized XQuery into SQL

Queries within the normalized subset of XQuery are transformed in SQL queries on the real data sources in two steps: first, they are translated into SQL queries on the virtual generic schema in figure 5.6, then, by a relational query rewriting step, they become SQL queries on the local data sources schema. In this section, we detail the translation step, which does not yet take into consideration the schemas of the local sources.
5.4.1 Virtual generic schema as support for translation

The simple generic, virtual, relational schema that we use is shown in figure 5.6; in each table, primary keys are in bold characters. This schema is constructed as a fully normalized relational version of the hierarchical structure of an XML document; foreign keys represent the relationships between different entities within a document. The last table, TransClosure, is redundant; it represents the transitive closure of the parent-child relationship modeled by the Child table. This table is useful for translating recursive XML path expressions, as described in section 5.4.2, and for rewriting the resulting queries, as shown in section 5.5.4; we stress the fact that it is virtual, i.e. it does not need to be materialized or maintained.

Using the virtual generic schema has several advantages. First, it connects the relational (and other) data sources and the XML global schema. This schema represents a middle ground for query translation: it is a minimal lossless schema with respect to the information contained in an XML document. Since this generic schema does not lose any of an XML document’s information content, XQuery constructs that cannot be translated to it cannot be translated to any relational schema, simply because their semantics cannot be adapted to the semantics of SQL. At the same time, it is a middle ground for view definitions: data sources described as views over this generic relational schema are in fact defined in terms of the global XML schema, thus following the LAV technique.

To handle the translation of XQuery constructs referring to a document order, we assume that among elements belonging to the same document, the elID virtual field in the virtual schema reflects this order. To actually return query results in correct document order, all data sources must provide the correspondent of an order-reflecting element ID.

5.4.2 Translating simple path expressions

Let us denote by \( T(E) = (S(E), F(E), W(E)) \) the translation function that, for a given expression \( E \), computes the select, from and where parts of the corresponding SQL query.

Rule \( TR_1 \) translates simple path expressions denoting a document root. \( E \) may be either a string constant, or a more complex XQuery expression, whose SQL translation is a row sub-query returning one string:

\[
TR_1 \quad T(\text{document}(E)) = \\
\begin{align*}
\text{select} \quad & d.\text{docID} \\
\text{from} \quad & \text{Document} \; d, \; \text{URI} \; u, \; \text{Value} \; v \\
\text{where} \quad & d.\text{docURIID} = u.\text{uriID} \; \text{and} \; u.\text{uriValID} = v.\text{vID} \; \text{and} \\
\text{v.val} & = T(E)
\end{align*}
\]

The following rules show how to translate path expressions, given the translation of the path shorter by one step. \( TR_2 \) shows how to add a final “child” step to the SQL translation of an expression; again, there are two slightly different cases, according to the name test being a constant or resulting from a complex expression. We show the rule for the most general case; if \( E_2 \) is a constant, simply replace \( T(E_2) \) with the constant. Since the path
expression is correctly typed, we know that $S(E_1)$ must be an element ID, and that $T(E_2)$ must return a single row with one string column.

$$\begin{align*}
\text{TR}_2 & \quad T(E_1/E_2) = \\
& \quad \text{select e.e1ID} \\
& \quad \text{from } F(E_1), \text{ Child c, Element e, Qname q, Value v} \\
& \quad \text{where } W(E_1) \text{ and c.pid}=S(E_1) \text{ and} \\
& \quad c.c1ID=e.e1ID \text{ and e.elQNameID}=q.qNameID \text{ and} \\
& \quad q.qnValID=v.v1ID \text{ and v.val} = T(E_2)
\end{align*}$$

We move on to translate the expressions whose final step is a “descendent” step, denoted by “//”. Note the use of the TransClosure table to express arbitrary depth nesting.

$$\begin{align*}
\text{TR}_3 & \quad T(E_1//E_2) = \\
& \quad \text{select e.e1ID from } F(E_1) \text{ TransClosure tc, } \\
& \quad \text{Element e, QName q, Value v} \\
& \quad \text{where } W(E_1) \text{ and } S(E_1)=tc.pid \text{ and} \\
& \quad tc.c1ID=e.e1ID \text{ and e.elQNameID}=q.qNameID \text{ and} \\
& \quad q.qnLocalID=v.v1ID \text{ and v.val} = T(E_2)
\end{align*}$$

Rule TR$_4$ shows how to translate a final “attribute” step; this rule also has two variants, depending on whether the attribute name is a string constant or results from a different expression.

$$\begin{align*}
\text{TR}_4 & \quad T(E_1/@attName) = \\
& \quad \text{select a.attrID} \\
& \quad \text{from } F(E_1), \text{ Attribute a, Value v} \\
& \quad \text{where } W(E_1) \text{ and a.attrElID}=S(E_1) \text{ and} \\
& \quad a.attNameID=v.v1ID \text{ and v.val} = T(@attName)
\end{align*}$$

Rule TR$_5$ translates a dereferencing step. Note that in the SQL translation, the query translator has inserted the name of the ID attribute in the target element, $id$, although it was not supplied in the original XQuery expression; this information is taken from the DTD of the document being queried. We only show the case when the attribute name is a constant; if it results from a more complex expression, the corresponding sub-query would replace attName in the translation.

$$\begin{align*}
\text{TR}_5 & \quad T(E_1/@attName \rightarrow elName) = \\
& \quad \text{select e.e1ID from } F(E_1), \text{ Attribute a1, Value v1, Value v2,} \\
& \quad \text{Element e, QName q, Value v3, Attribute a2, Value v4, Value v5} \\
& \quad \text{where } W(E_1) \text{ and a1.attrElID}=S(E_1) \text{ and} \\
& \quad a1.attNameID=v1.v1ID \text{ and} \\
& \quad v1.val=attName \text{ and a1.attrValID}=v2.v1ID \text{ and e.elQNameID}=q.qNameID \text{ and} \\
& \quad q.qnLocalID=v3.v1ID \text{ and v3.val}=elName \text{ and a2.attrElID}=e.e1ID \text{ and} \\
& \quad a2.attNameID=v4.v1ID \text{ and v4.val}=id \text{ and a2.attrValID}=v5.v1ID \text{ and} \\
& \quad v2.val=v5.val
\end{align*}$$
In general, the results of path expressions should come in document order; SQL queries, however, do not guarantee result order, unless an explicit ORDER BY clause is added. Since we require element IDs to reflect document order, if we need to enforce the result order, we add, for example, to $T(E_1/E_2)$, “order by $S(E_1)$, e.elID”. Even if $T(E_1)$ was already sorted on $S(E_1)$, after the extra joins the ordering needs to be re-established. These ordering steps are very inefficient from a database point of view, if the expression appears within an unordered() operator, the order can be ignored.

Translating the range operator  Rule TR₆ shows the SQL translation for a range path expression step.

$$TR₆ \ T(E[range \ n \ to \ p]) = \text{with } V \text{ as } (T(E))$$

$$\text{select } x.\text{elID} \text{ from } V \ x$$

$$\text{where}$$

$$\text{select count(*) from } V \ y \ \text{where } x.\text{elID} > y.\text{elID} \geq n$$

$$\text{and}$$

$$\text{select count(*) from } V \ y \ \text{where } x.\text{elID} > y.\text{elID} < p$$

This rule can be easily extended to cases when the order results from a cross-product. The comparison $x.\text{elID} > y.\text{elID}$ needs to be replaced with the proper comparisons for establishing that $(x_1, x_2, \ldots, x_n) > (y_1, y_2, \ldots, y_n)$ in the sense of the desired lexicographic order.

5.4.3 Translating FLWR expressions

Recall that in a FLWR expression, the for clause produces tuples of bindings for the variables in the query, the where clause poses conditions that discard some of these tuples, and the return clause uses the tuples of bindings that satisfy the selection conditions to construct
the result, either under the form of complex structured XML elements or as tuples of flat values.

Rule TR7 translates a simple FLWR expression, whose for and where clauses contain only simple path expressions, and that returns all the variables bound in for-where. Figure 5.7 shows a translation example.

\[
\text{TR}_7\quad T(\text{for } x_1 \in E_1, x_2 \in E_2(x_1), \ldots,
\quad x_n \in E_n(x_1, \ldots, x_{n-1})
\quad \text{where } C(x_1, \ldots, x_n) \text{ return } x_1, \ldots, x_n =
\quad \text{select } S(E_1), S(E_2) \ldots S(E_n)
\quad \text{from } F(E_1), \ldots, F(E_n)
\quad \text{where } W(E_1) \text{ and } \ldots\text{ and } W(E_n) \text{ and }
\quad \text{exists } T(C(x_1, \ldots, x_n))
\]

To respect the semantics of XQuery, the evaluation of such a path expression should result into \((x_1, x_2, \ldots, x_n)\) tuples sorted in the lexicographic order derived from the order in each \(E_i\). If the order of tuples is important, a final sort by \(x_1\) ascending, ..., \(x_n\) ascending is added.

To explain the translation of queries returning newly constructed XML elements, we first show how to translate a single element constructor. An element constructor appearing in an XQuery query may depend on variables that have been previously bound in the query. To correctly structure and order the information needed in order to build an XML element, we borrow the sorted outer union approach presented in [107]. Translation rule TR8 can be applied, with the following notations.

- Let \(E_0\) be the part of the query providing bindings for the query variables \(\overline{x} = x_1, \ldots, x_n\) (in the case of FLWR expressions, the for-where clauses); tuples resulting from \(T(E_0)\) contain bindings for the variables in \(\overline{x}\).
- We denote the tag of the outermost result element by \(E_1(\overline{x})\).
- Let \(E_2, \ldots, E_{2k}\) be the expressions providing names for the element’s attributes, while \(E_{3, \ldots, E_{2k+1}}\) provide attribute values.
- Let \(H_1, \ldots, H_j\) be the expressions corresponding to the result element’s children.
- Finally, let \(G_1, \ldots, G_l\) be the elementary expressions (no element constructor) appearing in the \(E_i\)s and \(H_i\)s that really depend on the bound variables \(\overline{x}\); each \(G_i\) provides values to be used as attribute or element names, attribute values, or character data.

The first union term at right in TR8 contains the translation of the for-where clause, padded with nulls; this term contains only the variable bindings, and is labeled 0. Each of the next \(l\) terms retrieves the information corresponding to one of the \(G_1, \ldots, G_l\) path expressions.
for $x_1$ in document("patients.xml")//patient,
    $x_2$ in document("records.xml")//record
where $x_1$/@SSno/data()=x_2/patientSSno/data()
return
<medFile>
    <personal>
        <patName>{$x_1/name/data()}</patName>
        <patAddress>{$x_1/address/data()}</patAddress>
    </personal>
    <medical>
        {string-value(x_2/entry)}
    </medical>
</medFile>

Figure 5.8: Sample query and its tagging template.

As a by-product of the translation from normalized XQuery to SQL, a tagging template is constructed, to inform the tagger module how to structure data from the sorted tuples into an XML result. As an example, consider the normalized query in figure 5.8, and its corresponding tagging template. Running this query on our medical database yields one binding for the variables $x_1, x_2, k=0$ (no attributes in the returned element), $j = 2$, $H_1$ is the element constructor with tag personal, $H_2$ is the element constructor with tag medical; $l = 3$, $G_1$ is $x_1/name$, $G_2$ is $x_1/address$, $G_3$ is string-value($x_2/entry$).

We briefly explain the construction of the tagging template, during the translation of a complex FLWR expression. First, we translate the FLWR expression having the same for and where clauses as the complex expression, and returning only the bound variables: this yields the sub-query $T(E_0)$ in rule TR₈ (an example for $T(E_0)$ is the SQL query in figure 5.7). Next, the structure of the returned element is copied into the tagging template as follows.
Constants appearing in the result are copied as such in the template. Every $G_i$ in the result yields: a new union term to the sorted union query, joining the result of the for-where block and the translation of $G_i$; and an elem entry in the template. This amounts to multiple outer joins between the bound variables and the expressions retrieving components of the result that depend on these variables.

Each block of the sorted union query will be rewritten and handed to the execution engine. The result metadata (column number, types and names) stay the same in the queries over the virtual and real schemas; therefore, the column information contained in the tagging template can be used by the tagger to structure the result. For every tuple labeled 0, the tagger starts a new element; then, by following the label field, it decides where to fill in the value from the non-null g<sub>i</sub> column. The tagger runs in linear time and constant space [107].

5.4.4 Difficulties for the translation process

Various features of XQuery are difficult or impossible to translate to SQL, no matter what relational schema is used for the target query, because the inner logic of these language features is incompatible with the semantics of SQL.

**Constants** A very simple example is that of scalar constant expressions that are legal XQuery queries, but do not have a proper equivalent in SQL.

**Order-related operations** Document order-preserving operators and the range predicate deserve a special discussion. Note that the data sources must provide order-reflecting element IDs for the elements they contain, otherwise, the document order of the documents in the global schema cannot be reconstructed. In the case of relational data sources, this ID needs to be reflected as a stored attribute. In the case of DOM sources, the DOM wrapper assigns order-reflecting IDs to the elements accessed through one of its restricted tables. The wrapper ensures that during the execution of a given query, such an ID consistently refers to the same elements. For example, if within a single query, the variable $p_1$ is bound to the expression document("patients.xml")//patient/name/data()="Alec, Mary") and the variable $p_2$ to the expression document("patients.xml")//patient/@ssNo/data()="101"], on our sample database, $p_1 = p_2$.

A first thing to note is that the order of the result in a simple XQuery expression, without nesting, may come only from some document or data order, a cross-product or concatenation of such orders. Assuming that the data sources contain persistent, order-reflecting element IDs, the result order of such a simple expression may be captured by an SQL query as we will show in the following section. However, this query involves aggregation and its rewriting is more complex. Thus, even if the document order is within the expressive power of SQL, operators related to order make query translation cumbersome or may even make it fail.

Second, an XQuery query may involve an order derived from the result of a black-box XQuery function, as we already explained. Consider a query retrieving the first to third
hospitals from the list returned by the listHospitals function, introduced in section 5.3.2. The order of elements within this list may be immaterial, i.e. not reflected in any data item of the returned elements, and therefore cannot be captured by a single SQL query ending with a ORDER BY step. Again, such queries can only be executed by several translation-execution passes.

**Run-time access to type** Another most important class of untranslatable XQuery constructs is that of run-time operators that access or cast an element’s type. XQuery provides a treat() operator that attempts to cast the type of a variable to a type more specific than its statically-determined type; thus, treat() performs downcasts. The explicit cast operator $e :- \tau$ is used to convert $e$ to a type that is more general than its static type. Finally, a type-switch operator compares the run-time type of a variable with several types and evaluates the expression that corresponds to the correct type. All three operators can throw run-time errors. The problem with these operators is that for data that is stored in relational data sources, it is difficult to “craft” a type, in the sense of the XQuery type system.

**Recursion** The recursion allowed by XQuery is more powerful than the linear recursion from the SQL standard; thus, queries involving XQuery functions that are non-linearly recursive cannot be translated to SQL.

**Union types** XQuery allows heterogeneous type unions; in SQL, these expressions can only be translated by outer union queries, padding the missing fields with nulls.

**Node identifiers** XQuery cannot be properly executed via SQL from the point of view of the required semantics for node identifiers. As we explained, if an XQuery query copies a node from an input document to the output, the default copy semantics assigns a new identity to the produced node. The query translation methodology that we use, since it is based on SQL, is oriented on values, not identity; therefore, the data extracted by the query can only contain the identifiers of nodes in the input data source. To obtain the behavior that is correct with respect to the semantics of XQuery, a run-time component should be included in the tagger module, that generates “new” element IDs respecting the correctness conditions provided in the XQuery specification [124].

**Order-related operations** Document order-preserving operators and the range predicate deserve a special discussion. Note that the data sources must provide order-reflecting element IDs for the elements they contain, otherwise, the document order of the documents in the global schema cannot be reconstructed. In the case of relational data sources, this ID needs to be reflected as a stored attribute. In the case of DOM sources, the DOM wrapper assigns order-reflecting IDs to the elements accessed through one of its restricted tables. The wrapper ensures that during the execution of a given query, such an ID consistently refers to the same elements. For example, if within a single query, the variable $p_1$ is bound
Figure 5.9: Relationships between local data sources and users’ view of the data in Agora.

to the expression `document("patients.xml")//patient[name/data()="Ale, Mary"]` and the variable $p_2$ to the expression `document("patients.xml")//patient[@ssNo/data()="101"]`, on our sample database, $p_1 = p_2$.

A first thing to note is that the order of the result in a simple XQuery expression, without nesting, may come only from some document or data order, a cross-product or concatenation of such orders. Assuming that the data sources contain persistent, order-reflecting elementIDs, the result order of such a simple expression may be captured by an SQL query as we will show in the following section. However, this query involves aggregation and its rewriting is more complex. Thus, even if the document order is within the expressive power of SQL, operators related to order make query translation cumbersome or may even make it fail.

Second, an XQuery query may involve an order derived from the result of a black-box XQuery function, as we already explained. Consider a query retrieving the first to third hospitals from the list returned by the `listHospitals` function, introduced in section 5.3.2. The order of elements within this list may be immaterial, i.e. not reflected in any data item of the returned elements, and therefore cannot be captured by a single SQL query ending with a ORDER BY step. Again, such queries can only be executed by several translation-execution passes.

5.5 Relational query rewriting

Until now, we have shown how to normalize XQuery queries, and how to translate them into SQL over the virtual schema, when the translation is possible. During normalization and translation, the local data sources are ignored, and all transformations are performed on the user query. This section describes the relational query rewriting phase, in which we finally connect the query to the data sources; the query is rewritten using the descriptions of local data sources as views over the generic relational schema.

First, we comment on the conceptual relations between local sources and the global
schema in Agora. In local as view (LAV) data integration, each data source is described as a materialized view over the global schema. In Agora, all data sources are presented to the execution engine by their respective wrappers as tables with binding patterns. Syntactically, these tables are described as SQL views over an internal virtual relational schema, corresponding to the generic structure of an XML document. Conceptually, every relational table $T_i$ is defined as a materialized view over one or several XML documents present in the user’s view of the data, as shown in figure 5.9. However, XML data sources are propagated as such in the XML global schema (this is the case, for example, for “doc1.xml” in figure 5.9). This limitation comes from the expressive power of the language used for view definitions. Indeed, in SQL one cannot define hierarchical, nested data as a view over a set of relations. While answering XML queries using XML materialized views is an interesting area of active research, we do not fully address it in this work. Defining the DOM sources as views over the relational generic schema only enables us to process together data from relational and XML data sources; it does not enable us to express the structure of the XML sources as views over the XML global schema.

Finally, it is interesting to note that relational and XML data sources may have overlapping contents, as is the case with $T_3$ and “doc1.xml” in figure 5.9. As explained in section 1.2, the capability of taking advantage of redundant storage is one of the benefits of the local as view approach.

We illustrate the rewriting process on the database shown in section 5.2.2.

5.5.1 View definitions for relational sources

Figure 5.10 shows part of the view definition for the $S_1://W_1/Entry$ table, where $W_1$ is the wrapper used to publish the Entry table on the site $S_1$. This view relates the information in the table to data items from the “records.xml” document. The first three tables in the from clause, and the first three predicates in the where, give the name of the document. The next few joins represent the information that the root element of the document, el, has a descendent record element. The joins on lines 9, 10 and 11 relate the record element to its child patientSSno element; the value of this element is represented by v4, and note that v4.val is retained by the select clause; thus, we have obtained one attribute of the Entry table. The rest of the view definition shown derives, in a similar manner, the values of the entryID and date fields in the table. We did not show how to relate the diagnosis, symptoms and medication attributes to their counterparts in the XML global schema; this is done the same way as for the attributes shown.

Note that this view definition does not export element IDs for the elements it contains. These IDs do not exist in the data source. Therefore, queries that need these IDs, e.g., to test for node equality, or to order the result following the document order in “records.xml”, cannot be answered. In order to be able to process such queries, the table should contain one or several fields corresponding to the IDs of the elements involved in its view definition.

The Entry table contains information from one document only. Of course, in general, a relational view can contain data resulting from a join of several documents.
\begin{verbatim}
S1://W1/Entry
select v4.val as patientSSno, v7.val as entryID, v9.val as date, ...
  e1.e1ID as e1, e2.e1ID as e2, e3.e1ID as e3, e4.e1ID as e4, e5.e1ID as e5, e6.e1ID as e6
from Document d1, URI u1, Value v1, TransClosure tc1, Element e1, QName q1,
  Value v2, Child c1, Element e2, QName q2, Value v3, Child c2, Value v4,
  Child c3, Element e3, QName q3, Value v5, Attribute a1, Value v6,
  Value v7, Child c4, Element e4, QName q4, Value v8, Child c5, Value v9, ...
where d1.docURIID=u1.uriID and u1.uriValID=v1.vld and v1.val="records.xml" and
d1.rootElemID=tc1.pID and tc1.clID=e1.e1ID and e1.e1QNameID=q1.qNameID and
  q1.qnLocalID=v2.vld and v2.val="record" and c1.pID=e1.e1ID and
c1.clID=e2.e1ID and e2.e1QNameID=q2.qNameID and q2.qnLocalID=v3.vld and
  v3.val="patientSSNo" and c2.pID=e2.e1ID and c2.cValID=v4.vld and
  c3.pID=e1.e1ID and c3.clID=e3.e1ID and e3.e1QNameID=q3.qNameID and
  q3.qnLocalID=v5.vld and v5.val="entry" and a1.attrElID=e3.e1ID and
  a1.attrNameID=v6.vld and v6.val="entryID" and a1.attrValID=v7.vld and
  c4.pID=e3.e1ID and c4.clID=e4.e1ID and e4.e1QNameID=q4.qNameID and
  q4.qnLocalID=v8.vld and v8.val="date" and c5.pID=e4.e1ID and
  c5.cValID=v9.vld ...
\end{verbatim}

Figure 5.10: View definition for the Entry table.

5.5.2 View definitions for DOM sources

In section 2.2.6 we have shown how DOM-compliant data sources are modeled as a collection of tables with binding patterns. A special wrapper, that we call the DOM wrapper, handles all these tables. Of course, the content of the tables is not materialized a-priori as a set of tuples; these tuples are only produced when a table is queried.

For example, the \textit{getElementsByTagName} method of the Element DOM class returns the list of all descendents of the element that have the given tag. In our framework, this method is modeled as a table \textit{getElementsByTagName}(pID\textsuperscript{a}tag\textsuperscript{b}cID\textsuperscript{a}); in which: pID is the ID of the element receiving the method call, tag is a string that provides the tag name, cID is the ID of each element returned.

The table S2://W2/getElementsByTagName table is described as a view over the virtual generic relational schema, just like the tables from relational data sources:

\begin{verbatim}
S2://W2/getElementsByTagName
select tc.pID as pID, v.val as tag, tc.clID as cID
from TransClosure tc, Element e, QName q, Value v
where tc.desc=e.e1ID and e.e1QNameID=q.qNameID and
  q.qnLocalID=v.vld
\end{verbatim}

We make here the following remarks. The tables exported by the DOM wrapper, and as a consequence, the corresponding view definitions, are \textit{generic}, in the sense that they are the same for every XML document that the wrapper manages. Thus, the \textit{content of
the XML data sources is not properly reflected by the view definitions allowing them to be accessed. When answering a query over the “records.xml” document, there is no way of telling which XML data sources (that is, which virtual tables on a DOM wrapper) contain useful information.

To solve this problem, we attach to each wrapper the list of the documents it holds. Thus, as we explain in the next section, in order to rewrite a query, we only use the view definitions from the wrappers that give access to the documents involved in the query.

This framework still does not allow us to define an XML source as an arbitrary view over an XML global schema, as explained in section 5.5.

5.5.3 View definition for a XML full-text index

A full-text index (FTI) is an tool commonly used to query document data, replacing or complementing a regular query language. At the language level, XQuery provides a contains textual predicate, applying on a node and a word, and returning true if the textual image of the node contains the word; the sample query in figure 5.2 uses this predicate. At the implementation level, there are two basic options for evaluating queries that mix navigation and keyword search:

- follow the navigation path, and test the presence of the keyword in the text nodes that are attained; in the sample query using contains, the path would be doc("records.xml")//record

- if a materialized FTI is available, use it to reach directly the elements containing the keyword, and then filter this element set to retain only those that are on the required path

Textual search on XML documents has two important benefits. From the user’s point of view, when the structure of the data is unknown or highly irregular, a contains predicate is convenient to express simple queries. From the query processor point of view, the presence of a materialized FTI yields alternative query evaluation strategies.

Independently of the way the index is materialized and stored, in Agora we view it as a table with a restricted access pattern, as described in section 2.2.7. This table is also described as a view over the virtual generic schema. The rewriting module will thus understand that the FTI provides an alternative access path to some of the elements present in the document, and use it as a regular view.

To that purpose, we need to augment the virtual generic schema with one table contains(eID, word). A FTI that indexes all diagnosis elements from the “records.xml” document will thus be described by the following view definition:
In the absence of a materialized index, the only evaluation strategy for keyword search queries is (1) call the string-value() function to obtain the text image of the node and (2) apply a string matching function occurs(string, word). In our context, both these functions have to be published as tables with binding patterns at least on one one of the available LeSelect servers. Furthermore, string-value(), as well as other built-in XQuery function, can only be applied on nodes that come from documents published by the same wrapper, since the element IDs have no meaning outside the wrapper that produced them. Thus, the site and the wrapper where a table is published are used to decide whether the view definition corresponding to that table can be used to answer the query.

5.5.4 Rewriting algorithm

Given the translated query and the view definitions, a query rewriting algorithm is used to produce a SQL query to be sent to the data sources. We make the closed-world assumption [1], i.e., we assume that the view extensions contain all tuples corresponding to the query defining the view over the generic schema. The current implementation of the Agora system uses an equivalent query rewriting algorithm, and therefore in this section equivalent query rewriting is considered. In a large-scale data integration application, maximally-contained rewriting algorithms may be a better choice, since different data sources may contain different subsets of the data presented to the user.

A survey of query rewriting algorithms is provided in [58]. In our context, we rewrite SQL queries on the virtual schema into SQL queries on the relational tables representing the data sources; it has been shown in [24] that two SQL queries are equivalent only if they are identical up to renaming and reordering. This results provides the basis for our query rewriting algorithm. However, several modifications need to be made given the particular context of our application.

Rewriting example  We illustrate query rewriting on the sample query below, that joins information from the “patients.xml” and “records.xml” documents:

S₃://W₃/FTI
select d.docID as docID, e.eID as eID, c.word as word
from Document d, URI u, Value v1, TransClosure tc, Element e, QName q, Value v2, Contains c
where d.docURIID=u.uriID and u.uriValID=v1.vlID and v1.val=’records.xml’ and
d.rootElemID=tc.pID and tc.cID=e.eID and e.eQNameID=q.qNameID and
q.qnLocalID=v.vlID and v.val=’diagnosis’ and e.eID=c.eID
for $x$ in document("patients.xml")/patient, $y$ in document("records.xml")//record, $z$ in $y$/entry
declare $x$/@ssNo/data() = $x$/@ssNo/data() and $z$/diagnosis/data() = "flu"
return <res>{
  $x$/name/data()
  $z$/date/data()
}</res>

This query returns for all patients that had a flu, their names and the date of the illness. Following the translation methodology of FLWR expression, this query will be translated into a sorted union of three SQL queries: each union term has five attributes, namely $xID$, $yID$, $zID$, name, date, and label.

- The first union term represents the join of the translation of subtrees selected by the for and where clauses in the two documents. This term enforces the conditions in the where clause, and its select clause returns the element identifiers corresponding to the variables $x$, $y$, $z$, and fills in the fields name and date with nulls. The label column is set to 0.

- The second union term is obtained from the first by extending it with the joins that are necessary in order to retrieve the name of $x$ and its value. In this term, the value of the name is inserted in the name column, while the date field is left null, and the value of label is 1.

- The third union term is obtained in the same manner from the first, and uses the binding for $z$ provides values for the date column; the associated value of label is 2.

The result of the union is sorted on the label field. To rewrite this query, we rewrite the three union terms, and perform the union and the sorting step at the end. Let us now focus on the rewriting, say, of the third union term. It is easier to understand the rewriting by considering the tree representation of the query shown at the top of figure 5.11. We have shown the tree structure of the patterns selecting data from the two documents; edges annotated with “//” correspond to a TransClosure connection between two nodes, while “@” designates node-attribute links, following the XPath notations; the rest of the links correspond to parent-child relationships. The dashed line shows the join condition between the two documents, on the values of the patient’s social security number.

In this figure, we have circled together fragments of the patterns in the two XML documents that are matched with some existing view definitions describing the existing data sources. The complete rewriting of this union term appears at the bottom of figure 5.11. To the six tables in the from clause of this rewritten query correspond to the six circled numbers in the tree representation of the query.

**Rewriting algorithm** The complete rewriting algorithm applied to each union term $Q$ can be outlined as follows. First, the documents referenced by the query $Q$ are extracted
Figure 5.11: Tree representation of a union term corresponding to the sample query, and its SQL rewriting on the real data sources.
from the query; call this set of documents $docs(Q)$. We construct a set $V$ of candidate views, which are those that reference documents from $docs(Q)$. A subset \{\(v_1, v_2, \ldots, v_k\)\} of the candidate view set $V$ can be used to answer $Q$ only if the documents referenced by this view subset contain all the documents from $docs(Q)$. We attempt to rewrite $Q$ using all such subsets of $V$.

In the presence of path expressions using "/" steps, translated in SQL using the TransClosure table, special care needs to be taken to ensure that all valid matches are recognized. To that purpose, the DTD of the document involved should be used.

- If both $v$ and $Q$ are defined using TransClosure, or neither of them use it, $v$ can be used to answer $Q$ if and only if we can find an isomorphism between $v$ and part of $Q$.

- If $v$ is defined using TransClosure, but the query only uses child and attribute path steps, we need to consult the DTD in order to decide whether the view contains exactly the nodes required by the query, or a superset of these nodes. Consider a view storing patientSSno elements at any depth within "records.xml", and a query asking for document("records.xml")/records/record/patientSSno. Using a DTD, the rewriter might infer that patientSSno elements can only be found on the path shown in the query. Without the path information present in a DTD, the view may contain a superset of the required data, and therefore cannot be used.

- If $v$ is defined with transitive closure, but $Q$ asks for a specific path in the data, the usability of the view depends on the presence of a DTD. Let $v$ store the patientSSno elements at any depth within "records.xml", and let $Q$ be

  \[
  \text{document("records.xml")/records/record/patientSSno}
  \]

Using a DTD, the rewriter might infer that all patientSSno elements are found only on the path shown in $Q$. Thus, a query fragment joining the Document virtual tables with three pairs of Child-Element tables can be rewritten using a view of only three tables: Document, TransClosure, Element. If the DTD is unavailable, or if patientSSno elements could be found on several paths in the document, the view can be mapped to the last Element in the query path; we still have to check that the path going backwards, from the patientSSno element to the document root, is the one required in the query. To that purpose, we need to use other views; for example, the DOM wrapper exports, beside the getChildren table that allows downward navigation, the getParentNode table that can be used to go back from an element to its parent.

Rewriting complex SQL queries The algorithm that we described can handle simple select-from-where queries. What we would need is a general algorithm for rewriting of arbitrary SQL queries using arbitrary SQL views. In the case of nested queries, we require an equivalent rewriting of each elementary block and then re-compose the rewritings into a nested query similar to the incoming one. This approach may not yield all the solutions, but is guaranteed to find only correct solutions. In [20], algorithms are given for SQL query
for $p$ in document("patients.xml")//patient, 
  $a$ in $p$/address 
where $p$/name="Doe, John"
return closestHospitals ($a$, "hospitals.xml") 
  //hospital [range 1 to 3]

Figure 5.12: Query necessitating intermediate XML result materialization

rewriting using materialized views, in the case when the query and the views are conjunctive
SQL queries. Algorithms proposed in [108] solve the rewriting problem in the cases when
the query contains aggregation and unions, and the views contain aggregation; recent work
presented in [49] focused on a scalable query rewriting algorithm in the context of SQL query
optimization.

5.6 Translating queries with intermediate XML results

In this section, we explain how Agora’s capacity to query native XML documents is used for
translating queries necessitating the materialization of intermediate XML results.

Consider the normalized query in figure 5.12, and its representation as an operator tree.
In this tree, $PE_1$ corresponds to document("patients")//patient, $PE_2$ is $p$/name, and $PE_3$ is
$p$/address; note that the bindings of $p$ from $PE_1$ need to be passed to $PE_3$. By examining
a node, it can be decided whether (a) this expression cannot be executed by a relational
processing system, and it is not a problem of intermediate XML results; in this case, the
whole query is untranslatable; (b) this node does not necessitate materialization of its inputs;
or (c) this node does necessitate the materialization of one or more of its inputs; in this case,
XML materialization nodes are inserted in the query tree between the current node and its
appropriate descendents. In this example, there is one such materialization node, as input
to the range operator.

At this point, the input query is partitioned into two subqueries. $Q_1$ extracts the patient
and passes the proper binding for $p$ as input to the function, which results in an XML
document. This document is assigned to a DOM wrapper described in section 5.5.2, as a
special temporary data source, given a new name, and provided as input to $Q_2$. Next, the
subqueries are sorted in the order dictated by the data sources they produce/consume; $Q_1$,
then $Q_2$, are translated into SQL queries on the generic schema, rewritten and executed.
The fact that one input is a temporary document does not hinder $Q_2$’s rewriting, since the
DOM wrapper publishes *generic* view definitions, in which the XML document name is a simple attribute and can be selected on.

## 5.7 Related work

### Data integration systems

The Tsimmis data integration system was the first to propose integrating heterogeneous data sources under the a semistructured global schema (the format was OEM) [44]. However, the Tsimmis project, as well as Garlic [54], Disco [113], and Yat [30] all adopt the GAV approach, and therefore do not compare directly to our system. The Information Manifold [65] data integration system has a LAV architecture; however, the local and global schemas are relational. The Ozone system [67] allows storing object-oriented and OEM data in a single repository, and querying them seamlessly using a hybrid of OQL and Lorel [3]. The benefits of optimizing the structured and unstructured parts of the query together are obvious; however, it is not always possible (for performance and copyright reasons) to store all the data in a single repository, and mediation (over independent, autonomous sources) still has to be performed.

### Querying XML views of relational data

This approach corresponds to a GAV architecture, when the global XML schema is defined as a view over the local relational data sources; therefore, our approach for translating XML queries into SQL queries on the data sources solve a different problem.

SilkRoute [35, 36] and XPERANTO [107, 106] focus on exporting relational databases under an XML interface. Since the mapping is done from tuples to XML, these projects adopt the GAV approach; also, they can only integrate relational data sources. In a work developed in parallel with ours, a translation methodology from XQuery to SQL is provided, in order to query XML views of relational data [106]. In contrast, our integration approach can handle diverse data sources, not only relational. The study in [35] investigates efficient ways of materializing a large XML document from the data contained in an RDBMS. In this context, a single sorted outer union SQL query may be suboptimal, and the authors describe a search space of several smaller SQL queries. We used the sorted outer union approach for several reasons. First, we expect that in a data integration setting, most queries return moderate-size results. Also, the search done in [35] is based on a RDBMS’s optimizer’s cost estimates for a given SQL query; in a centralized context, these estimates are easy to obtain. However, in a data integration context, it is difficult to get precise and comparable estimates from wrappers.

Microsoft’s SQL server provides a solution very similar to SilkRoute. As described in [101], SQL Server provides several “default” mappings from relational tables into XML documents, as well as a proprietary language for customizing this mapping. An interesting feature included in SQL Server is the ability to update the relational data behind an XML
virtual view, by providing read/write cursors over the XML view. The XML query language supported for the time being is XPath.

Relational storage schemas for XML documents

Several proposals have been made for computing “good” relational storage schemas for data contained in (a set of) XML documents. Let us denote the correspondences between parts of the XML documents to be stored and their relational counterparts by the generic term of mapping. In the existing approaches, the quality of the mapping is evaluated using the number of tables in the relational schema, data statistics, perhaps a query workload, the number of nulls required to represent the irregular XML data in a relational database etc.

The Stored [32] project uses a data mining algorithm for identifying regular structures in a collection of XML documents without DTDs. For each “stable” pattern found in the data, Stored allocates one table; the authors propose to store the data items that do not belong to any stable pattern in a semistructured storage, while we consider that all the XML data we intend to query is stored within the RDBMS. Whatever the storage for the overflow, exploring it to answer queries entails a serious performance hit, since there is no schema for that data; the author’s hypothesis is that the overflow graph is “often” small. For each mapping that the data mining algorithm discovers, translation rules are generated for pushing XML queries to the RDBMS. We note that the query language used does not construct new structure, it can only retrieve existing elements; also, to ensure losslessness of mappings, the mapping language is restricted (e.g. no joins): our method applies to a wider range of mappings, including those that join in a single table information from two different documents. Also, we do not forbid “lossy” mappings; if only part of the data contained in the XML document is available, we are still able to query it. If the relational views do not contain enough information to answer the query, this is discovered in the last relational rewriting phase described in section 5.5.

The study in [37] models XML documents as ordered, oriented graphs, and investigates several ways of storing first, links between nodes, and second, flat values. For storing link information, the simplest storage scheme materializes a single relation for all graph edges; a more refined approach partitions the edge table on the edge label. For storing values, a simple solution is to have one table per value type; such type partitioning is beneficial for domain-specific indexing. A second approach for storing values is to add in the edge tables one column per type of value possible; the destination field of every edge is either a node or a value of a given type. This paper does not provide algorithms for query translation. Performance measures on a relational RDBMS for several types of XML queries show that the partitioned edge approach, combined with in-lining values, offer the best performance. This is understandable since (a) self-joins on the complete Edge table are avoided and (b) joins to retrieve values are unnecessary if they are nested within elements.

Work done in [105] discusses three mappings derived from a simplified graph structure of the DTDs. Nodes in the graph are constructed for each element type; a parent-child or ID-IDREF relationship in the DTD is modeled by a directed edge from the parent node to the child node, therefore such graphs may have cycles. Three mappings are proposed,
ranging between fully normalized and highly redundant; the interest of the mappings is assessed by counting the number of joins required to reconstruct the document. We note that this measure does not account for dataset particularities or a query mix. The mappings that provide the best performance, called hybrid, allocates a table for one element type as soon as its corresponding node in the graph (a) has zero in-degree or (b) it belongs to at least a loop in the graph, and: either has in-degree greater than one, or can appear any number of times (“*”) in a parent. All elements that do not meet these criteria are stored within the closest ancestor for which a table was created. The authors sketch the translation from XML-QL queries on the shared mapping to SQL; our method generalizes their results for a larger family of mappings, and provides a more thorough investigation on translating features of a full-fledged XML query language.

Industrial products like Microsoft’s SQL Server [101] and Oracle 8i [87] provide (a) a simple default way of storing XML elements into tables, in general one table per element type and (b) some mechanism for customizing the mapping by selecting the subset of the XML document that should be stored. In the case of SQL Server, the language used for identifying one such subset is XPath.

The mappings produced by these methods fall in the category of LAV, since the relational storage is treated as a collection of (materialized) relational views over the XML document. Therefore, our translation methodology is general enough to cover these mappings.

**Query rewriting and unnesting**

There is a large body of research on rewriting queries using views, for a wide range of query and view definition languages. The decidability and data complexity of this problem for Datalog, conjunctive queries, and first-order queries has been studied in [1]. Efficient algorithms for rewriting conjunctive queries using conjunctive views have been proposed recently in [92] and [4]. In section 5.5 we had surveyed several results on rewriting SQL queries using SQL views.

While we used SQL as an intermediate format for query translation, the rewriting problem that we solve can be alternatively described as follows. Ignoring several problematic features (discussed in section 5.4.4), an XQuery query can be thought of as the composition, through iteration and join, of several path expressions, coming perhaps from different documents (note that these path expressions allow for a limited form of recursion, corresponding to “//”). The SQL definitions of the data sources as views over the global schema have the same expressive power. Thus, the core rewriting problem is to reformulate a path expression in terms of a set of materialized paths. Containment of path expressions has been first shown to be decidable in [41], for a restricted subset of the STROQL query language, and proven \(NP\)-hard. The class of XQuery path expressions that we consider is similar to the one used in [41].

More recently, query rewriting and query answering for regular path queries have been studied in [17, 15, 16] using a graph data model. The complexity of finding a query rewriting when the query and the views are regular path expressions is shown to be \(2\text{EXPTIME}\), and checking if the rewriting is equivalent is done in \(2\text{EXPSPACE}\) [17]. A regular path expression
language allowing for recursive inverse navigation is studied in [15, 16]; it is shown that query rewriting is still decidable for such path queries and views.

A recent study performed in parallel with our work addresses the problem of deciding containment among XPath expressions, in the presence of several types of semantic constraints [33]. The approach they take is very similar to ours, in that they also use an intermediate relational schema as a middle format for reasoning about containment. Like in Agora, the problem of containment for two path expressions is brought to the problem of containment of two relational queries. Unlike Agora, however, they assume set semantics; this may be more appropriate for the semantics of their (and our) virtual relational schema.

In parallel with our work, the problem of rewriting a tree query using materialized path expressions has been addressed in [7]. The input query is formulated in terms of an ontology; the data sources are related to the ontology by the means of a set of mapping rules. The rewriting problem they address, as well as our path rewriting problem, are conceptually related, although for us both the query and the materialized views are expressed in terms of path expressions. The mapping rules proposed in [7] are matched successively, following the query pattern, starting from the root of the pattern (a document node). The presence of these rules makes the rewriting process easier than in our case, due to the presence of variables in the rules. For example, if one view $v_1$ contains $\text{document("patients.xml")} // patient$, and view $v_2$ contains $ssNo$ attributes for these patients, the fact that $v_1$ and $v_2$ are to be used together is explicit in the mapping rules by the presence of a common variable in the view definitions. In contrast, if we attempt to rewrite our query in the generic schema using materialized views over this schema, it is up to the rewriter to determine whether or not two views can be combined to reconstruct information from the same path expression.

As explained before, the mechanism for describing the data sources as views over the XML global schema does not allow us to define XML views of this XML schema, since the view definition language used does not have enough expressive power. In a integration application in which all data sources are XML, an XML query rewriting algorithm using XML views would be needed. A rewriting algorithm for the Tsimmis Semistructured Language (TSL) has been proposed in [90]; TSL allows the creation of new tree structures as answers to the query, but forbids path expressions with recursion.

Unnesting rules for queries on object bases are presented in [28], and they are in spirit similar to the unnesting rules we present for XQuery queries in section 5.3. While there exist some similarities, the rules we present are different due to the differences between the object-oriented algebra used in [28], and the XQuery algebra [124]. However, the unnesting rules are only part of our framework; we use them to allow the processing of a XQuery query by a relational framework.

**Indexing XML**

Combining navigation and keyword search for querying semi-structured data has been studied in the context of SGML data [26, 27]. There are several proposals for XML indexing and for query processing techniques that use these indexes [81, 29, 5]. These indexes encapsulate both textual information (which words appears within which element) as well as structural
information (the position of the word within the element, the position of the element within
the document etc.) Such indexes could be exploited by our framework as well, as far as they
can be described as views over our virtual generic schema. One limitation may come from
the semantics encapsulated by the element identifiers; in our schema, we only assume the
element ID reflects the order of an element in a document, but we do not consider it allows
to decide, e.g., if an element is the parent of another element or not.

Structural indexes for semistructured data have been proposed, under the form of dataguides [47]
or T-indexes [82]. In our context, structural information is assumed available under the form
of DTDs.
Chapter 6

Conclusion

6.1 Summary

In this thesis, we have addressed several aspects of query processing in a data and program integration system. This work took place in the context of the LeSelect relational data integration system. The classes of resources that can be published and shared through LeSelect may be heterogeneous with respect to their storage, format, and query processing capabilities. Since LeSelect is based on the relational model, all resources are presented by the wrappers to the mediators as tables; to describe their limited access capabilities, we used the formalism of tables with binding patterns.

Modeling We described the modeling currently used in LeSelect in order to describe heterogeneous resources as tables with binding patterns. Furthermore, we provided conditions for the feasibility and of SQL queries (with bag semantics) over a set of tables with binding patterns. In the presence of bag semantics, the data sources accessed in order to answer a query must satisfy some soundness conditions, that we described.

Query optimization We have addressed the problem of cost-based query optimization over a set of tables with binding patterns. We have shown that the presence of binding pattern limitations entails a different search space than the one considered by a traditional optimizer. We have studied the size of this search space, both analytically for a restricted set of queries and source binding patterns, and through a series of experiments. Based on this analysis, we have proposed two classes of cost-based optimization algorithms using binding patterns, one based on dynamic programming, and another one that uses best-first search in order to produce some QEPs very fast, even if the size of the search space is important. We have compared the trade-offs between these algorithms through several experiments.

Query execution In the context of the LeSelect system, we have implemented a simple version of the dynamic programming algorithm using binding patterns; the simplifications that this algorithm makes are due to some restrictions in LeSelect’s execution model. Of
course, this algorithm takes into account the distribution and the query processing capabilities of the resources involved in a query.

Usage scenarios of LeSelect in real-life scientific integration applications have lead to the observation that expensive program invocation and transfers of large data objects make up a significant performance problem. To solve this problem, we have proposed a physical BindJoin algorithm, that uses cache to avoid repeated useless computation. The counterpart of the BindJoin is the BindAccess operator, that wraps the access to a restricted resource under the iterator interface adopted by all operators in LeSelect. We have designed the BindJoin and BindAccess operators so that they distribute well over different sites, and allow for a maximum of intra-operator parallelism. Furthermore, the BindJoin-BindAccess pair can take advantage of a steady tuple input, and duplicates in the input, to output tuples much faster than the access speed to the restricted resource. We assessed the advantages of the BindJoin operator through several simulations; its implementation in LeSelect is currently on-going.

**XML data integration**  Finally, while LeSelect has no global schema and follows the relational data model, we designed the Agora data integration system, that extends LeSelect to allow the integration of relational and DOM-compliant data sources under an XML global schema. To process together relational and tree-structured data sources using relational query optimization and query execution techniques, we relied on our modeling of heterogeneous resources as tables with binding patterns; note that DOM data sources are not stored as tables. To relate the data sources, modeled as tables, to the XML global schema, we describe them as views over an intermediate virtual relational schema, that corresponds to the generic structure of an XML document. Query processing in Agora is thus done as follows: an incoming XML query, formulated in terms of an XML global schema, is translated into a SQL query that LeSelect can process; this query is optimized and executed, and its resulting tuples are structured into the desired XML result, using a method previously proposed by a research team from IBM. The innovative query processing step in Agora is the query translation. We perform this translation in three stages: first, we apply simple equivalence rules to unnest the XML query, then, we translate it into SQL over the virtual generic schema, and finally, we rewrite it into an SQL query over the tables corresponding to the data sources. Given the data model and the complexity of the XML query language used, XQuery [126], not all features of the language can be properly translated and executed using a relational framework. We identify these translation difficulties, that are due to the mismatch between the semantics and the expressive powers of SQL and XQuery, respectively.

### 6.2 Directions for future work

The results presented in this thesis open two interesting research directions.

First, in the context of data and costly program integration, it would be interesting to extend the execution model of LeSelect so that it can handle complex QEPs that can only be executed when provided a set of bindings for some of their variables. In the current execution
model, only an access plan (i.e., a leaf in the QEP) may have access restrictions. Allowing complex parameterized QEPs may provide better execution performance, since instead of caching the result of a simple program or blob access, we might cache the result of the whole sub-query. The observation that a sub-query can be considered an expensive function has been made in [62]; using a cache for a complex subqueries is equivalent to materializing a view, and using it instead of recomputing. Query processing using materialized views has been addressed in several existing works [66]. It would be interesting to blend the caching and materialized view approaches in order to find efficient QEPs over a set of tables with binding patterns. Sub-query caching is easy to implement in a centralized context, but a distributed execution model for complex QEPs requiring input values is more difficult to find; as we explained in chapter 4, this execution model requires information passing that does not fit in the basic iterator model.

Second, in the Agora data integration system, due to the relational view definition mechanism, we were not able to define the XML data sources as views over the XML global schema. The goal of Agora was to process XQuery queries through SQL, and this limitation was due to the semantic mismatch between the two languages. If we consider the problem of integrating XML sources under an XML global schema, where the XML sources are described as materialized views over the XML global schema, the language used for these view descriptions should ideally have the expressive power of XQuery itself (in this new context, we disregard relational sources and abandon the relational query processing framework). Thus, the problem of rewriting an XML query using materialized XML views becomes more complex. When the standardization process is over and the semantics of the language are completely specified, this problem is likely to receive a lot of attention, due to the popularity of XML as a data integration format and to the strength of a W3C language standard.
Bibliography


[70] Chen Li. Computing complete answers to queries in the presence of limited access patterns. Research Report, Department of Computer Science, Stanford University, 1999.


[72] Chen Li and Edward Chang. Query planning with limited source capabilities. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, 2000.


