Università degli Studi della Calabria

Dipartimento di Matematica

Dottorato di Ricerca in Matematica ed Informatica

Settore Scientifico-Disciplinare INF/01 INFORMATICA

XXI Ciclo

TESI DI DOTTORATO

Advanced Tecniques and Systems for Data and Process Management

Luigi Granata

[illegible]

SUPERVISORI
Prof. Nicola Leone
[illegible]
Dr. Gianluigi Greco
[illegible]

COORDINATORE
Prof. Nicola Leone
[illegible]

ANNO ACCADEMICO 2008 - 2009

to my family
and
the people I love

Advanced Techniques and Systems for Data and Process Management.

Luigi Granata
Dipartimento di Matematica,
Università della Calabria
87036 *Rende, Italy*
email : *l.granata@mat.unical.it*

Sommario

Il lavoro di tesi viene suddiviso in due parti che trattano rispettivamente di tecniche avanzate e sistemi per la gestione dei dati e per il mining dei processi. Sono state affrontate problematiche relative all'efficienza della risposta alle interrogazioni su una base di dati, l'integrazioni di più sorgenti di dati, e la progettazione di un sistema per il mining di processi. In particolare, i principali contributi della tesi sono:

(1) Un nuovo modo di calcolare alberi di decomposizione di una interrogazione i cui *query plan* garantiscano un tempo di esecuzione al più di complessità polinomiale.

(2) Lo studio di tecniche e metodologie innovative, basate su logica computazionale, per i sistemi per l'integrazione di sorgenti informative e lo sviluppo di un prototipo che le implementi.

(3) Lo studio di tecniche ed algoritmi per il mining di processi e lo sviluppo di una suite che le implementi.

(1) Tecniche di risposta alle interrogazioni su basi di dati

Rispondere ad interrogazioni su una basa di dati può essere un processo molto costoso da un punto di vista computazionale. Per far fronte a questa problematica, in letteratura sono stati proposti vari approcci. Alcuni di essi sono basati su moduli per l'ottimizzazione delle interrogazioni che sfruttino le informazioni quantitative e statistiche sull'istanza della base di dati, mentre altre tecniche sfruttano le proprietà strutturali degli ipergrafi delle interrogazioni.

I nostri sforzi si sono rivolti in quest'ultima direzione estendendo il metodo di *hypertree decomposition*, considerato al momento il più potente tra quelli strutturali. Questa nuova versione, chiamata *query-oriented hypertree decomposition*, mira a gestire esplicitamente le variabili di output e gli operatori aggregati. Basandoci su queste nozioni, è stato implementato un ottimizzatore ibrido. Esso può essere utilizzato dai DBMS correntemente disponibili per poter calcolare i piani di esecuzione per le interrogazioni. Tale prototipo è stato integrato nel noto DBMS open source PostgreSQL. In fine questa estensione è stata validata attraverso una intensa fase sperimentale, portata avanti con PostgreSQL ed un noto DBMS commerciale, che mostra come entrambi i sistemi migliorino significativamente le loro prestazioni utilizzando le hypertree decomposition per l'ottimizzazione delle interrogazioni.

(2) Tecniche per l'integrazione di sorgenti informative

Per integrazione di informazioni si intende il problema di combinare i dati residenti in varie sorgenti informative, fornendo agli utenti una vista unificata di questi dati, chiamata *global schema*.

Il nostro lavoro è stato svolto all'interno del progetto INFOMIX. Il suo scopo principale è stato quello di fornire tecniche avanzate e metodologie innovative per per gli information integration systems. In breve, il progetto ha sviluppato una teoria, comprendente un modello esauriente ed algoritmi per l'integrazione delle informazioni ed l'implementazione di un prototipo di un sistema knowledge based traminte l'utilizzo della logica computazionale che integri i risultati della ricerca sull'acquisizione e la trasformazione dei dati. Un'attenzione speciale è stata dedicata alla definizione di un meccanismo per l'interazione dichiarativa da parte dell'utente e alle tecniche per la gestione di dati semistrutturati e sorgenti di dati incomplete o inconsistenti.

(3) Tecniche per il mining di processi

Nel contesto della enterprise automation, il *process mining* è recentemente emerso come uno strumento utilissimo per l'analisi e la progettazione di processi di business complessi. Lo scenario tipico per il process mining è dato da un insieme di tracce che registrano, tramite un sistema transazionale, le attività svolte durante più esecuzioni di un processo e dall'obiettivo ricavare in maniera (semi)automatica un modello che possa spiegare tutti gli episodi registrati nelle tracce.

Noi abbiamo sviluppato una Suite per le applicazioni del process mining con un'architettura aperta ed estendibile che introduce tre elementi innovativi per soddisfare i desiderata di flessibilità e scalabilità che sorgono negli scenari industriali attuali.

- Il concetto di "flusso di mining", i.e., essa permette di specificare delle catene di mininig complesse basate sulla connessione di task elementari.
- La costruzione di applicazioni interattive basate sulla possibilità di personalizzare tipi di dati, algoritmi e l'interfaccia grafica utilizzata per l'analisi.
- Scalabilità su grandi moli di dati.

Contents

Abstract

In this thesis, we deal with techniques for query answering exploiting structural properties, with the integration of multiple data sources, and with the design and the implementation of a suite for process mining. The main contributions of this thesis are the following:

(1) A new algorithm that computes a hypertree decomposition of a query, by accounting for grouping operators and statistics on the data. .

(2) The study of advanced techniques and innovative methodologies for information integration systems and a prototype implementation of a knowledge based system for advanced information integration, by using computational logic and integrating research results on data acquisition and transformation.

(3) The study of techniques and algorithms for process mining and a suite implementing them.

(1) Techniques for query evaluation

Answering queries is computationally very expensive, and many approaches have been proposed in the literature to face this fundamental problem. Some of them are based on optimization modules that exploit quantitative information on the database instance, while other approaches exploit structural properties of the query hypergraph.

Our efforts were carried on this last direction extending the notion of hypertree decomposition, which is currently the most powerful structural method. This new version, called query-oriented hypertree decomposition, is a suitable relaxation of hypertree decomposition designed for query optimization, and such that output variables and aggregate operators can be dealt with. Based on this notion, a hybrid optimizer is implemented, which can be used on top of available DBMSs to compute query plans. The prototype is also integrated into the well-known open-source DBMS PostgreSQL. Finally, we validate our proposal with a thorough experimental activity, conducted on PostgreSQL and on a commercial DBMS, which shows that both systems may significantly benefit from using hypertree decompositions for query optimization.

(2) Techniques for data integration systems

Information integration is the problem of combining the data residing at different sources, and providing the user with a unified view of these data, called *global schema*. Our work was performed within of the INFOMIX project. Its principal goal was to provide advanced techniques and innovative methodologies for information integration systems. In a nutshell, the project developed a theory, comprising a comprehensive information model and information integration algorithms, and a prototype implementation of a knowledge based system for advanced information integration, by using computational logic and integrating research results on data acquisition and transformation. Special attention was devoted to the definition of declarative user-interaction mechanisms, and techniques for handling semi-structured data, and incomplete and inconsistent data sources.

(3) Techniques for process mining

In the context of enterprise automation, *process mining* has recently emerged as a powerful approach to support the analysis and the design of complex business processes. In a typical process mining scenario, a set of traces registering the activities performed along several enactments of a transactional system is given to hand, and the goal is to (semi)automatically derive a model explaining all the episodes recorded in them.

We developed a novel Suite for Process Mining applications having an open and extendable architecture and introducing three innovative designing elements to meet the desiderata of flexibility and scalability arising in actual industrial scenarios.

- The concept of “flow of mining”, i.e., it allows to specify complex mining chains based on interconnecting elementary tasks
- Building interactive applications based on the possibility of customizing data types, algorithms, and graphical user interfaces used in the analysis.
- Ensuring scalability over large volumes of data.

Acknowledgments

- Leone, N. and Eiter, T. and Faber, W. and Fink, M. and Gottlob, G. and Granata, L. and Greco, G. and Kalka, E. and Ianni, G. and Lembo, D. and others. Data integration: a challenging ASP application. In *Lecture notes in computer science*, volume 3662, page 379. Springer 2005.

- Ghionna, L. and Granata, L. and Greco, G. and Scarcello, F. Hypertree decompositions for query optimization. In *IEEE 23rd International Conference on Data Engineering, 2007. ICDE 2007*. Pages 36-45.

- Ghionna, L. and Granata, L. and Greco, G. and Scarcello, F. Estensione del DBMS OPEN SOURCE PostgreSQL con un nuovo algoritmo per le interrogazioni. In *Proceedings of Conferenza Nazionale Software Libero (confSL 2007)*.

Introduction

Advanced Techniques and Systems for Data Management

Database Management Systems

The Data Base Management System (DBMS) is the foundation of almost every modern business information system. Virtually every administrative process in business, science or government relies on a data base. The rise of the Internet has only accelerated this trend; today a flurry of database transactions powers each content update of a major website, literature search, or internet shopping trip.

A data base management system is a very complex piece of system software. A single DBMS can manage multiple data bases, each one usually consisting of many different tables full of data. The DBMS includes mechanisms for application programs to store, retrieve and modify this data and also allows people to query it interactively to answer specific questions.

One of the most important features of the DBMS is its ability to shield the people and programs using the data from the details of its physical storage. Because all access to stored data is mediated through the DBMS, a data base can be restructured or moved to a different computer without disrupting the programs written to use it. The DBMS polices access to the stored data, giving access only to tables and records for which a given user has been authorized.

The data base concept originated around 1960, approximately ten years before the idea of a DBMS gained general currency. It originated among the well-funded cold war technologists of the military command and control, and so was associated with the enormously complex and expensive technologies of on-line, real-time, interactive computer applications.

Querying: optimization and data integration

Query optimization is a function of many relational database management systems in which multiple query plans for satisfying a query are examined and a good query plan is identified. This may or not be the absolute best strategy because there are many ways of doing plans. There is a trade-off between the amount of time spent figuring out the best plan and the amount running the plan. Different qualities of database management systems have different ways of balancing these two. Cost based query optimizers evaluate the resource footprint of various query plans and use this as the basis for plan selection.

The performance of a query plan is determined largely by the order in which the tables are joined. For example, when joining 3 tables A, B, C of size 10 rows, 10,000 rows, and 1,000,000 rows, respectively, a query plan that joins B and C first can take several orders-of-magnitude more time to execute than one that joins A and C first.

A SQL query to a modern relational DBMS does more than just selections and joins. In particular, SQL queries often nest several layers of SPJ blocks (Select-Project-Join) , by means of group by, exists, and not exists operators. In some cases such nested SQL queries can be flattened into a select-project-join query, but not always. Query plans for nested SQL queries can also be chosen using the same dynamic programming algorithm as used for join ordering, but this can lead to an enormous escalation in query optimization time. So some database management systems use an alternative rule-based approach that uses a query graph model.

Many approaches have been proposed in the literature to face the problem of choosing the optimal query plan. Some of them are based on optimization modules that exploit quantitative information on the database instance, while other approaches exploit structural properties of the query hypergraph.

On the other hand,the problem of combining the data residing at different sources, and providing the user with a unified view of these data, called *global schema*, arise and is faced by Data integration techniques. The interest in data integration systems has been continuously growing in the last years. Many organizations face the problem of integrating data residing at several sources. Companies that build a Data Warehouse, a Data Mining, or an Enterprise Resource Planning system must address this problem. Also, integrating data in the World Wide Web is the subject of several investigations and projects nowadays. Finally, applications requiring accessing or re-engineering legacy systems must deal with the problem of integrating data stored in different sources.

The design of a data integration system is a very complex task, which comprises several different issues such as dealing with heterogeneity of the sources, the mapping between the global schema and the sources, data cleaning and reconciliation, how to process queries expressed on the global schema and many other.

Advanced Techniques and Systems for Process Management

Process mining is a process management technique, that allow for the analysis of business processes based on event logs. The basic idea is to extract knowledge from event logs recorded by an information system. Process mining aims at improving this by providing techniques and tools for discovering process, control, data, organizational, and social structures from event logs. Moreover, it is possible to use process mining to monitor deviations (e.g., comparing the observed events with predefined models or business rules in the context of SOX).

Process mining techniques are often used when no formal description of the process can be obtained by other means, or when the quality of an existing documentation is questionable. For example, the audit trails of a workflow management system, the transaction logs of an enterprise resource planning system, and the electronic patient records in a hospital can be used to discover models describing processes, organizations, and products.

Process mining is closely related to BAM (Business Activity Monitoring), BOM (Business Operations Management), BPI (Business Process Intelligence), and data/workflow mining. Unlike classical data mining techniques the focus is on processes and questions that transcend the simple performance-related queries supported by tools such as Business Objects, Cognos BI, and Hyperion.

Main Contribution

In this thesis we address the following relevant issues in data and process management:

- Exploit structural query properties in order to build optimized query plan.
- Provide a formal framework for Data Integration and a prototype implementing them.

- Design process mining algorithms using a streaming approach and a suite providing an extensible and interactive framework for them.

Query plan optimization

The database community has investigated many structure-driven methods, which guarantee that large classes of queries may be answered in (input-output) polynomial-time. However, despite their very nice computational properties, these methods are not currently used for practical applications, since they do not care about output variables and aggregate operators, and do not exploit quantitative information on the data. In fact, none of these methods has been implemented inside any available DBMS. This thesis aims at filling this gap between theory and practice. First, we define an extension of the notion of hypertree decomposition, which is currently the most powerful structural method. This new version, called query-oriented hypertree decomposition, is a suitable relaxation of hypertree decomposition designed for query optimization, and such that output variables and aggregate operators can be dealt with. Based on this notion, a hybrid optimizer is implemented, which can be used on top of available DBMSs to compute query plans. The prototype is also integrated into the well-known open-source DBMS PostgreSQL. Finally, we validate our proposal with a thorough experimental activity, conducted on PostgreSQL and on a commercial DBMS, which shows that both systems may significantly benefit from using hypertree decompositions for query optimization.

Data Intgration

The work described in this thesis was carried on within the European Community funded INFOMIX project. The main goal of the INFOMIX project was to provide advanced techniques and innovative methodologies for information integration systems. In a nutshell, the project developed a theory, comprising a comprehensive information model and information integration algorithms, and a prototype implementation of a knowledge based system for advanced information integration, by using computational logic and integrating research results on data acquisition and transformation. Special attention was devoted to the definition of declarative user-interaction mechanisms, and techniques for handling semi-structured data, and incomplete and inconsistent data sources.

These objectives, which advanced the state of the art in several respects, are detailed as follows.

- **Comprehensive Information Model.** A comprehensive information model has to be provided, which incorporates static and dynamic aspects of information integration, and supports advanced *human like* reasoning, based on a rich semantics. Current information integration systems are rather poor in this respect, and provide only limited support (if any) for expressing constraint relationships between the local sources and a global view of the data.

- **Information Integration Algorithms.** A host of efficient algorithms for information integration must be provided, which can be applied to homogenized data from heterogeneous data sources.

- **Usage of Computational Logic.** Exploit advanced methodologies and techniques from computational logic as a toolbox for information integration.

- **Prototype System.** Definition and implementation of a component-based integration system prototype, and providing an infrastructure by using software agent technology.

Process Mining

We aimed to develop a novel Suite for Process Mining applications having an open and extensible architecture and introducing three innovative designing elements to meet the desiderata of flexibility and scalability arising in actual industrial scenarios. Indeed, the suite has been specifically conceived to support:

- the definition of complex mining applications, where various mining tasks can be combined and automatically orchestrated at run-time.

- building interactive applications based on the possibility of customizing data types, algorithms, and graphical user interfaces used in the analysis.

- ensuring scalability over large volumes of data.

Organization of the Thesis

This thesis consists in two parts. The first part deals with techniques and systems for data management, while in the second part techniques and systems for process management are described. More in detail, the thesis is organized as follows:

- In the first chapter will introduce DBMS and the query evaluation problem. After that, original results about Hypertree Decomposition are shown.
- The second chapter will show, after an introduction on Data integration, the INFOMIX project and its original contribution.
- The third chapter will introduce Process Mining and show results obtained with the implemented suite.

Part I

Advanced Techniques and Systems for Data Management

Chapter 1

Query Evaluation

Answering queries is computationally very expensive, and many approaches have been proposed in the literature to face this fundamental problem. Some of them are based on optimization modules that exploit quantitative information on the database instance, while other approaches exploit structural properties of the query hypergraph. For instance, acyclic queries can be answered in polynomial time, and also query containment is efficiently decid able for acyclic queries.

In this chapter we will first describe the overall architecture of a data base management system (DBMS), then we will survey *quantitative methods* for query plan optimization and finally we will show our studies on the way to exploit structural property for query answering.

1.1 DBMS Architecture

Database Management Systems are very complex, sophisticated software applications that provide reliable management of large amounts of data. To better understand general database concepts and the structure and capabilities of a DBMS, it is useful to examine the architecture of a typical database management system.

There are two different ways to look at the architecture of a DBMS: the logical DBMS architecture and the physical DBMS architecture. The logical architecture deals with the way data is stored and presented to users, while the physical architecture is concerned with the software components that make up a DBMS.

1.1.1 Logical Architecture

The logical architecture describes how data in the database is perceived by users. It is not concerned with how the data is handled and processed by the DBMS, but only with how it looks. Users are shielded from the way data is stored on the underlying file system, and can manipulate the data without worrying about where it is located or how it is actually stored. This results in the database having different levels of abstraction.

The majority of commercial Database Management Systems available today are based on the ANSI/SPARC generalized DBMS architecture, as proposed by the ANSI/SPARC Study Group on Data Base Management Systems.

The ANSI/SPARC architecture divides the system into three levels of abstraction: the internal or physical level, the conceptual level, and the external or view level.

- **The Internal or Physical Level.** The collection of files permanently stored on secondary storage devices is known as the physical database. The physical or internal level is the one closest to physical storage, and it provides a low-level description of the physical database, and an interface between the operating system's file system and the record structures used in higher levels of abstraction. It is at this level that record types and methods of storage are defined, as well as how stored fields are represented, what physical sequence the stored records are in, and what other physical structures exist.

- **The Conceptual Level.** The conceptual level presents a logical view of the entire database as a unified whole, which allows you to bring all the data in the database together and see it in a consistent manner. The first stage in the design of a database is to define the conceptual view, and a DBMS provides a data definition language for this purpose.

  It is the conceptual level that allows a DBMS to provide data independence. The data definition language used to create the conceptual level must not specify any physical storage considerations that should be handled by the physical level. It should not provide any storage or access details, but should define the information content only.

- **The External or View Level.** The external or view level provides a window on the conceptual view which allows the user to see only the data of interest to them. The user can be either an application program or an end user. Any number of external schema can be defined and they can overlap each other.

The System Administrator and the Database Administrator are special cases. Because they have responsibilities for the design and maintenance for the design and maintenance of the database, they, some times, need to be able to see the entire database. The external and the conceptual view are functionally equivalent for these two users.

- **Mappings Between Levels.** Obviously, the three levels of abstraction in the database do not exist independently of each other. There must be some correspondence, or mapping, between the levels. There are actually two mappings: the conceptual/internal mapping and the external/conceptual mapping.

  The conceptual/internal mapping lies between the conceptual and internal levels, and defines the correspondence between the records and the fields of the conceptual view and the files and data structures of the internal view. If the structure of the stored database is changed, then the conceptual/ internal mapping must also be changed accordingly so that the view from the conceptual level remains constant. It is this mapping that provides physical data independence for the database.

  The external/conceptual view lies between the external and conceptual levels, and defines the correspondence between a particular external view and the conceptual view. Although these two levels are similar, some elements found in a particular external view may be different from the conceptual view. For example, several fields can be combined into a single (virtual) field, which can also have different names from the original fields. If the structure of the database at the conceptual level is changed, then the external/conceptual mapping must change accordingly so the view from the external level remains constant. It is this mapping that provides logical data independence for the database.

  It is also possible to have another mapping, where one external view is expressed in terms of other external views (this could be called an external/external mapping). This is useful if several external views are closely related to one another, as it allows you to avoid mapping each of the similar external views directly to the conceptual level.

1.1.2 Physical Architecture

The physical architecture describes the software components used to enter and process data, and how these software components are related and interconnected. Although it is not possible to generalize the component structure of a DBMS, it is possible to identify a number of key functions which are common to most database management systems. The components that normally implement these functions are shown in the diagram on the following page, which depicts the physical architecture for a typical DBMS. At its most basic level the physical DBMS architecture can be broken down into two parts: the back end and the front end.

The back end is responsible for managing the physical database and providing the necessary support and mappings for the internal, conceptual, and external levels described earlier. Other benefits of a DBMS, such as security, integrity, and access control, are also the responsibility of the back end.

The front end is really just any application that runs on top of the DBMS. These may be applications provided by the DBMS vendor, the user, or a third party. The user interacts with the front end, and may not even be aware that the back end exists.

Application and Utilities

Applications and utilities are the main interface to the DBMS for most users. There are three main sources of applications and utilities for a DBMS: the vendor, the user, and third parties.

Vendor applications and utilities are provided for working with or maintaining the database, and usually allow users to create and manipulate a database without the need to write custom applications. However, these are usually general-purpose applications and are not the best tools to use for doing specific, repetitive tasks.

User applications are generally custom-made application programs written for a specific purpose using a conventional programming language. This programming language is coupled to the DBMS query language through the application program interface (API). This allows the user to utilize the power of the DBMS query language with the flexibility of a custom application.

Third party applications may be similar to those provided by the vendor, but with enhancements, or they may fill a perceived need that the vendor hasn't created an application for. They can also be similar to user applications, being written for a specific purpose they think a large majority of users will need.

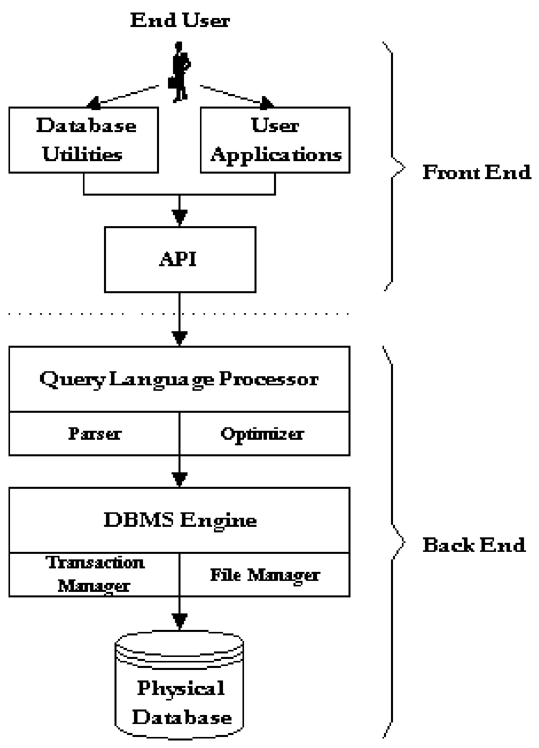


Figure 1.1: Physical Architecture of a DBMS

The most common applications and utilities used with a database can be divided into several well-defined categories. These are:

- **Command Line Interfaces:** these are character-based, interactive interfaces that let you use the full power and functionality of the DBMS query language directly. They allow you to manipulate the database and perform ad-hoc queries and see the results immediately. They are often the only method of exploiting the full power of the database without creating programs using a conventional programming language.

- **Graphical User Interface (GUI) tools:** these are graphical, interactive interfaces that hide the complexity of the DBMS and query language behind an intuitive, easy to understand, and convenient interface. This allows casual users the ability to access the database without having to learn the query language, and it allows advanced users to quickly manage and manipulate the database without the trouble of entering formal commands using the query language. However, graphical interfaces usually do not provide the same level of functionality as a command line interface because it is not always possible to implement all commands or options using a graphical interface.

- **Backup/Restore Utilities:** these are designed to minimize the effects of a database failure and ensure a database is restored to a consistent state if a failure does occur. Manual backup/restore utilities require the user to initiate the backup, while automatic utilities will back up the database at regular intervals without any intervention from the user. Proper use of a backup/restore utility allows a DBMS to recover from a system failure correctly and reliably.

- **Reporting/Analysis Utilities:** these are used to analyze and report on the data contained in the database. This may include analyzing trends in data, computing values from data, or displaying data that meets some specified criteria, and then displaying or printing a report containing this information.

The Application Program Interface

The application program interface (API) is a library of low-level routines which operate directly on the database engine. The API is usually used when creating software applications with a general-purpose programming language such as Java or C++. This allows you to write custom software applications to suit the needs of your business, without having to develop the storage architecture as well. The storage of the data is handled by the database engine, while the input and any special analysis or reporting functions are handled by the custom application.

An API is specific to each DBMS, and a program written using the API of one DBMS cannot be used with another DBMS. This is because each API usually has its own unique functions calls that are tied very tightly to the operation of the database. Even if two databases have the same function, they may use different parameters and function in different ways, depending on how the database designer decided to implement the function in each database. One exception to this is the Microsoft Open Database Connectivity API, which is designed to work with any DBMS that supports it.

The Query Language Processor

The query language processor is responsible for receiving query language statements and changing them from the English-like syntax of the query language to a form the DBMS can understand. The query language processor usually consists of two separate parts: the parser and the query optimizer.

The parser receives query language statements from application programs or command-line utilities and examines the syntax of the statements to ensure they are correct. To do this, the parser breaks a statement down into basic units of syntax and examines them to make sure each statement consists of the proper component parts. If the statements follow the syntax rules, the tokens are passed to the query optimizer.

The query optimizer examines the query language statement, and tries to choose the best and most efficient way of executing the query. To do this, the query optimizer will generate several query plans in which operations are performed in different orders, and then try estimate which plan will execute most efficiently. When making this estimate, the query optimizer may examine factors such as: CPU time, disk time, network time, sorting methods, and scanning methods.

The DBMS Engine

The DBMS engine is the heart of the DBMS, and it is responsible for all of the data management in the DBMS. The DBMS engine usually consists of two separate parts: the transaction manager and the file manager.

The transaction manager maintains tables of authorization and currency control information. The DBMS may use authorization tables to allow the transaction manager to ensure the user has permission to execute the query language statement on the database. The authorization tables can only be modified by properly authorized user commands, which are themselves checked against the authorization tables. In addition, a database may also support concurrency control tables to prevent conflicts when simultaneous, conflicting commands are executed. The DBMS checks the concurrency control tables before executing a query language statement to ensure that it is not locked by another statement.

The file manager is the component responsible for all physical input/output operations on the database. It is concerned with the physical address of the data on the disk, and is responsible for any interaction (reads or writes) with the host operating system.

1.2 Query Answering

As we have seen, DBMSs have specialized modules, called query optimizers, looking for good ways to deal with any given query. For the same query, there are many execution plans that a DBMS can choose to compute its answer. All these plans are equivalent in terms of output relation, however, the differences among their execution times can be several orders of magnitude large. The query optimizer examines the alternative plans and then choose the best one, according to some cost model. Actually, the optimizer performs two main tasks. First, it rewrites the query into an equivalent one that can be easily answered, by pushing selections and projections, replacing views by their definitions, flattening out nested queries and so on. Then, it starts the planning phase, where it selects a query execution plan, taking into account the actual query costs for the specific database and DBMS. We call quantitative methods all these query planners based on quantitative information about sizes of relations, indices, and so on. Note that computing an optimal plan is an NPhard problem and hence it is unlikely to find an efficient algorithm for selecting the best plans. Indeed, all the commercial DBMS just compute approximations optimal query plans.

1.2.1 Quatitative Methods

One central component of a query optimizer is its search strategy or enumeration algorithm. The enumeration algorithm of the optimizer determines which plans to enumerate, and the clas sic enumeration algorithm is based on dynamic programming. This algorithm was pioneered in IBM's System R project, and it is used in most query optimizers today. Dynamic pro gramming works very well if all queries are standard *SQL92* queries, the queries are moderately complex, and only simple textbook query execution techniques are used by the database system. Dynamic programming, however, does not work well if these conditions do not hold; e.g., if the database system must support very complex applications whose queries often involve many tables or new query optimization and execution techniques need to be integrated into the system in order to optimize queries in a distributed and/or heterogeneous programming environment. In these situations, the search space of query optimization can become very large and dynamic programming is not always viable because of its very high complexity. In general, there is a tradeoff between the complexity of an enumeration algorithm and the quality of the plans generated by the algorithm. Dynamic programming represents one extreme point: dynamic programming has exponential time and space complexity and generates optimal plans. Other algorithms have lower complexity than dynamic programming, but these algorithms are not able to find as lowcost plans as dynamic programming. Since the problem of finding an optimal plan is NPhard, implementors of query optimizers will probably always have to take this fundamental tradeoff between algorithm complexity and quality of plans into account when they decide which enumeration algorithm to use. Due to its importance, a large number of different algorithms have already been developed for query optimization in database systems. All algorithms proposed so far fall into one of three different classes or are combinations of such basic algorithms. In the following, we will briefly discuss each class of algorithms; a more complete overview and comparison of many of the existing algorithms can be found in [67].

Exhaustive search

All published algorithms of this class have exponential time and space complexity and are guaranteed to find the best plan according to the optimizer's cost model. The most prominent representative of this class of algorithms is (bottom-up) dynamic programming [65], which is currently used in most database systems. This algorithm works in a bottom-up way as follows. First, dynamic programming generates so-called access plans for every table involved in the query. Typically, such an access plan consists of one or two operators, and there are several different access plans for a table. In the second phase, dynamic programming considers all possible ways to join the tables. First, it considers all two-way join plans by using the access plans of the tables as building blocks and calling the joinPlans function to build a join plan from these building blocks. From the two-way join plans and the access plans, dynamic programming then produces three-way join plans. After that, it generates four-way join plans by considering all combinations of two two way join plans and all combinations of a three-way join plan with an access plan. In the same way, dynamic programming continues to produce five-way, six-way join plans and so on up to n-way join plans. In the third phase, the n-way join plans are massaged so that they become complete plans for the query; e.g., project, sort,or groupby operators are attached, if necessary. Note that in every step of the second phase, dynamic programming uses the same function to produce more and more complex plans using simpler plans as building blocks. Just as there are usually several alternative access plans, there are usually several different ways to join two tables (e.g., nested loop joins, hash joins, etc.) and the joinPlans function will return a plan for every alternative join method. The beauty of dynamic programming is that it discards inferior building blocks after every step (pruning). While enumerating two-way join plans, for example dynamic programming would consider an $A \bowtie B$ plan and a $B \bowtie A$ plan, but only the cheaper of the two plans would be retained, so that only the cheaper of the two plans would be considered as a building block for three-way, four-way, ... join plans involving A and B. Pruning is possible because the $A \bowtie B$ plan and the $B \bowtie A$ plan do the same work; if the $A \bowtie B$ plan is cheaper than the $B \bowtie A$ plan, then any complete plan for the whole query that has $A \bowtie B$ as a building block (e.g., $C \bowtie (A \bowtie B)$) will be cheaper than the same plan with $B \bowtie A$ as a building block (e.g., $C \bowtie (B \bowtie A)$). As a result of pruning, dynamic programming does not enumerate inferior plans such as $C \bowtie (B \bowtie A)$ and runs significantly faster than a naive exhaustive search. Note tht such an algorithm enumerates all bushy plans.

Heuristics

Typically, the algorithms of this class have polynomial time and space complexity, but they they typically produce worse plans [67]. Representatives of this class of algorithms are mini mum selectivity and other greedy algorithms , the KBZ algorithm ; and the AB algorithm. Basically, at each step, a partially determined order is extended by choosing the most promis ing relational operation to be executed, according to some preference criterion. Obviously, the quality of plans produced by the greedy algorithm strongly depends on the plan evaluation function that guides the preference criterion.

Randomized algorithms

The big advantage of randomized algorithms is that they have constant space overhead. The running time of most randomized algorithms cannot be predicted because these algorithms are indeterministic; typically, randomized algorithms are slower than heuristics and dynamic programming for simple queries and faster than both for very large queries. The best known randomized algorithm is called 2PO and is a combination of applying iterative improvement and simulated annealing. In many situations, 2PO produces good plans. However, there are situations in which 2PO produces plans that are orders of magnitude more expensive than an optimal plan.

Iterative Dynamic Programming

This technique is based on iteratively applying dynamic programming and can be seen as a combination of dynamic and greedy programming. The essence of this heuristic is that instead of fully enumerating all query processing plans, a resource limit is established (defined by a parameter k). During each dynamic programming stage, all query processing plans are enumerated up to k-way joins. At that point, one or more of the best plans are chosen. These partial plans will be used as building blocks to initiate the next stage of dynamic programming, which also produces query-processing plans until the resource limit is reached. The sequence of dynamic programming stages continues until a complete plan is generated.

1.3 Query Answering Exploiting Structural Properties

A completely different approach to query answering is based on structural properties of queries, rather than on quantitative information about data values. Exploiting such properties is possible to answer large classes of queries efficiently, that is, with a polynomial-time upper bound. The structure of a query Q is best represented by its *query hypergraph* $\mathcal{H}(Q) = (V, H)$, whose set V of vertices consists of all variables occurring in Q, and where the set H of hyperedges contains, for each query atom A, the set $var(A)$ of all variables occurring in A. As an example, consider the following query
Q_0: $ans \leftarrow s_1(A, B, D) \wedge s_2(B, C, D) \wedge s_3(B, E) \wedge s_4(D, G) \wedge s_5(E, F, G) \wedge s_6(E, H) \wedge s_7(F, I) \wedge s_8(G, J)$. Figure 1.2 shows its associated hypergraph $\mathcal{H}(Q_0)$.

1.3.1 Queries and Acyclic Hypergraphs

We will adopt the standard convention of identifying a relational database instance with a logical theory consisting of ground facts. Thus, a tuple $\langle a_1, \ldots a_k \rangle$, belonging to relation r, will be identified with the ground atom $r(a_1, \ldots, a_k)$. The fact that a tuple $\langle a_1, \ldots, a_k \rangle$ belongs to relation r of a database instance **DB** is thus simply denoted by $r(a_1, \ldots, a_k) \in$ **DB**.

A (rule-based) *conjunctive query* Q on a database schema $DS = \{R_1, \ldots, R_m\}$

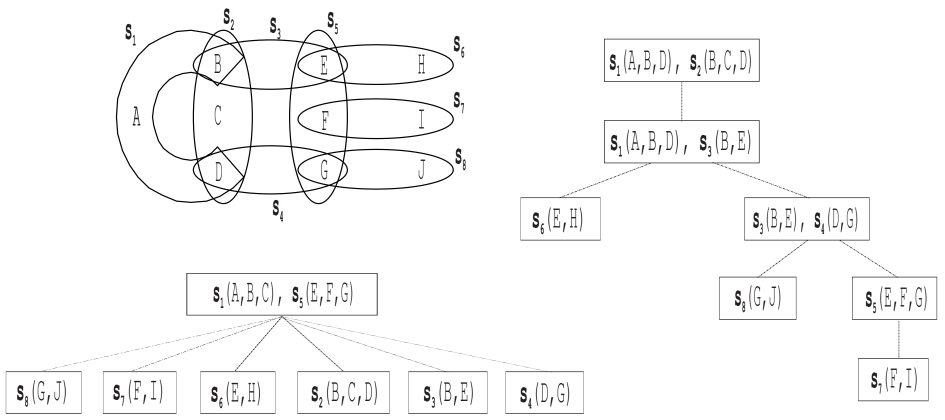


Figure 1.2: Hypergraph $\mathcal{H}(Q_0)$ (left), two hypertree decompositions of width 2 of $\mathcal{H}(Q_0)$ (right and bottom).

consists of a rule of the form

$$Q: \quad ans(\mathbf{u}) \leftarrow r_1(\mathbf{u_1}) \wedge \cdots \wedge r_n(\mathbf{u_n}),$$

where $n \geq 0$; $r_1, \ldots r_n$ are relation names (not necessarily distinct) of DS; ans is a relation name not in DS; and $\mathbf{u}, \mathbf{u_1}, \ldots, \mathbf{u_n}$ are lists of terms (i.e., variables or constants) of appropriate length. The set of variables occurring in Q is denoted by $var(Q)$. The set of atoms contained in the body of Q is referred to as $atoms(Q)$.

The *answer* of Q on a database instance **DB** with associated universe U, consists of a relation ans, whose arity is equal to the length of $\mathbf{u}$, defined as follows. Relation ans contains all tuples $\mathbf{u}\theta$ such that $\theta : var(Q) \longrightarrow U$ is a substitution replacing each variable in $var(Q)$ by a value of U and such that for $1 \leq i \leq n$, $r_i(\mathbf{u_i})\theta \in$ **DB**. (For an atom A, $A\theta$ denotes the atom obtained from A by uniformly substituting $\theta(X)$ for each variable X occurring in A.)

If Q is a conjunctive query, we define the hypergraph $H(Q) = (V, E)$ associated to Q as follows. The set of vertices V, denoted by $var(H(Q))$, consists of all variables occurring in Q. The set E, denoted by $edges(H(Q))$, contains for each atom $r_i(\mathbf{u_i})$ in the body of Q a hyperedge consisting of all variables occurring in $\mathbf{u_i}$. Note that the cardinality of $edges(H(Q))$ can be smaller than the cardinality of ***atoms(Q)***, because two query atoms having exactly the same set of variables in their arguments give rise to only one edge in $edges(H(Q))$. For example, the three query atoms $r(X, Y)$, $r(Y, X)$, and $s(X, X, Y)$ all correspond to a unique hyperedge $\{X, Y\}$.

A query Q is acyclic if and only if its hypergraph $H(Q)$ is acyclic or, equivalently, if it has has a join forest. A *join forest* for the hypergraph $H(Q)$ is a forest G whose set of vertices V_G is the set $edges(H(Q))$ and such that, for each pair of hyperedges h_1 and h_2 in V_G having variables in common (i.e., such that $h_1 \cap h_2 \neq \emptyset$), the following conditions hold:

1. h_1 and h_2 belong to the same connected component of G, and
2. all variables common to h_1 and h_2 occur in every vertex on the (unique) path in G from h_1 to h_2.

If G is a tree, then it is called a *join tree* for $H(Q)$.

Intuitively, the efficient behavior of acyclic instances is due to the fact that they can be evaluated by processing any of their join trees bottom-up by performing upward semijoins, thus keeping the size of the intermediate relations small (while it could become exponential, if regular join were performed).

Let us recall the highly desirable computational properties of acyclic queries:

1. Acyclic instances can be efficiently solved. Yannakakis provided a (sequential) polynomial time algorithm for Boolean acyclic queries[1]. Moreover, he showed that the answer of a non-Boolean acyclic conjunctive query can be *computed* in time polynomial in the combined size of the input instance and of the output relation [86].

2. We have shown that answering queries is highly parallelizable on acyclic queries, as this problem (actually, the decision problem of answering Boolean queries) is complete for the low complexity class LOGCFL [36]. Efficient parallel algorithms for Boolean and non-Boolean queries have been proposed in [36] and [35]. They run on parallel database machines that exploit the *inter-operation parallelism* [84], i.e., machines that execute different relational operations in parallel. These algorithms can be also employed for solving acyclic queries efficiently in a distributed environment.

3. Acyclicity is efficiently recognizable: deciding whether a hypergraph is acyclic is feasible in linear time [68] and belongs to the class L (deterministic logspace). The latter result is new: it follows from the fact that hypergraph acyclicity belongs to SL [37], and from the very recent proof that SL is in fact equal to L [59].

[1]Note that, since both the database **DB** and the query Q are part of an input-instance, what we are considering is the *combined complexity* of the query [82].

1.3.2 Hypertree Decompositions

We recall the formal definition and the most important results about *hypertree width* and *hypertree decompositions*.

A *hypertree for a hypergraph* $\mathcal{H}$ is a triple $\langle T, \chi, \lambda \rangle$, where $T = (N, E)$ is a rooted tree, and χ and λ are labeling functions which associate to each vertex $p \in N$ two sets $\chi(p) \subseteq var(\mathcal{H})$ and $\lambda(p) \subseteq edges(\mathcal{H})$. The *width* of a hypertree is the cardinality of its largest λ label, i.e., $max_{p \in N} |\lambda(p)|$.

We denote the set of vertices of any rooted tree T by $vertices(T)$, and its root by $root(T)$. Moreover, for any $p \in vertices(T)$, T_p denotes the subtree of T rooted at p. If T' is a subtree of T, we define $\chi(T') = \bigcup_{v \in vertices(T')} \chi(v)$.

Definition 1.3.1 *[39]*

A generalized hypertree decomposition *of a hypergraph $\mathcal{H}$ is a hypertree $HD = \langle T, \chi, \lambda \rangle$ for $\mathcal{H}$ which satisfies the following conditions:*

1. *For each edge $h \in edges(\mathcal{H})$, all of its variables occur together in some vertex of the decomposition tree, that is, there exists $p \in vertices(T)$ such that $h \subseteq \chi(p)$ (we say that p* covers *h).*

2. *Connectedness Condition: for each variable $Y \in var(\mathcal{H})$, the set $\{p \in vertices(T) \mid Y \in \chi(p)\}$ induces a (connected) subtree of T.*

3. *For each vertex $p \in vertices(T)$, variables in the χ labeling should belong to edges in the λ labeling, that is, $\chi(p) \subseteq var(\lambda(p))$.*

A hypertree decomposition *is a generalized hypertree decomposition that satisfies the following additional condition:*

4. *Special Descendant Condition: for each $p \in vertices(T)$, $var(\lambda(p)) \cap \chi(T_p) \subseteq \chi(p)$.*

The HYPERTREE width *$hw(\mathcal{H})$ (resp., generalized hypertree width $ghw(\mathcal{H})$) of $\mathcal{H}$ is the minimum width over all its hypertree decompositions (resp., generalized hypertree decompositions).*

An edge $h \in edges(\mathcal{H})$ is strongly covered *in HD if there exists $p \in vertices(T)$ such that $var(h) \subseteq \chi(p)$ and $h \in \lambda(p)$. In this case, we say that p strongly covers h. A decomposition HD of hypergraph $\mathcal{H}$ is a* complete decomposition *of $\mathcal{H}$ if every edge of $\mathcal{H}$ is strongly covered in HD. From any (generalized) hypertree decomposition HD of $\mathcal{H}$, we can easily compute a complete (generalized) hypertree decomposition of $\mathcal{H}$ having the same width.*

Note that the notions of hypertree width and generalized hypertree width are true generalizations of acyclicity, as the acyclic hypergraphs are precisely those hypergraphs having hypertree width and generalized hypertree width one. In particular, as we will see in the next section, the classes of conjunctive queries having bounded (generalized) hypertree width have the same desirable computational properties as acyclic queries [38].

At first glance, a generalized hypertree decomposition of a hypergraph may simply be viewed as a clustering of the hyperedges (i.e., query atoms) where the classical connectedness condition of join trees holds. However, a generalized hypertree decomposition may deviate in two ways from this principle: **(1)** A hyperedge already used in some cluster may be reused in some other cluster; **(2)** Some variables occurring in reused hyperedges are not required to fulfill any condition.

For a better understanding of this notion, let us focus on the two labels associated with each vertex p: the set of hyperedges $\lambda(p)$, and the set of *effective* variables $\chi(p)$, which are subject to the connectedness condition (2). Note that all variables that appear in the hyperedges of $\lambda(p)$ but that are not included in $\chi(p)$ are "ineffective" for v and do not count w.r.t. the connectedness condition. Thus, the χ labeling plays the crucial role of providing a join-tree like re-arranging of all connections among variables. Besides the connectedness condition, this re-arranging should fulfill the fundamental Condition 1: every hyperedge (i.e., query atom, in our context) has to be properly considered in the decomposition, as for graph edges in tree-decompositions and for hyperedges in join trees (where this condition is actually even stronger, as hyperedges are in a one-to-one correspondence with vertices of the tree). Since the only relevant variables are those contained in the χ labels of vertices in the decomposition tree, the λ labels are "just" in charge of covering such relevant variables (Condition 3) with as few hyperedges as possible. Indeed, the width of the decomposition is determined by the largest λ label in the tree. This is the most important novelty of this approach, and comes from the specific properties of hypergraph-based problems, where hyperedges often play a predominant role. For instance, think of our database framework: the cost of evaluating a natural join operation with k atoms (read: k hyperedges) is $O(n^k)$, no matter of the number of variables occurring in the query.

Example 1.3.2 *Consider the following conjunctive query Q_1:*

$$\begin{aligned} ans \leftarrow\ & a(S,X,X',C,F) \ \wedge\ b(S,Y,Y',C',F') \\ & \wedge\, c(C,C',Z) \ \wedge\ d(X,Z) \ \wedge \\ & e(Y,Z) \ \wedge\ f(F,F',Z') \ \wedge\ g(X',Z') \ \wedge \\ & h(Y',Z') \ \wedge\ j(J,X,Y,X',Y'). \end{aligned}$$

Let $\mathcal{H}_1$ be the hypergraph associated to Q_1. Since $\mathcal{H}_1$ is cyclic, $hw(\mathcal{H}_1) > 1$ holds. Figure 1.3 shows a (complete) hypertree decomposition HD_1 of $\mathcal{H}_1$ having width 2, hence $hw(\mathcal{H}_1) = 2$.

In order to help the intuition, Figure 1.4 shows an alternative representation of this decomposition, called atom *(or* hyperedge*)* representation *[38]: each node p in the tree is labeled by a set of atoms representing $\lambda(p)$; $\chi(p)$ is the set of all variables, distinct from '_', appearing in these hyperedges. Thus, in this representation, possible occurrences of the anonymous variable '_' take the place of variables in $var(\lambda(p)) - \chi(p)$.*

Another example is depicted in Figure 1.2, which shows two hypertree decompositions of query Q_0 in Section 1.3. Both decompositions have width two and are complete decompositions of Q_0. $\square$

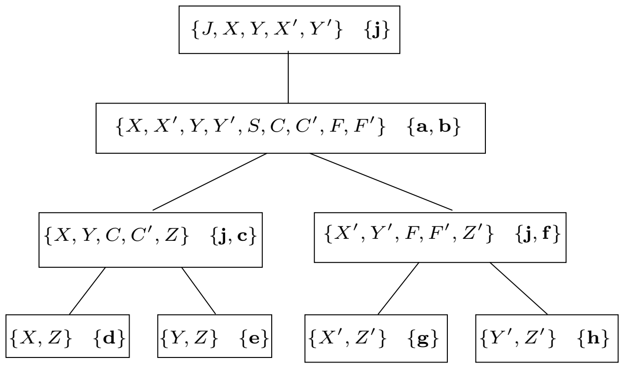

$\{J, X, Y, X', Y'\}$ $\{\mathbf{j}\}$
$\{X, X', Y, Y', S, C, C', F, F'\}$ $\{\mathbf{a}, \mathbf{b}\}$
$\{X, Y, C, C', Z\}$ $\{\mathbf{j}, \mathbf{c}\}$
$\{X', Y', F, F', Z'\}$ $\{\mathbf{j}, \mathbf{f}\}$
$\{X, Z\}$ $\{\mathbf{d}\}$
$\{Y, Z\}$ $\{\mathbf{e}\}$
$\{X', Z'\}$ $\{\mathbf{g}\}$
$\{Y', Z'\}$ $\{\mathbf{h}\}$

Figure 1.3: A 2-width hypertree decomposition of hypergraph $\mathcal{H}_1$ in Example 1.3.2

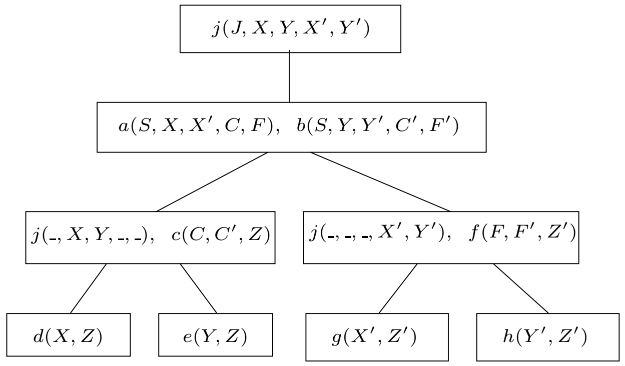

$j(J, X, Y, X', Y')$
$a(S, X, X', C, F)$, $b(S, Y, Y', C', F')$
$j(\_, X, Y, \_, \_)$, $c(C, C', Z)$
$j(\_, \_, \_, X', Y')$, $f(F, F', Z')$
$d(X, Z)$
$e(Y, Z)$
$g(X', Z')$
$h(Y', Z')$

Figure 1.4: Atom representation of the hypertree decomposition in Figure 1.3

Let k be a fixed positive integer. We say that a CQ instance I has k-bounded (generalized) hypertree width if $(g)hw(\mathcal{H}(I)) \leq k$. A class of queries has bounded (generalized) hypertree width if there is some $k \geq 1$ such that all instances in the class have k-bounded (generalized) hypertree width.

Clearly enough, choosing a tree and a clever combination of χ and λ labeling for its vertices in order to get a decomposition below a fixed threshold width k is not that easy, and is definitely more difficult than computing a simple tree decomposition, where only variables are associated with each vertex. In fact, the tractability of generalized hypertree width is an interesting open problem, as no polynomial time algorithm is known for deciding whether a hypergraph has generalized hypertree width at most k, for any fixed $k \geq 2$.

It is thus very nice and somehow surprising that dealing with the hypertree width is a very easy task. More precisely, for any fixed $k \geq 1$, deciding whether a given hypergraph has hypertree width at most k is in LOGCFL, and thus it is a tractable and highly parallelizable problem. Correspondingly, the search problem of computing a k-bounded hypertree decomposition belongs to the functional version of LOGCFL, which is $\mathrm{L}^{\mathrm{LOGCFL}}$ [38]. See the Hypertree Decomposition Homepage [2], for available implementations of algorithms for computing hypertree decompositions, and further links to heuristics and other papers on this subject.

Let us briefly discuss the only difference of hypertree decomposition with respect to generalized hypertree decomposition, that is, the *descendant condition* (Condition 4 in Definition 1.3.1). Consider a vertex p of a hypertree decomposition and a hyperedge $h \in \lambda(p)$ such that some variables $\bar{X} \subseteq h$ occur in the χ labeling of some vertices in the subtree T_p rooted at p. Then, according to this condition, these variables must occur in $\chi(p)$, too. This means, intuitively, that we have to deal with variables in $\bar{X}$ at this point of the decomposition tree, if we want to put h in $\lambda(p)$. For instance, as a consequence of this condition, for the root r of any hypertree decomposition we always have $\chi(r) = var(\lambda(r))$. However, once a hyperedge has been covered by some vertex of the decomposition tree, any subset of its variables can be used freely in order to decompose the remaining cycles in the hypergraph.

To shed more light on this restriction, consider what happens in the related hypergraph-based notions: in query decompositions [20], all variables are relevant; at the opposite side, in generalized hypertree decompositions, we can choose as relevant variables any subset of variables occurring in λ, without any limitation; in hypertree decompositions, we can choose any subset of relevant variables

[2]http://wwwinfo.deis.unical.it/ frank/Hypertrees/

as long as the above descendant condition is satisfied. Therefore, the notion of hypertree width is clearly more powerful than the (intractable) notion of query width, but less general than the (probably intractable) notion of generalized hypertree width, which is the most liberal notion.

For instance, look at Figure 1.4: the variables in the hyperedge corresponding to atom j in $\mathcal{H}_1$ are jointly included only in the root of the decomposition, while we exploit two different subsets of this hyperedge in the rest of the decomposition tree. Note that the descendant condition is satisfied. Take the vertex at level 2, on the left: the variables J, X' and Y' are not in the χ label of this vertex (they are replaced by the anonymous variable '_'), but they do not occur anymore in the subtree rooted at this vertex. On the other hand, if we were forced to take all the variables occurring in every atom in the decomposition tree, it would not be possible to find a decomposition of width 2. Indeed, j is the only atom containing both pairs X, Y and X', Y', and it cannot be used again entirely, for its variable J cannot occur below the vertex labeled by a and b, otherwise it would violate the connectedness condition (i.e., Condition 2 of Definition 1.3.1). In fact, every query decomposition of this hypergraph has width 3, while the hypertree width is 2. In this case the generalized hypertree width is 2, as well, but in general it may be less than the hypertree width. However, after a recent interesting result by Adler et al. [2], the difference of these two notions of width is within a constant factor: for any hypergraph $\mathcal{H}$, $ghw(\mathcal{H}) \leq hw(\mathcal{H}) \leq 3ghw(\mathcal{H}) + 1$. It follows that a class of hypergraphs has bounded generalized hypertree width if and only if it has bounded hypertree width, and thus the two notions identify the same set of tractable classes.

Though the formal definition of hypertree width is rather involved, it is worthwhile noting that this notion has very natural characterizations in terms of games and logics [39]:

- **The robber and marshals game (R&Ms game).** It is played by one robber and a number of marshals on a hypergraph. The robber moves on variables, while marshals move on hyperedges. At each step, any marshal controls an entire hyperedge. During a move of the marshals from the set of hyperedges E to to the set of hyperedges E', the robber cannot pass through the vertices in $B = (\cup E) \cap (\cup E')$, where, for a set of hyperedges F, $\cup F$ denotes the union of all hyperedges in F. Intuitively, the vertices in B are those not released by the marshals during the move. As in the monotonic robber and cops game defined for treewidth [66], it is required that the marshals capture the robber by monotonically shrinking the moving space of the robber. The game is won by the marshals if they corner the robber somewhere in the hypergraph. A hypergraph $\mathcal{H}$ has k-bounded hypertree width if and only if k marshals win the R&Ms game on $\mathcal{H}$.

- **Logical characterization of hypertree width.** Let L denote the existential conjunctive fragment of positive first order logic (FO). Then, the class of queries having k-bounded hypertree width is equivalent to the k-guarded fragment of L, denoted by $\mathrm{GF}_k(\mathrm{L})$. Roughly, we say that a formula Φ belongs to $\mathrm{GF}_k(\mathrm{L})$ if, for any subformula ϕ of Φ, there is a conjunction of up to k atoms jointly acting as a guard, that is, covering the free variables of ϕ. Note that this notion is related to the *loosely guarded fragment* as defined (in the context of full FO) by Van Benthem [70], where an arbitrary number of atoms may jointly act as guards.

Query Decompositions and Query Plans

In this section we describe the basic idea to exploit (generalized) hypertree decompositions for answering conjunctive queries.

Let $k \geq 1$ be a fixed constant, Q a conjunctive query over a database **DB**, and $HD = \langle T, \chi, \lambda \rangle$ a generalized hypertree decomposition of Q of width $w \leq k$. Then, we can answer Q in two steps:

1. For each vertex $p \in vertices(T)$, compute the join operations among relations occurring together in $\lambda(p)$, and project onto the variables in $\chi(p)$. At the end of this phase, the conjunction of these intermediate results forms an acyclic conjunctive query, say Q', equivalent to Q. Moreover, the decomposition tree T represents a join tree of Q'.

2. Answer Q', and hence Q, by using any algorithm for acyclic queries, e.g. Yannakakis's algorithm.

For instance, Figure 1.5 shows the tree $\mathcal{JT}_1$ obtained after Step 1 above, from the query Q_1 in Example 1.3.2 and the generalized hypertree decomposition in Figure 1.4. E.g. observe how the vertex labeled by atom p_3 is built. It comes from the join of atoms j and c (occurring in its corresponding vertex in Figure 1.4), and from the subsequent projection onto the variables X, Y, C, C', and Z (belonging to the χ label of that vertex). By construction, $\mathcal{JT}_1$ satisfies the connectedness condition. Therefore, the conjunction of atoms labeling this tree is an acyclic query, say Q_1', such that $\mathcal{JT}_1$ is one of its join trees. Moreover, it is easy to see that Q_1' has the same answer as Q_1 [38].

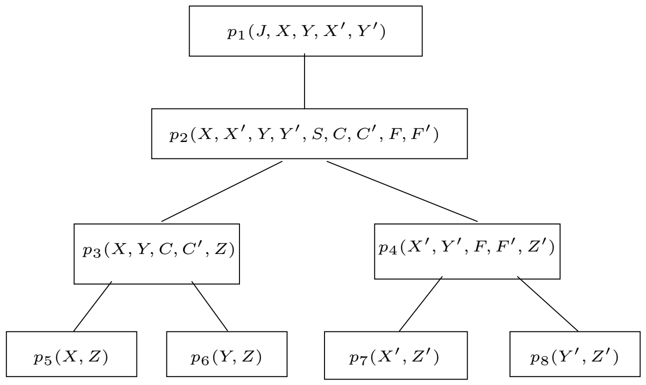


Figure 1.5: Join tree JT_1 computed for query Q_1'

Step 1 is feasible in $O(m|r_{max}|^w)$ time, where m is the number of vertices of T, and r_{max} is the relation of having the largest size. For Boolean queries, Yannakakis's algorithm in Step 2 takes $O(m|r_{max}|^w \log |r_{max}|)$ time, and thus its cost is an upper bound for the entire query evaluation process. For non-Boolean queries, Yannakakis's algorithm works in time polynomial in the combined size of the input and of the output, and thus we should add to the above cost a term that depends on the answer of the given query (which may be exponential w.r.t. the input size). For instance, if we consider query Q_1, the above upper bound is $O(7|r_{max}|^2 \log |r_{max}|)$, whereas typical query answering algorithms (which do not exploit structural properties) would take $O(|r_{max}|^7)$ time, in the worst case.

It has been observed that, according to Definition 1.3.1, a hypergraph may have some (usually) undesirable hypertree decompositions [38], possibly with a large number m of vertices in the decomposition tree. For instance, a decomposition may contain two vertices with exactly the same labels. Therefore, a *normal form* for hypertree decompositions has been defined in [38], and then strengthened in [63], in order to avoid such kind of redundancies. Hypertree decompositions in normal form having width at most k may be computed in time polynomial in the size of the given hypergraph $\mathcal{H}$ (but exponential in the parameter k). The number m of vertices cannot exceed the number of variables in $\mathcal{H}$, and is typically much smaller. Moreover, $\mathcal{H}$ has a hypertree decomposition of width w if and only if it has a normal-form hypertree decomposition of the same width w.

It follows that, for any fixed $k \geq 1$, the class of all queries having k-bounded hypertree width may be answered in polynomial time (actually, in input-output polynomial time, for non-Boolean queries). Indeed, given a query Q, both computing a hypertree decomposition *HD* of width at most k of $\mathcal{H}(Q)$, and then answering Q exploiting HD are polynomial-time tasks.

As far as generalized hypertree decompositions are concerned, we currently miss a polynomial-time algorithm for recognizing queries having k-bounded generalized hypertree-width. However, there is a great deal of interest in these decompositions, and some first results are coming.

1.3.3 Weighted Hypertree Decompositions

As described in the previous section, given a query Q on a database **DB** and a small-width decomposition HD for Q, we know that there is a polynomial time upper bound for answering Q, while in general this problem is NP-hard and all the available algorithms requires exponential time, in the worst case. However, HD is not just a theoretical indication of tractability for Q. Rather, the above two steps for evaluating Q actually represent a query plan for it, though not completely specified. For instance, no actual join method (merge, nested-loop, etc.) is chosen, but this final more physical phase can be easily implemented using well-known database techniques. We remark that such optimizations are executed just on relations belonging to the same vertex, and hence on w relations at most, if w is the width of HD. Thus, also optimal methods based on dynamic programming or sophisticated heuristics can be employed, as the size of the problem is small.

The remaining interesting problem is before this evaluation phase, where we have to compute a decomposition for $\mathcal{H}(Q)$. Indeed, in general there is an exponential number of hypertree decompositions of a hypergraph. Every decomposition encodes a way of aggregating groups of atoms and arranging them in a tree-like fashion. As far as the polynomial-time upper bound is concerned, we may be happy with any minimum-width decomposition. However, in practical real-world applications we have to exploit all available information. In particular, for database queries, we cannot get rid of information on the database **DB**. Indeed, looking only at the query structure is not the best we can do, if we may additionally exploit the knowledge of relation sizes, attribute selectivity, and so on.

1.3.4 Minimal Decompositions

In this section, we thus consider hypertree decompositions with an associated weight, which encodes our preferences, and allows us to take into account further requirements, besides the width. We will see how to answer queries more efficiently, by looking for their best decompositions.

Formally, given a hypergraph $\mathcal{H}$, a *hypertree weighting function* (short: HWF) $\omega_{\mathcal{H}}$ is any polynomial-time function that maps each generalized hypertree decomposition $HD = \langle T, \chi, \lambda \rangle$ of $\mathcal{H}$ to a real number, called the *weight* of HD.

For instance, a very simple HWF is the function $\omega^{w}_{\mathcal{H}}(HD) = \max_{p \in vertices(T)} |\lambda(p)|$, that weights a decomposition HD just on the basis of its worse vertex, that is the vertex with the largest λ label, which also determines the width of the decomposition.

In many applications, finding such a decomposition having the minimum width is not the best we can do. We can think of minimizing the number of vertices having the largest width w and, for decompositions having the same numbers of such vertices, minimizing the number of vertices having width $w-1$, and continuing so on, in a lexicographical way. To this end, we can define the HWF $\omega^{lex}_{\mathcal{H}}(HD) = \sum_{i=1}^{w} |\{p \in N \text{ such that } |\lambda(p)| = i\}| \times B^{i-1}$, where $N = vertices(T)$, $B = |edges(\mathcal{H})| + 1$, and w is the width of HD. Note that any output of this function can be represented in a compact way as a radix B number of length w, which is clearly bounded by the number of edges in $\mathcal{H}$. Consider again the query Q_0 of the Introduction, and the hypertree decomposition, say HD', of $\mathcal{H}(Q_0)$ shown in Figure 1.2, on the right. It is easy to see that HD' is not the best decomposition w.r.t. $\omega^{lex}_{\mathcal{H}}$ and the class of hypertree decompositions in normal form. Indeed, $\omega^{lex}_{\mathcal{H}}(HD') = 4 \times 9^0 + 3 \times 9^1$, and thus the decomposition HD'' shown on the bottom of Figure 1.2 is better than HD', as $\omega^{lex}_{\mathcal{H}}(HD'') = 6 \times 9^0 + 1 \times 9^1$.

Let $k > 0$ be a fixed integer and $\mathcal{H}$ a hypergraph. We define the class $kHD_{\mathcal{H}}$ (resp., $kNFD_{\mathcal{H}}$) as the set of all hypertree decompositions (resp., normal-form hypertree decompositions) of $\mathcal{H}$ having width at most k.

Definition 1.3.3 *[63]*Let $\mathcal{H}$ be a hypergraph, $\omega_{\mathcal{H}}$ a weighting function, and $\mathcal{C}_{\mathcal{H}}$ a class of generalized hypertree decompositions of $\mathcal{H}$. Then, a decomposition $HD \in \mathcal{C}_{\mathcal{H}}$ is *minimal* w.r.t. $\omega_{\mathcal{H}}$ and $\mathcal{C}_{\mathcal{H}}$, denoted by $[\omega_{\mathcal{H}}, \mathcal{C}_{\mathcal{H}}]$-$minimal$, if there is no $HD' \in \mathcal{C}_{\mathcal{H}}$ such that $\omega_{\mathcal{H}}(HD') < \omega_{\mathcal{H}}(HD)$. $\square$

For instance, the $[\omega^{w}_{\mathcal{H}}, kHD_{\mathcal{H}}]$-$minimal$ decompositions are exactly the k-bounded hypertree decompositions having the minimum possible width, while the $[\omega^{lex}_{\mathcal{H}}, kHD_{\mathcal{H}}]$-$minimal$ hypertree decompositions are a subset of them, corresponding to the lexicographically minimal decompositions described above.

It is not difficult to show that, for general weighting functions, the computation of minimal decompositions is a difficult problem even if we consider just bounded hypertree decompositions [63]. We thus restrict our attention to simpler HWFs.

Let $\langle \mathbb{R}^+, \oplus, \min, \perp, +\infty \rangle$ be a *semiring*, that is, $\oplus$ is a commutative, associative, and closed binary operator, $\perp$ is the neuter element for $\oplus$ (e.g., 0 for $+$, 1 for $\times$, etc.) and the absorbing element for $\min$, and $\min$ distributes over $\oplus$.[3] Given a function g and a set of elements $S = \{p_1, ..., p_n\}$, we denote by $\bigoplus_{p_i \in S} g(p_i)$ the value $g(p_1) \oplus \ldots \oplus g(p_n)$.

Definition 1.3.4 *[63]*Let $\mathcal{H}$ be a hypergraph. Then, a *tree aggregation function* (short: TAF) is any hypertree weighting function of the form

$$\mathsf{F}_{\mathcal{H}}^{\oplus,v,e}(HD) = \bigoplus_{p \in N} (v_{\mathcal{H}}(p) \ \oplus \bigoplus_{(p,p') \in E} e_{\mathcal{H}}(p,p')),$$

associating an $\mathbb{R}^+$ value to the hypertree decomposition $HD = \langle (N, E), \chi, \lambda \rangle$, where $v_{\mathcal{H}} : N \mapsto \mathbb{R}^+$ and $e_{\mathcal{H}} : N \times N \mapsto \mathbb{R}^+$ are two polynomial functions evaluating vertices and edges of hypertrees, respectively. $\square$

We next focus on a tree aggregation function that is useful for query optimization. We refer the interested reader to [63] for further examples and applications.

Given a query Q over a database **DB**, let $HD = \langle T, \chi, \lambda \rangle$ be a hypertree decomposition in normal form for $\mathcal{H}(Q)$. For any vertex p of T, let $E(p)$ denote the relational expression $E(p) = \ \bowtie_{h \in \lambda(p)} \prod_{\chi(p)} rel(h)$, i.e., the join of all relations in **DB** corresponding to hyperedges in $\lambda(p)$, suitably projected onto the variables in $\chi(p)$. Given also an incoming node p' of p in the decomposition HD, we define $v^*_{\mathcal{H}(Q)}(p)$ and $e^*_{\mathcal{H}(Q)}(p, p')$ as follows:

- $v^*_{\mathcal{H}(Q)}(p)$ is the estimate of the cost of evaluating the expression $E(p)$, and
- $e^*_{\mathcal{H}(Q)}(p, p')$ is the estimate of the cost of evaluating the semi-join $E(p) \ltimes E(p')$.

Let $cost_{\mathcal{H}(Q)}$ be the TAF $\mathsf{F}_{\mathcal{H}(Q)}^{+,v^*,e^*}(HD)$, determined by the above functions. Intuitively, $cost_{\mathcal{H}(Q)}$ weights the hypertree decompositions of the query hypergraph $\mathcal{H}(Q)$ in such a way that minimal hypertree decompositions correspond to "optimal" query evaluation plans for Q over **DB**. Note that any method for computing the estimates for the evaluation of relational algebra operations from the quantitative information on **DB** (relations sizes, attributes selectivity, and so on) may be employed for v^* and e^*.

Clearly, all these powerful weighting functions would be of limited practical applicability, without a polynomial time algorithm for the computation of

[3]For the sake of presentation, we refer to $\min$ and hence to minimal hypertree decompositions. However, it is easy to see that all the results presented in this paper can be generalized easily to any semiring, possibly changing $\min$, $\mathbb{R}^+$, and $+\infty$.

minimal decompositions. Surprisingly, it turns out that, unlike the traditional (non-weighted) framework, working with normal-form hypertree decompositions, rather than with any kind of bounded-width hypertree decomposition, does matter. Indeed, computing such minimal hypertree decompositions with respect to any tree aggregation function is a tractable problem, while it has been proved that the problem is still NP-hard if the whole class of bounded-width hypertree decomposition is considered. A polynomial time algorithm for this problem, called `minimal-`k`-decomp`, is presented in [63].

1.3.5 Hypertree Decompositions for Queries

In this section, we describe a new extension of the notion of hypertree decomposition specifically designed for the query evaluation purposes described in the previous sections. In particular, we show how to compute decomposition trees whose associated query plans guarantee a polynomial time upper bound with a single bottom-up evaluation phase, and we improve the query answering exploiting the hypertreee decomposition phase, by avoiding the naive construction of the acyclic query equivalent instance Q'.

Definition 1.3.5 A *query-oriented hypertree decomposition* (short: *q-hypertree decomposition*) of a conjunctive query Q is a hypertree $HD = \langle T, \chi, \lambda \rangle$ of the hypergraph $\mathcal{H}(Q)$ which satisfies the following conditions:

1. For each edge $h \in edges(\mathcal{H})$, there exists $p \in vertices(T)$ such that $h \subseteq \chi(p)$.
2. There exists $p \in vertices(T)$ such that $out(Q) \subseteq \chi(p)$.
3. Connectedness Condition: For each variable $Y \in var(\mathcal{H})$, the set $\{p \in vertices(T) \mid Y \in \chi(p)\}$ induces a (connected) subtree of T. □

In fact, a q-hypertree decomposition HD of a (non-Boolean) query Q is a (generalized) hypertree decomposition of $\mathcal{H}(Q)$, except for the following important features:

a) The decomposition HD is forced to have a vertex p that covers all the output variables of Q. Therefore, if we root the decomposition tree at p, the query may be answered by performing only one bottom-up evaluation of the decomposition tree. Indeed, at the end of this procedure, the relation at vertex p, projected onto $out(Q)$, directly provides the answer of Q. Thus, steps (ii) and (iii) described above are no longer needed.

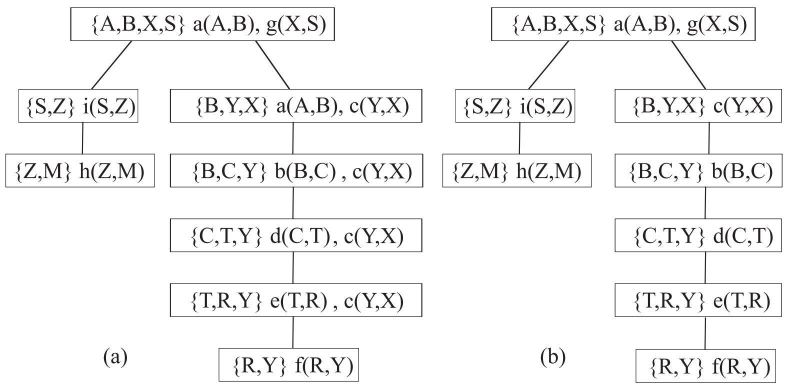


Figure 1.6: HD_1 (a) and HD'_1 (b), in Example 1.3.6.

b) Condition 3 of Definition 1.3.1 is not required here, that is, some variables in the χ labeling may be not covered by atoms in the λ labeling. This relaxation allows us to perform an important optimization of decomposition-based query plans: we may save join operations at a vertex p, as long as there are variables in $\chi(p)$ whose sets of possible tuples are bounded by atoms in some child of p.

To evaluate a conjunctive query Q on a database **DB**, given a q-hypertree decomposition $HD = \langle T, \chi, \lambda \rangle$ for it, we use an adaptation of the Yannakakis's algorithm described next.

A *q-hypertree evaluator* is a procedure that, given as its input $\langle HD, Q, \mathbf{DB} \rangle$, performs the following steps: (P') For each vertex $p \in vertices(T)$, compute the join of the relations associated to atoms in $\lambda(p)$, and project the result onto the variables in $\chi(p)$; (P'') Following a topological order of T, evaluate each vertex p of the decomposition tree by taking the join of the relation at p with each one of its children and projecting the result onto $\chi(p)$'s variables; (P''') Output the projection of the relation at the root of T onto the variables in $out(Q)$.

Example 1.3.6 *Consider the following SQL query Q_1:*

```
SELECT A, S, max(X) FROM a,b,c,d,e,f,g,h,i
WHERE a.B=b.B and b.C=d.C and d.T=e.T and e.R=f.R and
      f.Y=c.Y and g.X=c.X and g.S=i.S and h.Z=i.Z
GROUP BY A,S
```

Figure 1.6 shows two q-hypertree decomposition HD_1 and HD'_1 of $CQ(Q_1)$. Note that both of them have width 2, though $hw(\mathcal{H}(Q_1)) = 1$, as the query hypergraph is acyclic. In fact this is the best we can do, because we want to start from a hypertree decomposition like HD_1 (to keep its nice properties), but we have to satisfy also Condition 2 in Definition 1.3.5. Intuitively, this condition may introduce *cycles in the hypergraph, through the connections involving output variables. This is what we are going to pay, to avoid the additional top-down and its subsequent bottom-up evaluation of the decomposition tree for non-Boolean queries (steps (ii) and (iii) in Section 1.3.2).*

Looking at the simpler hypertree HD'_1 in Figure 1.6, we can see how the feature (b) *of q-hypertree decompositions works, by allowing the optimizer to get better plans. Compare the right subtrees of the roots of HD_1 and HD'_1: The atom a in the first vertex, and the atom c in the subsequent three vertices do not occur in the λ labels of their corresponding vertices in HD'_1. Hence, the preliminary evaluation Step P' requires for HD'_1 only one join operation (at the root) instead of 5, as it is for the hypertree HD_1.* $\square$ ■

Note that, in general, there are different ways of evaluating Q on **DB** w.r.t. HD, depending on the choice of the topological order of the decomposition tree. Moreover, observe that, at Step P'', we take joins instead of semi-joins, because some variables at a vertex p may be covered by its children, rather than by its own atoms. In fact, differently from the corresponding Step S'_2 described in Section 1.3.2, the output of Step P' is not an acyclic query equivalent to Q. It follows that, in principle, the size of such intermediate joins—and hence the cost of P''—may increase exponentially, even if the width is bounded by the constant k.

Thus, a key issue is the computation of a q-hypertree decomposition that can be evaluated efficiently, that is, such that the size of intermediate relations cannot blow-up, and the whole procedure has a polynomial-time upper bound in the combined size of the input and of the output.

Definition 1.3.7 A q-hypertree decomposition HD of a conjunctive query Q is said *good*, if there is a q-hypertree evaluator that, given as its input $\langle HD, Q, \mathbf{DB}\rangle$, takes polynomial-time in $\|Q\| + \|\mathbf{DB}\| + \|Q(\mathbf{DB})\|$ to compute the answer of Q on **DB**. $\square$

Putting it all together, given an SQL query Q on a database **DB**, our algorithm for evaluating Q by exploiting hypertree decompositions consists of the following steps: (1) we compute the conjunctive query $CQ(Q)$, as described in Section 2.5.4; (2) we compute a good q-hypertree decomposition HD of $CQ(Q)$; (3) we compute the answer of $CQ(Q)$ on **DB** by means of a suitable q-hypertree evaluation of Q on **DB** w.r.t. HD; and (4) we evaluate possible aggregate operators (including group-by computations) working on the answer of $CQ(Q)$.

Step (4) is implemented in any standard way, as, by definition, variables in $out(Q)$ include all variables involved in such aggregate operators. It remains to

```
Input: A conjunctive query $Q$.
Output: A good q-hypertree decomposition $HD = \langle T, \chi, \lambda \rangle$ of $Q$.

  Compute a "minimal" NF hypertree decomposition $HD$ of $\mathcal{H}(Q)$ such that
      Condition 2 in Definition 1.3.5 is satisfied, the width of $HD$ is at most $k$,
      and, for the root $r$ of $HD$, $out(Q) \subseteq \chi(r)$ holds;
  If no such a decomposition exists,
      Output "Failure";
  else
      Execute Optimize $(HD, r)$;
      Output $HD$.
------------------------------------------------------------------------
 Procedure Optimize(VAR $HD, p \in vert(HD)$)
  For each hyperedge $a \in \lambda(p)$ Do
    If $\exists q \in child(p)$ and $b \in \lambda(q)$ such that $a \cap \chi(p) \subseteq b \cap \chi(q)$
      Then Remove $a$ from $\lambda(p)$;
  For each $q \in child(p)$ Do Optimize($HD, q$).
```

Figure 1.7: **ALGORITHM** q-HypertreeDecomp.

describe how to implement Step (2) and Step (3), that will be the subjects of the following sections.

Computing good q-Hypertree Decompositions

An algorithm that given a conjunctive query Q returns a good q-hypertree decomposition of Q is reported in Figure 1.7. This algorithm depends on a fixed constant k, which bounds the width of the decompositions to be considered (typically, $k = 4$ is enough for database queries). Firstly, the algorithm computes a width-k hypertree decomposition HD of $\mathcal{H}(Q)$ that satisfies Condition 2 in Definition 1.3.5, if any. Note that in general there is an exponential number of hypertree decompositions of a hypergraph, leading to different query-evaluation performances. Therefore, we have implemented an algorithm (based on the ideas in [63]) that evaluates different hypertrees according to a cost model for physical operators. Specifically, the cost model is based on a number of estimates about the operations on the input database, computed with standard techniques described in [33]. Then, rather than looking at all possible hypertree decompositions, we focus on those hypertree decompositions in having the minimum associated cost, as they correspond to "optimal" query evaluation plans for Q over . Notably, as shown in [63], computing such a best query plan can be done efficiently, if we consider normal form (NF) decompositions (again, it is feasible in $\mathrm{L}^{\mathrm{LOGCFL}}$).

After the computation of this hypertree decomposition of $\mathcal{H}(Q)$, we execute the *Procedure Optimize*, which simplifies the hypertree HD by removing hyperedges from the λ labels, in order to get a more efficient query plan for the query, without giving up the guarantee on the polynomial-time upper bound on the evaluation of $CQ(Q)$.

Let $k \geq 1$ be a fixed constant. Given a conjunctive query Q, Algorithm q-

HypertreeDecomp in Figure 1.7 runs in polynomial time, and outputs a good q-hypertree decomposition of Q, or “Failure”. The latter output is returned if and only if there is no hypertree decomposition of $\mathcal{H}(Q)$ having width at most k that satisfies Condition 2 in Definition 1.3.5.

For space limitations, we just give a rough idea on how the procedure *Optimize* works, guided by the example in Figure 1.6. In HD_1, consider the right child p_r of the root, where atom a is removed from $\lambda(p_r)$. In principle, this atom is useful here, because it provides a bound on the possible values for variable B coming from the bottom of the tree (but non for variable A, that does not belong to $\chi(p_r)$). However, B is also contained in the atom b in the child of this vertex. Then, we may think of an equivalent hypertree decomposition where a is replaced by b in this vertex. Now, it is clear that, in any evaluation of the decomposition tree, there is no sense in computing the join operation between the two bs in these adjacent vertices. Indeed, the result of such an operation would be exactly the relation corresponding to the atom b (possibly already filtered by previous join operations executed in the bottom-up evaluation). Thus, a may be replaced by b, and b is useless, whence we can just delete a from that vertex, as far as the polynomial-time upper bound is concerned. Intuitively the bounding effect on the variables in $a \cap \chi(p_r)$ (in this case, only B) is guaranteed by the atom b in the child of p_r.

It is worthwhile noting that, in more complex examples where such a simplified atom has many children, the topological order used in the evaluation of the join tree should take care of the children used for the simplification, that have to be joined with their parent before the other siblings. Otherwise, intermediate relations with exponentially many tuples can be temporary computed.

1.3.6 System Architecture

The structural approach described in so far has been implemented and integrated into a prototype system, which can be used either as a stand-alone application, or as a module plugged-in the open source PostgreSQL DBMS.

In the former case, the system rewrites the user query in a set of SQL views (based on its structural decomposition), which can be evaluated on top of any DBMS — in this case, possible statistical information about data should be provided explicitly by the user, and the DBMS optimizer is responsible for the translation of the logical query plan into the physical one.

In the latter case, instead, the decomposition algorithms have been tightly integrated in the PostgreSQL optimizer, so that *(1)* the optimization process is completely transparent to the user, and *(2)* additional information about data can automatically be exploited, to find a good query plan.

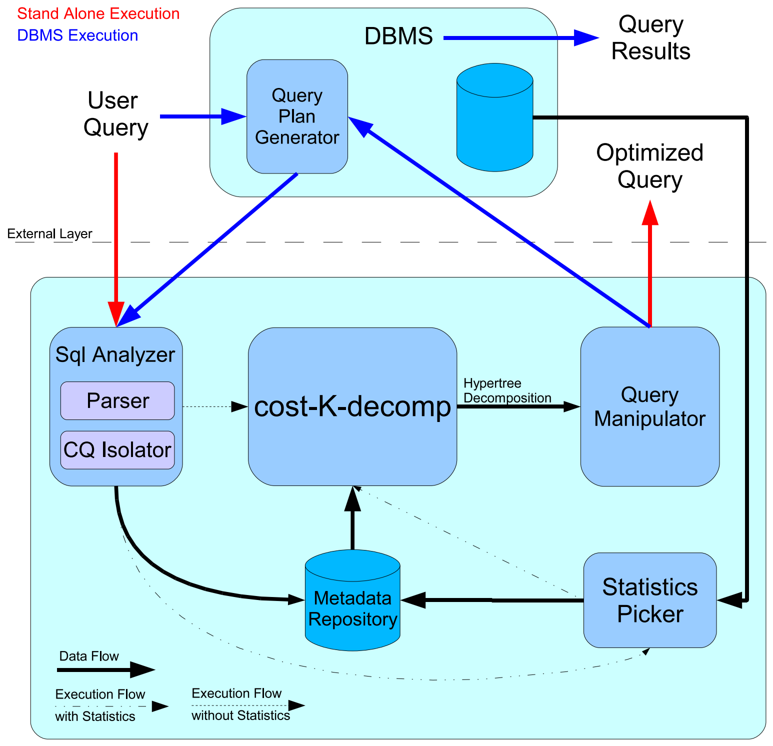


Figure 1.8: System architecture.

Figure 1.8 illustrates a functional view of the system architecture. Basically, it is formed by the following modules:

Sql Analyzer. It is responsible for preprocessing the query. First, the *Sql Parser* verifies its syntactical correctness, and then the *Conjunctive Query Isolator* computes the associated query hypergraph, which is the basis for the structural optimization.

Statistics Picker. This module is responsible for collecting the statistics about the relations involved in the query. When the system is coupled with PostgreSQL, these statistics are directly accessible from the DBMS optimizer. Otherwise, i.e., in the stand-alone usage, the user may optionally indicate the cardinality of the involved relations, and the selectivity of their attributes.

`cost-`k`-decomp`. This is the fundamental module of our architecture. It picks from the *Metadata Repository* statistics about data, together with the query hypergraph generated by the *Sql Analyzer*, and produces a q-hypertree decomposition according to the ideas described in Section 1.3.5.

Query Manipulator. In the case of the direct integration in PostgreSQL, the module produces a suitable data structure which is used to implement the bottom-up strategy discussed in the paper. Otherwise, i.e., in the stand-alone usage, the query plan is returned to the user in terms of a rewritten SQL query, which can be evaluated on top of any DBMS (possibly, disabling its internal optimizer).

Tight Coupling with PostgreSQL

Since our system is fairly the first attempt to integrate structural optimizations into the core of standard (quantitative) query optimizers, it is relevant to discuss in more details how this coupling has been practically achieved.

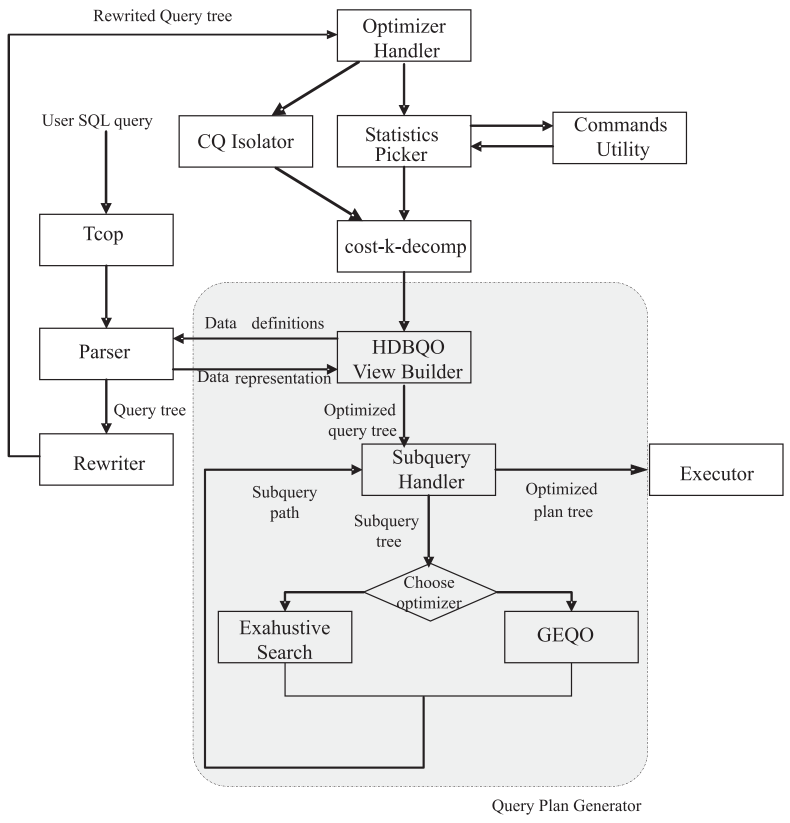


Figure 1.9: Integration in PostgreSQL.

In PostgreSQL, queries are processed as follows (see Figure 1.9). The *Tcop* module intercepts user requests and forwards them to *Parser*, which performs the syntactical and semantical analysis and produces a structured representation called *Query tree*. The *Rewriter* elaborates the query tree for optimization purposes and sends the result to the *Optimizer handler*, which is in charge of selecting the way the optimization has to be carried out. In the current implementation of PostgreSQL, two distinct and alternative optimizers are available: one performing an *exhaustive search*, and another using a genetic algorithm (*GEQO*).

To make the coupling possible, we modified the *Optimizer handler*, so that the control is no longer directly passed to either of the optimizers. Rather, both the *CQ Isolator* and the *Statistics picker* are invoked. In particular, statistics about data are now collected from the PostgreSQL *Commands Utility*.

Then, the *Query Plan Generator* is invoked: first, the *HDBQO ViewsBuilder* is responsible for building an optimized query tree based on the q-hypertree decomposition produced by `cost-`k`-decomp`, expressed in terms of nested SQL subqueries; then, each subquery is processed by the *HDBQO SubQueryHandler*, which is in charge of its execution on top of the built-in PostgreSQL optimizer.

1.3.7 Experimental Results

Compared Methods. Query plans for a number of queries were generated and their performances were compared with those produced by a commercial DBMS, that will be called CommDB for license reason, and PostgreSQL. Specifically, we used CommDB to evaluate the performances of our stand-alone architecture, and PostgreSQL to assess the advantages of a direct coupling inside a DBMS.

Benchmark Queries and Data. Tests included different kinds of queries. For each of them, we varied the hypertree width, the number of involved relations, their cardinality, and the selectivity of their attributes. Our attention was primarily devoted to test the optimization strategies on the standard *TPC-H* benchmarks. In addition, we considered the following kinds of query:

- *Acyclic Queries.* These are acyclic queries whose hypergraph has the form of a line: $q(\mathbf{y}) \leftarrow p_1(\mathbf{x_1}), p_2(\mathbf{x_2}), ..., p_n(\mathbf{x_n})$, where $\mathbf{x_i}$ denotes the set of variables occurring in the query atom p_i. We considered queries such that $\mathbf{x_i} \cap \mathbf{x_{i+1}} \neq \emptyset$, for any $1 \leq i < n$, and $\mathbf{x_i} \cap \mathbf{x_j} = \emptyset$, for any $i \notin \{j+1, j-1\}$. We experimented with queries whose length n ranges from 2 to 10.

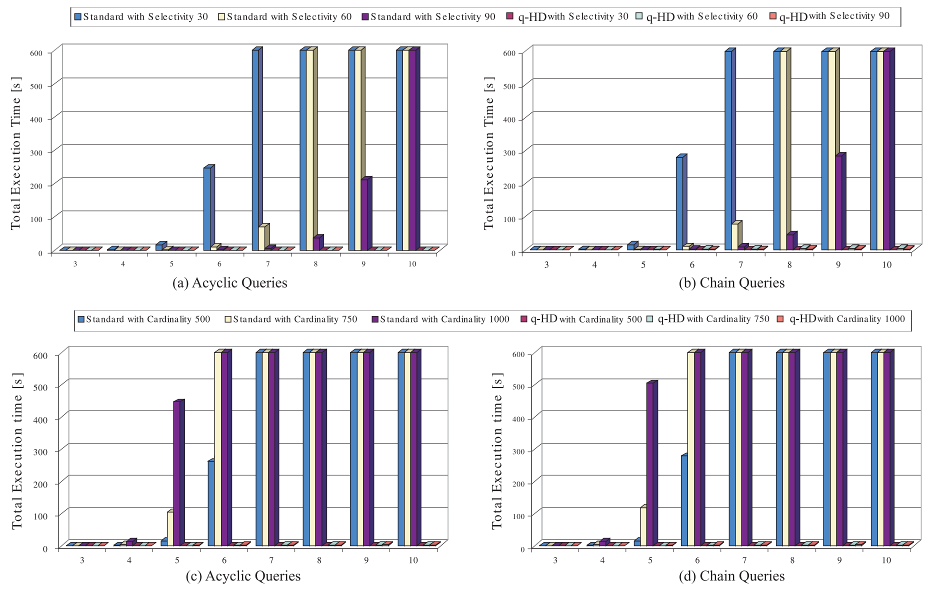


Figure 1.10: Execution times w.r.t. number of body atoms: (a) and (b) for attributes selectivity values 30, 60 and 90 — cardinality 500. (c) and (d) for databases of 500, 750 and 1000 tuples — selectivity 30.

- *Chain Queries.* They are the simplest cyclic variation of the above lines, where the first and the last query atoms have a variable in common ($\mathbf{x_1} \cap \mathbf{x_n} \neq \emptyset$).

TPC-H queries were evaluated over data generated by the *dbgen* tool provided by TPC. For the other kinds of queries, synthetic data were used, which has been generated randomly by using an uniform distribution over a fixed range of values, and setting the desired values for the cardinality of each relation and the selectivity of each attribute.

All experiments were performed on a 2,66Ghz Pentium 4 laptop, equipped with 512 Mb of ram and a 5400 rpm hard disk, running Windows XP Professional.

Experimenting with CommDB

We executed TPC-H queries on CommDB, both with and without its standard optimizer, to execute queries according to the q-hypertree decomposition method (q-HD). In the latter case, we report the *total* execution time, i.e., the summation of the stand-alone optimization time and of the CommDB evaluation time.

Figure 1.11 shows results for two TPC-H queries, Q_5 and Q_8, having hypertree width 2 (that is, two cyclic queries). For these queries, the use of statistics for q-HD had no impact on the computed query plans, which means that, for these queries and according to our cost model, exploiting the structure was estimated more important than exploiting the information on the database. Thus, the results shown in this figure for q-HD, are in fact the same as those obtained without any information on the data, that is, by using q-HD as a purely structural method.

The picture is completely different for CommDB: standard execution time, when it is not allowed to use statistics on the data, dramatically grows with the database size, and the evaluation quickly becomes infeasible. Even when CommDB is allowed to exploit statistics on the data, the use of q-hypertree decomposition (q-HD) improves the query evaluation performances. Of course, such good results are closely related with the peculiarities of queries Q_5 and Q_8, which are cyclic and involve many join operations. In fact, on queries where the structure plays instead a marginal role, q-HD used as a purely structural method is generally not competitive with CommDB exploiting statistics.

Anyway, it is relevant to notice that gathering statistics is expensive (for 1GB, 800 seconds are needed) while building a structure-based query plan takes an average time of 1.5 seconds—not affected by the database size, and usually leads to good performances. Hence, besides the cases of long or cyclic queries like Q_5 and Q_8, q-HD could be very useful in all those applications where statistics are not available (or not yet).

Experimental results for Acyclic and Chain queries further confirmed the above intuitions. The results are depicted in Figure 1.10, which reports query execution

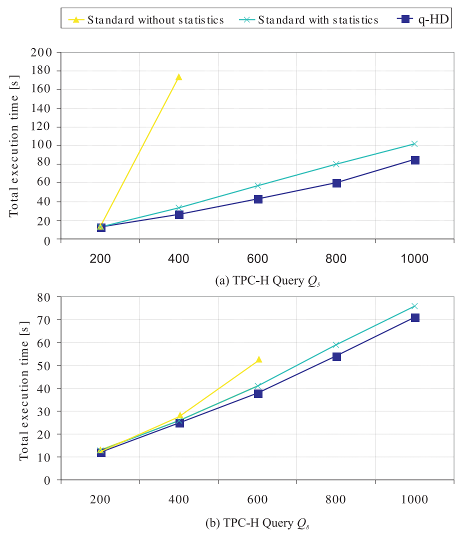


Figure 1.11: Execution time on TPCH-Queries: CommDB vs `cost-`k`-decomp`. Execution Times with database size ranging from 200mb to 1000mb: (a) Query Q_5. (b) Query Q_8.

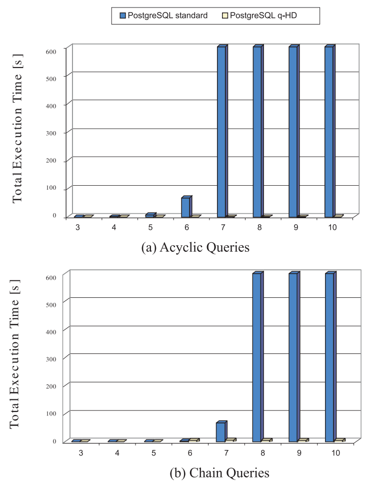


Figure 1.12: PostgreSQL: Execution times w.r.t. number of body atoms — selectivity 60, cardinality 450.

times, varying the number of body atoms and with different values for cardinality of relations and selectivity of attributes—where CommDB is able to use statistics on the data.

Notice that, for queries with 10 atoms in the body, CommDB executions do not terminate after more than 10 minutes while the q-HD driven executions take just a few seconds. This evidences that, when the size of the query grows (especially if combined with its its intricacy), current DBMS optimizers often fail in finding good query plans and structural decomposition methods can significantly improve their performances.

Experimenting with PostgreSQL

All the experiments described above have been repeated with our prototype directly implemented inside PostgreSQL 8.3. The relative gain turned out to be even higher than for the tests with CommDB, since in this scenario query evaluation can benefit of both the structural methods and the quantitative statistics about data.

As an example, we reported in Figure 1.12 the execution times on Acyclic and Chain queries, for a synthetic database where each relation contains 450 tuples and whose attributes have selectivity 60 (by lowering the selectivity, the gain of the structural approach is even more evident).

The reader may notice that the basic PostgreSQL optimizer performs quite poorly when compared with CommDB, since evaluating an acyclic query with

6 body atoms takes about 80 second in this scenario, while it is feasible in a few seconds by CommDB (cf. Figure 1.10.(a)). However, when the structural methods are integrated in PostgreSQL (q-HD), we get some quite surprising results, since its query evaluation nicely scales up to 10 body atoms, while CommDB (without the use of structural optimizations) does not terminate after 10 minutes, even for 8 atoms only.

Finally, in the last set of experiments, we explore the benefits of using the procedure *Optimize* in Figure 1.7, and hence of exploiting feature (b) of q-hypertree decompositions. The results for chain queries are shown in Figure 1.13 (over the same dataset as in Figure 1.12).

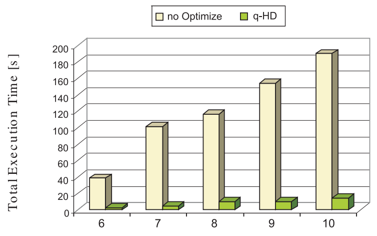


Figure 1.13: Impact of Procedure *Optimize*.

Chapter 2

Data Integration Systems

2.1 Data Integration Systems

Information integration is the problem of combining the data residing at different sources, and providing the user with a unified view of these data, called *global schema*. The global schema is therefore a reconciled view of the information, which can be queried by the user. It can be thought of as a set of virtual relations, in the sense that their extensions are not actually stored anywhere. A data integration system frees the user from having to locate the sources relevant to a query, interact with each source in isolation, and manually combine the data from different sources.

The interest in this kind of systems has been continuously growing in the last years. Many organizations face the problem of integrating data residing at several sources. Companies that build a Data Warehouse, a Data Mining, or an Enterprise Resource Planning system must address this problem. Also, integrating data in the World Wide Web is the subject of several investigations and projects nowadays. Finally, applications requiring accessing or re-engineering legacy systems must deal with the problem of integrating data stored in different sources.

The design of a data integration system is a very complex task, which comprises several different issues, including the following:

1. heterogeneity of the sources,

2. mapping between the global schema and the sources,

3. limitations on the mechanisms for accessing the sources,

4. materialized vs. virtual integration,

5. data cleaning and reconciliation,
6. how to process queries expressed on the global schema,
7. how to deal with integrity constraints.

Problem (1) arises because sources are typically heterogeneous, meaning that they adopt different models and systems for storing data. This poses challenging problems in both representing the sources in a common format within the integration system, and specifying the global schema. As for the first issue, data integration systems make use of suitable software components, called wrappers, that present data at the sources in the form adopted within the system, hiding the original structure of the sources and the way in which they are modeled. With regard to the specification of the global schema, the goal is to design such a schema so as to provide an appropriate abstraction of all the data residing at the sources. One aspect deserving special attention is the choice of the language used to express the global schema. Since such a view should mediate among different representations of overlapping worlds, the language should provide flexible and powerful representation mechanisms.

With regard to Problem (2), two basic approaches have been used to specify the mapping between the sources and the global schema. The first approach, called query-centric or *global-as-view* (GAV), requires that the global schema is expressed in terms of the data sources. More precisely, to every concept of the global schema, a view over the data sources is associated, so that its meaning is specified in terms of the data residing at the sources. The second approach, called source-centric or *local-as-view* (LAV), requires the global schema to be specified independently of the sources. The relationships between the global schema and the sources are established by associating each element of the sources with a view over the global schema. Thus, in the local-as-view approach, we specify the meaning of the sources in terms of the concepts in the global schema. It is clear that the latter approach favors the extensibility of the integration system, and provides a more appropriate setting for its maintenance. For example, adding a new source to the system requires only to provide the definition of the source, and does not necessarily involve changes in the global view. On the contrary, in the global-as-view approach, adding a new source may in principle require changing the definition of the concepts in the global schema. A different approach could be to specify the mapping by combining LAV and GAV views together. This approach, called *global-local-as-view* (GLAV), is quite recent and has so far drawn little attention in the literature (see, e.g., [54]), thus it will not be discussed further in this thesis.

Examples of LAV systems are Information Manifold [46], Infomaster [24], and the systems presented in [53] and [17]. Information Manifold and the system described in [17] express the global schema in terms of Description Logics [11, 7], and adopt the language of conjunctive queries to specify both user queries and views in the mapping. The system described in [53] uses an XML global schema, and adopts XML-based query languages for both the mapping and the queries on the global schema. More powerful schema languages for expressing the global schema are reported in [24, 42, 16, 15]. In particular, [24, 42] discuss the case where various forms of relational integrity constraints are expressible in the global schema, including functional and inclusion dependencies, whereas [16, 15] consider a setting where the global schema is expressed in terms of Description Logics, which allow for the specification of various types of constraints.

Examples of GAV systems are TSIMMIS [32], Garlic [18], COIN [34], MOMIS [9], Squirrel [88], and IBIS [14]. These systems usually adopt simple languages for expressing both the global and the source schemas. IBIS is the only system we are aware of that takes into account integrity constraints in the global schema.

Problem (3) refers to the fact that, both in the local-as-view and in the global-as-view approach, it may happen that a source presents some limitations on the types of accesses it supports. A typical example is a web source accessible through a form where one of the fields must necessarily be filled in by the user. This can be modeled by specifying the source as a relation supporting only queries with a selection on a column. Suitable notations have been proposed for such situations [57], and the consequences of these access limitations on query processing in integration systems have been investigated in several papers [57, 51, 30, 87, 50].

Problem (4) deals with a further criterion that one should take into account in the design of a data integration system. In particular, with respect to the data explicitly managed by the system, one can follow two different approaches, called *materialized* and *virtual*. In the materialized approach, the system computes the extensions of the concepts in the global schema by replicating the data at the sources. In the virtual approach, data residing at the sources are accessed during query processing, but they are not replicated in the integration system. Obviously, in the materialized approach, the problem of refreshing the materialized views in order to keep them up-to-date is a major issue [45]. Unless otherwise specified, in the following we only deal with the virtual approach.

Whereas the construction of the global schema concerns the intentional level of the data integration system, Problem (5) refers to a number of issues arising when considering integration at the extensional/instance level. A first issue in this context is the interpretation and merging of the data provided by the sources. Interpreting data can be regarded as the task of casting them into a common representation. Moreover, the data returned by various sources need to be converted/reconciled/combined to provide the data integration system with the requested information. The complexity of this reconciliation step is due to several problems, such as possible mismatches between data referring to the same real world object, possible errors in the data stored in the sources, possible inconsistencies between values representing the properties of real world objects in different sources [31]. The above task is known in the literature as *Data Cleaning and Reconciliation*, and the interested reader is referred to [31, 17, 12] for more details on this subject.

Problem (6) is concerned with one of the most important issues in a data integration system, i.e., the choice of the method for computing the answer to queries posed in terms of the global schema only on the basis of the data residing at the sources. The main issue is that the system should be able to re-express such queries in terms of a suitable set of queries posed to the sources, hand them to the sources, and assemble the results into the final answer.

Finally, Problem (7) arises because the language adopted to represent the integration domain should be powerful enough to cope with the issues highlighted at point (1), i.e. should be based on integrity constraints. Generally, in data integration, data at the sources are assumed to be coherent with the integrity constraints specified over the sources to which they belong, thus such constraints can be overlooked during query processing. On the other hand, data originally stored in autonomous sources may not satisfy the integrity constraints expressed in the global schema, and inconsistencies may arise in the integration system. Furthermore, integrity constraints represent fundamental knowledge about the real world and important requirements that the reconciled data have to respect, so that they cannot be neglected during query processing. Generally speaking, in the presence of some inconsistencies, traditional semantics for data integration systems consider the entire system inconsistent and are unable to support query processing even if most of the data at the sources satisfy the integrity constraints in the global schema. More recent approaches [52, 4, 5, 40] consider different semantics able to provide database instances for the global schema even in the presence of inconsistencies, and define techniques to compute answers to queries in such a scenario.

2.2 The Infomix Project

The principal goal of the INFOMIX project was to provide advanced techniques and innovative methodologies for information integration systems. In a nutshell, the project developed a theory, comprising a comprehensive information model and information integration algorithms, and a prototype implementation of a knowledge based system for advanced information integration, by using computational logic and integrating research results on data acquisition and transformation. Special attention was devoted to the definition of declarative user-interaction mechanisms, and techniques for handling semi-structured data, and incomplete and inconsistent data sources.

These objectives, which advanced the state of the art in several respects, are detailed as follows.

- **Comprehensive Information Model.** A comprehensive information model has to be provided, which incorporates static and dynamic aspects of information integration, and supports advanced *human like* reasoning, based on a rich semantics. Current information integration systems are rather poor in this respect, and provide only limited support (if any) for expressing constraint relationships between the local sources and a global view of the data. The source data are integrated in such systems under implicit assumptions such as soundness and completeness; arising inconsistencies are handled at low levels in a procedural manner, without a clear understanding to the user about the effects on the semantics of the overall system. What we barely need is a much richer information model in which knowledge about the sources, their semantics and relationships can be declaratively expressed, such that on the basis of a clear semantics, reasoning about the sources is possible and can be exploited for meaningful integration. Furthermore, the information model should be capable of expressing criteria such as source preference, or strategies for data integration that the user might select. From the declarative specification, the integration process may then provide results which are transparently obtained by exploiting all available knowledge in a meaningful way.

- **Information Integration Algorithms.** A host of efficient algorithms for information integration must be provided, which can be applied to homogenized data from heterogeneous data sources. On the computational side, we need a number of algorithms and techniques for advanced information integration, which cover different tasks. Besides general, high-level algorithms for the integration process, we need particular algorithms for solving advanced reasoning tasks in the integration process, such as checking consistency of source specifications, finding explanations for data inaccuracies and suggesting repairs etc. The design and development of such algorithms which are useable in practice is a nontrivial and challenging task, since more advanced reasoning tasks require often higher computational resources, and, moreover, the volume of resource data may be large. Thus, suitable optimization techniques for information integration must be devised in order to ensure the scalability of the approach.

- **Usage of Computational Logic.** Exploit advanced methodologies and techniques from computational logic as a toolbox for information integration. In the recent years, research in computational logic has produced a number of implemented systems by which various advanced reasoning problems such as diagnosis, configuration, etc can be declaratively solved in logic-based languages. The underlying computational engines have been developed (mostly in Europe) with quite some effort, and comprise a body of sophisticated tools and algorithms. Exploiting them for solving reasoning tasks in advanced information integration is a natural approach, but will require extensions and adaptations as for the needs of this application. The INFOMIX project, by its usage of computational logic, will contribute in strengthening the leading role of Europe in this key technology for building advanced reasoning systems.

- **Integration of results on data acquisition and transformation.** Selected research results from the area of data source acquisition and transformation should be integrated. Research on multi and federated database systems has made available a number of techniques and systems for accessing heterogeneous data at a homogenized level. Any advanced information integration system which should be used in a wider context must be capable of incorporating sources that provide data in different formats, such as relational data, object-oriented data, or semi-structured data. One of the INFOMIX objectives is to make use of selected existing results and techniques in the area of heterogeneous data acquisition and transformation, and to integrate them into the architecture of an advanced information integration system. As a result, a much more powerful system for combining heterogeneous data will be provided.

- **Prototype System.** Definition and implementation of a component-based integration system prototype, and providing an infrastructure by using software agent technology. It is only a prototype system by which we may validate the suitability of the methods and techniques developed. The prototype will implement the architecture of an INFOMIX information integration system and serve as a testbed for experimentation, from which conclusions about the project results and further research goals will be obtained. Agent technology will be used for system components, and in particular for incorporating heterogeneous data sources. Furthermore, the prototype system plays an important role in dissemination of the foundational results to the R&D industry.

The formal framework for data integration, that will be deeply investigated in Section 2.3, was defined in terms of a global schema, which provides the unified and centralized view of the data, a source schema, comprising the schemas of all sources involved in the integration application, and the mapping that specifies the relationship between the two. Generally speaking, the framework allows for the specification of constraints of general form on both the global schema and the sources, and the definition of complex forms of mappings between the global schema and the source schema. More specifically, the mapping is given in terms of a set of assertions where each assertion associates a view over the global schema to a view over the sources. Such an approach captures both LAV and GAV mappings, and allows also for the specification of more complex dependencies between elements of the global schema and elements of the sources.

With regard to the semantics, we deeply concentrated on the semantics of the mapping, and analyzed several assumptions that can be adopted on mapping assertions, in order to specify how to interpret data that can be retrieved at the sources with respect to data that satisfy the corresponding portion of the global schema. With this respect, we first considered classical *exact*, *sound*, or *complete* assertions, that correspond to the different situations in which data in the answer to a view over the global schema are exactly the data in the answer to the corresponding view over the sources, or are a superset or a subset of such data. Then, we addressed the more general case in which data retrieved at the sources do not respect integrity constraints expressed over the global schema, and cannot be reconciled in such a way that both integrity constraints and mapping assertions are satisfied. Classical assumptions on mapping assertions do not allowed us to properly handle such inconsistencies, and generally they bring about a situation where no database instance exists for the global schema. In this respect, we proposed a more general approach in which the classical assumptions on mapping assertions can be suitably relaxed.

It is worth noticing that the semantic problem that arises in this context is similar to the one underlying the notion of *database repair* introduced by several works in the area of *inconsistent databases*. However, many of this studies basically apply to a single database setting [21, 10, 6], and the proposed techniques can be employed in a data integration setting only by assuming an “exact” interpretation of mapping assertions [52, 23].

Finally, based on our formal model of a data integration system, and on the preliminary structure of the system outlined in the INFOMIX proposal, we have identified the general functionalities that the system should provide, thus providing a functional specification of the data integration system. In particular, we have divided the system features into four levels of capabilities:

- **Final User Level**, which comprises functionalities that allow users both to pose their queries to the system and to suitably access the results computed by the system.
- **Information Service Level**, which comprises functionalities that allow for the modeling of the global schema, the source schema, and the mapping, and for the reformulation of queries expressed over the global schema in terms of queries on the sources.
- **Internal Integration Level**, which comprises functionalities to optimize and execute the query plan computed by the query reformulation process.
- **Data Acquisition and Transformation Level**, which comprises functionalities to properly access the sources, retrieve data from them, and suitably transform the acquired data into the internal homogeneous data format adopted in the system.

2.3 Formal Framework

In this section we define a logical framework for data integration. The main components of a data integration system are the sources, the global schema and the mapping between the two. We first present the syntax, and then the semantics of a data integration system.

2.3.1 Syntax

We consider to have a fixed infinite alphabet Γ of constants representing real world objects, and assume that the structures constituting the databases involved in our framework are defined over the fixed interpretation domain Γ, i.e., Γ represents all the possible elements in a database instance.

Formally, a data integration system $\mathcal{I}$ is a triple $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, where:

- $\mathcal{G}$ is the *global schema*, expressed in the global language $\mathcal{L}_{\mathcal{G}}$ over the alphabet $\mathcal{A}^{\mathcal{G}}$. The alphabet comprises a symbol for each element of $\mathcal{G}$ (i.e., relation if $\mathcal{G}$ is relational, class if $\mathcal{G}$ is object-oriented, etc.). The language $\mathcal{L}_{\mathcal{G}}$ determines the expressiveness allowed for specifying the global schema, i.e., the set of constraints that can be defined over it;

- $\mathcal{S}$ is the *source schema*, composed by the schemas of the various sources that are part of the data integration system. $\mathcal{S}$ is modeled in the source language $\mathcal{L}_{\mathcal{S}}$ over the alphabet $\mathcal{A}^{\mathcal{S}}$. We request that, $\mathcal{A}^{\mathcal{S}}$ is disjoint from the alphabet of the global schema $\mathcal{A}^{\mathcal{G}}$. As in the previous case, the alphabet comprises a symbol for each element of the sources, and $\mathcal{L}_{\mathcal{S}}$ determines the expressiveness allowed for specifying the source schema;

- $\mathcal{M}$ is the *mapping* between $\mathcal{G}$ and $\mathcal{S}$, i.e., the specification of the relationship between the sources and the global schema. It is constituted by a set of *assertions* of the form

$$q_{\mathcal{S}} \sqsubseteq q_{\mathcal{G}},$$
$$q_{\mathcal{G}} \sqsubseteq q_{\mathcal{S}}$$

  where $q_{\mathcal{S}}$ and $q_{\mathcal{G}}$ are two queries, respectively over the source schema $\mathcal{S}$, and over the global schema $\mathcal{G}$. Queries $q_{\mathcal{S}}$ are expressed in a query language $\mathcal{L}_{\mathcal{M},\mathcal{S}}$ over the alphabet $\mathcal{A}_{\mathcal{S}}$, and queries $q_{\mathcal{G}}$ are expressed in a query language $\mathcal{L}_{\mathcal{M},\mathcal{G}}$ over the alphabet $\mathcal{A}_{\mathcal{G}}$. Intuitively, an assertion $q_{\mathcal{S}} \sqsubseteq q_{\mathcal{G}}$ specifies that the concept represented by the query $q_{\mathcal{S}}$ over the sources is put in correspondence with the concept in the global schema represented by the query $q_{\mathcal{G}}$ (similarly for an assertion of type $q_{\mathcal{G}} \sqsubseteq q_{\mathcal{S}}$). The exact meaning of such a correspondence will be described in the next subsection.

Thus, from the syntactic viewpoint, the specification of an integration system depends on the following parameters:

- The form of the global schema, i.e., the formalism used for expressing data and the global relationships between data.

- The form of the source schema, i.e., the formalism used for expressing data at the sources. Moreover, we assume that the data at the sources satisfy all constraints specified on $\mathcal{S}$, thus, in the following we do not consider anymore these constraints.

- The form of the mapping. In the data integration literature two possible forms for the mapping are studied, called respectively *global-as-view* (GAV) and *local-as-view* (LAV). The GAV approach requires that the global schema is defined in terms of the data sources: more precisely, every element of the global schema is expressed as a view over the sources, so that its meaning is specified in terms of the data residing at the sources. With respect to the mapping syntax above defined, the GAV approach corresponds to restricting the queries $q_{\mathcal{G}}$ to unary queries, i.e., queries containing a single element of the global schema. In the LAV approach, the meaning of the sources is specified in terms of the elements of the global schema: more exactly, the mapping $\mathcal{M}$ between the sources and the global schema is provided in terms of a set of views over the global schema, one for each source element. With respect to the mapping syntax above defined, the LAV approach corresponds to restricting the queries $q_{\mathcal{S}}$ to unary queries, i.e., queries containing a single element of the source schema. Therefore, the above definition of mapping of our framework corresponds to a generalized form that comprises LAV and GAV as special cases.

Finally, we consider *queries* posed to a data integration system and define their syntax. Each such query is a formula that is intended to provide the specification of which data to extract from the integration system, i.e., q is intended to extract a set of elements of Γ. Each query is issued over the global schema $\mathcal{G}$, and is expressed in a specific query language, denoted by $\mathcal{L}_{\mathcal{Q}}$, over the global alphabet $\mathcal{A}^{\mathcal{G}}$.

2.3.2 Semantics

Intuitively, to specify the semantics of a data integration system, we have to start with a set of data at the sources, and, given such data at the sources, we have to specify which are the data that satisfy the global schema. Thus, in order to assign the semantics to a data integration system $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, we start by considering a *source database* for $\mathcal{I}$, i.e., a database $\mathcal{D}$ for the source schema $\mathcal{S}$. Given a query q over $\mathcal{S}$ and a source database $\mathcal{D}$, we denote as $q^{\mathcal{D}}$ the set of objects in the answer to q over $\mathcal{D}$, i.e. the set of objects that satisfy q over $\mathcal{D}$.

Based on $\mathcal{D}$, we now specify which is the information content of the global schema $\mathcal{G}$. We call any database $\mathcal{B}$ for $\mathcal{G}$ a *global database* for $\mathcal{I}$. Furthermore, given a query q over $\mathcal{G}$ and a global database $\mathcal{B}$, we denote as $q^{\mathcal{B}}$ the set of objects in the answer to q over $\mathcal{B}$, i.e. the set of objects that satisfy q over $\mathcal{B}$.

A global database $\mathcal{B}$ for $\mathcal{I}$ is said to be *legal* with respect to $\mathcal{D}$ if:

1. $\mathcal{B}$ is coherent with $\mathcal{G}$;

2. $\mathcal{B}$ satisfies the mapping $\mathcal{M}$ with respect to $\mathcal{D}$, namely the objects in $\mathcal{B}$ satisfy the relationships between the global and the source elements defined by the mapping. More precisely, we say that $\mathcal{B}$ *satisfies* $\mathcal{M}$ *with respect to* $\mathcal{D}$ if:

   (a) for each assertion in $\mathcal{M}$ of the form $q_{\mathcal{S}} \sqsubseteq q_{\mathcal{G}}$, each object in $q_{\mathcal{S}}^{\mathcal{D}}$ is also an element of $q_{\mathcal{G}}^{\mathcal{B}}$, i.e., $q_{\mathcal{S}}^{\mathcal{D}} \subseteq q_{\mathcal{G}}^{\mathcal{B}}$;

   (b) for each assertion in $\mathcal{M}$ of the form $q_{\mathcal{G}} \sqsubseteq q_{\mathcal{S}}$, each object in $q_{\mathcal{G}}^{\mathcal{B}}$ is also an element of $q_{\mathcal{S}}^{\mathcal{D}}$, i.e., $q_{\mathcal{G}}^{\mathcal{B}} \subseteq q_{\mathcal{S}}^{\mathcal{D}}$.

Notice that, from the above semantics of the mapping $\mathcal{M}$, it follows that in our framework it is possible to express all the kinds of interpretations of the mapping assertions studied in data integration, namely the sound, complete, and exact interpretation. In particular, if we want to formulate a generic mapping assertion V defining a relationship between the query over the global schema $q_{\mathcal{G}}$ and the query over the source schema $q_{\mathcal{S}}$:

- a *sound* interpretation of V corresponds in our framework to the assertion $q_{\mathcal{S}} \sqsubseteq q_{\mathcal{G}}$;

- a *complete* interpretation of V corresponds to the assertion $q_{\mathcal{G}} \sqsubseteq q_{\mathcal{S}}$;

- an *exact* interpretation of V corresponds to the pair of assertions $q_{\mathcal{S}} \sqsubseteq q_{\mathcal{G}}$, $q_{\mathcal{G}} \sqsubseteq q_{\mathcal{S}}$.

In an analogous way, one can express sound, complete or exact interpretations of a mapping assertion defining a relationship between $q_{\mathcal{S}}$ and $q_{\mathcal{G}}$.

Given a source database $\mathcal{D}$ for $\mathcal{I}$, the semantics of $\mathcal{I}$ with respect to $\mathcal{D}$, denoted $sem(\mathcal{I}, \mathcal{D})$, is defined as follows:

$$sem(\mathcal{I}, \mathcal{D}) = \{ \mathcal{B} \mid \mathcal{B} \text{ is a legal global database for } \mathcal{I} \text{ w.r.t. } \mathcal{D} \}$$

Let us now turn our attention to queries. In order to define the semantics of a query q over the global schema $\mathcal{G}$, we have to take into account all the global databases legal for $\mathcal{I}$ with respect to $\mathcal{D}$. We call *certain answers* (or simply *answers*) of a query q with respect to $\mathcal{I}$ and $\mathcal{D}$, the set $q^{\mathcal{I},\mathcal{D}}$ of objects t such that $t \in q^{\mathcal{DB}}$ for *every* database $\mathcal{DB} \in sem(\mathcal{I},\mathcal{D})$.

Furthermore, we call *possible answers* of a query q the set $q^{\mathcal{I},\mathcal{D}}$ of objects t such that $t \in q^{\mathcal{DB}}$ for *some* database $\mathcal{DB} \in sem(\mathcal{I},\mathcal{D})$.

From the above definitions, it is easy to see that, in data integration, answering queries is essentially an extended form of reasoning in the presence of incomplete information [79]. Indeed, when we answer the query, we know only the extensions of the sources, and this provides us with only partial information on the global database.

2.3.3 Dealing with inconsistent data sources

According to the semantics above defined, in which we adopted a first-order logic interpretation of the mapping, it may be the case that the data retrieved from the sources cannot be reconciled in the global schema in such a way that both the constraints of the global schema and the mapping are satisfied [48]. This happens, for instance, in a relational context, when a key constraint specified for the relation r in the global schema is violated by the tuples retrieved by the view associated to r, since the assumption of sound views does not allow us to disregard tuples from r with duplicate keys. If we do not want to conclude in this case that no global database exists that is legal for $\mathcal{I}$ with respect to $\mathcal{D}$, we need a different characterization of the mapping. In particular, we need a characterization that allows us to support query processing even when the data at the sources are incoherent with respect to the integrity constraints on the global schema.

A possible solution is to characterize the data integration system $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ in terms of those global databases that:

1. satisfy the integrity constraints of $\mathcal{G}$, and

2. approximate at best the satisfaction of the assertions in the mapping $\mathcal{M}$, i.e., that are *as close as possible* to the mapping $\mathcal{M}$.

In other words, the integrity constraints of $\mathcal{G}$ are considered “strong”, whereas the mapping is considered “soft”.

We now propose a modified definition of the semantics for the integration system that reflects the above idea. Given a source database $\mathcal{D}$ for $\mathcal{I}$, we define a partial order (based on set-containment) between the global databases for $\mathcal{I}$. If $\mathcal{B}_1$ and $\mathcal{B}_2$ are two databases that are legal with respect to $\mathcal{G}$, we say that $\mathcal{B}_1$ is *better* than $\mathcal{B}_2$ with respect to $\mathcal{D}$, denoted as $\mathcal{B}_1 \gg_{\mathcal{D}} \mathcal{B}_2$, if:

1. for each assertion $q_{\mathcal{S}} \sqsubseteq q_{\mathcal{G}}$ in $\mathcal{M}$, $q_{\mathcal{G}}^{\mathcal{B}_1} \cap q_{\mathcal{S}}^{\mathcal{D}} \supseteq q_{\mathcal{G}}^{\mathcal{B}_2} \cap q_{\mathcal{S}}^{\mathcal{D}}$;
2. for each assertion $q_{\mathcal{G}} \sqsubseteq q_{\mathcal{S}}$ in $\mathcal{M}$, $q_{\mathcal{G}}^{\mathcal{B}_1} - q_{\mathcal{S}}^{\mathcal{D}} \subseteq q_{\mathcal{G}}^{\mathcal{B}_2} - q_{\mathcal{S}}^{\mathcal{D}}$;
3. at least one of the following conditions holds:
   (a) there exists an assertion $q_{\mathcal{S}} \sqsubseteq q_{\mathcal{G}}$ in $\mathcal{M}$ such that $q_{\mathcal{G}}^{\mathcal{B}_1} \cap q_{\mathcal{S}}^{\mathcal{D}} \supset q_{\mathcal{G}}^{\mathcal{B}_2} \cap q_{\mathcal{S}}^{\mathcal{D}}$;
   (b) there exists an assertion $q_{\mathcal{G}} \sqsubseteq q_{\mathcal{S}}$ in $\mathcal{M}$ such that $q_{\mathcal{G}}^{\mathcal{B}_1} - q_{\mathcal{S}}^{\mathcal{D}} \subset q_{\mathcal{G}}^{\mathcal{B}_2} - q_{\mathcal{S}}^{\mathcal{D}}$.

Intuitively, this means that there is at least one assertion for which $\mathcal{B}_1$ satisfies the sound mapping better than $\mathcal{B}_2$, while for no other assertion $\mathcal{B}_2$ is better than $\mathcal{B}_1$. In other words, $\mathcal{B}_1$ approximates the mapping better than $\mathcal{B}_2$.

It is easy to verify that the relation $\gg_{\mathcal{D}}$ is a partial order. With this notion in place, we can now define the notion of $\mathcal{B}$ satisfying the mapping $\mathcal{M}$ with respect to $\mathcal{D}$ in our setting: a database $\mathcal{B}$ that is legal with respect to $\mathcal{G}$ satisfies the mapping $\mathcal{M}$ with respect to $\mathcal{D}$ if $\mathcal{B}$ is maximal with respect to $\gg_{\mathcal{D}}$, i.e., for no other global database $\mathcal{B}'$ that is legal with respect to $\mathcal{G}$, we have that $\mathcal{B}' \gg_{\mathcal{D}} \mathcal{B}$.

The notion of legal database for $\mathcal{I}$ with respect to $\mathcal{D}$, and the notions of answers remain the same, given the new definition of satisfaction of mapping. It is immediate to verify that, if there exists a legal database for $\mathcal{I}$ with respect to $\mathcal{D}$ under the first-order logic interpretation of the mapping, then the new semantics and the old one coincide, in the sense that, for each query q, the set $q^{\mathcal{I},\mathcal{D}}$ of certain answers computed under the first-order semantics coincides with the set of certain answers computed under the new semantics.

2.4 The System

INFOMIX has been conceived as an extensible software environment, where new modules can be easily added, e.g., for dealing with further kinds of data sources. In the following a detailed description of the overall INFOMIX system architecture is specified, comprising information flows, data management tasks and interactions between the various modules composing the system.

The general architecture of the INFOMIX system is quite complex. In order to simplify the description of the several system components, we have identified two fairly independent components of the architecture, namely the *Design-Time* and the *Run-Time* system architecture. The former includes activities and features involving the *design* phases of the data integration process; the latter refers to querying activities carried out by the users during the system exploitation.

For each component introduced we give a general description of the associated architecture.

2.4.1 Design-Time System Architecture

The activities involved in the design of a data integration system are:

- the identification of the information sources to be considered in the data integration tasks;
- the definition of suitable wrappers allowing to retrieve and suitably transform data residing at the sources;
- the design of a global scheme representing in a uniform and consistent way all the information stored in the selected sources,
- the definition of mappings stating how data at the sources are to be transformed and integrated in order to obtain global objects.

These activities are mostly performed during the initial steps of the data integration system definition. However, in order to cope with the dynamic nature of the integration applications, designers have the possibility to modify previous settings. The access to design activities is allowed only to designers and not to end users.

In Figure 2.1 the INFOMIX system architecture relative to design activities is illustrated. All the features provided by this part of the INFOMIX system are accessed by a graphical user interface. The *Wrapper Generator Interface* allows the designer to specify the information sources that are involved in the data integration system; once sources have been identified and specified, the *Wrapper Generator* module can be activated. It receives a set of sources to be wrapped and, for each of them, generates suitable wrappers. Associations between sources and wrappers, as well as information about the structure of the data which are present in the various sources are stored in the *Meta Data Repository*. The designer can monitor the wrapper generation activities through the *Wrapper Generation Interface* by

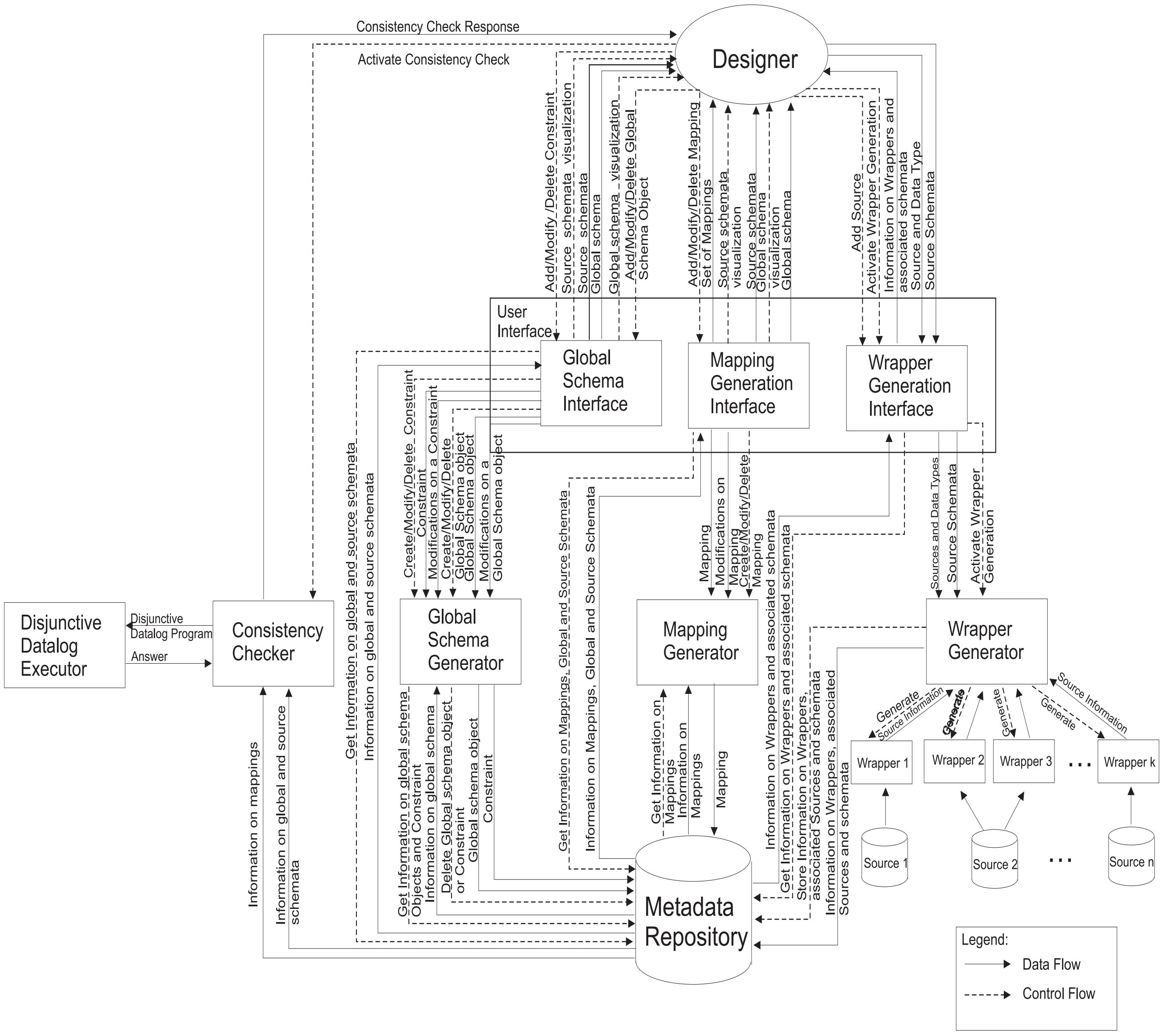


Figure 2.1: Design-Time Architecture of the INFOMIX system

displaying and managing the information stored in the *Metadata Repository* by the *Wrapper Generator*.

The global scheme is designed by using the *Global Scheme Interface*. In particular, this module gives the designer access to information about the schemes of the sources participating in the system, in order to define global scheme objects and constraints. While defining the global scheme, the *Global Scheme Interface* shows an up to date version of the global scheme. Insertions and modifications of global scheme objects and constraints are performed by the *Global Scheme Generator*; it represents global scheme objects and constraints in a suitable language and stores them in the *Metadata Repository*. Stored objects can be possibly deleted.

Once the global scheme has been designed, mappings between the source scheme objects and the global scheme objects can be specified by means of the *Mapping Generation Interface*. This module shows to the designer both the global scheme and the source schemes in order to facilitate the mapping definition phase. Mapping creation or modification requests performed by the designer are given as input to the *Mapping Generator* module. This module translates mappings in a suitable internal language, stores them in the *Metadata Repository* and handles modifications and updates required by the designer in order to guarantee that the *Metadata Repository* stores always the up to date version of them.

Finally, the designer may activate the *Consistency Checker* which verifies the correctness and the consistency of global scheme and mapping definitions. If an inconsistency is detected, the module informs the designer who has to modify global source objects and mapping definitions involved in the inconsistency loop.

In the following we give a detailed description of the functionalities and the interfaces of the modules introduced above.

Wrapper Generation Interface

The *Wrapper Generation Interface* provides the designer with a graphical interface allowing to specify the information sources to be considered in the data integration system. This module provides the following functionalities:

- Specification of sources to be wrapped. This feature requires the designer to provide as input the set of sources to be considered for the data integration system. In particular, for each source, the designer specifies the source location and the corresponding data format. The designer should also be able to either provide the relational scheme which the source should be wrapped to, or, supervise a scheme automatically proposed by the Wrapper Generator.

- Activation of the *Wrapper Generator*. In this case, the input consists just in an activation message prompt, provided by the designer, indicating that the wrapping generation step can take place. The output is the set of sources previously identified by the designer; these are given as input to the *Wrapper Generator* module.

- Visualization and browsing of the information about generated wrappers and associated schemes. In order to perform this task, the *Wrapper Generation Interface* retrieves from the *Data Repository* the information produced by the *Wrapper Generator* in the generation phase and shows it in a suitable format to the designer.

Global Scheme Interface

The *Global Scheme Interface* allows the designer to define the global scheme. In particular, the designer can specify the objects belonging to the global scheme and the constraints involving global scheme objects. In order to aid the definition task, the *Global Scheme Interface* shows the characteristics of the sources involved in the data integration system. The functionalities provided by this module are:

- Insertion of a global scheme object. This functionality requires the designer to specify the new object to add to the global scheme. This is then given as input to the *Global Scheme Generator*.

- Modification of a global scheme object. This functionality requires the designer to specify the modifications to be performed on an existing global scheme object. These are then given as input to the *Global Scheme Generator*.

- Deletion of a global scheme object. This functionality requires the designer to specify the global scheme object to delete. This is then given as input to the *Global Scheme Generator*.

- Insertion of a constraint. The designer can specify constraints holding among objects of the global scheme. Also in this case constraints are given as input to the *Global Scheme Generator*.

- Modification of a constraint. By this functionality, the designer can modify previously defined constraints. Modifications provided by the designer are given as input to the *Global Scheme Generator*.

- Deletion of a constraint. This functionality requires the designer to specify the existing constraint to delete. This is then given as input to the *Global Scheme Generator*.

- Visualization of the source schemata. In order to simplify the definition of the global scheme, source schemata participating in the data integration system can be shown to the designer. Information about source schemata is retrieved from the *Metadata Repository*.

- Visualization of the global schema. This functionality can be exploited in order to check the global schema definition status. Information on the global schema is retrieved from the *Metadata Repository*.

Mapping Generation Interface

The *Mapping Generation Interface* provides a graphical interface which allows the designer to specify the mappings between global scheme objects and source scheme objects. In order to simplify the definition task, the *Mapping Generation Interface* can show both the source and the global schemes. The functionalities provided by this module are:

- Insertion of a mapping. This functionality requires the designer to specify the new mapping to add. Necessary information for creating the involved mapping is then sent to the *Mapping Generator* in a suitable format.

- Modification of a mapping. This functionality requires the designer to specify the modifications to be performed on an existing mapping. Necessary information for modifying the involved mapping is then sent to the *Mapping Generator* in a suitable format.

- Deletion of a mapping. This functionality requires the designer to specify the mapping to delete. This is then sent to the *Mapping Generator*.

- Visualization of the source schemes. In order to simplify the definition of the mappings between source scheme and global scheme objects, source schemes participating in the data integration system are shown to the designer. Information about source schemes is retrieved from the *Metadata Repository*.

- Visualization of the global scheme. Analogously to the previous functionality, the global scheme is displayed to the user; necessary information is retrieved from the *Metadata Repository*.
- Visualization of the defined mappings. In order to allow the designer to check the status of the mapping definition phase, the module can show the set of mappings already defined. Information about mappings is retrieved from the *Metadata Repository*.

Consistency Checker

While global scheme, mappings and sources are being specified, it is important to check whether the definitions provided are consistent. When the *Consistency Checker* is activated by the designer, it retrieves from the *Metadata Repository* all the information relative to the source scheme objects, the global scheme objects, the constraints involving global scheme objects, and the mappings between global and local scheme objects. Then, the *Consistency Checker* might either generate from these data a disjunctive logic program which is then executed by a *Disjunctive Datalog Executor* or perform some simpler computations; the choice of which action to perform might be taken on the basis of the task complexity. The *Disjunctive Datalog Executor* is an external module implementing an existing program which is incorporated in the INFOMIX system. The result of the computation determines whether the definitions stored in the *Metadata Repository* are consistent or not.

The result of this check is presented to the user which should modify ill specified definitions.

Global Scheme Generator

The *Global Scheme Generator* receives from the *Global Scheme Interface* messages about global scheme objects and constraints to be added or modified. This module is in charge of the representation of the global scheme objects specified by the user in the *global language* $\mathcal{L}_{\mathcal{G}}$, the representation of the constraints in a suitable representation language and the management of insertions, modifications and deletion of both scheme object and constraint in the *Metadata Repository*.

The functionalities provided by this module are:

- Creation of a global scheme object. This functionality receives the information on the global scheme object to create from the *Global Scheme Interface*; it generates a representation of the object in the *global language* $\mathcal{L}_{\mathcal{G}}$ and stores it in the *Metadata Repository*.
- Modification of a global scheme object. This functionality receives the information relative to the global scheme object to modify and the required modifications from the *Global Scheme Interface*; it retrieves the object from the *Metadata Repository* and performs the required modifications. Finally it stores the modified global scheme object in the *Metadata Repository*.
- Deletion of a global scheme object. This functionality receives the information on the global scheme object to delete from the *Global Scheme Interface*; it removes it from the *Metadata Repository*.
- Creation of a constraint. This functionality receives the information relative to the constraint to create from the *Global Scheme Interface*; it generates a representation of the constraint in the associated language and stores it in the *Metadata Repository*.
- Modification of a constraint. This functionality receives the information relative to the constraint to modify and the required modifications from the *Global Scheme Interface*; it retrieves the constraint from the *Metadata Repository* and performs the required modifications. Finally it stores the modified constraint in the *Metadata Repository*.
- Deletion of a constraint. This functionality receives the information relative to the constraint to delete from the *Global Scheme Interface*; it removes the constraint from the *Metadata Repository*.

The *Mapping Generator* receives from the *Mapping Generation Interface* information on the creation, modification or deletion of mappings between source scheme objects and global scheme objects. This module represents the mappings specified by the user as assertions of the form

$$\begin{array}{c} q_{\mathcal{S}} \sqsubseteq q_{\mathcal{G}} \\ q_{\mathcal{G}} \sqsubseteq q_{\mathcal{S}} \end{array}$$

where $q_{\mathcal{S}}$ and $q_{\mathcal{G}}$ are two queries, respectively over the source schemes, and over the global scheme. Queries $q_{\mathcal{S}}$ are expressed in a query language $\mathcal{L}_{\mathcal{M},\mathcal{S}}$, and queries $q_{\mathcal{G}}$ are expressed in a query language $\mathcal{L}_{\mathcal{M},\mathcal{G}}$. Intuitively, an assertion $q_{\mathcal{S}} \sqsubseteq q_{\mathcal{G}}$ specifies that the concept represented by the query $q_{\mathcal{S}}$ over the sources is put in correspondence with the concept in the global scheme represented by the query $q_{\mathcal{G}}$ (similarly for an assertion of type $q_{\mathcal{G}} \sqsubseteq q_{\mathcal{S}}$). The way these assertions are formulated, indicates the kind of interpretation of the mappings, i.e. *exact*, *sound or complete* (see Section 2.3 for a formal definition of these interpretations).

The *Mapping Generator* manages also storage and update of the mappings in the *Metadata Repository*.

The functionalities provided by this module are:

- Creation of a mapping. This functionality receives the information on the mapping to be created from the *Mapping Generation Interface*; it generates a representation of the mapping as described above and stores it in the *Metadata Repository*.

- Modification of a mapping. This functionality receives the information relative to the mapping to be modified and the required modifications from the *Mapping Generation Interface*; it retrieves the mapping from the *Metadata Repository* and performs the required modifications. Finally, it stores the modified mapping in the *Metadata Repository*.

- Deletion of a mapping. This functionality receives the information on the mapping to be deleted from the *Mapping Generation Interface*; it removes it from the *Metadata Repository*.

Wrapper Generator

The *Wrapper Generator* module is in charge of both the generation of the wrappers for the sources participating in the data integration system and the storage, in the *Metadata Repository*, of source schemes and associations between source relations and wrappers. In particular, the designer specifies, by means of the *Wrapper Generator Interface*, the set of sources and the format of the data stored in the sources; examples of data formats are relational database, XML document, HTML Web page and so on. The designer might possibly specify also some source schemes. Then, the *Wrapper Generator* is activated. For each source, the wrapper generator checks the data format and activates a suitable wrapper generation procedure. This procedure first identifies the relations stored in the source, then it generates a wrapper for each relation. The result of this procedure is, therefore, a set of wrappers associated to the relations exported by the source. Each source is wrapped such as it is seen from the upper abstraction levels as a relational table. Finally, the *Wrapper Generator* stores in the *Metadata Repository*:

- the information on the selected sources, i.e. source location, data format and so on,
- the source schemes, employed in order to describe the source original organization format in relational form.
- the associations between the source relations and the generated wrappers.

2.4.2 Run-Time System Architecture

Once the data integration system is designed, it can be exploited by the users for querying activities. It is important to point out that, in general, the user may be unaware of the sources participating in the system; therefore, the system has to allow to pose queries on the objects of the global scheme.

Querying activities usually involve:

- Global scheme browsing; in this phase the user can identify which information can be extracted from the system.
- Query formulation; in this phase the user is supported by the system in the formulation of the query.
- Query re-formulation; the system rewrites the query posed by the user over the global scheme into a set of queries over the local schemes.
- Wrapper execution, in which data from the sources involved in the query are retrieved.
- Query evaluation; in this phase the system composes the results of single source queries and takes into account possible data inconsistency and incompleteness to obtain the answer to the user query.

The architecture of the INFOMIX system component devoted to querying activities is shown in Figure 2.2. The user can browse the global scheme by means of the *Global Scheme Browser*; then she/he can pose the query through the *Query*

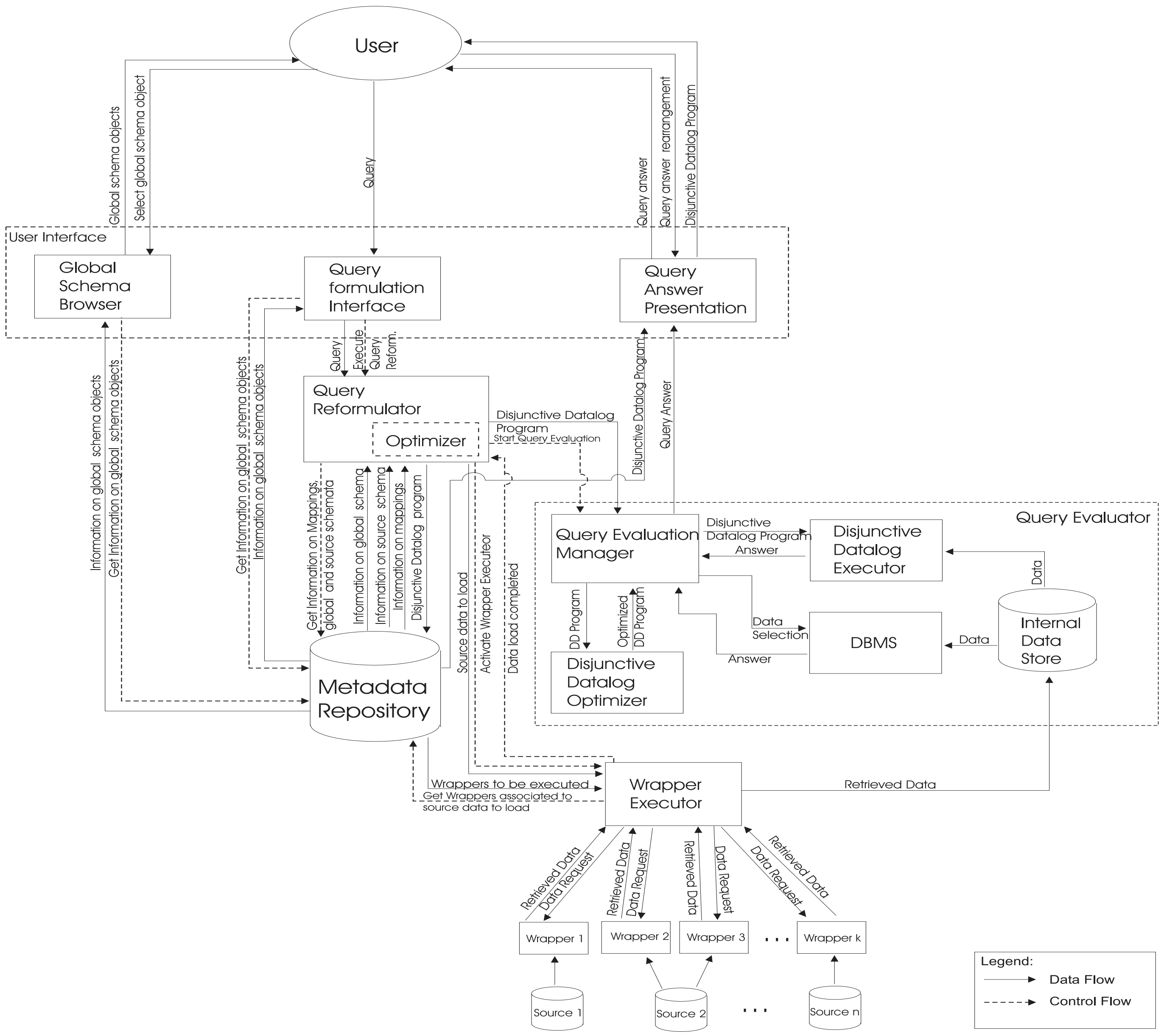

User
Global schema objects
Select global schema object
Query
Query answer
Query answer rearrangement
Disjunctive Datalog Program
User Interface
Global Schema Browser
Query formulation Interface
Query Answer Presentation
Query
Execute Query Reform.
Query Reformulator
Optimizer
Disjunctive Datalog Program
Start Query Evaluation
Disjunctive Datalog Program
Query Answer
Information on global schema objects
Get Information on global schema objects
Get Information on global schema objects
Information on global schema objects
Get Information on Mappings, global and source schemata
Information on global schema
Information on source schema
Information on mappings
Disjunctive Datalog program
Source data to load
Activate Wrapper Executeor
Data load completed
Metadata Repository
Query Evaluator
Query Evaluation Manager
Disjunctive Datalog Program
Answer
Disjunctive Datalog Executor
Data
DD Program
Optimized DD Program
Disjunctive Datalog Optimizer
Data Selection
Answer
DBMS
Data
Internal Data Store
Wrappers to be executed
Get Wrappers associated to source data to load
Wrapper Executor
Retrieved Data
Retrieved Data
Data Request
Wrapper 1
Wrapper 2
Wrapper 3
Wrapper k
Source 1
Source 2
Source n
Legend:
Data Flow
Control Flow

Figure 2.2: Architecture of INFOMIX system - Run-Time part

Formulation Interface. The query is then transferred to the *Query Reformulator* module which is responsible of translating the query posed by the user into a set of queries over the source schemes taking into account the constraints defined on the global scheme objects. This module performs also some query optimization; the result of this phase is a disjunctive datalog program. The *Query Reformulator* activates both the *Wrapper Executor*, in order to retrieve the data from the sources, and the *Query Evaluator*, in order to execute the query. The *Wrapper Executor* stores the retrieved data into an *Internal Data Store*, whereas the *Query Evaluator* first performs some further optimization on the disjunctive datalog program received from the *Query Reformulator*, then it decompose the query evaluation in two parts: the first can be executed directly by a *DBMS* and is relative to the part of the query which does not need complex reasoning; the second is submitted to a *Disjunctive Datalog Executor* and is relative to the part of the query handling constraints, data incompleteness and data inconsistencies. The results of these two parts are then composed and presented to the user by the *Query Answer Presentation* module. Finally, if the query does not return the expected results, the designer might analyze the disjunctive datalog program generated by the query reformulator to see if there are clues to what might be the problem and fix it.

In the following we describe the single modules in detail, one per subsection.

Global Scheme Browser

The *Global Scheme Browser* allows the final user to suitably inspect the global scheme by means of graphical navigation facilities. This allows the user to exactly understand the structure of the global scheme, and analyze the relationships and dependencies holding among the global elements. In other words, the *Global Scheme Browser* makes the user understand which information can be extracted from the system, and supports him in formulating queries.

Query Formulation Interface

The *Query Formulation Interface* allows users to define their queries over the global scheme objects. A graphical interface makes this task easier. The user should be able to specify which interpretation must be exploited for computing the query answer; possible interpretations are *exact*, *sound* and *complete*.

Once the query has been defined, the *Query Formulation Interface* sends it to the *Query Reformulator* which starts the query evaluation task.

Query Answer Presentation

User queries are answered by means of the *Query Answer Presentation* module. This module cares about presenting and organizing queries results, produced by the *Query evaluator*, in a suitable format. The user should be able to rearrange results visualization in order to better carry out analysis. Moreover, the interaction with a designer could be required, whenever a query does not return expected results; in particular, a designer should be made able to analyze the disjunctive datalog programs generated by the query reformulator in order to debug and fix problems.

The functionalities provided by this module are, therefore, the following:

- Query answer presentation. This functionality receives as input the query answer from the *Query Evaluator* and arranges it in a suitable, graphical way.
- Query answer rearrangement. In this case the user specifies how the query answer must be presented and the system rearranges the visualization taking into account user needs.
- Disjunctive Datalog Program Visualization. This functionality lets the designer analyze the disjunctive datalog program generated by the *Query Reformulator* to have an insight in the query evaluation process. The disjunctive datalog program is taken from the *Metadata Repository*.

Query Reformulator

Answers to user queries expressed in terms of the global scheme have to be computed by the system only on the basis of data stored in the sources. For this purpose, the *Query Reformulator* module re-express each user query in terms of a suitable set of queries posed to the sources. In this reformulation process, the query is unfolded and integrity constraints are taken into account in the formulation of the set of queries. Moreover, the kind of interpretation selected by the user, i.e. *certain answers* or *possible answers* is considered to produce the correct rules for combining data from different sources possibly storing incomplete or inconsistent data.

The query reformulator performs some query optimization activities which aim at selecting a set of data as small as possible from the sources. This allows to both minimize source accesses and reduce the amount of data to be analyzed by the *Query Evaluator*.

The reformulated and optimized query is then represented as a disjunctive datalog program which is given as input to the *Query Evaluator*. At the same time, the reformulated query allows to determine which sources are involved in the query and which data must be loaded from them. This information is given as input to the *Wrapper Executor* that performs suitable calls to the corresponding wrappers and stores the data into the *Internal Data Store*.

Finally, the disjunctive datalog program produced in this phase is stored in the *Metadata Repository* for future analysis.

The functionalities provided by this module are the following:

- Query Reformulation. This functionality receives in input the user query and a kind of interpretation. The outputs of this functionality are a query expressed only on the source schemes taking into account the constraints and the kind of interpretation, and the source data to be loaded by the *Wrapper Executor*.

- Query Optimization. This functionality receives in input the query generated by the query reformulation task and produces an optimized query expressed as a disjunctive datalog program. This program is stored in the *Metadata Repository* and is given as input to the *Query Evaluator*.

Wrapper Executor

The *Wrapper Executor* module receives from the *Query Reformulator* an indication about the relations to be retrieved and, whenever possible, a statement expressing which subsets of these relations must be retrieved from the sources. The *Wrapper Executor* identifies the wrappers to activate from the associations between the sources and the wrappers stored in the *Metadata Repository*. Then, it sends suitable data requests to each of these wrappers to retrieve needed data. These data are finally stored in the *Internal Data Store*. The Wrapper Generator should be able to take advantage of caching methods in order to speed up its activities.

Query Evaluator

The *Query Evaluator* module is responsible of the query answer generation. This module receives a disjunctive datalog program from the *Query Reformulator* and is activated when the *Wrapper Executor* has loaded the source data involved in the query into the *Internal Data Store*. The *Query Evaluator* is subdivided in some sub-modules, namely:

- The *Query Evaluation Manager* which coordinates and handles the other sub-modules.
- The *Disjunctive Datalog Optimizer* which performs some optimization tasks.
- The *DBMS* which executes parts of the query directly on the *Internal Data Store*.
- The *Disjunctive Datalog Executor* which executes automatic reasoning techniques for managing incomplete and inconsistent data.

Both the *DBMS* and the *Disjunctive Datalog Executor* are modules implementing existing software which are incorporated in the INFOMIX system.

The query evaluation task is carried out as follows: the *Query Evaluation Manager* receives from the *Query Reformulator* a disjunctive datalog program corresponding to an unfolded user query, enriched with information on the constraints and optimized. First it applies further optimization techniques, based on program rewriting techniques (e.g. “Magic Set” techniques [8, 58, 61]), on the disjunctive datalog program to further improve the efficiency of the evaluation task. Then it singles out two parts of the query: one which does not need automatic reasoning and that can be evaluated directly from a DBMS; the other one, which takes into account problems arising in the management of both incomplete and inconsistent data and constraint satisfaction; the evaluation of this part needs the support of a disjunctive datalog executor. The two parts are then submitted to the *DBMS* and the *Disjunctive Datalog Executor*, respectively, and their results are composed by the *Query Evaluator Manager*. Both the *DBMS* and the *Disjunctive Datalog Executor* retrieve the data to operate upon from the *Internal Data Store*.

The final query results are then forwarded as input to the *Query Answer Presentation* module.

2.4.3 The Metadata Repository

In both the Design-Time and Run-Time INFOMIX architectures the *Metadata Repository* is exploited to store all information necessary for carrying out data integration tasks. In particulare, the *Metadata Repository* have to store:

- information about sources participating in the data integration system, such as source location and data format;
- the relations exported by each source and, therefore, source schemes;
- the associations between exported source relations and the corresponding wrappers;
- information on generated wrappers;
- the global scheme objects;
- the constraints defined on global scheme objects;
- the mappings between global scheme objects and source scheme objects;
- the disjunctive datalog programs generated during the query reformulation tasks;

While this list covers all the information needed by the INFOMIX system devised in this report, it might be not exhaustive; it will be refined as soon as the single modules of the architecture are fully designed and specified.

DBMS

This module can be implemented by any of the available database management systems which support full SQL and ODBC. In prototype implementation, among the available free software products, *POSTGRES* was chosen.

Disjunctive Datalog Executor

This module must be implemented by a system allowing to interpret and execute disjunctive datalog programs. Answer Set Programming Systems (ASP) are better suited than deductive database systems for data integration, most importantly because the latter miss the computational power to solve some hard (co-NP-complete) tasks arising in data-integration. Moreover, among the most widely used ASP systems, we have noticed that DLV [27] seems to be the best-suited computational logic system to be incorporated in a data integration platform, since it is able to solve the computationally hard problems arising in the context of data integration, while showing better performance than the other ASP systems over large input data. Therefore, the INFOMIX system will exploit DLV as Disjunctive Datalog Executor.

2.5 Application and Experiments

In order to show the effectiveness of the INFOMIX prototype system, we set up an application scenario referring to a university information system. Rather than to invent an abstract academic example, we consider data sources which are available at the University of Rome “La Sapienza”, and build an information integration system on top of them. The choice of this application scenario has been driven by the facts that, on the one hand, real data sources from an enterprise environment should be used, including web pages, but that on the other hand, such data is very sensible and companies are often not willing to release their data. Furthermore, encryption and clearing efforts for such data might be high, and for the case that technical issues concerning the data need to be resolved only little (if any) support will be available.

Taking the view of a university as a service-oriented institution whose students are customers, there is quite some resemblance with an enterprise, though. In the modeling of the application scenario, we thus focus on data comprising students, professors, and exams in the different faculties of the university.[1]

Our aim is to collect all the information dispersed either over many data sources within the different secretary offices of the faculties or over the web pages of “La Sapienza”, and to build a data integration system $\mathcal{I}_0 = \langle \mathcal{G}_0, \mathcal{S}_0, \mathcal{M}_0 \rangle$ providing transparent access to this information.

There are three legacy databases in relational format, each of which comprises a number of relations; in total, there are about 25 different such relations, each of which can be viewed as a logical source.

Besides these legacy databases, there are numerous web pages on the web servers of “La Sapienza” which provide a wealth of informations on departments, people, offices etc. These informations are either provided explicitly on web pages itself, or can be obtained through simple query interfaces (returning, e.g. as the one on `http://www.amm.uniroma1.it/elenco/`, the phone number and the department in which a given person works). For this purpose, a number of wrappers have been designed and developed using LiXto tools which extract interesting informations from the web pages and provide them as virtual source data.

In the following three sections, we describe the components of the data integration system $\mathcal{I}_0 = \langle \mathcal{G}_0, \mathcal{S}_0, \mathcal{M}_0 \rangle$.

[1]Even for this application, encryption of sensitive personal data is required by Italian law.

2.5.1 Global Schema

The global schema $\mathcal{G}_0 = \langle \mathcal{H}_0, \mathcal{C}_0 \rangle$ which can be queried by an end user (e.g., administrative staff) comprises the following relations (in $\mathcal{H}_0$):

$$
\begin{array}{l}
\texttt{student}(S\_ID, FirstName, SecondName, CityOfResidence,\\
\qquad Address, Telephone, HighSchoolSpecialization)\\
\texttt{enrollment}(S\_ID, FacultyName, Year)\\
\texttt{course}(C\_Code, Description)\\
\texttt{professor}(ProfFirstName, ProfSecondName)\\
\texttt{university}(U\_Name, City)\\
\texttt{exam\_record}(S\_ID, C\_Code, ProfFirstName, ProfSecondName,\\
\qquad Mark, Date, CourseYear)\\
\texttt{teaching}(C\_Code, ProfFirstName, ProfSecondName,\\
\qquad AcademicYear)\\
\texttt{student\_course\_plan}(SCP\_Code, S\_ID, PlanType, RequestDate,\\
\qquad Status)\\
\texttt{plan\_data}(SCP\_Code, C\_Code, CourseType)\\
\texttt{faculty}(FacultyName, deanFirstName, deanLastName)\\
\texttt{department}(DeptName)\\
\texttt{university\_degree}(Degree, FacultyName)\\
\texttt{employee}(EmpFirstName, EmpSecondName, Structure, Phone)\\
\texttt{secretary\_office}(FacultyName, Place, Phone, Email)
\end{array}
$$

Notice that the above schema models a situation in which there are a number of Faculties each one having its own secretary's office and its own dean. Each faculty comprises a set of university degrees[2]. Finally, students enroll in a faculty in a given year, and then, they may take some examinations.

The schema $\mathcal{G}_0$ is also equipped with the set $\mathcal{C}_0$ of global constraints. $\mathcal{C}_0$ contains

[2]University degree corresponds to the Italian *corso di laurea*. For example, Computer Engineering is a course degree of the faculty of Engineering

- the following key constraints:

$$\begin{array}{l}
key(\texttt{student}) = \{1\} \\
key(\texttt{enrollment}) = \{1\} \\
key(\texttt{course}) = \{1\} \\
key(\texttt{professor}) = \{1,2\} \\
key(\texttt{university}) = \{1\} \\
key(\texttt{exam\_record}) = \{1,2,3,4\} \\
key(\texttt{student\_course\_plan}) = \{1\} \\
key(\texttt{plan\_data}) = \{1,2\} \\
key(\texttt{faculty}) = \{1\} \\
key(\texttt{department}) = \{1\} \\
key(\texttt{university\_degree}) = \{1\} \\
key(\texttt{employee}) = \{1,2\} \\
key(\texttt{secretary\_office}) = \{1\}.
\end{array}$$

- the following inclusion dependencies:

$$\begin{array}{rcl}
\texttt{enrollment}[1] & \subseteq & \texttt{student}[1] \\
\texttt{enrollment}[2] & \subseteq & \texttt{faculty}[1] \\
\texttt{exam\_record}[1] & \subseteq & \texttt{student}[1] \\
\texttt{exam\_record}[2] & \subseteq & \texttt{course}[1] \\
\texttt{exam\_record}[3,4] & \subseteq & \texttt{professor}[1,2] \\
\texttt{teaching}[2,3] & \subseteq & \texttt{professor}[1,2] \\
\texttt{teaching}[1] & \subseteq & \texttt{course}[1] \\
\texttt{student\_course\_plan}[2] & \subseteq & \texttt{student}[1] \\
\texttt{plan\_data}[1] & \subseteq & \texttt{student\_course\_plan}[1] \\
\texttt{plan\_data}[2] & \subseteq & \texttt{course}[1] \\
\texttt{university\_degree}[2] & \subseteq & \texttt{faculty}[1] \\
\texttt{secretary\_office}[1] & \subseteq & \texttt{faculty}[1] \\
\texttt{professor}[1,2] & \subseteq & \texttt{teaching}[2,3] \\
\texttt{teaching}[1] & \subseteq & \texttt{exam\_record}[2].
\end{array}$$

The last two dependencies impose that a professor has to teach at least one course, and that for a course taught there must exist at least one registered exam, respectively. Other dependencies are classical foreign keys, as they are produced by a standard design process for relational databases. We point out that the inclusion dependencies specified on the schema forms cycles (see for example the cycle involving relation `teaching`, `professor`, and `exam_record`). Nonetheless, the schema is non-key-conflicting [49].

- the following exclusion dependencies

$$\begin{array}{rcl}\texttt{student}[2,3] & \cap & \texttt{professor}[1,2] \neq \emptyset, \\ \texttt{employee}[4] & \cap & \texttt{secretary\_office}[3] \neq \emptyset,\end{array}$$

  which state that a student cannot be also a professor, and that phone numbers of employees and secretary offices must be different.

2.5.2 Data Sources

The system integrates data coming from three legacy databases containing information about students, professors and exams at the University "La Sapienza" of Rome, and data retrieved from several web sites of the same university. In the following, we separately describe these two main kinds of data sources.

Legacy Databases

We will denote in the following the three legacy databases with DB_1, DB_2 and DB_3. The specification of the source schema is reported below; the table names and attributes are in Italian, as in the original sources.[3]

```
#source schema DB1

studente(MATRICOLA,COGNOME,NOME,DATA_NASCITA,LUOGO_NASCITA,PROVINCIA_NASCITA,
  INDIRIZZO_RECAPITO,NUMERO_CIVICO_RECAPITO,CAP_RECAPITO,CITTA_RECAPITO,
  PROVINCIA_RECAPITO,PREFISSO_RECAPITO,TELEFONO_RECAPITO,INDIRIZZO_RESIDENZA,
  NUMERO_CIVICO_RESIDENZA,CAP_RESIDENZA,CITTA_RESIDENZA,PROVINCIA_RESIDENZA,
  PREFISSO_RESIDENZA,TELEFONO_RESIDENZA,CODICE_FISCALE,TIPO_DIPLOMA,
  VOTO_DIPLOMA)

diploma_maturita(CODICE,DESCRIZIONE)

carriera(MATRICOLA,ANNO_ACCADEMICO,ANNO_DI_CORSO,TIPO_ISCRIZIONE,FACOLTA,
  CORSO_DI_LAUREA,UNIVERSITA,STATO_DI_CARRIERA,VALIDITA_ANNO_CARRIERA,
  FASCIA_CONTRIBUTIVA,COMPONENTI_NUCLEO_FAMILIARE)

facolta(CODICE_UNIVERSITA,CODICE,DESCRIZIONE)

corso_laurea(SEDE_UNIVERSITA,FACOLTA,CODICE,DESCRIZIONE)

universita(CODICE,SEDE,DESCRIZIONE)

stato_carriera(CODICE,DESCRIZIONE)

iscrizione(CODICE,DESCRIZIONE)
```

[3]Notice that the three databases (for the sake of easiness implemented as a single relational database) have been constructed from a set of original text files. Data cleaning has been manually performed on the original files.

```
regolarita_esame(CODICE,DESCRIZIONE)

esame(CODICE_FACOLTA,CODICE,DESCRIZIONE,ATTIVAZIONE)

insegnamento(CODICE_FACOLTA,CODICE,DESCRIZIONE)

insegnamento_esame(FACOLTA_INSEGNAMENTO,CODICE_INSEGNAMENTO,FACOLTA_ESAME,
  CODICE_ESAME)

dati_esami(MATRICOLA,CODICE_INSEGNAMENTO,CODICE_ESAME,DATA,VOTO,REGOLARITA,
  ANNO_ACCADEMICO)

laurea(MATRICOLA,TITOLO_TESI,DATA,VOTO,RELATORE)


#source schema DB2

esame_ingegneria(CODICE,DESCRIZIONE,TIPO,ANNO_ESAME)

tipo_esame(CODICE,DESCRIZIONE)

piano_studi(CODICE,MATRICOLA,ORIENTAMENTO,DATA_PRESENTAZIONE,STATO,NOTE,
  PROPRESP,BASE,INDIRIZZO_A,INDIRIZZO_B)

stato(CODICE,DESCRIZIONE)

orientamento(CODICE,DESCRIZIONE)

dati_piano_studi(CODICE,CODICE_ESAME,NOME)

affidamenti_ing_informatica(CODICE_ESAME,CODICE_PROFESSORE,ANNO_ACCADEMICO)

dati_professori(CODICE,COGNOME,NOME)


#source schema DB3

verbali_esami_diploma(MATRICOLA,COGNOME,NOME,ESAME,DOCENTE,SESSIONE,APPELLO,
  ANNO,MODALITA,VOTO,LODE,ANNO_ACCADEMICO)

modalita(CODICE,DESCRIZIONE)

sessione(CODICE,DESCRIZIONE)

professore(CODICE,NOME,COGNOME,MATERIA)

esame_diploma(CODICE,DESCRIZIONE)
```

Here we report the number of tuples stored in each source relation, specifying to which database the source belongs.

carriera	50,633	DB_1
corso_laurea	1,716	DB_1
dati_esami	19,827	DB_1
diploma_maturita	69	DB_1
esame	17,144	DB_1
facolta	511	DB_1
insegnamento	4,722	DB_1
insegnamento_esame	7,204	DB_1
iscrizione	5	DB_1
laurea	397	DB_1
regolarita_esame	4	DB_1
stato_carriera	15	DB_1
studente	16,082	DB_1
universita	163	DB_1
affidamenti_ing_informatica	402	DB_2
dati_piano_studi	27,130	DB_2
dati_professori	67	DB_2
esame_ingegneria	67	DB_2
orientamento	29	DB_2
piano_studi	1,089	DB_2
stato	3	DB_2
tipo_esame	3	DB_2
esame_diploma	28	DB_3
modalita	2	DB_3
professore	146	DB_3
sessione	4	DB_3
verbali_esami_diploma	17,001	DB_3

Web Sources

In addition, we have identified several relevant web pages provided by the university and its faculties and departments. The main page of the university is `http://www.uniroma1.it/`, from which all of these pages can be reached. We will next describe the pages and data available from them in a concise way.

In particular, we will not report the INFOMIX Source Data Format (ISDF) schema of these web sources in detail and refer to the appendix. Instead we report the more concise Internal Integration Data Format (IIDF) schema of these sources. Note that these web wrappers for technical reasons possess a non-flat ISDF schema (cf. Appendix). When applying the standard conversion from ISDF to IIDF, as defined in [26], two relations are generated: One holds only newly created IDs, which are then used as foreign keys in a second relation, which holds the real data. The first relation is clearly redundant and a technical artifact, and hence we do not report it; only the second relation containing the mentioned foreign key attribute is given below.

- `http://www.uniroma1.it/facolta/default.htm`
  This web page contains a collection of some of the faculties of the university. For each faculty, it contains its name, the URL of its web page, and information on its dean (comprised of title, first name and surname). The source schema in relational format is

  ```
  facultyWeb__faculty(id_fk, facultyName, facultyURL,
                       deanTitle, deanFirstName, deanLastName)
  ```

- `http://w3.ing.uniroma1.it/dip/elenco.htm`
  `http://www.arc1.uniroma1.it/organizzazione/dipartimenti.htm`
  `http://www.eco.uniroma1.it/dipartimenti.htm`
  `http://www.filosofia.uniroma1.it/dipartimenti/index.asp`
  `http://www.comunicazione.uniroma1.it/dipartimenti.asp`
  `http://www.scienzemfn.uniroma1.it/cdipa.htm`
  `http://www.sta.uniroma1.it/strutture/dipartimenti.jsp`
  These are web pages of some faculties, each of them lists departments of the respective faculty. A lot of information about departments can be wrapped, such as the name of the department, the URL of its home page, its director, its address, the faculty it belongs to, and contact information such as fax and phone numbers and an email address. The source schema in relational format is

  ```
  departmentWeb__department(id_fk, deptName, faculty, deptURL,
              deptFax, address, eMail, deptPhone, deptabbrname)
  ```

- `http://www.uniroma1.it/dip\_ist/default.htm`
  This web page contains a university-wide list of departments. Here, only the name of the department and the URL of its homepage are available, the source schema in relational format is therefore much more restricted:

  ```
  departmentWeb2__department_w00(id_fk, deptURL, deptName)
  ```

- `http://www.uniroma1.it/studenti/corsi/default.htm`
  This web page contains a list of links to web pages, on which all the university degrees of one faculty are listed. Note that in our terminology, “school”

is equivalent to the term "university degree" of the global schema. One can therefore identify the name of the school, the name of the associated faculty, and the URL of the web page of the school. The source schema in relational format is

```
schoolWeb__school(id_fk, schoolName, facultyName, schoolURL)
```

- `http://w3.ing.uniroma1.it/ccl/elenco.htm`
  This web page contains a list of schools offered by the Facoltà di Ingegneria. The source schema does not explicitly contain the faculty name.

  ```
  schoolWeb2__school_w01(id_fk, schoolName, schoolURL)
  ```

- `http://www.uniroma1.it/studenti/corsi/docenti.asp`
  This web page contains a listing of the professors of the university (first name, surname, and a title). For each professor, there is also a link to its personal home page, from which one can usually obtain further data, such as an email address, phone and fax numbers. While the personal web pages usually also include an address, it is much harder to identify, since the structure of the web page is unknown. The address is therefore not wrapped. The source schema in relational format is

  ```
  professorWeb__professor(id_fk, surName, firstName, title,
                      homePage, phone, fax, eMail, address)
  ```

- `http://www.sociologia.uniroma1.it/offerta/ElencoDocenti.`
  `asp?IdQualifica=3`
  `http://w3.uniroma1.it/drsfp/docenti/index.shtml`
  `http://w3.ing.uniroma1.it/\%7Espacedpt/docenti.html`
  `http://www.dau.uniroma1.it/organizzazione/organizzazione.`
  `htm`
  These web pages list professors of four faculties, obtaining similar information as from the university-wide listing. An advantage here is that contact information need not be wrapped from personal web pages. Hence also the address can be provided in the schema:

  ```
  professorWeb2_professor_w00(id_fk, surName, firstName, title,
                          homePage, phone, fax, eMail, address)
  ```

- `http://www.amm.uniroma1.it/elenco/`
  This page is actually a form, and the wrapper proceeds by submitting values for departments (stored in a resource local to the wrapper) and analyzing the resulting web pages. These contain an entry for each employee of the department, which is made up of the first and surname and an external and internal phone number. The source schema in relational format is

  ```
  employeeWeb__employee(id_fk, surName, firstName, department,
                        extPhone, intPhone)
  ```

- `https://www.infostud.uniroma1.it/segreterie.asp`
  Also this page is a form. The wrapper submits values for faculty and analyzes the resulting data, which contains the location, phone, email address and office hours of the respective secretary's offices. The source schema in relational format is

  ```
  secretaryWeb__secretary(id_fk, faculty, place, phone, eMail,
  ```

- `http://www2.unibo.it/infostud/fare/eurouni/itauni/italia.htm`
  This web page is the only one which does not reside on a server of the University "La Sapienza" of Rome. It has a listing of all the universities of Italy, containing its name, the city it is located, and the URL of its web page. The source schema in relational format is

  ```
  universityWeb__university(id_fk, name, city, homePage)
  ```

- `http://www.dis.uniroma1.it/alphaindex.php?file=phdstud`
  A page with the PhD students of the DIS department. Apart from the name, one can obtain further information, such as phone, email, homepage and at which site the PhD student is at. The source schema in relational format is

  ```
  studentWeb__student(id_fk, homePage, site, phone,
                      lastName, firstName, eMail)
  ```

- `http://www.mat.uniroma1.it/persone/INDEX`
  A page with the PhD students of the mathematics department. Here, no phone numbers and site information can be obtained, otherwise it is similar to the previous wrapper. The source schema in relational format is

```
doctorantWeb__doctorant(id_fk, eMail, firstName, lastName,
                                                homePage)
```

Note that we do not use the last two sources at the moment, as no concept of PhD student currently exists in the global schema. However, we still report them here, as they have been created already and could be used in a future version of the scenario.

2.5.3 Mapping

In this section, we describe the mapping $\mathcal{M}_0$ of the integration system $\mathcal{I}_0$.

We recall that in the general formal framework for information integration underlying INFOMIX, the mapping comprises expressions of the form (1) $q_S \sqsubseteq q_G$ and (2) $q_G \sqsubseteq q_S$, respectively, where q_G and q_S are queries over the global schema and the source schema, respectively.

As for the first INFOMIX prototype, the mapping language consists of GAV mappings, that is, expressions of the form (1) where in essence, q_G is a single global relation, r_G, and such that q_S amounts to a union of conjunctive queries (UOCs). In principle, for GAV integration no special constraints would need to be imposed on q_S to maintain the algorithmic feasibility of the approach apart from decidability of the query language over the sources. However, on the one hand, union of conjunctive queries constitute already an expressive fragment with respect to practical applications and, on the other hand, seems more amenable to effective optimization techniques than richer languages. Furthermore, this assumption is no real limitation, since it is easily possible to transform any data integration system using general GAV mappings to one in which only GAV mappings occur that use UOCs, by composing wrappers. Note that such a transformation would not be feasible in a LAV setting.

We note that formally, in the GAV mapping $q_S \sqsubseteq r_G$ the query q_S maps data over the source schemata $\mathcal{S}_0$, which are formulated in the INFOMIX Source Data Format (ISDF), to data in the INFOMIX Global Data Format (IGDF), which is a relational format. Since ISDF is a fragment of XML-Schema, this means that for specifying queries q_S, a special-purpose XML query language might be defined, or alternatively a suitable fragment of an arbitrary existing XML query language might be used to map the XML data from the sources to the XML rendering of a relation.

We refrain here from defining such a language or fragment of an existing XML query language. Instead, we report the mappings for the internal representation of the sources (IIDF) which is in relational format, in Datalog like syntax. We recall that the internal representation is obtained by a simple mapping from ISDF to IIDF, which in particular for relational raw data behind a virtual ISDF source just recreates the respective relational schema (modulo type conversions). The design of a data integration system is in the hands of an expert administrator, who might

For the specification of the mapping $\mathcal{M}_0$ in the “La Sapienza” scenario, we use here the following notation. Each assertion

$$q_S \sqsubseteq r_G,$$

where q_S is of the form

$$q(\vec{x}) \leftarrow q_1(\vec{x}, \vec{y}_1) \vee \cdots \vee q_k(\vec{x}, \vec{y}_k),$$

and each $q_i(\vec{x}, \vec{y}_i)$ is a conjunctive query, is represented by rules

```
r_G(X1,X2,...,Xm) :- q_1(X1,X_2,...,Xm,Y11,...,Y1m_1)
                        ...
r_G(X1,X2,...,Xm) :- q_k(X1,X_2,...,Xm,Yk1,...,Ykm_k)
```

where $\vec{x}$ = `X1,...,Xm` and $\vec{y}_i$ = `Yi1,...,Yim_i`, for all $i = 1, \ldots, k$.

In the bodies of rules, we use “_” for anonymous variables, i.e., for variables which occur only once in that rule.

The mappings are now defined as follows.

```
student(X1,X2,X3,X4,X5,X6,X7) :-
  studente(X1,X3,X2,_,_,_,_,_,_,_,_,_,X6,X5,_,_,X4,_,_,_,_,Y,_),
  diploma_maturita(Y,X7).

enrollment(X1,X2,X3) :- carriera(X1,X3,_,_,Y,_,_,_,_,_,_), facolta(_,Y,X2).

course(X1,X2) :- esame(_,X1,X2,_).
course(X1,X2) :- esame_diploma(X1,X2).

professor(X1,X2) :- professore(_,X1,X2,_).
professor(X1,X2) :- dati_professori(_,X1,X2).
professor(X1,X2) :- professorWeb__professor(_,X1,X2,_,_,_,_,_,_).
professor(X1,X2) :- professorWeb2__professor_w00(_,X1,X2,_,_,_,_,_,_).

university(X1,X2) :- universita(_,X2,X1).
university(X1,X2) :- universityWeb__university(_,X1,X2,_).

exam_record(X1,X2,Z,W,X4,X5,Y) :- dati_esami(X1,_,X2,X5,X4,_,Y),
                                  affidamenti_ing_informatica(X2,X3,Y),
                                  dati_professore(X3,Z,W).

teaching(X1,Z,W,X3) :- affidamenti_ing_informatica(X1,X2,X3),
                        dati_professore(X2,Z,W).

student_course_plan(X1,X2,X3,X4,X5) :- piano_studi(X1,X2,Y1,X4,Y2,_,_,_,_,_),
                              orientamento(Y1,X3), stato(Y2,X5).

plan_data(X1,X2,X3) :- dati_piano_studi(X1,X2,_),
                      esame_ingegneria(X2,_,Y2,_), tipo_esame(Y2,X3).

faculty(X1,X2,X3) :- facultyWeb__faculty(_,X1,_,_,X2,X3).

department(X1) :- departmentWeb__department(_,X1,_,_,_,_,_,_,_).
department(X1) :- departmentWeb2__department_w00(_,_,X1).

university_degree(X1,X2) :- schoolWeb__school(_,X1,X2,_).
university_degree(X1,'Facolta di Ingeneria') :-
                schoolWeb2__school_w01(_,X1,_).

employee(X1,X2,X3,X4) :- employeeWeb_employee(_,X2,X1,X3,_,X4).

secretary_office(X1,X2,X3,X4) :- secretaryWeb__secretary(_,X1,X2,X3,X4,_).
```

2.5.4 Queries

We have tested the system with 10 queries, which we describe below. Their notation is in the INFOMIX Query Language (IQL), which is a subset of Datalog.

(Q_1) This query asks for the exams in the exam plan of the student with ID 09089903.

```
q1(D) :- student_course_plan(C,'09089903',_,_,_),
            plan_data(C,E,ET), course(E,D).
```

(Q_2) This query asks for the exams done by the student with ID 09089903.

```
q2(CD,D) :- exam_record('09089903',C,_,_,_,D,_),
            course(C,CD).
```

(Q_3) This query asks for the names of professors that teach a course.

```
q3(Pfn,Pln) :- teaching(_,Pfn,Pln,_).
```

(Q_4) This query asks for the personal data of the student with ID 09089903. Such queries are frequently posed by university authorities who have to contact a student.

```
q4(Sfn,Sln,Cor,Add,Tel,Hss) :-
     student('09089903',Sfn,Sln,Cor,Add,Tel,Hss).
```

(Q_5) This query returns the information about students and their exam plans with status ”APPROVATO SENZA MODIFICHE”, for students who have as first name ZNEPB[4].

```
q5(SID,Sln,R) :-
     student(SID,’ZNEPB’,Sln,Cor,Add,Tel,Hss),
     student_course_plan(SCP,SID,T,R,
                  ’APPROVATO SENZA MODIFICHE’).
```

(Q_6) This query asks for Universities in ROMA.

```
q6(U) :- university(U,’ROMA’).
```

(Q_7) This query retrieves the information about students living in ROMA having RETI LOGICHE in their their exam plans.

```
q7(F,S) :- student(SID,F,S,’ROMA’,A,T,H),
           student_course_plan(SCID,SID,PT,R,ST),
           plan_data(SCID,CID,CT),
           course(CID,’RETI LOGICHE’).
```

(Q_8) This query asks for professors who teach or have taught courses which were not taught in 1990.

```
q8(Pfn,Pln) :- teaching(C,Pfn,Pln,_), not t90(C).
t90(C) :- teaching(C,Pfn,Pln,1990).
```

(Q_9) This query asks for cities in which at least two universities are located.

```
q9(C) :- university(N,C),
         #count{N1:university(N1,C)} >= 2.
```

[4]Student names have been encrypted for privacy reasons.

Part II

Advanced Techniques and Systems for Process Management

Chapter 3

Process Mining

3.1 Workflow Mining

During the last decade workflow management technology [74, 71, 29, 44, 47] has become readily available. Workflow management systems such as Staffware, IBM MQSeries, COSA, etc. offer generic modeling and enactment capabilities for structured business processes. By making process definitions, i.e., models describing the life-cycle of a typical case (workflow instance) in isolation, one can configure these systems to support business processes. These process definitions need to be executable and are typically graphical. Besides pure workflow management systems many other software systems have adopted workflow technology. Consider for example ERP (Enterprise Resource Planning) systems such as SAP, PeopleSoft, Baan and Oracle, CRM (Customer Relationship Management) software, SCM (Supply Chain Management) systems, B2B (Business to Business) applications, etc. which embed workflow technology. Despite its promise, many problems are encountered when applying workflow technology. One of the problems is that these systems require a workflow design, i.e., a designer has to construct a detailed model accurately describing the routing of work. Modeling a workflow is far from trivial: It requires deep knowledge of the business process at hand (i.e., lengthy discussions with the workers and management are needed) and the workflow language being used.

To compare workflow mining with the traditional approach towards workflow design and enactment, consider the four phases composing the traditional *workflow life cycle*:

(A) **workflow design**

(B) **workflow configuration**

(C) **workflow enactment**

(D) **workflow diagnosis**

In the traditional approach the design phase is used for constructing a workflow model. This is typically done by a business consultant and is driven by ideas of management on improving the business processes at hand. If the design is finished, the workflow system (or any other system that is "process aware") is configured as specified in the design phase. In the configuration phases one has to deal with limitation and particularities of the workflow management system being used. In the enactment phase, cases (i.e., workflow instances) are handled by the workflow system as specified in the design phase and realized in the configuration phase. Based on a running workflow, it is possible to collect diagnostic information which is analyzed in the diagnosis phase. The diagnosis phase can again provide input for the design phase thus completing the workflow life cycle. In the traditional approach the focus is on the design and configuration phases. Less attention is paid to the enactment phase and few organizations systematically collect runtime data which is analyzed as input for redesign (i.e., the diagnosis phase is typically missing).

The goal of workflow mining is to reverse the process and collect data at runtime to support workflow design and analysis. Note that in most cases, prior to the deployment of a workflow system, the workflow was already there. Also note that in most information systems transactional data is registered (consider for example the transaction logs of ERP systems like SAP). The information collected at run-time can be used to derive a model explaining the events recorded. Such a model can be used in both the diagnosis phase and the (re)design phase.

Modeling an existing process is influenced by perceptions, e.g., models are often normative in the sense that they state what "should" be done rather than describing the actual process. As a result models tend to be rather subjective. A more objective way of modeling is to use data related to the actual events that took place. Note that workflow mining is not biased by perceptions or normative behavior. However, if people bypass the system doing things differently, the log can still deviate from the actual work being done. Nevertheless, it is useful to confront man-made models with models discovered through workflow mining.

Closely monitoring the events taking place at runtime also enables *Delta analysis*, i.e., detecting discrepancies between the design constructed in the design phase and the actual execution registered in the enactment phase.Workflow mining results in an “a posteriori” process model which can be compared with the “a priori” model. Workflow technology is moving into the direction of more operational flexibility to deal with workflow evolution and workflow exception handling [74, 76, 3, 19, 28]. As a result workers can deviate from the prespecified workflow design. Clearly one wants to monitor these deviations. For example, a deviation may become common practice rather than being a rare exception. In such a case, the added value of a workflow system becomes questionable and an adaptation is required. Clearly, workflow mining techniques can be used to create a feedback loop to adapt the workflow model to changing circumstances and detect imperfections of the design.

3.2 Process Mining

The goal of workflow mining is to extract information about processes from transaction logs. Instead of starting with a workflow design, we start by gathering information about the workflow processes as they take place. We assume that it is possible to record events such that (i) each event refers to a task (i.e., a well-defined step in the workflow), (ii) each event refers to a case (i.e., a workflow instance), and (iii) events are totally ordered. Any information system using transactional systems such as ERP, CRM, or workflow management systems will offer this information in some form. Note that we do not assume the presence of a workflow management system. The only assumption we make, is that it is possible to collect workflow logs with event data. These workflow logs are used to construct a process specification which adequately models the behavior registered.

The term *process mining* refers to methods for distilling a structured process description from a set of real executions. Because these methods focus on so-called case-driven process that are supported by contemporary workflow management systems, we also use the term workflow mining.

To illustrate the principle of process mining in more detail, we consider the workflow log and a corresponding process model shown in Figure 3.1. This log abstracts from the time, date, and event type, and limits the information to the

s_1 : abdelmo
s_2 : achino
s_3 : acihno
s_4 : abdfmo
s_5 : abdgmo

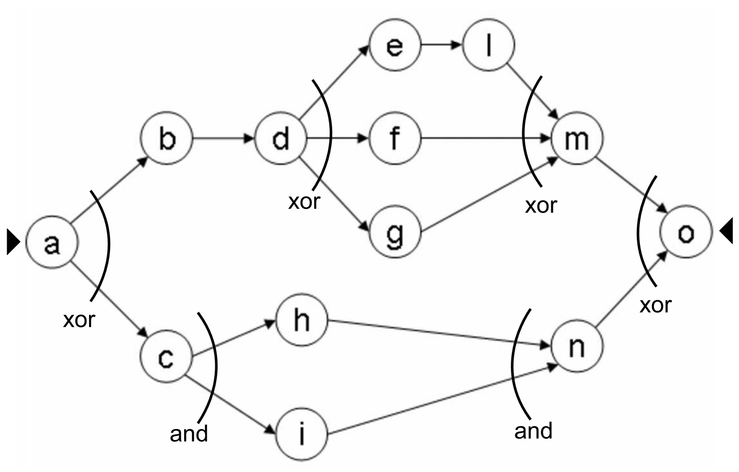


Figure 3.1: A workflow log and a corresponding model.

order in which tasks are being executed. The log contains information about 5 cases (i.e., workflow instances). Each case starts with the execution of *a* and ends with the execution of *o*. It is possible to notice that there are some activities whose execution prevents the execution of other ones in the same process instance (e.g. *b* and *c* or *e*,*f* and *g*). Furthermore, if *b* is executed, also *h* and *i* are executed in any order, this let us hypotize that *h* and *i* can be considered as parallel activities. Based on the information shown in the table and by making some assumptions about the completeness of the log (i.e., assuming that the cases are representative and a sufficient large subset of possible behaviors is observed), we can deduce, for example, the shown process model. It is represented by a *Model Graph* [1] consisting in an oriented graph where nodes represent activities and contain information about the flow management; edges connecting two activities show which activities can be executed after the execution of their source activity. In order to mark starting and ending activities, a triangle pointing right (resp. left) can be used. A boolean function, and eventually a numeric constraint, can be associated to edges outgoing (*split*), or incoming (*join*), from the same node. This filters the flow stating which activities can be executed or waited (in case of incoming edges). For example, edges outgoing from *a* have a *xor* split condition stating that only one activity among *b* and *c* can be executed after *a*. Otherwise, edges incoming in *n* have an *and* join condition stating that both *h* and *i* have to complete their

[1]Beside with Model Graph, other graphical representation formalism for workflow exist as *Petri nets* [55, 75] and *Event-driven Process Chain*(*EPC*) [73, 64].

execution before starting *n*.

The table shown in Figure 3.1 contains the minimal information we assume to be present. In many applications, the workflow log contains a time stamp for each event and this information can be used to extract additional causality information. In addition, a typical log also contains information about the type of event, e.g., a start event (a person selecting an task from a worklist), a complete event (the completion of a task), a withdraw event (a scheduled task is removed), etc. Moreover, we are also interested in the relation between attributes of the case and the actual route taken by a particular case. For example, when handling traffic violations: Is the make of a car relevant for the routing of the corresponding traffic violation? (E.g., People driving a Ferrari always pay their fines in time.) For this simple example, it is quite easy to construct a process model that is able to regenerate the workflow log (e.g., Figure 3.1). For more realistic situations there are however a number of complicating factors:

- For larger workflow models mining is much more difficult. For example, if the model exhibits alternative and parallel routing, then the workflow log will typically not contain all possible combinations. Consider 10 tasks which can be executed in parallel. The total number of interleavings is 10! = 3628800. It is not realistic that each interleaving is present in the log. Moreover, certain paths through the process model may have a low probability and therefore remain undetected.

- Workflow logs will typically contain noise, i.e., parts of the log can be incorrect, incomplete, or refer to exceptions. Events can be logged incorrectly because of human or technical errors. Events can be missing in the log if some of the tasks are manual or handled by another system/organizational unit. Events can also refer to rare or undesired events. Consider for example the workflow in a hospital. If due to time pressure the order of two events (e.g., make X-ray and remove drain) is reversed, this does not imply that this would be part of the regular medical protocol and should be supported by the hospital's workflow system. Also two causally unrelated events (e.g., take blood sample and death of patient) may happen next to each other without implying a causal relation (i.e., taking a sample did not result in the death of the patient; it was sheer coincidence). Clearly, exceptions which are recorded only once should not automatically become part of the regular workflow.

- The table in Figure 3.1 only shows the order of events without giving information about the type of event, the time of the event, and attributes of the event (i.e., data about the case and/or task). Clearly, it is a challenge to exploit such additional information.

3.3 Existing Tools

In the last few years, several efforts have been spent by the research community as well as by the industry to implement robust and scalable platforms for process mining applications, where various mining techniques are integrated and made available to the users. A breakthrough in the design of such kinds of systems was represented by the *ProM* framework [80]. Indeed, differently from earlier approaches (see, e.g., [77, 83] and the references therein), *ProM* comes as an open and extensible architecture, which enables users to write and import their own mining algorithms into the framework as plug-ins. This capability tremendously enriches the power of the entire system, which exports a virtually unbound set of resources to complain a wide range of process mining applications (e.g., control-flow mining, decision tree induction, or clustering, to cite a few) and analysis tasks (e.g., verification of process models, performance analysis, or statistical evaluations). Thanks to this valuable packaging, *ProM* is the most advanced tool for process mining applications, and some successful industrial experiences exploiting its mining capabilities have already been discussed in the literature (see, e.g.,[78, 69]).

3.3.1 The ProM Framework

In this section, we provide a minimal overview of the architecture of ProM Framework. The basic module of the architecture is the Log filter component, which is able to read process log encoded in XML format. This component exports a wide range of capabilities to inspect a log (e.g., show statistical information originators, activities, data appearing) and perform cleaning tasks (e.g., remove all events or traces with a specific event type) before the actual mining starts. Additionally, ProM implements different plug-ins aimed to read and load log and model coming from different transactional systems, which usually use different representation formats. ProM exports facilities for importing data from management systems such as Staffware, Oracle BPEL, Eastman Workflow, WebSphere, InConcert, FLOWer, Caramba, and YAWL, simulation tools such as ARIS, EPC Tools, Yasper, and CPN Tools, ERP systems like PeopleSoft and SAP, analysis tools such as AGNA, NetMiner, Viscovery, AlphaMiner, and ARIS PPM. On the top of the above modules, ProM define a bunch of transactional plug-in covering different tasks. Four different transactional plug-ins are available:

- **Mining plug-ins.** ProM implements a rich bunch of mining plug-ins to answer common tasks in process mining analysis. Notably, ProM provides plug-ins for each of the three process mining perspectives (i.e. process perspective, organization perspective, and case perspective). Additionally, support to data decision mining has been recently added. For the process perspective, various plug-ins are available. The following represent the most relevant:

  - **α-algorithm.** It implements the basic α-algorithm and all its extensions as developed by the authors .
  - **Tshinghua-α-algorithm.** This plug-in uses timestamps in the log files to construct a Petri net.
  - **Genetic algorithm.** This plug-in is based on a recent research guideline using genetic algorithms to tackle possible noise in the log file. Its output format is a heuristics net (which can be converted into an EPC or a Petri net).
  - **Heuristics Miner.** This plug-in encodes strategies to deal with noise and incompleteness in the log.
  - **Multi-phase mining.** It implements a series of process mining algorithms that use instance graphs (comparable to runs) as an intermediate format. The two-phase approach resembles the aggregation process in Aris PPM.

  For the organizational perspective, some plug-ins are available. We focus here on the following:

  - **Social network miner.** which uses the log file to determine a social network of people [72]. It requires the log file to contain the specification about the originators of activities.
  - **Semantic Organizational Miner.** It uses the semantic information in the log to mine groups of users based on task similarity.
  - **Role Hierarchy Miner.** This plug-ins uses the information about which originators have executed which tasks to identify who are the

specialist/generalist for a given process. The mined taxonomy can be exported as a WSML[13] ontology with concepts, is-relationships and instances.

Finally, for the decision mining perspective, also one plug-in is available:

  - **Decision Miner.** plug-in determines the decision points contained in a Petri net model (e.g., a task is performed only if a condition holds), and specifies the possible decisions with respect to the log while being able to deal with invisible and duplicate activities in the way described in [60]. While the Decision Miner formulates the learning problem, the actual analysis is carried out with the help of the J48 decision tree classifier, which is the implementation of the C4.5 algorithm [62] provided by the Weka software library [85].

- Analysis plug-ins. These plug-ins implement further mining analysis on the result produced by a mining plug-in. We cite here:
  - **LTL Checker.** This plug-in checks a Linear Temporal Logic (LTL) formula on a log (e.g., test if a given originator executes a specific task of the process).
  - **Conformance Checker.** It evaluates the conformance between a given process model (encoded in terms of a Petri net) and a log.
  - **Performance Analysis with Petri net.**This plug-in run statistical tests on a Petri net model which exploit time-related aspects of the process instances. As an example, this plug-in can evaluate the routing probabilities for each split/join task, or the average/minimum/maximum throughput time of cases.

- **Export plug-ins.** These plug-ins implement some "save as" functionality for data objects (such as graphs). For example, facilities to export models in EPCs, Petri nets (e.g., in PNML format), spreadsheets are provided.

- **Conversion plug-ins.** These plug-ins implement basic conversions between different data formats, e.g., from EPCs to Petri nets.

The results coming from applying transactional plug-ins are stored in *Result Frame* objects, which can be used for visualization or conversion.

Chapter 4

PROMETHEUS

4.1 Introduction

In the context of enterprise automation, *process mining* has recently emerged as a powerful approach to support the analysis and the design of complex business processes [77]. In a typical process mining scenario, a set of traces registering the activities performed along several enactments of a transactional system—such as a Workflow Management (WFM), an Enterprise Resource Planning (ERP), a Customer Relationship Management (CRM), a Business to Business (B2B), or a Supply Chain Management (SCM) system—is given to hand, and the goal is to (semi)automatically derive a model explaining all the episodes recorded in them. Eventually, the "mined" model can be used to design a detailed process schema capable to support forthcoming enactments, or to shed lights into its actual behavior. Thus, process mining is particularly useful when no formal description of the process is available beforehand, or when its observed enactment deviates from the expected one (see, e.g., [41, 25]).

In this chapter, we move from the success story of *ProM* in order to develop PROMETHEUS, a novel Suite for Process Mining applications that, while sharing with *ProM* the perspective of the open and extensible architecture, introduces three innovative designing elements to meet the desiderata of flexibility and scalability arising in actual industrial scenarios. Indeed, PROMETHEUS has been specifically conceived to support:

(1) the definition of complex mining applications, where various mining tasks can be combined and automatically orchestrated at run-time.

Process mining applications may involve dozens of different tasks, ranging from data acquisition, to data manipulation, information extraction based on different mining algorithms, recombination of mining results, and visualization. These different kinds of task can be well managed in the *ProM* framework, but at the price of requiring human intervention in their coordination. Indeed, *ProM* supports the execution of one task at time, so that to construct complex mining applications requires manually invoking the various tasks by collecting and storing each intermediate result (in the *ProM* workspace) and by reusing them as the input for some further tasks. In order to automatize and easily deploy mining applications involving different tasks, PROMETHEUS introduces instead the concept of "flow of mining", i.e., it allows to specify complex mining chains based on interconnecting elementary tasks. In fact, PROMETHEUS comes equipped with a run-time engine that supports and monitors the execution of the mining flow and that orchestrates the compositions of the various elementary bricks.

(2) building interactive applications based on the possibility of customizing data types, algorithms, and graphical user interfaces used in the analysis.

A plug-in based architecture is a crucial factor to provide flexibility for actual applications. However, each plug-in is current viewed in the *ProM* framework as a monolithic box, where interaction is limited to the start up phase in which users configure the execution environment of each algorithm by setting all parameters. In addition, plug-ins can be defined to work over a fixed set of data types only, and there is no way to modify their appearance when executing them in the platform. PROMETHEUS extends the flexibility of each plug-in, by introducing the concept of "interactive" execution (in addition to the standard "batch" execution), i.e., it support an approach to process mining where users may continually interact with the mining algorithms and provide their feedback trough the graphical user interface. Eventually, novel data types can be defined in addition to the standard ones pre-defined in the suite, and they can be transparently used by the various plug-ins. Moreover, PROMETHEUS enables developers to define, for each plug-in, a customizable set of graphical components that will be integrated as a part of the main interface when using the plug-in at run-time. For instance, developers can quickly define windowed components to guide plug-in's execution in the interactive mode, manipulate produced output, and visualize multiple perspectives on results.

(3) ensuring scalability over large volumes of data.

In real industrial environments, enormous volumes of data are available for mining analysis. Yet, very few efforts (see [43], for a noticeable exception focused on pre-processing data originating in ERP applications) have been spent to provide an adequate support for data-intensive applications, mainly because of the architectural design of current tools, which are based on importing the whole input into the main memory. This is in particular the case of *ProM*. To avoid these scalability issues, PROMETHEUS, instead, adopts a powerful data management subsystem based on a stream handling model for data acquisition. Indeed, rather than building a complete in-memory representation of data, this model stores statistical sketches only, while supporting on-demand streaming access to the details that are kept resident on disk.

Discussing the architectural design and the implementation issues arising when providing support to the above three elements is the main aim of this paper. In particular, in Section 4.2, we shall overview these functionalities, while deferring the discussion of their technical design to Section 4.3.

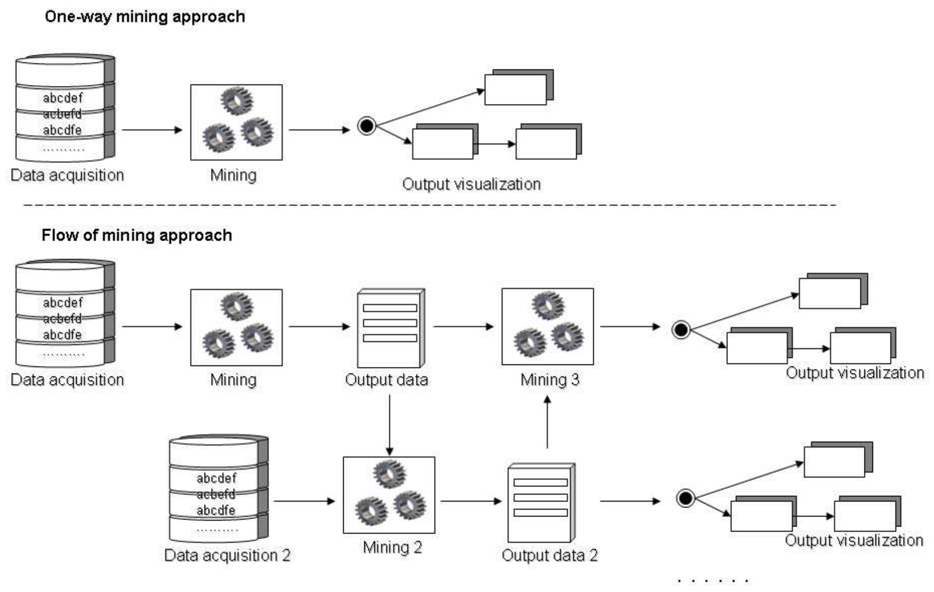


Figure 4.1: Flow of mining in process mining applications

4.2 An Overview of PROMETHEUS

PROMETHEUS is as an extendable suite for process mining applications, entirely programmed in JAVA. In this section, we overview its main functionalities and its innovative designing elements.

4.2.1 Flow of Mining and Customization

The concept of flow of mining is a very natural and manageable way of designing complex mining processes, which possibly involve different and heterogeneous mining components. Indeed, in a traditional mining perspective, each computational module is an independent component with its own input and output. However, to build a large-scale application often requires coordinating several computational modules, which is demanded to the user in current process mining platforms. PROMETHEUS supports instead the deployment of mining applications in their entity, by allowing to design mining process as complex flows of elementary bricks. Each brick produces an output that may be used as input for other bricks in the flow. Consequently, users may incrementally build the desired flow, by connecting existing blocks or adding new ones to manipulate produced outputs.

A comparison between the traditional approach to define process mining applications, and the concept of flow of mining is graphically depicted in Figure 4.1.

To support this concept of flow of mining, at design-time, PROMETHEUS provides the user with an intuitive graphical interface in which computational resources can be dragged and dropped on a work area panel. Moreover, connections between the nodes can be easily established to create the flow over the various the computational steps of the analysis. In particular, data involved in the mining flow (i.e., required in input or produced as output of some computational resource) can be of *Log* type, which is simply an abstraction of a log file, where the enactments of a transactional system are stored, of *Model* type, which represents the abstraction of a process model, and of *Custom* type, which allows users to define their own data-types and to freely import their definition in the suite.

Computational resources in PROMETHEUS operates over the above kinds of data type, and are coupled with it as importable plug-ins. In fact, PROMETHEUS defines the hierarchy of plug-in templates depicted in Figure 4.2, where the following three main templates are actually defined (as an extension of the interface *FlowComponent*):

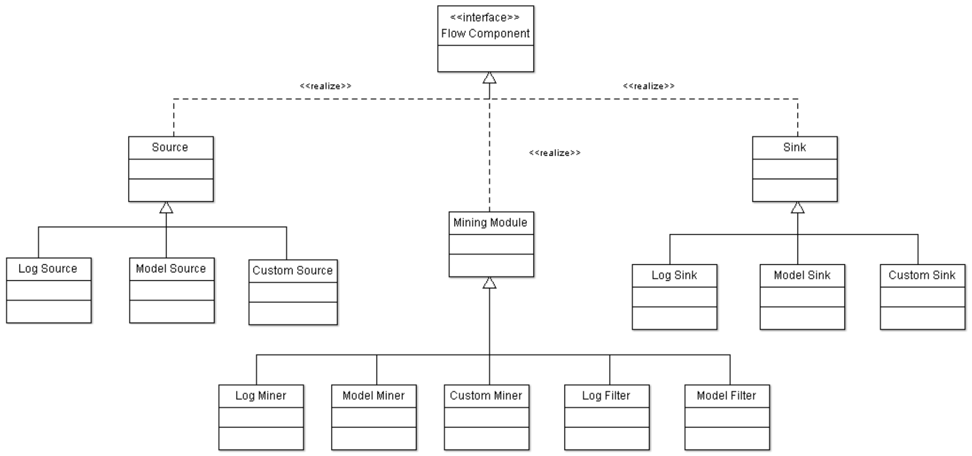


Figure 4.2: UML class diagram for plug-ins hierarchy.

Sources. A source is a plug-in class template conceived to access the input data on which the mining analysis has to be performed. PROMETHEUS defines three kinds of templates for this class. A *Log Source* template is designed to handle the *Log* type in input, and is indeed the most frequent kind of input for process mining applications. A *Model Source* template is designed to handle mining models as input data, thereby enriching the capabilities of the platform to design complex mining flows, where models may serve as the basis for further computation (e.g., comparison with some other model) rather than for visualization only. Finally, a *Custom Source* templates is also designed, in order to provide an abstract source template serving to handle arbitrary *Custom* data.

Mining Modules. A mining module template gives a high-level design of the computational modules in the mining flow. These modules are responsible of performing mining algorithms and statistical evaluations on the input provided by source modules. PROMETHEUS supports five kinds of mining modules templates. A *Log Miner* template is designed to manage a single *Log* as input, and to produce as output one or more instances of *Log*; thus, this module is useful to pre-process input logs, even by complex mining algorithm based, e.g., on clustering and outlier detection. For simpler kinds of pre-processing (e.g., size reduction, or duplicates removing), the *Log Filter* template is available which takes as input a *Log* and produces a novel *Log* based on it. A *Model Miner* template works on a log data source as input, and produces a process model as output. A *Model Filter* template is instead a plug-in operating over a process model and producing a novel process model. Finally, a *Custom Module* template provides an abstract mining module template to fit all the other kinds of needs emerging with *Custom* types.

Sinks. Sinks templates are intended to manage the final results of mining process. These templates are useful for data visualization, analysis and storage of the results. Three sink templates are supported. A *Log Sink* is designed to work on a log data source, a *Model Sink* is designed to work on models, and finally a *Custom Sink* is also provided for custom data input.

Interestingly, plug-ins may be composed in high-level blocks of components performing user-defined operations. In many occasions, in fact, it might be necessary to perform the same operation many times in the same mining flow or in different flows as well. In order to efficiently suite this need, PROMETHEUS supports the grouping of connected plug-ins into *macros* that can be used as ordinary plug-ins with their input and outputs. In practice, macros act as defining sub-routines that frequently occur in mining applications.

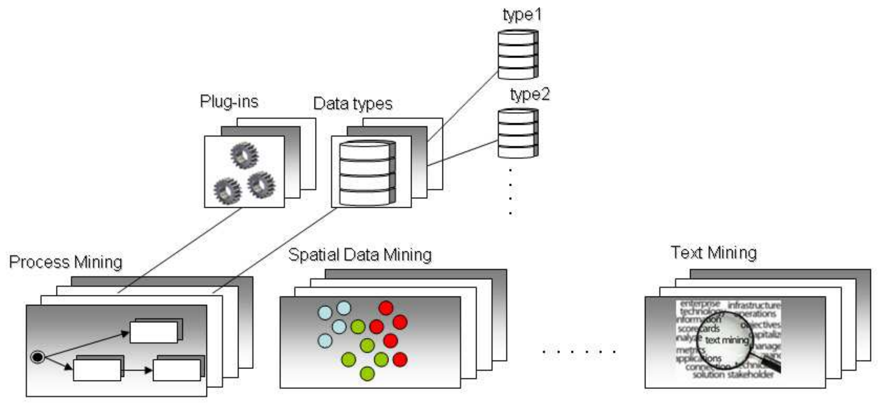


Figure 4.3: Workspace oriented design in data customization.

We leave this section, by stressing that (as emerged from the discussion above) data type customization is a very relevant feature of PROMETHEUS, which makes this platform suited to deal with arbitrary mining analysis, even not focused on processes. As an example, users might be interested in performing text mining or clustering over numerical data, and in creating mining flow chains for them. To easily support such kinds of heterogeneous mining applications, PROMETHEUS introduces the concept of *workspace*, as a place where to conceptually store related data types, and computational modules (see Figure 4.3). Thus, users can freely define sources, mining modules, sinks and data types, organize them in custom workspaces of resources, and then transparently use such resources in PROMETHEUS. Moreover, users can easily connect together resources belonging to different workspaces to obtain mixed mining flows. As an example, users can use a text mining ad-hoc module to extract a log source from a flat textual description, and then use such source as an input for a process mining module.

4.2.2 Flow of Mining in Action: Interactive VS Batch Mode

PROMETHEUS offers two operational modes for supporting the mining analysis: *bach* and *interactive* modes.

In the interactive mode, users control the enactment of the mining flow in a supervised way. Basically, they can choose to execute the whole flow at once or to execute portions of it, by selecting single tasks or groups of them. Additionally, users can stop running tasks, or restart their execution if needed. Moreover, in the interactive mode, users can quickly modify the flow of mining, by adding at any time additional plug-ins and then recover the analysis from the point it was interrupted. And, finally, users may interact with each single mining algorithm via the graphical user interface, by modifying its parameters and even by changing its own execution logic when providing feedbacks on the current execution and results. This kind of approach is particularly useful in the design of complex mining applications, for debugging purposes, and for interactively and incrementally build the mining flow.

When the application scenario is fully understood and the flow of mining needed to obtain the desired process model is consolidated, users may save the mining flow and use it for a batch execution. In this mode, there is no need to run the graphical user interface, since PROMETHEUS comes equipped with a run-time engine which is capable of coordinating the various tasks. This execution mode allows to deploy the mining application as an independent JAVA library, which can be easily coupled within larger kinds of application.

4.2.3 Stream Oriented Log-handling

Available process mining tools are often unsuited for real-world environments, where mining applications have to face large amounts of data. The main fault of such tools is related to the log handling subsystem, which usually builds a

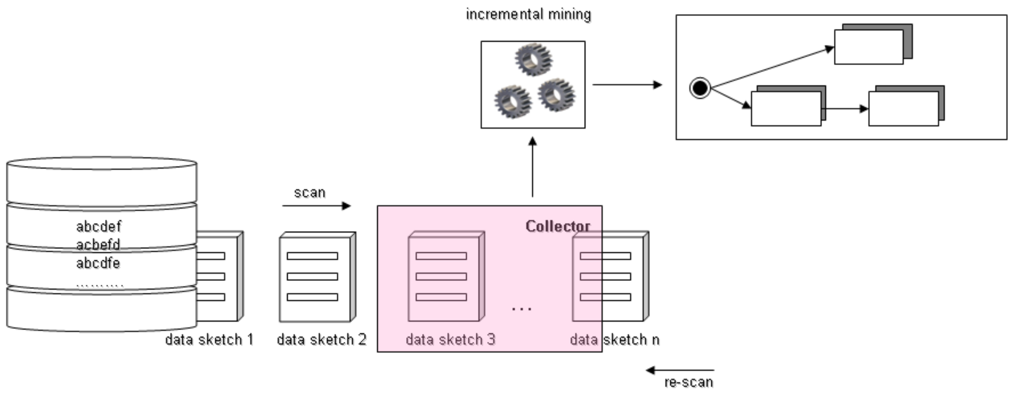


Figure 4.4: Stream oriented log-handling.

complete in-memory representation of input data. Clearly enough, this approach is unviable as far as the input data grow, and memory leaks usually represent the most relevant source of errors while evaluating gigabytes of data.

Moving from these observations, PROMETHEUS implements an innovative streaming approach for log-handling, whose target is to ensure scalability over huge amounts of data. The conceptual idea behind this approach is the following. Basically, a *collector* subdivides the input flow in small data-sketches and processes each one of them separately. Each sketch produces only a partial result in the overall mining process. Once a data sketch has been processed, the collector demands for the next one. Using this approach, the whole mining process is carried out in a step-by-step fashion, and only needed sketches are stored in memory. Note that the strategy above requires the use of ad-hoc mining techniques that work on sketches of data only.

4.3 System Architecture

This section explores the architecture of PROMETHEUS, by highlighting some relevant implementation issues. An overview of this architecture is reported in Figure 4.5. The reader may notice that PROMETHEUS is implemented over four distinct logic layers.

Starting from the bottom of Figure 4.5, the *data* layer is responsible of dealing with physical input/output operations involved in data acquisition and storage. As discussed in Section 4.2.1, data can be of *Log*, *Model*, and *Custom* type; in fact, the second *API* layer provides a transparent access to the physical operations over

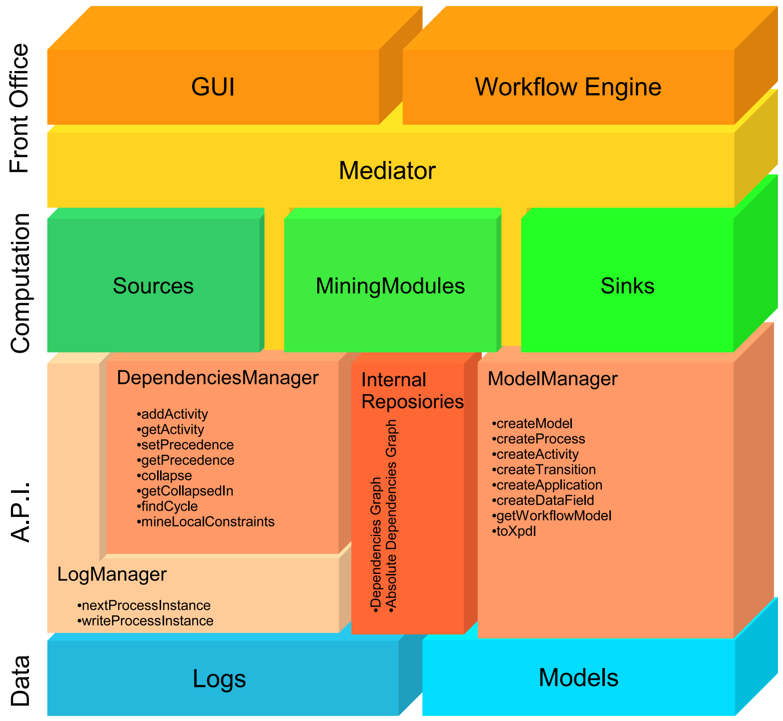


Figure 4.5: PROMETHEUS architecture.

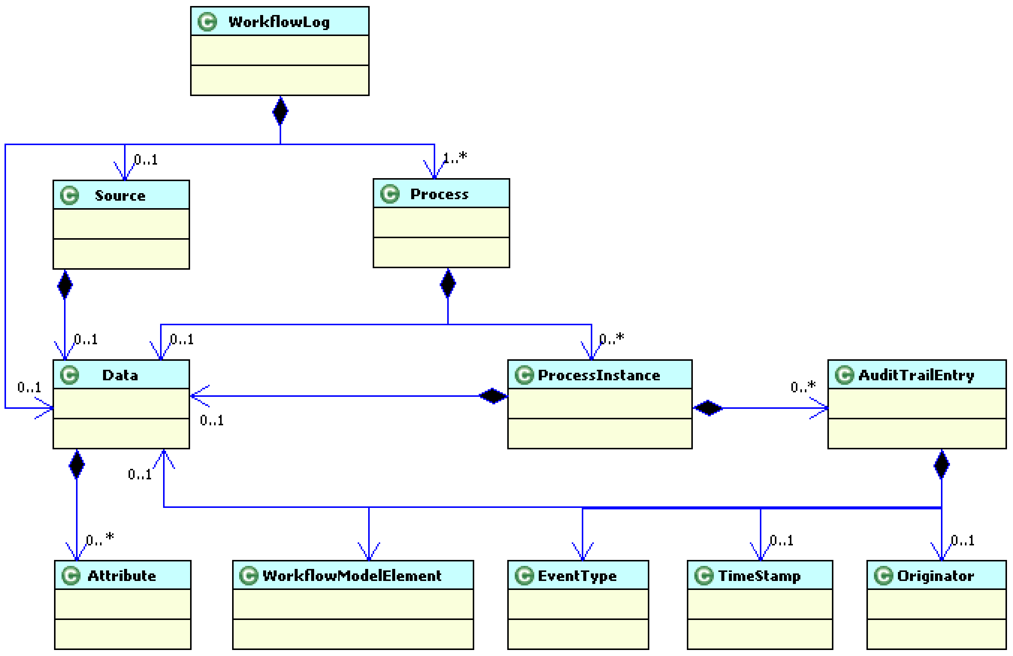


Figure 4.6: MXML data model.

these data types, by exporting several abstract primitives for their stream-oriented manipulation (in particular, w.r.t. predefined data types) and by using some optimized internal data-structures for efficient and compact data (sketches) representation. Above the API layer, it is placed the computational core of PROMETHEUS. The core is constituted by the various plug-ins (sources, mining modules, and sinks) that can be written on the basis of the primitives exported by the API layer. These plug-ins are glued by a *Mediator*, which manages communications between plug-ins and the *Front Office* later. The last layer is indeed constituted by a *GUI* component to be used at design-time and at run-time in the interactive mode, and by a *Workflow Engine* to be used at run-time in the batch mode.

In the rest of the section, we discuss some relevant details for each of these layers.

4.3.1 Data Layer

Regarding the model representation, PROMETHEUS natively supports a subset of XPDL 2.0, which is considered to be the reference data representation for workflow models by W3C community [56].

Regarding input data representation, PROMETHEUS supports the MXML data model, which is an XML-based language considered as the standard de-facto for process mining applications [81]. The UML specification for the MXML data model is reported in Figure 4.6. Basically, this data models views a log for transactional systems as a group of processes, each one associated with many instances (*ProcessInstance*) corresponding to its actual enactments. In fact, each instance consists of a sequence of entries (*AuditTrailEntry*) storing information about the events occurred in the enactment (such as the *Originator* and the *TimeStamp*).

For both process models and logs, PROMETHEUS provides at the data layer I/O primitives and primitives for data manipulation. These low-level primitives are transparently accessed by the API layer, where the basic data types are represented at a higher level of abstraction.

4.3.2 API Layer

The API layer is responsible of carrying out two basic functionalities of PROMETHEUS. On the one hand, it serves to provide support for the efficient internal storage of the data to be used in the analysis. Thus, it handles a main-memory repository where two structures are stored, called *DependencyGraph* and *AbsoluteDependencyGraph*. Both structures are directed graphs whose nodes represent the activities in the process log being analyzed, and whose edges represent the relationship of precedence among them; in fact, the former structure stores the direct relation of precedence, whereas the latter consider the transitive closure of such relation. Note that these two structures are internally built by scanning once the input log (i.e., by following the stream-approach), and serve to provide summary information on the process model being mined, without the need of further I/O operations.

On the other hand, the API layer is responsible of providing transparent access to the data layer. To this end, it implements three main modules:

LogManager. This component provides all the basic capabilities to iterate over a disk-resident MXML log file. The main methods, whose name are intended to be self explicative, are *hasNextProcessInstance* and *nextProcessInstance*. A facility to write a log file in the MXML standard is also provided.

DependenciesManager. This module supplies all APIs necessary to manage activities and dependencies between them. The main methods for editing dependency graphs are *addActivity*, *removeActivity*, *collapseActivity*, and *setPrecedence*. Furthermore, facilities to collapse an activity into another one, find cycles in precedence relations, and mine local constraints in the precedence relations are also implemented.

ModelManager. This module supports all the facilities needed to create and manage process models conforming to the internal meta-model. In details, creation of processes and related elements like applications, data fields, activities and transitions are transparently complained. Once the model is created, each model element can be translated in the correspondent XPDL element by the use of *toXpdl* method.

4.3.3 Computation Layer

Each plug-in may be implemented by using the primitives provided by the API layer. Then, collaboration and message passing between them is supported at the computation layer. Indeed, a *Mediator* acts as a traffic cop, by processing and messaging all the requests from the various plug-ins in the interactive as well as in the batch mode.

In particular, during the batch execution mode, the mediator automatically checks for the dependencies among the involved plug-ins, by keeping updated a map of consistency with the details of the state of the various executions and the execute availability of their input. Indeed, a plug-in may be executed only when all its input data are available, and hence only when they have been completely produced by some antecedent plug-in in the flow of mining. In this scenario, the crucial aim of the mediator is to give to each plug-in of the flow a global view of the state of the plug-ins of interest for it, through a transparent communication infrastructure. Basically, for a given configuration, the mediator chooses which plug-ins are ready to run, executes them, and waits for some notification. When a plug-in changes its state during its execution (e.g., completes the computation of its output), it notifies the mediator through a one way system signal. Then, the mediator propagates the update information about the plug-in state to all those plug-ins whose execution strictly depends on it, and decides which to run, depending on their states as well. Note that, in this approach, only mediator has a global view over the plug-ins involved in the flow and no coupling between them is explicitly needed. Beside these aspects, such design may dramatically improve overall system performances in parallel architectures [22]. There, in fact, the mediator may run unrelated plug-ins at the same time, reducing the overall time to complete the flow of mining.

Importantly, the mediator plays a crucial role during the interactive mode as

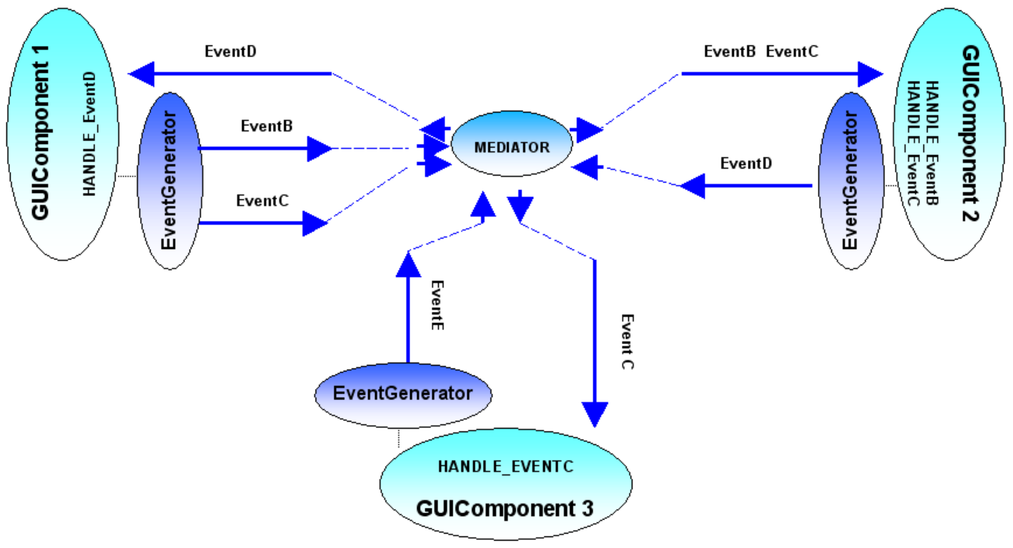


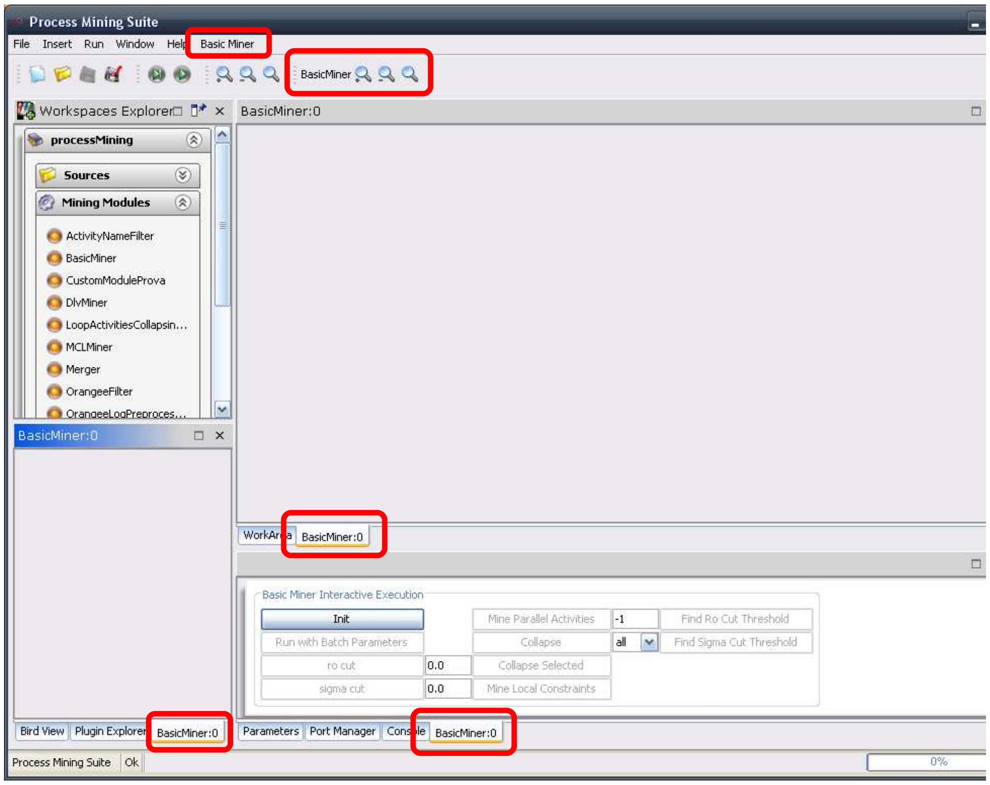
Figure 4.7: Message passing through the mediator.

well. In this execution mode, indeed, PROMETHEUS takes cares of the interaction between the various graphical elements associated with the different plug-ins. Basically, when the state of a component of a plug-in is modified, it generates an event by an *EventGenerator* object. Any component interested in reacting to an event has to implement an HANDLE method for it, further propagating a result event if necessary. According to the observer pattern, the mediator listens all the events generated in PROMETHEUS: When a plug-in graphical interface is modified by the user or by a system signal, the resulting event is notified to the mediator (see Figure 4.7). As a result, only the mediator encapsulates how a given set of GUI components interact together, ensuring loose coupling among the graphical components.

4.3.4 Front Office Layer

The most abstract layer in the architecture of PROMETHEUS is the Front Office layer, which exports (in addition to the workflow engine serving in the batch execution mode) functionalities related to the creation of a process mining flow, to the configuration of execution environment parameters, and to the visualization of analysis results.

All these functionalities involve the graphical interface, which consists of several graphical elements, most of which are shown in Figure 4.8:



ro cut 0.0
sigma cut 0.0
Mine Parallel Activities -1
Collapse all

Figure 4.8: Graphical elements.

Workspaces Explorer. The *workspace explorer* is a graphical component designed to show the available system workspaces. The default "process mining" workspace is shown at the top of the sliding menu. Each workspace is shown as an entry of a navigation menu, and reports all the plug-ins defined in it, organized according their type (i.e., source, mining modules or sinks).

Workarea and Plug-in Explorer. The *workarea* is a graphical component that plays a crucial role in PROMETHEUS. Indeed, it offers a design panel on which users can freely customize mining flow properties. Users can quickly add/remove concrete instances of plug-in definitions (by dragging them from the workspace explorer), edit connections between plug-ins, combine input/outputs, control execution flow, and so on. Once a plug-in instance is placed, users can configure its execution environment in three steps:

- *Parameters Configuration*: if the selected plug-in requires to set some input parameters, users can proceed to their configuration by simply double-clicking on the plug-in instance.
- *Edge Configuration*: users can edit edges (i.e., precedences between tasks) by simply switching the system in the "edge mode". Users can insert a new connection between modules, can rearrange a defined connection, or can remove it from the mining flow.
- *Execution*: once a suitable flow configuration is created, user can proceed to its execution. Runnable plug-ins are identified by a "ready state" icon; users can decide to run all executable plug-ins at once simply selecting, or execute only selected ones.

The workarea provides a graph-based view of the flow of mining. In some cases, it is instead desirable to have a tree-like view of such flow, which is accomplished by the *plug-in explorer* graphical element. In particular, for each concrete plug-in instance, it shows the input data and the tasks which directly depends on it.

Inspectors. To visualize the actual value of input/output data, PROMETHEUS introduces the *inspector* graphical component. An inspector is a very generic data explorer able to produce a suitable representation of a specific data type. Inspector modules definition allows to create multiple different views on the same data set, each one depicting some portion of the data information of interest. PROMETHEUS comes equipped with default inspectors to analyze *Log* and *Model* data types. However, users are free to program their own inspectors for custom data types.

Plug-ins Graphical Elements. Plug-ins can be equipped in PROMETHEUS with various graphical components. In particular, each of them can be associated with a *munu*, with a *toolbar*, with the *main pane* (intended to provide the main graphical interface to support interactive execution), with the *bottom pane* (intended to provide an interface for setting parameters), and with the *quick view* (intended to provide a synthetic view of the status of the plug-in).

Chapter 5

Conclusion

In this thesis we have first presented an extension of the notion of hypertree decomposition, which is currently the most powerful structural method. This new version, called query-oriented hypertree decomposition, is a suitable relaxation of hypertree decomposition designed for query optimization, and such that output variables and aggregate operators can be dealt with. Based on this notion, a hybrid optimizer is implemented, which can be used on top of available DBMSs to compute query plans. The prototype is also integrated into the well-known open-source DBMS PostgreSQL. The experimental activity, conducted on PostgreSQL and on a commercial DBMS, shows that both systems may significantly benefit from using hypertree decompositions for query optimization proving that these techniques can be successfully integrated in commercial products.

Then we have presented the INFOMIX project showing the advanced techniques and innovative methodologies developed in it. We pointed out how it advanced the state of the art in several respects, in particular it provided:

- **Comprehensive Information Model.** A comprehensive information model has to be provided, which incorporates static and dynamic aspects of information integration, and supports advanced *human like* reasoning, based on a rich semantics.

- **Information Integration Algorithms.** A host of efficient algorithms for information integration must be provided, which can be applied to homogenized data from heterogeneous data sources.

- **Usage of Computational Logic.** Exploit advanced methodologies and techniques from computational logic as a toolbox for information integration.

- **Prototype System.** Definition and implementation of a component-based integration system prototype, and providing an infrastructure by using software agent technology.

Finally, we presented PROMETHEUS, a suite for Process Mining and shown its open and extensible architecture and how it introduce some innovative designing elements to meet the desiderata of flexibility and scalability arising in actual industrial scenarios. In particular it supports:

- the definition of complex mining applications, where various mining tasks can be combined and automatically orchestrated at run-time.
- building interactive applications based on the possibility of customizing data types, algorithms, and graphical user interfaces used in the analysis.
- scalability over large volumes of data.

As possible future extensions of this work, we think that some improvements can be achieved further extending the Hypertree Decomposition techniques in order to support also aggregate and other SQL features. Furthermore, we think that it is possible to engineer the INFOMIX prototype and PROMETHEUS in order to obtain industrial products.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] I. Adler, G. Gottlob, and M. Grohe. Hypertree width and related hypergraph invariants. *European Journal of Combinatorics*, 28(8):2167–2181, 2007.

[3] A. Agostini and G. De Michelis. Improving flexibility of workflow management systems. *Lecture Notes In Computer Science; Vol. 1806*, pages 218–234, 2000.

[4] M. Arenas, L. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 68–79. ACM New York, NY, USA, 1999.

[5] M. Arenas, L. Bertossi, and J. Chomicki. Specifying and querying database repairs using logic programs with exceptions. In *Flexible query answering systems: recent advances: proceedings of the Fourth International Conference on Flexible Query Answering Systems, FQAS'2000, October 25-28, 2000, Warsaw, Poland*, page 27. Physica Verlag, 2000.

[6] M. Arenas, L. Bertossi, and J. Chomicki. Specifying and querying database repairs using logic programs with exceptions. In *Flexible query answering systems: recent advances: proceedings of the Fourth International Conference on Flexible Query Answering Systems, FQAS'2000, October 25-28, 2000, Warsaw, Poland*, page 27. Physica Verlag, 2000.

[7] F. Baader, D. Calvanese, D.L. McGuinness, P. Patel-Schneider, and D. Nardi. *The description logic handbook: theory, implementation, and applications*. Cambridge Univ Pr, 2003.

[8] F. Bancilhon, F. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Proc. of the ACM Symposium on Principles of Database Systems (PODS'86)*, pages 1–16, Cambridge, Massachusetts, 1986. ACM Press.

[9] D. Beneventano, S. Bergamaschi, S. Castano, A. Corni, R. Guidetti, G. Malvezzi, M. Melchiori, and M. Vincini. Information integration: the MOMIS project demonstration. *language*, 1:3.

[10] L. Bertossi and J. Chomicki. Consistent query answers in inconsistent databases. In *In PODS*, 1999.

[11] A. Borgida. Description logics in data management. *IEEE transactions on knowledge and data engineering*, 7(5):671–682, 1995.

[12] M. Bouzeghoub and M. Lenzerini. Introduction to: data extraction, cleaning, and reconciliation a special issue of Information Systems, An International Journal. *Information Systems*, 26(8):535–536, 2001.

[13] Jos De Bruijn, Holger Lausen, and Axel Polleres. The web service modeling language wsml: an overview. Technical report, DERI, June 2005. Available at: http://www.wsmo.org/wsml/wsml-resources/deri-tr-2005-06-16.pdf.

[14] A. Calì, D. Calvanese, G. De Giacomo, and M. Lenzerini. Data integration under integrity constraints. In *Advanced information systems engineering: 14th International Conference, CAiSE 2002, Toronto, Canada, May 27-31, 2002: proceedings*, page 262. Springer Verlag, 2002.

[15] D. Calvanese, G. De Giacomo, and M. Lenzerini. Answering queries using views over description logics knowledge bases. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, pages 386–391. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2000.

[16] D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, and R. Rosati. Description logic framework for information integration. In *PRINCIPLES OF KNOWLEDGE REPRESENTATION AND REASONING-INTERNATIONAL CONFERENCE-*, pages 2–13. MORGAN KAUFMANN PUBLISHERS, 1998.

[17] D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, and R. Rosati. Data integration in data warehousing. *International Journal of Cooperative Information Systems*, 10(3):237–272, 2001.

[18] M.J. Carey, L.M. Haas, P.M. Schwarz, M. Arya, W.F. Cody, R. Fagin, M. Flickner, A.W. Luniewski, W. Niblack, D. Petkovic, et al. Towards heterogeneous multimedia information systems: The Garlic approach. In *Proceedings of the 5th International Workshop on Research Issues in Data Engineering-Distributed Object Management (RIDE-DOM'95)*, page 124, 1995.

[19] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow evolution. *Data & Knowledge Engineering*, 24(3):211–238, 1998.

[20] C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. *Theoretical Computer Science*, 239(2):211–229, 2000.

[21] F. cois Bry. Query Answering in Information Systems with Integrity Constraints.

[22] Riehle D. Composite design patterns. In *In Proceedings of the 1997 ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 218–228. ACM Press, 1997.

[23] P.M. Dung. Integrating data from possibly inconsistent databases. In *International Conference on Cooperative Information Systems, Brussels, Belgium*. Citeseer, 1996.

[24] O.M. Duschka. Query planning and optimization in information integration. 1998.

[25] van der Aalst W. M. P. Dustdar S., Hoffmann T. Mining of ad-hoc business processes with teamlog. *Data Knowledge Engineering*, 55(2):129–158, 2005.

[26] Thomas Eiter, Wolfgang Faber, Michael Fink, Stefan Woltran, and Lilianna Zalewska. Methods for data acquisition and transformation. Technical Report D6.2, Infomix Consortium, December 2003.

[27] Thomas Eiter, Wolfgang Faber, Georg Gottlob, Christoph Koch, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and F. Scarcello. The DLV System. In Jack Minker, editor, *Workshop on Logic-Based Artificial Intelligence*,

Washington, DC, College Park, Maryland, June 1999. Computer Science Department, University of Maryland. Workshop Notes.

[28] C.A. Ellis and K. Keddara. A workflow change is a workflow. *Lecture Notes In Computer Science; Vol. 1806*, pages 201–217, 2000.

[29] L. Fischer. *Workflow handbook 2001*. Future Strategies Inc., 2000.

[30] D. Florescu, A. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 311–322. ACM New York, NY, USA, 1999.

[31] H. Galhardas, D. Florescu, D. Shasha, and E. Simon. An extensible framework for data cleaning. In *In ICDE*, 2000.

[32] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom. The TSIMMIS approach to mediation: Data models and languages. *Journal of intelligent information systems*, 8(2):117–132, 1997.

[33] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database system implementation*. Prentice Hall, 2000.

[34] C.H. Goh, S. Bressan, S. Madnick, and M. Siegel. Context interchange: New features and formalisms for the intelligent integration of information. *ACM Transactions on Information Systems*, 17(3):270–293, 1999.

[35] G. Gottlob, N. Leone, and F. Scarcello. Advanced parallel algorithms for processing acyclic conjunctive queries, rules, and constraints. In *Proc. of the Conference on Software Engineering and Knowledge Engineering (SEKE00)*, pages 167–176.

[36] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions: A survey. *Lecture Notes in Computer Science*, pages 37–57, 2001.

[37] G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. *Journal of the ACM (JACM)*, 48(3):431–498, 2001.

[38] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences*, 64(3):579–627, 2002.

[39] G. Gottlob, N. Leone, and F. Scarcello. Robbers, marshals, and guards: game theoretic and logical characterizations of hypertree width. *Journal of computer and system sciences*, 66(4):775–808, 2003.

[40] G. Greco, S. Greco, and E. Zumpano. A logic programming approach to the integration, repairing and querying of inconsistent databases. *Lecture notes in computer science*, pages 348–364, 2001.

[41] Pontieri L. Greco G., Guzzo A. Discovering expressive process models by clustering log traces. *IEEE Transactions on Knowledge and Data Engineering.*, 18(8):1010–1027, 2006.

[42] J. Gryz. Query folding with inclusion dependencies. In *Proceedings Of The International Conference On Data Engineering*, pages 126–133. Institute Of Electrical And Electronics Engineers, 1998.

[43] Gulla J.A. Ingvaldsen J. Preprocessing support for large scale process mining of sap transactions. 2008.

[44] S. Jablonski and C. Bussler. Workflow management: modeling concepts, architecture and implementation. 1996.

[45] M. Jarke, M. Lenzerini, Y. Vassiliou, and P. Vassiliadis. *Fundamentals of data warehouses*. Springer Verlag, 1999.

[46] T. Kirk, A.Y. Levy, Y. Sagiv, and D. Srivastava. The information manifold. In *Proceedings of the AAAI 1995 Spring Symp. on Information Gathering from Heterogeneous, Distributed Enviroments*, volume 7, pages 85–91. Citeseer, 1995.

[47] M. Klein, C. Dellarocas, and A. Bernstein. Towards Adaptive Workflow Systems, CSCW-98 Workshop, Proceedings, 1998.

[48] D. Lembo, M. Lenzerini, and R. Rosati. Source inconsistency and incompleteness in data integration. *KRDB02*, 2002.

[49] Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Formalism for IIM specification. Technical Report D3.1, Infomix Consortium, April 2003.

[50] C. Li and E. Chang. Query planning with limited source capabilities. In *Proceedings Of The International Conference On Data Engineering*, pages 401–412. IEEE Computer Society Press; 1998, 2000.

[51] C. Li, R. Yerneni, V. Vassalos, H. Garcia-Molina, Y. Papakonstantinou, J. Ullman, and M. Valiveti. Capability based mediation in TSIMMIS. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 564–566. ACM New York, NY, USA, 1998.

[52] J. Lin and A.O. Mendelzon. Merging databases under constraints. *International Journal of Cooperative Information Systems*, 1998.

[53] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries over heterogeneous data sources. In *Proceedings of the International Conference on Very Large Data Bases*, pages 241–250, 2001.

[54] Alon Levy Marc Friedman and Todd Millstein. Navigational plans for data integration.

[55] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[56] Newman J. Ping J., Mair Q. Using uml to design distributed collaborative workflows: from uml to xpdl. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003. WET ICE 2003.*, pages 71 – 76.

[57] A. Rajaraman, Y. Sagiv, and J.D. Ullman. Answering queries using templates with binding patterns (extended abstract). In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 105–112. ACM New York, NY, USA, 1995.

[58] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. The CORAL deductive system. *VLDB J.*, 3(2):161–210, 1994.

[59] O. Reingold. Undirected st-connectivity in log-space. In *Proceedings of the thirty-seventh annual ACM Symposium on Theory of computing*, pages 376–385. ACM New York, NY, USA, 2005.

[60] A. Rozinat and W.M.P. van der Aalst. Decision mining in ProM. *Lecture Notes in Computer Science*, 4102:420, 2006.

[61] D. Saccà and C. Zaniolo. The generalized counting method for recursive logic queries. *Theoretical Computer Science*, 62, 1988.

[62] Steven L. Salzberg. Book review: C4.5: Programs for machine learning by j. ross quinlan. morgan kaufmann publishers, inc., 1993. *Machine Learning*, 16(3):235–240, 1994.

[63] F. Scarcello, G. Greco, and N. Leone. Weighted hypertree decompositions and optimal query plans. *Journal of Computer and System Sciences*, 73(3):475–506, 2007.

[64] A.W. Scheer, O. Thomas, and O. Adam. Process modeling using event-driven process chains. *Process-aware Information Systems: Bridging People and Software through Process Technology*, pages 119–145, 2005.

[65] P.G. Selinger, MM Astrahan, DD Chamberlin, RA Lorie, and TG Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM New York, NY, USA, 1979.

[66] P.D. Seymour and R. Thomas. Graph searching and a min-max theorem for tree-width. *Journal of Combinatorial Theory, Series B*, 58(1):22–33, 1993.

[67] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208, 1997.

[68] R.E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13:566, 1984.

[69] Maruster L. van Beest N. R. T. P. A process mining approach to redesign business processes - a case study in gas industry. In *SYNASC '07: Proceedings of the Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 541–548, 2007.

[70] J. van Benthem. Dynamic bits and pieces. *ILLC research report*, 1997.

[71] W. Van der Aalst and K.M. van Hee. *Workflow management: models, methods, and systems*. The MIT press, 2004.

[72] Wil M. P. van der Aalst, Hajo A. Reijers, and Minseok Song. Discovering social networks from event logs. *Computer Supported Cooperative Work*, 14(6):549–593, 2005.

[73] WMP Van der Aalst. Formalization and verification of event-driven process chains. *Information and Software technology*, 41(10):639–650, 1999.

[74] W.M.P. van der Aalst, J. Desel, and A. Oberweis. *Business Process Management, Models, Techniques, and Empirical Studies*. Springer-Verlag London, UK, 2000.

[75] W.M.P. Van der Aalst et al. The application of Petri nets to workflow management. *Journal of Circuits Systems and Computers*, 8:21–66, 1998.

[76] W.M.P. van der Aalst and S. Jablonski. Dealing with workflow change: identification of issues and solutions. *Computer systems science and engineering*, 15(5):267–276, 2000.

[77] Herbst J. Maruster L. Schimm G. Weijters A. J. M. M. van der Aalst W. M. P., van Dongen B. F. Workflow mining: a survey of issues and approaches. *Data Knowledge Engineering*, 47(2):237–267, 2003.

[78] Weijters A. J. M. M. van Dongen B. F. Alves de Medeiros A. K. Song M. Verbeek H. M. W. van der Aalst W. M. P., Reijers H. A. Business process mining: An industrial application. *Information Systems*, 32(5):713–732, 2007.

[79] R. van der Meyden. Logical approaches to incomplete information: a survey, Logics for databases and information systems, 1998.

[80] de Medeiros A.K.A. Verbeek H.M.W. van der Aalst W.M.P. van Dongen B.F., WeijtersA.J.M.M. The prom framework: A new era in process mining tool support. pages 444–454, 2005.

[81] van der Aalst W.M.P. van Dongen B.F. A meta model for process mining data. In *Proceedings of the CAiSE 05 Workshops (EMOI-INTEROP Workshop)*, volume 2. Casto J.,Teniente E. (Eds.), 2005.

[82] M.Y. Vardi. The complexity of relational query languages (Extended Abstract). In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 137–146. ACM New York, NY, USA, 1982.

[83] van der Aalst W. M. P. Weijters A. J. M. M. Rediscovering workflow models from event-based data using little thumb. *Integrated Computer-Aided Engineering*, 10(2):151–162, 2003.

[84] A.N. Wilschut, J. Flokstra, and P.M.G. Apers. Parallel evaluation of multi-join queries. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 115–126. ACM New York, NY, USA, 1995.

[85] Ian H. Witten and Frank Eibe. Data mining: practical machine learning tools and techniques with java implementations. *SIGMOD Record*, 31(1):76–77, 2002.

[86] M. Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the seventh international conference on Very Large Data Bases-Volume 7*, page 94. VLDB Endowment, 1981.

[87] R. Yerneni, C. Li, J. Ullman, and H. Garcia-Molina. Optimizing large join queries in mediation systems. *Lecture Notes in Computer Science*, pages 348–364, 1999.

[88] G. Zhou, R. Hull, R. King, and J.C. Franchitti. Data integration and warehousing using H2O. *Bulletin of the Technical Committee on*, 51:29.

Appendix A

Relational Databases

In this appendix we illustrate here the basic notions of the relational model for a data base. In the relational model, predicate symbols are used to denote the relations in the database, whereas constant symbols denote the objects and the values stored in relations. We assume to have a fixed (infinite) alphabet Γ of constants, and we consider only databases over such an alphabet. We adopt the so-called *unique name assumption*, i.e., we assume that different constants denote different objects.

A *relational schema* $\mathcal{H}$ is constituted by:

- An alphabet $\mathcal{A}$ of *predicate* (or relation) symbols, each one with an associated arity denoting the number of arguments of the predicate (or attributes of the relation).

- A set $\mathcal{C}$ of *integrity constraints*, i.e., assertions on the symbols of the alphabet $\mathcal{A}$ that are intended to be satisfied in every database coherent with the schema.

A *relational database* (or simply, database, DB) $\mathcal{DB}$ over a schema $\mathcal{H}$ is simply a set of relations with constants as atomic values. We have one relation of arity n for each predicate symbol of arity n in the alphabet $\mathcal{A}$. The relation $r^{\mathcal{DB}}$ in $\mathcal{DB}$ corresponding to the predicate symbol R is constituted by a set of tuples of constants, those that satisfy the predicate R. A database $\mathcal{DB}$ over a schema $\mathcal{H}$ is said to be *coherent* with $\mathcal{H}$ if every constraint in $\mathcal{C}$ is satisfied by $\mathcal{DB}$. The notion of satisfaction depends on the type of constraints defined over the schema.

The integrity constraints that we consider are inclusion dependencies (IDs), key dependencies (KDs) and exclusion dependencies (EDs). More specifically,

- an inclusion dependency is an assertion of the form $r_1[A] \subseteq r_2[B]$, where r_1, r_2 are relations in $\mathcal{A}$, $A = A_1, \ldots, A_n (n \geq 0)$ is a sequence of attributes of r_1, and $B = B_1, \ldots, B_n$ is a sequence of distinct attributes of r_2. Therefore, we allow for repetition of attributes in the left-hand side of the inclusion. A database $\mathcal{DB}$ for $\mathcal{H}$ satisfies an inclusion dependency $r_1[A] \subseteq r_2[B]$ if for each tuple $t_1 \in r_1^{\mathcal{DB}}$ there exists a tuple $t_2 \in r_2^{\mathcal{DB}}$ such that $t_1[A] = t_2[B]$, where $t[A]$ indicates the projection of the tuple t over A;

- a key dependency is an assertion the form $key(r) = A$, where r is a relation in $\mathcal{A}$, and $\mathcal{A}$, $A = A_1, \ldots, A_n$ is a sequence of distinct attributes of r. A database $\mathcal{DB}$ for $\mathcal{H}$ satisfies a key dependency $key(r) = A$ if for each $t_1, t_2 \in r^{\mathcal{DB}}$ with $t_1 \neq t2$ we have $t_1[A] \neq t_2[A]$. We assume that at most one key dependency is specified for each relation;

- an exclusion dependency is an assertion of the form $(r_1[A] \cap r_2[B]) = \emptyset$, where r_1, r_2 are relations in $\mathcal{A}$, $A = A_1, \ldots, A_n$ and $B = B_1, \ldots, B_n$ are sequences of attributes of r_1 and r_2, respectively. A database $\mathcal{DB}$ for $\mathcal{H}$ satisfies an exclusion dependency $(r_1[A] \cap r_2[B]) = \emptyset$ if there do not exist two tuples $t_1 \in r_1^{\mathcal{DB}}$ and $t_2 \in r_2^{\mathcal{DB}}$ such that $t_1[A] = t_2[B]$.

A relational query is a formula that specifies a set of data to be retrieved from a database. In the sequel we mainly refer to the class of conjunctive queries, union of conjunctive queries and Datalog queries. A *conjunctive query* (CQ) q of arity n over the schema $\mathcal{H}$ is written in the form

$$q(\vec{\mathbf{x}}) \leftarrow conj(\vec{\mathbf{x}}, \vec{\mathbf{y}})$$

where

- q belongs to a new alphabet $\mathcal{Q}$ (the alphabet of queries, that is disjoint from both Γ and $\mathcal{A}$),

- $q(\vec{\mathbf{x}})$ is the *head* of the conjunctive query,

- $conj(\vec{\mathbf{x}}, \vec{\mathbf{y}})$ is the *body* of the conjunctive query and is a conjunction of atoms involving the variables $\vec{\mathbf{x}} = X_1, \ldots, X_n$ and $\vec{\mathbf{y}} = Y_1, \ldots, Y_n$, and constants from Γ,

- the predicate symbols of the atoms are in $\mathcal{A}$,

- the number of variables of $\vec{\mathbf{x}}$ is called the *arity* of q, and is the arity of the relation denoted by the query q.

Notice that, the body of the query may also contain atoms whose predicates are arithmetic comparison predicates, i.e., built-in predicates, with the restriction that variables involved in such predicates must appear also in atoms whose predicate symbols are in $\mathcal{A}$.

Given a database $\mathcal{DB}$, the answer to q over $\mathcal{DB}$, denoted $q^{\mathcal{DB}}$, is the set of n-tuples of constants $(c_1, \ldots, c_n)$, such that, when substituting each c_i for x_i, the formula

$$\exists \vec{\mathbf{y}}.conj(\vec{\mathbf{x}}, \vec{\mathbf{y}})$$

evaluates to true in $\mathcal{DB}$.

A set of conjunctive queries with the same head predicate is a *Union of Conjunctive Queries* (UCQ). More formally, a UCQ is written in the form

$$q(\vec{\mathbf{x}}) \ \leftarrow \ conj_1(\vec{\mathbf{x}}, \vec{\mathbf{y}}_1) \vee \cdots \vee conj_m(\vec{\mathbf{x}}, \vec{\mathbf{y}}_m)$$

The answer to a UCQ q over a database $\mathcal{DB}$, as usually denoted $q^{\mathcal{DB}}$, is the set of n-tuples of constants $(c_1, \ldots, c_n)$, such that, when substituting each c_i for x_i, the formula

$$\exists \vec{\mathbf{y}}_1.conj_1(\vec{\mathbf{x}}, \vec{\mathbf{y}}_1) \vee \cdots \vee \exists \vec{\mathbf{y}}_m.conj_m(\vec{\mathbf{x}}, \vec{\mathbf{y}}_m)$$

evaluates to true in $\mathcal{DB}$.

Finally a Datalog query is a collection of rules, each having the same form as a conjunctive query, except that predicate symbols in the body of the rules can be in $\mathcal{Q}$ as well. In a Datalog query, each head predicate of the rules refers to an intermediate relation, and has not to contain predicates referring to database relations. The intermediate predicates are called *Intensional DataBase* (IDB) predicates, whereas predicates referring to stored relations are called *Extensional DataBase* (EDB) predicates. Given a Datalog query q and a database $\mathcal{DB}$, the answer $q^{\mathcal{DB}}$ of q over $\mathcal{DB}$ is the minimal fixpoint model of q and $\mathcal{DB}$ [1].

Given a CQ, UCQ, or Datalog query q, we also say that $q^{\mathcal{DB}}$ denotes the set of tuples that *satisfy* q over $\mathcal{DB}$.