Query Planning in Mediator Based Information Systems

vorgelegt von
Diplom - Informatiker

Ulf Leser

Vom Fachbereich 13 – Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr. Ing. –

Promotionsausschuß:	Prof. Dr. Hommel Prof. Dr. Weber Prof Dr. Freytag
Tag der wissenschaftlichen Aussprache:	28 Juni 2000

Berlin, 17 September 2000

D 83

mediation: *a practice under which, in a conflict, the services of a third party are utilized to reduce the differences or to seek a solution. Mediation differs from "good offices" in that the mediator usually takes more initiative in proposing terms of settlement. It differs from arbitration in that the opposing parties are not bound by prior agreement to accept the suggestions made.*

Encyclopaedia Britannica

There can be no understanding
between the brain and the hands,
unless the heart acts as mediator.

From the movie “Metropolis”

Abstract

Information integration has gained new importance since the widespread success of the World Wide Web. The simplicity of data publishing on the web and the promises of the emerging eCommerce markets pose a strong incentive for data providers to offer their services on the Internet. Due to the exponential growth rate of the number of web sites, users are already faced with an overwhelming amount of accessible information. Finding the desired piece of information is difficult and time-consuming due to the inherently chaotic organisation of the Web.

For this reason, information integration services are becoming increasingly important. The idea of such a service is to offer to a user a single point of access that provides him or her exactly with the information he or she is interested in. To achieve this goal, the service dynamically integrates and customises data from various data providers. For instance, a business information service would integrate news tickers, specialised business databases, and stock information. However, integration services face serious technical problems. Two of them are particularly hard to overcome: The heterogeneity between data sources and the high volatility that interfaces to data providers on the web typically exhibit.

Mediator based information systems (MBIS) offer remedies for those problems. A MBIS tackles heterogeneity on two levels: Mediators carry out structural and semantic integration of information stemming from different origins, whereas wrappers solve technical and syntactical problems. Users only communicate with mediators, which use wrappers to access data sources. To this end, mediators and wrappers are connected by declarative rules that semantically describe the content of data sources. This decoupling supports the stability of interfaces and thus increases the maintainability of the overall system.

This thesis discusses query-centred MBIS, i.e., MBIS in which mediators represent their domain through a schema. The core functionality of a mediator is to receive and answer user queries against its schema. Two ingredients are essential to accomplish this task: First, it requires a powerful language for specifying the rules that connect mediators and wrappers. The higher the expressiveness of this these rules, the more types of heterogeneity can be overcome declaratively. Second, the mediator must be equipped with algorithms that are – guided by the semantic rules – capable of efficiently rewriting user queries into queries against wrappers.

We contribute to both issues. We introduce query correspondence assertions (QCA) as a flexible and expressive language to describe the content of heterogeneous data sources with respect to a given schema. QCAs are able to bridge more types of conflicts between schemas than previous languages. We describe algorithms that rewrite queries against a mediator into sequences of queries against wrappers, based on the knowledge encoded in QCAs. Our algorithms are considerably more efficient than previously published algorithms for query rewriting in MBIS. Furthermore, we define a formal semantics for queries in MBIS, which allows us to derive statements about properties of rewriting algorithms. Based on this semantics, we prove that our algorithm is sound and complete. Finally, we show how to reduce the main cost factor of query answering in MBIS, i.e., the number of accesses to remote data sources. To this end, we device algorithms that are capable of detecting and removing redundant remote accesses.

Zusammenfassung

Die in den letzten Jahren rasant zunehmende Vernetzung von Informationssystemen hat neue Perspektiven für die Informationsintegration geschaffen. Dies hat im wesentlichen zwei Gründe: Zum einen die Verfügbarkeit einer kritischen Masse an Information, deren Integration einen echten Mehrwert darstellt, und zum anderen die Unübersehbarkeit und Unübersichtlichkeit der vorhandenen Informationsquellen, deren getrennte Handhabung für einen Einzelnen unzumutbar wird. Beide Aspekte zusammen bewirken einen zunehmenden Marktwert für Services der Art ‚Integrierter Zugriff‘.

Wer Integration als Service versteht, plaziert sich als Vermittler (Mediator) zwischen Informationslieferanten und Informationskonsumenten. Der Integrationsservice selbst nimmt beide Rollen ein: Für die eigentlichen Content Provider ist er Konsument, für den Endkonsumenten ist er Lieferant. Gegenüber seinen Konsumenten muß er einen Mehrwert erbringen, z.B. einen transparenten Zugriff auf verschiedene Informationsquellen oder die semantische Integration unterschiedlicher Informationsquellen in eine einheitliche Weltsicht.

Die Unterstützung zur Schaffung eines solchen Mehrwertes, d.h., die Unterstützung einer hochwertigen Informationsintegration, ist eine Herausforderung für die Informatik. Abstrakt gesprochen, ist das Ziel die Bereitstellung eines einheitlichen Zugriffs auf eine Menge von heterogenen, autonomen und verteilten Informationsquellen. Sowohl Heterogenität als auch Autonomie spielen dabei heute eine größere Rolle als noch vor einigen Jahren: Während man damals unter Heterogenität oftmals nur unterschiedliche Dialekte relationaler Anfragesprachen verstand, wird man heute oft mit Datenquellen konfrontiert, die schlichtweg überhaupt keine Anfragesprache im eigentlichen Sinne unterstützen. Quellautonomie bezeichnete damals vor allem die Tatsache, daß man die Eigenschaften einer Quelle als gegeben und unveränderlich hinnehmen muß; heute verbindet man mit Autonomie vor allem das Problem der Integration sich laufend ändernder Quellen. Typische Beispiele für anfragebeschränkte und sich im stetigen Wandel befindliche Datenquellen sind Web-basierte Informationsdienste. Die Bereitstellung eines kostengünstigen und wartbaren Integrationsservices muß daher durch flexible und ausdrucksstarke Verfahren unterstützt werden.

Die vorliegende Dissertation beschreibt solche Verfahren. Sie konzentriert sich auf zwei Komponenten einer Mediator-basierten Integrationsarchitektur. Zum einen stellt sie eine Sprache zur Beschreibung des Inhalts von Datenquellen sowie Ihrer Schnittstellen vor, die eine Abbildung auf ein einheitliche Schema ermöglicht. Zum anderen beschreibt die Arbeit Algorithmen, die in der Lage sind, Anfragen an ein einheitliche Schema effizient in Aufrufe der Datenquellen zu übersetzen. Die vorgestellten Algorithmen sind sowohl formal fundiert als auch bisher veröffentlichten Verfahren bzgl. Komplexität und Flexibilität überlegen.

Die Ergebnisse der vorliegenden Arbeit unterstützen damit maßgeblich die Bewältigung der oben genannten Probleme. Die Entkopplung der Quellbeschreibungen von den Algorithmen zur Anfrageübersetzung erlaubt die flexible und schnelle Reaktion auf Änderungen in Quellen. Die neu entwickelte Beschreibungssprache ist in der Lage, Web-basierte Informationsdienste gleichermaßen wie relationale Datenbanken einzubinden, da sie nur minimale Annahmen über die von Datenquellen bereitgestellte Funktionalität macht. Dem Benutzer bleibt die Heterogenität der Quellen gänzlich verborgen, denn der Mediator präsentiert sie gemäß einer homogenen Weltsicht.

Acknowledgements

First of all I thank my two supervisors for their constant support and advice during the preparation of this thesis. Professor Weber and his department at the Technical University of Berlin provided the indispensable shelter and scientific infrastructure. Professor Freytag of the Humboldt University of Berlin co-supervised my work and improved it through many valuable comments on technical issues.

I always very much enjoyed the lively and controversial discussions with my colleagues at CIS, the department for Computer-Based Information Systems at the Technical University of Berlin. Especially, I thank Dr. Ralf–Detlef Kutsche for continuous moral support and Susanne Busse for the inspiring seminar on Mediator-Based Information Systems we held together in 1998.

This thesis would not have been written without the financial support of the Berlin and Brandenburg Graduate School for Distributed Information Systems. Through the regular evaluation of my ideas the graduate school was also a helpful forum to filter the good from the bad.

Furthermore, I thank my diploma students Heiko Müller, Christoph Holzheuer, Oliver Kinne, and Holger Last, whose questions forced me to work out the details of my ideas; my temporary colleagues at the European Bioinformatics Institute, especially Patricia Rodriguez-Tome, Kim Jungfer, Steffen Möller, and Philip Lijnzaad; Lutz Friedel, the best system administrator I ever had; Sabine Lembke for listening whenever it was necessary; and Hugues Roest-Crollius at the Max-Planck-Institute for Molecular Genetics, who introduced me to the problem of data integration in a heterogeneous and distributed environment.

Finally, I am particularly grateful to Felix Naumann, without whom this work would have been only half the fun it was.

Table of Contents

1. INTRODUCTION

Some of the great challenges for current computer science emerge from the need for integration. In software engineering, integration is necessary for the development of software systems that are built on top of existing components; in information systems, integration appears as the task of building systems whose content is provided by and managed in other, pre-existing, and autonomous information systems.

Integrating heterogeneous components into a new system is the more complicated the more heterogeneity exists between existing systems and the requirements for the new system. In information systems, heterogeneity appears for instance in differently structured schemas, different scopes and meanings of schema elements, and different access interfaces. Coping with heterogeneity is always cumbersome. The necessary effort grows with the degree of autonomy of systems being integrated. At worst, a developer is confronted with heterogeneous, autonomous, and independently evolving information systems.

In 1985, Heimbinger & McLeod first described an approach to achieve integration in such an environment [HM85]. The authors introduced the term "*database federation*" to emphasise that participating systems on one hand need to cooperate – to solve a global "problem", i.e., a query – but on the other hand try to keep a high degree of autonomy. Since then, federated database systems have been a highly active area of research (see [EP90; BE95; BBE98]).

In [SL90], Sheth & Larson summarise achieved results in the field and review at that time existing research and commercial systems. From this survey, one can see that until the late 1980s most projects made the following assumptions: (a) The projects studied the integration of data sources within a single organisation, such as the integration of personnel databases from different departments. (b) They aimed at systems that allow read and write access to data sources. (c) It was presupposed that data sources must be completely integrated, i.e., each piece of information stored in a data source must also be accessible from the integrated system. (d) The projects expected that data sources are full-fledged database systems.

These assumptions are not adequate for current applications of data integration. Information system integration nowadays addresses worldwide distributed data sources, crosses organisational borders, and involves data sources that are not accessible through a query language. On the other hand, integrated systems only require read access to only selected parts of data sources, since their goal is to satisfy a certain information requirement.

The major incentive for current projects studying information integration is the ubiquitous use of the World Wide Web (in the following: the web) as data publishing system. Through the web, the number of freely accessible data sources has increased by magnitudes. Due to the chaotic and confusing nature of the web, it is however difficult to dig up this wealth of information. Integration services tackle at this point. For instance, companies such as Junglee[1] integrate job offers taken from organisations web sites; projects such as the HyperView [FS98] aim at providing alerting services for digital libraries by regularly checking publisher's infor-

[1] See http://www.junglee.com

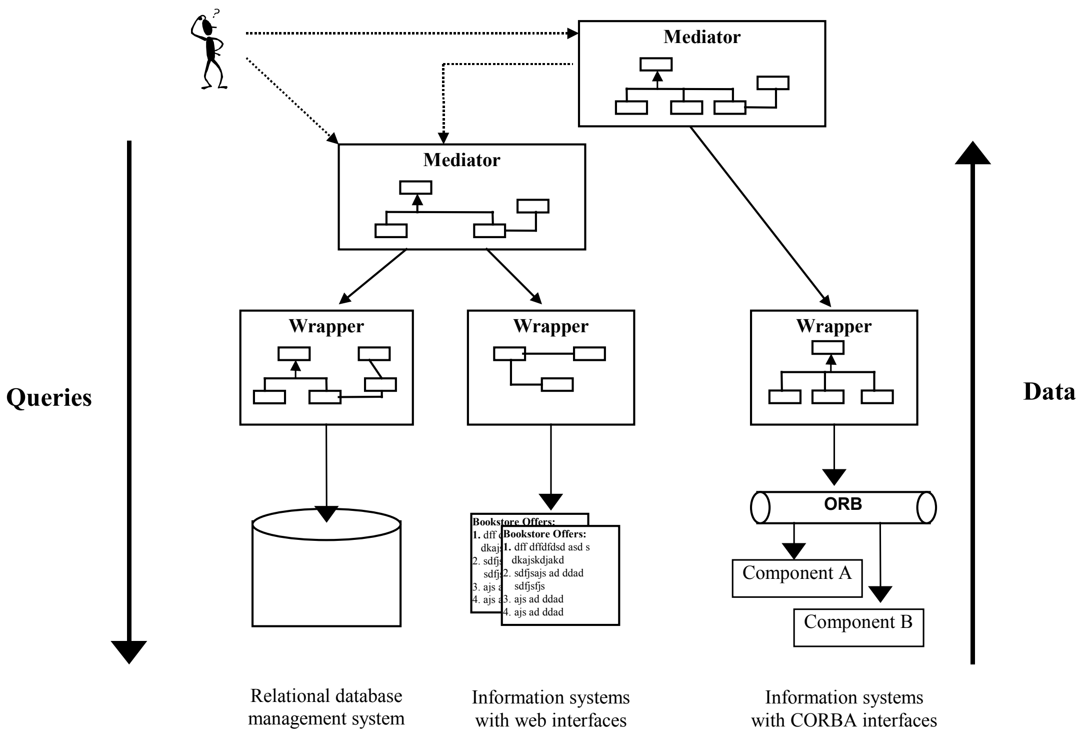


Figure 1. A mediator based information system.

mation systems for new articles or books; integrated databases in molecular biology, such as the Integrated X Chromosome Database [LWG+98], offer homogeneous access to information taken from dozens of different sources; stock-watch services, such as Stockwatch.de[2], update their quotes in minute intervals by parsing web sites of stock markets all over the world. Information integration in those scenarios constitutes a value on its own. Therein, "integration" is a product of high commercial value.

However, information integration is a difficult subject. The pure number of sources and the high diversity in the types of interfaces through which they are accessible impose requirements that call for sophisticated techniques and methods. For instance, *scalability*, i.e., the ability to perform sufficiently with a growing number of data sources, is very important if an integration service wants to keep pace with the growing number of content providers. Another example is *maintainability*, i.e., the ability of provide a stable system in the presence of an continuously changing environment. Especially on the web, changes in data sources are all too frequent and must be compensated. Finally, *flexibility* is vital to cover the manifold of different data sources that are potential candidates for being integrated. For instance, a company's intranet portal probably must access information stored in flat-files, applications with low-level programming interfaces, web pages, or relational database management systems.

In [Wie92], Wiederhold proposed an integration architecture that meets these requirements. In this architecture, integration is accomplished by many small, manageable, and interacting components. The most important such components are *mediators* and *wrappers*. Mediators are domain-specific, i.e., they offer integrated access to data from a certain domain. Therefore, mediators dynamically collect information from wrappers, which encapsulate data sources. A wrapper is source specific. Its task is to present the data and interfaces of its wrapped data source in the form that a mediator requires. A set of cooperating wrappers and mediators constitutes a *mediator based information systems* (see Figure 1).

[2] See http://www.stockwatch.de

This work focuses on query-centric mediator based information systems. In such systems, a mediator administers its own schema and is capable of answering queries against this schema. Users communicate with mediators by posing queries against their schemas. Therein, users are shielded from the properties of underlying data sources. Mediators compute answers to such user queries dynamically, i.e., the data is physically kept and managed in the data sources and only accessed at query time. Typical application scenarios for mediator based information systems are:

- Travel information systems that allow the combined booking of flight- and train tickets, car rental, hotel reservation, excursions, etc.
- Scientific information systems that provide integrated access to experimentally obtained data produced and published all over the world.
- Company-intern information portals that offer a central point of access to corporate knowledge, such as telephone lists, room reservation, knowledge bases, bug reports, etc.
- Web portals that aim to gather under a single interface information for a specific audience, where the information is originally stored and provided by autonomous content providers.
- Shopping assistants and bargain finder on the web that dynamically compare offers from different electronic shops.

To answer a user query, a mediator must translate that query into sequences of remote method executions or remote queries. This task is called query planning. Query planning is not trivial since it must deal with the structural and semantic heterogeneity between the mediator and the autonomous data sources. For instance, query planning encompasses the translation of names, the transformation of values, and the combination of information obtained from different sources. Query planning is impossible without powerful languages for the specification of the relationships between a mediator and the content and interfaces of data sources.

Query planning in the presence of heterogeneous data sources with restricted query capabilities is one of the main current research areas in information integration. The present thesis is devoted to this topic.

1.1 Motivating Example: Data Integration in Molecular Biology

Bioengineering is considered as one of the key technologies of the 21st century. Advances in bioengineering are mainly based on research in molecular biology and aim, for instance, at the identification of the causes of hereditary diseases or at the design of new drugs.

The principal object of analysis in molecular biology are DNA sequences (see Figure 2) and protein sequences. The goal of the *human genome project* (HGP), which started in the mid 1980s, is it to establish the complete sequence of the human genome. The HGP is a highly distributed research project in which laboratories, companies, and national science organisation from all over the world participate [Fre91]. The main work carried out in the HGP is experimental in nature [Pri96]. However, computers, and in particular databases, play an increasingly important role. Researchers use computers to store experimental results, to derive conclusions from experimental results, and to exchange data. Robbins, a former program director of the US' national science foundation, puts it this way:

> *"If the informatics is not handled well, the HGI [human genome initiative] could spend billions of dollars and researchers might still find it easier to obtain data by repeating experiments than by querying the database. If this happens, someone blew it."* (taken from [Fre91]).

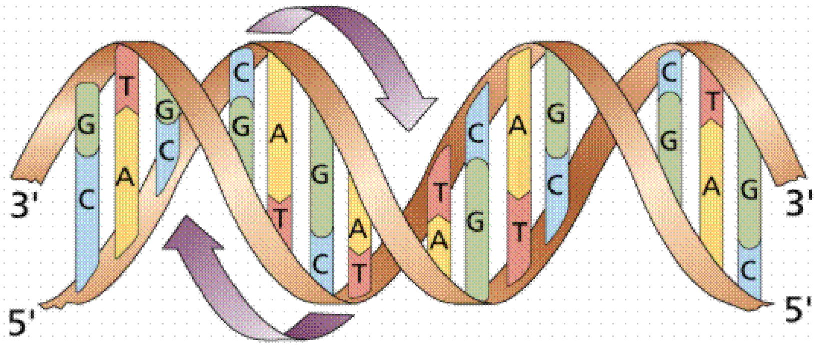


Figure 2. A string of DNA having the famous double-helix structure.

The different groups participating in the HGP mostly work un-coordinated and in competition for funding and academic honours. Apart from leading to a potential duplication of work, the lack of coordination triggers a complex problem for everybody who tries to integrate data from different sources: A large number of data sources has to be considered – a current list contains more than 500 entries [DBBV00] – and all these data sources are heterogeneous wrt. their access interface, syntax, and semantics. Furthermore, data sources have a large degree of *intensional overlap* (see Figure 3) due to overlaps in the research goals. For a class of objects there are very likely many sources containing related information.

Today, data integration in molecular biology is considered as "the most pressing problem" [Rob95] in the field of genome research. There are a number of projects working on it. A survey of ongoing work may, for instance, be found in the proceedings of the workshops on "Interconnecting Molecular Biology Databases", edited by Peter Karp [Karp94; Karp95c]. Most of the projects concentrate on a small fraction of the domain, and physically integrate data into a single database [LLRC98]. Since integrated databases in turn also act as data source they add the problem of *extensional overlap*.

In the following list, we describe various aspects of the data integration problem in molecular biology and point out the solutions we present in this thesis:

- The requirement is to provide *comfortable access* to all or most available information in the field.
  Our solution provides full location, language and schema transparency for users.

- The data *physically* resides on computers distributed all over the world.
  Our solution integrates data at query time and does not depend on data replication.

- Data sources are *heterogeneous* in terms of the access mechanisms they offer, the schemas they use to describe their data, the meaning they give to schema elements, and the format in which data is eventually provided.
  Our solution assigns the treatment of different forms of heterogeneity to different components, separating technical and syntactic issues from semantic and structural problems. We concentrate on the latter.

- Data sources are *intensionally and extensionally overlapping*.
  Our research focuses on intensional overlap. Intension is represented in schemas; extension is represented in instances. Only schemas are usually small enough to make a detailed analysis feasible.

- Data in different data sources may be *inconsistent*.
  Extensional overlap between data sources may lead to inconsistent data values. We shall not try to remove or resolve inconsistencies. Instead, we collect all available values for a specific piece of information.

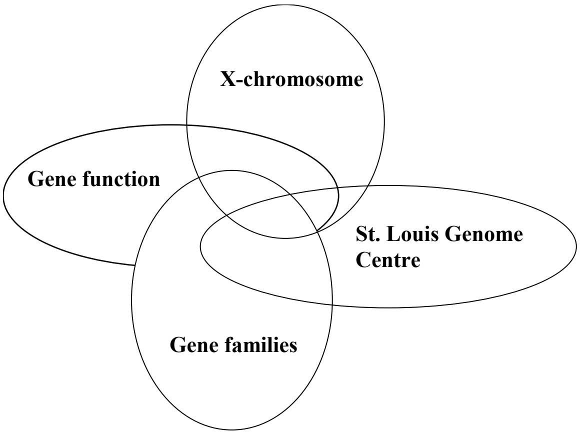


Figure 3. Intensional overlap between data sources. Each oval represents a data source.

- Data sources evolve *frequently and independently*.
  Our solution is highly flexible wrt. change. This flexibility is achieved through a loose coupling combined with declarative descriptions.

The approach to data integration we develop in this thesis is by no means restricted to molecular biology. Instead, it is completely domain independent. However, the motivation for its development was largely taken from experienced or published problems in integrating molecular biology data sources, as reported by Leser et al. [LLRC98] and Robbins [Rob92].

1.2 Objective

We study the problem of query planning in mediator based information systems. Query planning is the task of computing answer to global queries in systems that integrate autonomous, heterogeneous, and distributed data sources. We consider query planning as an abstraction of the data integration problem found in many domains, such as molecular biology. We develop solutions for the two major challenges entailed by query planning:

- One challenge emerges from the conflict between the requirement for a high degree of transparency for a user, and the existing heterogeneity between data sources. Query processing must hide heterogeneity, which, for instance, requires source selection, query decomposition and translation, and result integration. Methods for query processing must pay special attention to the highly dynamic nature of many data sources.
  Our solution for this problem is based on a declarative rule language for the description of data sources, called query correspondence assertions. We provide an algorithm that efficiently answers queries using such rules.

- The other challenge is simply performance. Performance has two components: The amount of time necessary for query planning inside a mediator, and the time necessary to receive answers from data sources. The query planning time benefits from efficient algorithms; the data collection time benefits from avoiding remote queries as much as possible.
  To achieve sufficient performance for query answering, we present an efficient algorithm for query planning, the improved bucket algorithm. Furthermore, we investigate methods to reduce remote query executions based on multiple query optimisation.

We only consider data sources that are accessible for computer programs, for instance through the World Wide Web. We use the word "data sources" to abstract from the hardware, protocol, program, etc. that grants the access.

1.3 Context

This section gives an overview of research in the area of database integration and summarise our contributions. Technical discussions of related work may be found at the end of each chapter. Since the number of publications in database integration is enormous, we cannot claim to be complete. Interested readers are referred to: [BE95; Kim95; Con97; Hull97; Ull97; BBE98; PS98].

1.3.1 Information Integration

Information integration is tackled by a number of different research communities. Examples include cooperative information systems [DDJ+98], agent technology [KB98], federated databases [SL90], broker and trader architectures [JP99], intelligent information integration [AHK+95], and mediator based systems [Wie92].

The integration architecture we apply is mediator based. In the following, we briefly characterise several approaches to information integration. Our goal is to highlight the distinguishing properties of mediator based information systems.

Federated information systems.
We use the term "*federated information systems*" (FIS) to denote all systems that provide integrated access to a set of heterogeneous, autonomous and distributed data sources, where "integrated access" only means the provision of a single point of access. The architecture of a FIS is depicted in Figure 4. Applications and users access a set of heterogeneous data sources through a uniform access layer. This layer could offer a federated schema, a uniform query language, a uniform set of source and content descriptions as metadata sets, etc.

Distributed databases.
Distributed databases [OV99] are FIS where all data sources are database systems with only little autonomy. The distribution of data over sources is determined by design, for instance to achieve high availability or good performance.

In contrast, the types of FIS we are considering consist of autonomous data sources that evolve independently. Data distribution is not centrally organised but inherently chaotic.

Federated database systems.
Federated database systems (FDBS) [SL90] also assume that data sources are database systems. They allow for a higher degree of source autonomy than distributed databases. *Tightly coupled FDBS* provide access to their data sources through a single, unified schema. The process of creating this schema, called schema integration, is the focus of research in FDBS [NS96]. Typically, the federated schema is created such that it completely covers the export schemas of all data sources.

FDBS fail in two points wrt. the requirements described in Section 1.1. First, we mostly assume data sources to be non-database systems. Second, we shall see that schema integration does not offer sufficient flexibility if sources evolve independently.

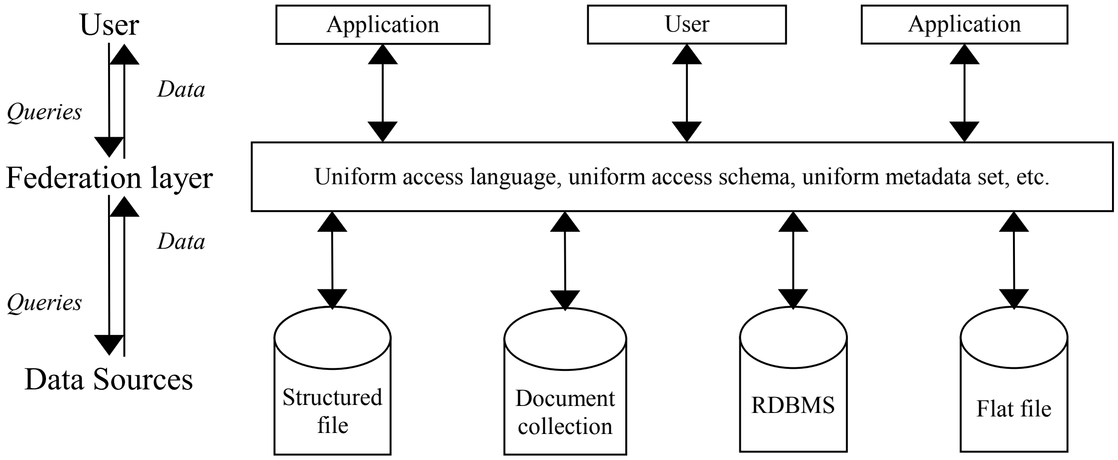


Figure 4. Architecture of a federated information system.

Mediator based information systems.

In the last years, *mediator based information systems* (MBIS) have been applied successfully in many domains [Wie92]. The two main types of components of a MBIS are mediators and wrappers: *Wrappers* encapsulate data sources to provide access in a predefined manner. A *mediator* accepts queries against its *mediator schema* and computes answers by sending appropriate queries to wrappers. MBIS provide full location, language and schema transparency for users. In contrast to FDBS, mediator schemas are not derived from export schemas but designed independently. The main advantage is greater flexibility in the presence of change; the main disadvantage is a higher degree of heterogeneity requiring more complex query translation methods.

We only deal with structured MBIS, i.e., we do not consider semistructured data sources. This implies that all wrappers provide the content of their wrapped data sources according to an explicitly and previously defined schema, the *wrapper schema*.

1.3.2 Schema Correspondences

The main task of a mediator inside a MBIS is to answer queries against its schema by using only queries executable by wrappers. Finding sequences of such wrapper queries is called *query planning*. Query planning relies on knowledge about the relationships between elements of the mediator schema and elements of the wrapper schemas. Such relationships are called *schema correspondences* and are expressed using a *correspondence specification language* (CSL).

Previously developed CSLs either use the "*Local-as-View*" (LaV) or the "*Global-as-View*" (GaV) approach [Hull97]. LaV languages only admit a correspondence to connect a single relation of a wrapper schema with a query on the mediator schema, i.e., each relation of a wrapper schema (the "local" component) is defined as a view on the mediator schema. GaV languages only admit correspondences that connect a single relation of the mediator schema (the "global" component) with a query on a wrapper schema, i.e., they define each relation of a mediator schema as one or more views on wrapper schemas.

1.3.3 Query Planning

Query planning in MBIS is different from conventional query planning in relational database management systems (RDBMS) due to the heterogeneity involved. In RDBMS, a query is posed against a single schema and "directly" executed by appropriately combing the data stored in the relations of that schema. In MBIS, a query is formulated in terms of the mediator schema and must be *translated* into sets of queries against wrapper schemas before execution. The higher the degree of heterogeneity between those schemas, the more complex is query planning.

Query planning in RDBMS.
In classical database optimisation, a plan is a specific way to execute a given query, determining for instance the order of join executions and the selection of appropriate join algorithms. The results of all plans are identical and only differ in performance. Cost-based query optimisation selects the best plan based on cost criteria, and all other plans are discarded (see [SAC+79; Cha98]).

Query planning in MBIS.
Mediators cannot directly execute a user query, but must first rewrite it into sequences of queries against wrapper schemas. In our framework, a *plan* is such a sequences of executable wrapper queries. A plan is *correct* if it only produces correct answers, where the correctness of an answer is essentially determined through the previously defined schema correspondences. Since different correct plans differ in the wrapper queries they use, it follows that different plans produce different results. If more than one correct plan exists, the answer to the user query is defined as the union of the results of all correct plans.

Rewriting queries into sets of correct plans is the focus of our work. To decide upon the correctness of a plan we need a description of the result of each wrapper query in that plan in terms of the mediator schema. We shall define a plan as correct for a user query if the expansion of the plan is *contained* in the user query, where the expansion of a plan is the conjunction of the descriptions of its wrapper queries in terms of the mediator schema.

Answering queries using views.
The complexity of query planning in MBIS correlates to the type of CSL that is used. Query processing in GaV languages is similar to view expansion in SQL; in contrast, query processing in LaV languages is more complex. This thesis mainly deals with the problems of query planning imposed by the LaV approach. To give an intuitive understanding of the nature of those problems, we draw an analogy.

Imagine a data warehouse consisting of a set of materialised views against one operational source database D [Wid95]. The views are updated regularly; their extension is stored in the warehouse, and their definition, i.e., the queries they perform on D, are known. Since there is only one source, all view definitions address the same schema. Now, imagine that D has a fatal crash and is destroyed. Can we still answer a query against the schema of the destroyed D that is not identical to any of the views? Figure 5 illustrates this situation. A user query u addresses arbitrary relations of the schema of D, but at query time, only a set materialised views, v_1, v_2, and v_3, is available.

This problem is known as "*answering queries using only views*" [LMSS95]. Basically, the answer is "yes", if both the view definitions and the query are conjunctive queries. The general idea to answer a query using only views is the following: We enumerate all conjunctions of view definitions and insert appropriate joins. For each conjunction we check whether it is equivalent to the query by testing *query containment* in both directions. Query containment is tested by searching a *containment mapping* [ASU79b], i.e., a mapping from the symbols of

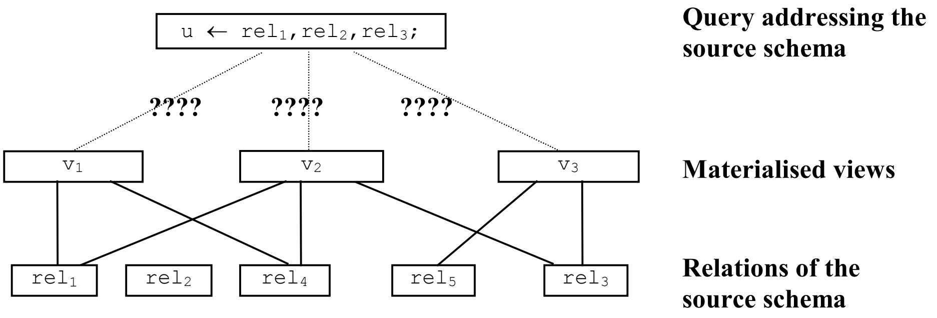


Figure 5. Answering a query using materialised views.

one query onto the symbols of the other query that fulfils certain properties. If a containment mapping exists, the problem is solved and the query can be answered; if no containment mapping exists, the query cannot be answered using only the views.

The analogy to information integration is the following: Consider the materialised views to be spread over different data sources. A mediator schema takes the role of the schema of `D` – although an instance of this schema never existed. We only *model* elements of wrapper schemas as views on `D`. We can then essentially use the same procedure as described above to answer queries against the mediator schema.

Describing data sources as views on a global schema and answering queries against this schema by finding appropriate view combinations was first suggested by Tsatalos et al. in [TSI94]. This technique is fundamental for our work.

1.4 Contributions

We make three main contributions:

- A new correspondence specification language, called *query correspondence assertions* (QCAs), which is more powerful and more flexible than previous approaches.
- An improved algorithm for query planning in MBIS. The best known algorithm so far has complexity $O((n+k)^k)$, where n is the number of views and k is the length of the query to be planned. In contrast, our *improved bucket algorithm* has complexity $O(n^k)$. We also give an average case analysis of several algorithms for query planning and query containment. Such an analysis was, to our best knowledge, not accomplished before.
- We suggest a post-processing for query plans based on *multiple query optimisation*. This processing dramatically reduces the number of queries that have to be shipped to data sources to compute the answer to a user query.

QCAs.
The correspondence specification language we propose and use in this thesis is *query correspondence assertions* (QCAs). QCAs are a combination of the GaV and LaV approach, i.e., QCAs allow correspondences to relate queries against the mediator schema to queries against a wrapper schema. We shall give examples showing that both the LaV and the GaV approach are not able to describe situations that are expressible through QCAs. Query planning with QCAs is in first place identical to query planning in LaV. However, QCAs also allow for efficient multiple query optimisation.

To place our query planning algorithms on solid ground we formally define the *semantics for user queries* in MBIS based on QCAs. Surprisingly, this fundamental issue is disregarded or only fuzzily described in many projects, which impedes their comparability [Mot95]. The challenge in defining a proper semantics for global queries lies in the possible occurrence of inconsistency between data sources, which renders a valuation-based approach as in central databases impossible [GM99].

Improved bucket algorithm.
We in detail discuss ways to prove query containment, including worst-case and average-case analysis. We highlight the difference between query containment and query planning in MBIS by showing that query planning sometimes requires modifications of plans. The necessity to consider plan transformations was mentioned in several publications before but never analysed in detail. We prove that it does not increase the complexity of query planning, although it does add considerable difficulties to the algorithms.

We describe and analyse two algorithms for query planning. The *generate & test algorithm* (GTA) is easy to capture and therefore more suitable to prove properties of query planning. The *improved bucket algorithm* (IBA) is more complicated but at the same time considerably more efficient, as revealed by a detailed complexity analysis. Its main achievement, compared to the GTA, is a better exploitation of the problem structure, which helps to avoid redundant computation. We formally prove soundness and completeness of both query planning algorithms wrt. our semantics of global queries.

Multiple query optimisation.
Query planning algorithms potentially generate a large number of plans to answer a user query. However, those plans are often redundant or share subplans that only need to be executed once. We approach this problem through *multiple query optimisation* [Jar85; SSN94]. We give three methods for the treatment of redundancy in or between query plans: (a) We show that redundant query plans can sometimes be removed entirely. (b) We describe a linear algorithm for the detection of identical subplans. (c) We also include a more complex algorithm for the detection of subsumed subplans. Although essentially all published query planning algorithms based on LaV correspondences face the problem of redundancy, no other project has yet investigated it.

1.5 Structure of the Thesis

This thesis is structured as follows:

- Chapter 2 defines basic concepts such as queries and schemas and introduces query containment. We discuss three algorithms for testing query containment, analysing two of them in detail.
- Chapter 3 discusses various approaches to information integration and characterises MBIS. We define formal abstractions of mediators and wrappers to facilitate our discussion of query planning. Finally, we introduce correspondence specification languages and show where current languages fail.
- Chapter 4 introduces query correspondence assertions and defines their syntax. We also give a formal semantics for user queries in MBIS based on QCAs.
- Chapter 5 discusses query planning algorithms. We start by showing that query planning indeed implements the previously defined semantics, and prove that we only have to consider a finite number of query plans. We then describe and analyse in detail the generate & test algorithm and the improved bucket algorithm. The detection and removal of different forms of redundancy is handled in a separate section.

- Chapter 6 concentrates on methodological issues of MBIS. We discuss issues of the construction of wrappers, focussing on the derivation of wrapper schemas, and show the flexibility and power of QCAs in bridging heterogeneity in schemas. Finally, we argue that the usage of QCAs leads to particularly robust systems by separately investigating the consequences of different types of changes in MBIS.
- Chapter 7 summarises our contributions and gives future research directions.

1.6 Notation and Terminology

Terminology.
A precise characterisation of MBIS shall be given in Section 3.3. In the meantime, we define some necessary terminology (see Figure 1 for illustration):

- In this thesis, the word "*user*" refers to any type of client of a mediator, may it be an applications, another mediator or a human being.
- The word "*data source*" refers to any type of system that a mediator uses to obtain data. Data sources may, for instance, be databases, web based information systems, and files. We assume that data sources are remote wrt. the mediator, i.e., reside on a different host, but are accessible through a network.
- We only deal with structured MBIS, i.e., we assume a mediator to have a schema, the *mediator schema*. We use the term "*global schema*" for the pendant of mediator schemas in FIS that are not MBIS. Furthermore, each wrapper has a schema, called *wrapper schema*.
- The queries a user poses against a mediator schema are called *user queries*. Answering user queries is the goal of a mediator.
- The queries that are executed by wrappers are called *wrapper queries*. Mediators answer user queries by collecting data through the execution of wrapper queries.
- Queries against mediator schemas that are used to describe wrapper queries are called *mediator queries* (see Chapter 4). A mediator uses mediator queries to rewrite user queries into sets of wrapper queries.

Notation.
We make the following syntactical conventions:

- Small case letter variables stand for singular things, upper case letters for sets: "v" is a variable, "V" a set of variables, "q" is a query, "Q" a set of queries, etc.
- In definitions and proofs, "v" in general stands for a variable, "c" for a constant, and "s" for a symbol, which may be either a variable or a constant.
- We use "$\mathtt{rel}_i$" for relation names, and "a", "b" .. and "x", "y" ... for symbols, "l" for literals, "q" for queries, "p" for plans, and "h" for mappings.
- We use "r" (rule) for query correspondence assertions.
- We use DATALOG notation for queries.
- We shall often construct queries by concatenating other queries. In such cases, we use the delimiters "<" and ">" to separate a constructed query from the rest of a term. For instance, "$\mathtt{cond(<}q_1\mathtt{,...,}q_n\mathtt{>,V)}$" denotes the function cond with two parameters. The first is a query constructed as the conjunction of the queries q_1, ..., q_n and the second is a set of variables.
- In queries, we use variable names that abbreviate attribute names, such as "gn" for "genename" or "cl" for "clonelength".

Algorithms.

The analysis of algorithms is an essential part of this thesis. In particular, we often want to prove that an algorithm is *sound and complete* for a given problem, and we want to derive the *complexity* of algorithms. However, soundness and complexity is difficult to prove for implementation–near algorithms, while a detailed complexity analysis requires some level of detail.

To escape this problem, we distinguish between two types of algorithms. The first type is a *textual description* of the solution for a problem. Algorithm 1 (testing query containment), Algorithm 6 (the GTA for query planning), Algorithm 8 (the IBA for query planning), and Algorithm 11 (testing replaceability of query plans) are of this type. For those algorithms we prove soundness and completeness.

The second type are implementations of algorithms of the first type, and are given in pseudo code. For instance, Algorithm 2 implements Algorithm 1, Algorithm 5 together with Algorithm 7 and Algorithm 2 implement Algorithm 6, and Algorithm 10 implements Algorithm 8. For those algorithms we carry out a detailed worst-case and average-case analysis, and derive their complexity.

We therein do not prove the complexity of algorithm of the first type, i.e., the complexity of a problem. For instance, we show that Algorithm 1 (page 28) is sound and complete for the query containment problem for certain queries. We then device two algorithms (Algorithm 2 and Algorithm 3) that implement Algorithm 1 and show that both are exponential in the length of the queries.. This does *not imply* that no implementation of Algorithm 1 exists that has a lower complexity.

2. QUERIES AND QUERY CONTAINMENT

This chapter provides technical background. In Section 2.1 we define essential notations regarding the relational data model and conjunctive queries. In Section 2.2 we introduce query containment and query equivalence. Both concepts are fundamental for this thesis. We provide extensive examples to easy understanding of their properties and peculiarities. Furthermore, we introduce containment mappings as the main tool for proving query containment.

Section 2.3 describes three algorithms for testing query containment. The first two build on a decomposition of the query containment problem into a set of literal containment problems. A containment mapping from one query into another query is composed from containment mappings from the literals of the first query into literals of the second query. We prove soundness and completeness of this approach in Section 2.3.1.

Finding proper combinations of containment mappings between literals is a search problem. We characterise the search space in Section 2.3.2. We propose two algorithms, which differ in how they traverse the search space: one algorithm performs a breadth-first search (Section 2.3.3), the other algorithm performs a depth-first search (Section 2.3.4). Both algorithms are analysed wrt. their average-case and worst-case behaviour and their complexity. Section 2.3.5 presents a comparison of both algorithms based on simulations.

In Section 2.3.6 we describe a third method for proving query containment that is not based on containment mappings. We include this method because it opens interesting perspectives on the nature of the containment problem. Many of the results mentioned in Section 2.4 for queries that extend conjunctive queries base on this method.

2.1 Conjunctive Queries

We use the relational data model throughout this thesis. Queries are given in a logical notation. In the following, we define basic terms such as schema, relation, arity of a relation, conjunctive queries, etc. Readers familiar with the relational data model may skip this section.

Definition (D2.1)-(D2.2) (Schema, instance, database, relation).

(D2.1) Let var be a set of variable symbols, const be a set of constant symbols, $\mathtt{rel}_E$ be a set of relation symbols, and att be a set of attribute symbols. var and const must be disjoint.

- A *schema* Σ is a finite set of *relations* from $\mathtt{rel}_E$ where every relation has a finite set of *attributes* from att.
- The *size* of Σ is the number of relations it contains, written $|\Sigma|$.

- The *arity* of a relation `rel` is the number of attributes of `rel`, written `arity(rel)`.
- An *instance* of a schema Σ, written I^{Σ}, is a finite set of tuples for the relations of Σ, where each value of each tuple is taken from `const`.
- A *database* `D` is a tuple containing a schema and an instance of this schema, written as $D=(\Sigma, I^{\Sigma})$.
- A *literal* `l` is an expression of the form $rel(s_1, ..., s_n)$, where `rel` is a n-ary relation symbol and $s_i \in (var \cup const)$. We say that `rel` is the relation of `l`.

(D2.2) Let $D=(\Sigma, I^{\Sigma})$, and let $rel \in \Sigma$.

- The *extension* of `rel` in `D` is the set of tuples for `rel` contained in I^{Σ}, written as $I^{\Sigma}|_{rel}$.
- The *intension* of `rel` in `D` is the set of all possible tuples corresponding to real-world objects represented by `rel`.

■

Remark:
The intension of a relation is the *set of all possible objects* that are intended to be stored in this relation. This is essentially an intuitive concept in the head of the schema designer, which may be infinite, fuzzily described, or may be without any strict borders. We do not try to formalise this concept. We usually represent the intension of a relation `rel` by the name "`rel`" itself, in the sense that `rel` is a name for a real-world concept, i.e., the intension of `rel` [Web82]. This implies that two relations with the same name in different schemas are in first place assumed to have the same intension, if not specified otherwise. In contrast, the extension of a relation depends on the actual instance of a database and is always a finite set of tuples.

■

In the following, we define different classes of queries: The class of *conjunctive queries without conditions* (CQ, for conjunctive query), the class of conjunctive queries with only *simple conditions* (CQ_S), and the class of queries with *complex conditions* (CQ_C). These classes are defined such that $CQ \subset CQ_S \subset CQ_C$.

The reason for distinguishing different classes of conjunctive queries is that the problems we study in this work require different algorithms for different types of queries. For instance, the existence of a containment mapping is a sufficient and necessary conditions to show that a CQ_S query is contained another CQ_S queries, but it is only sufficient – and not necessary – for CQ_C queries (see Section 2.3).

Definition (D2.3)-(D2.6) (Conjunctive queries, simple and complex conditions).
Let Σ be a schema. Let rel_Q be a set of *query symbols* disjoint from `var`, `const`, rel_E and `att`.

(D2.3) The language CQ^{Σ} comprises all *conjunctive queries* of the form:

$$q(v_1, ..., v_k) \leftarrow l_1, l_2, ..., l_n;$$

where:

- $q \in rel_Q$.
- The l_i are literals of relations in Σ.

- $\mathtt{v_1,...,v_k}$ are called *exported variables* of $\mathtt{q}$.
- $\mathtt{q}$ must be *safe*, i.e., all exported variables must appear in at least one literal of $\mathtt{q}$.

(D2.4) The language $\mathtt{CQ}_\mathtt{C}^\Sigma$ comprises all conjunctive queries of the form:

$$\mathtt{q(v_1,...,v_k)} \leftarrow \mathtt{l_1,l_2,...,l_n,c_1,...,c_m;}$$

where $\mathtt{q}$ without the $\mathtt{c}$'s is a $\mathtt{CQ}^\Sigma$ query and each $\mathtt{c}$ is a *condition* of the form "$\mathtt{s_1\ op\ s_2}$" where:

- $\mathtt{s_1} \in \mathtt{var}$ and $s_2 \in \mathtt{const} \cup \mathtt{var}$.
- Any variable in a condition must be *bound*, i.e., the variable must also appear in a literal of $\mathtt{q}$.
- The set of allowed operations is: $\mathtt{op} \in \{ \text{'<'}, \text{'≤'}, \text{'>'}, \text{'≥'}, \text{'='} \}$.
- A condition "$\mathtt{s_1} = \mathtt{s_2}$", where both operands are variables or both operands are constants, is not allowed (see below).

(D2.5) The language $\mathtt{CQ}_\mathtt{S}^\Sigma$ is the subset from $\mathtt{CQ}_\mathtt{C}^\Sigma$ where no condition contains two variables.

(D2.6) Let $\mathtt{q}$ be a query:

- $\mathtt{export(q)}$ returns the set of exported variables of $\mathtt{q}$.
- The *size* of $\mathtt{q}$, written as $\mathtt{|q|}$, is the number of literals in $\mathtt{q}$.
- $\mathtt{variables(q)}$ returns the set of all variable symbols occurring in $\mathtt{q}$.
- $\mathtt{constants(q)}$ returns the set of all constants occurring in $\mathtt{q}$.
- $\mathtt{sym(q)} = \mathtt{variables(q)} \cup \mathtt{constants(q)}$, i.e., the set of all symbols occurring in $\mathtt{q}$.
- $\mathtt{cond(q)}$ denotes the conjunction of all conditions $\mathtt{c_1,...,c_m}$ of $\mathtt{q}$.
- $\mathtt{cond(q,v)}, \mathtt{v} \in \mathtt{variables(q)}$, is the conjunction of all conditions of $\mathtt{q}$ that contain no other variable than $\mathtt{v}$. $\mathtt{cond}$ may be extended to sets of variables.
- The *length* of a set $\mathtt{C}$ of conditions, written $\mathtt{|C|}$, is the number of conjuncts in $\mathtt{C}$.

■

Remarks:

- The "s" in $\mathtt{CQ}_\mathtt{S}^\Sigma$ is an abbreviation for "simple conditions", the "c" in $\mathtt{CQ}_\mathtt{C}^\Sigma$ for "complex conditions".
- The focus of this work is on $\mathtt{CQ_S}$ queries. Many problems for which we find satisfying solutions for queries with simple conditions are considerably harder if complex conditions are allowed. For instance, the method we shall use for proving query containment is not complete for $\mathtt{CQ_C}$ queries but for $\mathtt{CQ_S}$ queries (see Section 2.3). The semantics of global queries in MBIS we shall define in Section 4.3 is only applicable for $\mathtt{CQ_S}$ queries, but not for $\mathtt{QC_C}$ queries. Finally, the query planning algorithms we shall present in Chapter 5 are sound and complete for $\mathtt{CQ_S}$ queries, and sound but not complete for $\mathtt{CQ_C}$ queries.
- We do not distinguish between the symbol of the head predicate of a query and the query itself.
- $\mathtt{rel_Q}$ and $\mathtt{rel_E}$ correspond to EDB – *extensional* - and IDB – *intensional* – predicates in DATALOG.
- The expression before the "←" is the *head* of $\mathtt{q}$, the expression after "←" is the *body* of $\mathtt{q}$.

- We do not require that all variables in the body of a query are distinguished. A variable appearing more than once implies an equi-join between the connected literals. On the other hand, equalities are not allowed as conditions. Throughout this work, equi-joins are treated separately from other conditions, although an equi join is in fact a condition.
- We exclude the case that both operands of a condition are constants since this either evaluates to `true` - then it can be ignored - or to `false` - then the query always computes the empty result, independent of the database content.
- As abbreviation, we use the symbol “-“ instead of a variable symbol in the body of a query, indicating that the value of this attribute is irrelevant: it is not exported, it is not constrained by any condition, and it is not used for a join. Every “-“ in a query can be replaced by a fresh variable name without affecting the semantics of the query.
- If the schema is clear from the context, we omit the superscript “$^{\Sigma}$”.
- All functions for queries can be restricted to subqueries or single literals.

■

The current syntax allows semantically identical `CQ` queries to be written in different forms. For instance, the query:

```
q(a,b) ← rel(a,b,c,d),c=”thisthing”,d=5;
```

is equivalent to:

```
q’(a,b) ← rel(a,b,”thisthing”,5);
```

In some cases, we need a unique way of writing down queries. Therefore, we define two forms of queries.

Definition (D2.7)-(D2.8) (Embedded and normal form of queries).
Let Σ be a schema and $q \in CQ_c^{\Sigma}$.

(D2.7) q is in *normal form* if q does not contain a ”-“ and no literal contains a constant.

(D2.8) q is in *embedded form* if q does not contain a ”-“ and no condition is of the form “v = c”, where $v \in \text{variables}(q)$ and $c \in \text{const}$.

■

Remarks:

- Any conjunctive query (`CQ`) is in embedded form.
- Any query in embedded form can be transformed into normal form. The reverse is not true since it might require “exported constants”. In this sense, CQ_C is more powerful than `CQ` even if only equalities are allowed as conditions.

■

So far, we only defined the syntax of queries. Now, we define what it means to *execute* a query against a given database. This definition follows the standard definition [AHV95] using inner-join semantics.

Definition (D2.9)-(D2.12) (Valuation, extension and intension of a query).
Let $D = (\Sigma, I^{\Sigma})$ and $\Sigma = (rel_1, ..., rel_k)$. Let $S \subseteq \text{const} \times \text{var}$.

(D2.9) A *valuation* v for S is a function:

$$v: \text{sym}(S) \mapsto \text{const}$$

v must be the identity on constants and is extended to queries in the natural fashion.

(D2.10) The *extension* of a query $\mathtt{q} \in \mathtt{CQ}_{\mathtt{c}}^{\Sigma}$ in $\mathtt{D}$, written $\mathtt{q(D)}$ is:

$$\begin{aligned} \mathtt{q(D)} = \{\ \mathtt{v(export(q))}\ |\ & \\ & \mathtt{v} \text{ is a valuation over } \mathtt{sym(q)}\ \wedge \\ & \forall\, \mathtt{l} \in \mathtt{q}\colon \mathtt{v(l)} \in \mathtt{I}^{\Sigma}|_{\mathtt{l}}\ \wedge \\ & \mathtt{v(cond(q))} \Rightarrow \mathtt{true}\}; \end{aligned}$$

(D2.11) The *intension* of a query $\mathtt{q} \in \mathtt{CQ}_{\mathtt{c}}^{\Sigma}$ is the union of the extensions of q in every possible database for Σ.

(D2.12) Let $\mathtt{V} \subseteq \mathtt{export(q)}$. Then $\mathtt{q}|_{\mathtt{V}}\mathtt{(D)}$ denotes the extension of q in D restricted to the variables in V.

■

Remark:
Intuitively, the intension of a query q is the set of all tuples corresponding to real-world objects in any database for Σ that fulfil the conditions of q. Using this definition, we can derive and prove statements about the relationship of intensions of different queries. For instance, the intension of the query "$\mathtt{q_1(x)} \leftarrow \mathtt{rel(x,y),y<100}$" is provably a subset of the intension of the query "$\mathtt{q_2(x)} \leftarrow \mathtt{rel(x,y)}$" since any real-world object that is returned by $\mathtt{q_1}$ must also be returned by $\mathtt{q_2}$. The intension of a query is the set of all possible answers; the extension of a query is the set of all answers that are computed in a given database.

■

Example 2.1.
Figure 6 is an entity-relationship diagram for genome mapping. We briefly explain the fundamentals of chromosomal mapping, since examples of this domain will be used throughout this work. The interested reader is referred to [LRTL93; Pri96; LLRC98].

The human genome consists of 24 chromosomes. A chromosome is a string over a four letter alphabet, the DNA. Human chromosomes are between 50 and 250 million base pairs (MB) long. Revealing the "string" of each chromosome is the final goal of the human genome project (see Section 1.1).

However, there is no known technique that could extract a chromosome from a cell and read it from start to end. Instead, chromosomes first have to be *mapped* (see Figure 7). Mapping is a fragile and complex technology. Essentially, it depends on the isolation of chromosome fragments of various lengths, and on subsequent experiments that determine the relative position of those fragments to each other.

One type of such fragments are clones. A clone is a continuous piece of DNA that is somehow isolated from a cell and then implanted into an organism, for instance in a bacteria, for proliferation. Clones are classified depending on the organisms that are used to host them, such as yeast for YACs (*yeast artificial chromosomes*) or bacteria for BACs (*bacterial artificial chromosomes*). The type of a clone correlates with its length: YACs are between 200 kilo bases (KB) and 2 MB long, BAC between 50 and 150 KB, etc. Clones are extracted at random positions of a chromosome, and related through each other through hybridisation experiments: If two clones hybridise with each other we can conclude that their sequence overlaps.

After a sufficient number of such experiences it is in principle possible to compute a complete map of a chromosome build out of such overlapping clone fragments. Hence, we can assign each clone a fixed position (entity clone and map, connected through clonelocation). However, since hybridisation is a technique of pitiful low accuracy, different sets of

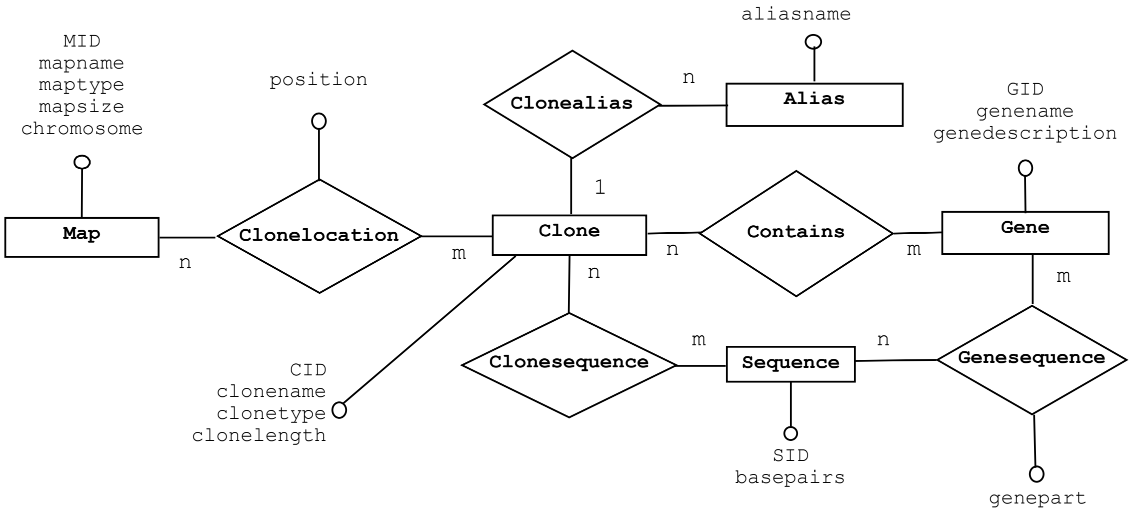


Figure 6. Entity-relationship diagram of a mediator schema.

experiments will lead to different maps. We need a m:n relationship between clones and maps.

Clones are distributed worldwide by shipping collections of them, so called libraries. Unfortunately biologists often rename a clone upon its reception, for instance to be ion conformance with laboratory specific naming conventions. This leads to a high number of `alias`'s for each single clone [LLRC98].

Not the entire chromosome if of interest. Only some parts are actually used by the human organism: the genes. A `gene` is a (not necessarily continuous) fragment of a chromosome that is sometimes during the lifetime of the organism read and, usually, "translated" into a protein. Finding genes is the real goal of most research in molecular biology. Genes are identified through different techniques, which all usually result in a short part of the gene `sequence`. To find the entire sequence, those sequences are compared with `clonesequence`'s. If a match is found, the position and complete sequence of the gene is disclosed through the position and sequence of the clone.

```
map(mid, mapname, maptype, mapsize, chromosome);
clonelocation(mid, cid, position);
clone(cid, clonename, clonetype, clonelength);
clonealias(cid, alias);
contains(cid, gid);
gene(gid, genename, genedescription);
genesequence(gid, sid, genepart);
clonesequence(sid, cid);
sequence(sid, basepairs);
```

Table 1. Exemplary relational schema, corresponding to the schema illustrated in Figure 6.

Table 1 contains a relational schema corresponding to the diagram of Figure 6. The relations `map, clone, sequence` and `gene` contain entities, the other relations represent relationships between entities. Queries against this schema are for instance:

- Give the names of all gene contained in the clone 'yWXD1':
  `genes(gn) ← clone(cid,'yWXD1',-,-),contains(cid,gid),gene(gid,gn,-);`
- Give me all gene parts plus sequence and the appropriate gene name for genes that are contained in a clone smaller than 100 KB, which are placed on a map of chromosome 'X':

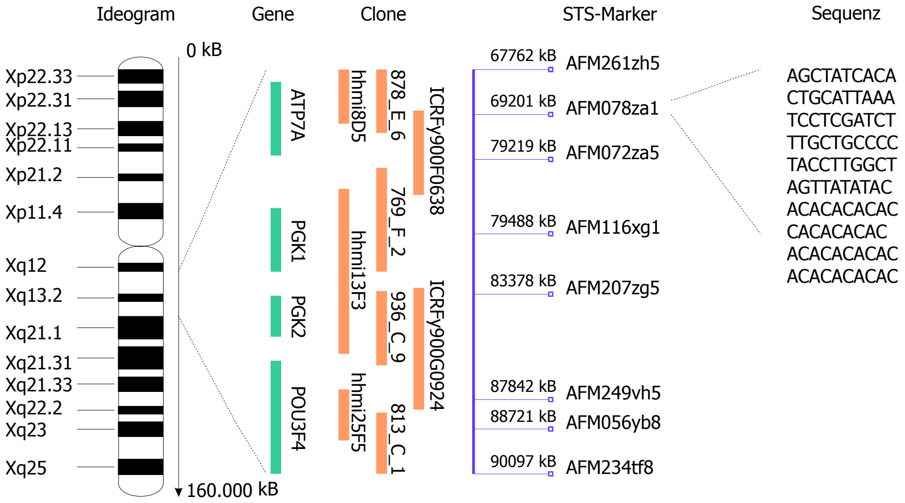


Figure 7. Fraction of a physical map of the human X chromosome.

```
seqs(gn,gp,bp) ←
   map(mid,-,-,-,'X'),clonelocation(mid,cid,-),clone(cid,-,-,cl),
   contains(cid,gid),gene(gid,gn,-),genesequence(gid,sid,gp),
   sequence(sid,bp,-), cl<100;
```

■

2.2 Query Containment and Query Equivalence

In this section we define *query containment* and *query equivalence*, i.e., whether or not the result of one query is contained in or identical to the result of another query. We give necessary and sufficient conditions for query containment. Query containment is the fundamental concept behind the query planning methods described in Chapter 5.

Definition (D2.13)-(D2.14) (Query equivalence and containment).
Let $q_1, q_2 \in CQ_C$ for a schema Σ.

(D2.13) q_1 is *equivalent* to q_2, written $q_1 \equiv q_2$, iff:

$$q_1 \equiv q_2 \;\Leftrightarrow\; \forall\, D = (\Sigma, I^{\Sigma}):\; q_1(D) = q_2(D)$$

(D2.14) q_1 is *contained* in q_2, written $q_1 \subseteq q_2$, iff:

$$q_1 \subseteq q_2 \;\Leftrightarrow\; \forall\, D = (\Sigma, I^{\Sigma}):\; q_1(D) \subseteq q_2(D)$$

■

Theorem (T2.1) (Query equivalence versus query containment).

(T2.1) $q_1 \equiv q_2 \;\Leftrightarrow\; q_1 \supseteq q_2 \wedge q_1 \subseteq q_2$

■

Some authors use the term “semantic query containment” instead of simply “query containment” [LRO96a]. This does not refer to any *intuitive semantics* of relations or queries, but can be justified to emphasise that containment is not primarily a syntactic property, and that it is independent of the actual values in a database. A query that is contained in another query can be syntactically quite different, as we shall see later. In this thesis we use the term "query containment".

Testing query containment does not require to enumerate and check all possible databases. Instead, we can test query containment syntactically using *containment mappings*.

Definition (D2.15)-(D2.16) (Symbol mapping, containment mapping).
Let $q_1, q_2 \in CQ_C$ for a schema Σ, and let q_1, q_2 be in embedded form.

(D2.15) Let $S_1, S_2 \subseteq \mathtt{const} \cup \mathtt{var}$. A mapping $h: S_1 \mapsto S_2$ is a *symbol mapping* from S_1 to S_2 if the following holds:

- h is a function, and
- h is the identify on constants.

S_1 is the *origin* of h, written org(h), and S_2 is the *image* of h, written img(h).

(D2.16) A *containment mapping* h (CM) from q_2 into q_1 is a symbol mapping $h: \mathrm{sym}(q_2) \mapsto \mathrm{sym}(q_1)$ such that:

(CM1) $\forall\, v \in \mathrm{export}(q_2) : h(v) \in \mathrm{export}(q_1)$
(CM2) $\forall\, c \in \mathrm{constants}(q_2) : h(c) = c$
(CM3) $\forall\, l \in q_2 : \exists\, l' \in q_1 : h(l) = l'$
(CM4) $\mathrm{cond}(q_1) \Rightarrow h(\mathrm{cond}(q_2))$

■

Remarks:

- h is the identity on relation symbols if extended to literals as in (CM3).
- Condition (CM2) is already captured by the definition of symbol mapping. However, in the following we need both the definition of symbol mapping and the four conditions for containment mappings. Therefore the redundancy.
- We purposely define a containment mapping from q_2 into q_1, and not vice versa. The reason becomes clear after the following theorem.
- We use the following notation for mappings: $h = [x \rightarrow a, b \rightarrow z]$ is a mapping that maps the symbol x into a and b into z.
- Condition (CM2) is not necessary for queries in normal form, since then no constants appear in the literals of the queries. The semantics remains the same, since constants in normal form queries appear as conditions being captured by (CM4).

■

The following theorems connect containment mappings and query containment.

Theorem (T2.2)-(T2.5) (From containment mapping to query containment).
Let q_1, q_2 be two queries against a schema Σ.

(T2.2) Let $q_1, q_2 \in CQ_S^{\Sigma}$. The existence of a containment mapping from q_2 into q_1 is a *necessary and sufficient* condition for showing $q_1 \subseteq q_2$.

(T2.3) Let $q_1, q_2 \in CQ_C^{\Sigma}$. The existence of a containment mapping from q_2 into q_1 is a *sufficient, but not necessary* condition for showing $q_1 \subseteq q_2$[3].

[3] More precisely: The condition remains necessary if $q_1 \in CQ_C$ and $q_2 \in CQ_S$. Only if $q_2 \notin CQ_S$ the condition is not necessary any more.

(T2.4) Let $q_1, q_2 \in CQ_S^\Sigma$. Testing whether $q_2 \subseteq q_1$ is NP-complete in the size of q_1 and q_2.

(T2.5) Let $q_1, q_2 \in CQ_C^\Sigma$. Testing whether $q_2 \subseteq q_1$ is complete in $\prod_2^p$.

■

Proof:
The proofs can be found in the literature. Theorem (T2.2) and (T2.4) were originally proven by Chandra & Merlin [CM77] and also appear in [ASU79b; Ull89]. Theorem (T2.5) was proven by van der Meyden [vdM92]. A characterisation of the complexity class $\prod_2^p$ may be found in [Pap94].

■

One has to be careful not to confuse the direction of the CM: q_1 is contained in q_2 if there is a containment mapping *from q_2 into q_1*, i.e., from the "more general" query to the "more restricted" query.

We show by counterexample that the existence of a containment mapping is not a necessary condition for a CQ_C^Σ query to be contained in another CQ_C^Σ query.

Example 2.2.
Consider the following two queries:

```
q1() ← rel(a,b),rel(b,a);
q2() ← rel(a,b),a≤b;
```

$q_1 \subseteq q_2$ because for every pair of values `(a,b)` for which both tuples `(a,b)` and `(b,a)` are present in the extension of `rel`, the query "`rel(x,y),x≤y`" succeeds: Either with the valuation `v(x) = a`, `v(y) = b` or with `v(x) = b`, `v(y) = a`. But there exists no containment mapping from q_2 into q_1 since the conditions of q_1 do not imply the conditions of q_2.

■

Complex conditions introduce an interaction between literals and conditions that is not entirely captured through the notion of containment mapping. A modification of the definition of containment mapping for queries with arithmetic comparisons that also contains a necessary condition for query containment may be found in [ZO93].

The existence of a containment mapping `h` from q_2 to q_1 implies the following facts:

- Since `h` is a function, every symbol of q_2 is mapped to exactly one symbol in q_1. The reverse does not hold: the same symbol in q_1 may be the image of several symbols in q_2.
- Wherever there appears a constant in q_2, the same constant appears (at least) at the same position in q_1.
- Every relation that appears in the body of q_2 also appears in the body of q_1. The reverse does not hold: q_1 may contain relations not appearing in q_2.
- Every exported variable of q_2 is mapped to an exported variable of q_1. Therefore, q_1 exports at least all variables that q_2 exports.
- Every symbol in q_2 that is neither a constant nor an exported variable may be mapped to any symbol in q_1.
- There exists at most one CM between two literals (up to variable renaming), but there can exist more than one CM between two queries. Consider the following two queries:

```
q1(a,b,z) ← rel(a,b),rel(z,z);
q2(x,y) ← rel(x,y);
```

$q_1 \subseteq q_2$ can be proven with $h_1 = [x\rightarrow a, y\rightarrow b]$ and with $h_2 = [x\rightarrow z, y\rightarrow z]$.

We formally prove some less obvious properties of query containment.

Lemma (L2.6)-(L2.7) (Transitivity and monotony of query containment).
Let q_1, q_2, $q_3 \in CQ_c^{\Sigma}$. Let $l_1,..,l_n$ be a set of literals for relations from Σ.

(L2.6) Query containment is *monotone*: $q_1 \subseteq q_2 \;\Rightarrow\; <q_1, l_1, ..., l_n> \subseteq q_2$.

(L2.7) Query containment is *transitive*: $q_1 \subseteq q_2 \;\wedge\; q_2 \subseteq q_3 \;\Rightarrow\; q_1 \subseteq q_3$.

■

Proof:

- (L2.6): Let $q_1' = <q_1, l_1, ..., l_n>$, and let h be the CM from q_2 into q_1. h is also a CM from q_2 into q_1' using only the literals from q_1. We test the four conditions that show that h is a containment mapping:
  - $export(q_1') \supseteq export(q_1) \;\Rightarrow\; \forall v \in export(q_2): h(v) \in export(q_1')$
  - q_1' has the same constants as q_1 at all positions used as mapping target in h, i.e., $\forall c \in const(q_2):\; h(c)=c$
  - q_1' has all target literals of q_1;
  - q_1' does not introduce new conditions on variables of q_1.
- (L2.7): Let h_1 be the CM from q_2 into q_1, and let h_2 be the CM from q_3 into q_2. Let $h = h_2 \circ h_1$. We show that h is a containment mapping from q_3 into q_1.
  - h maps each exported variable from q_3 onto an exported variable from q_1 via an exported variable of q_2;
  - h maps each constant from q_3 onto the same constant in q_1 via the same constant in q_2;
  - h maps each literal of q_3 onto a literal of q_1 via a literal of q_2;
  - Implication is transitive.

■

To specify a containment mapping, we must give a mapping for every symbol that appears in a query. If a query uses the abbreviation symbol '-', we first replace each occurrence of '-' with a fresh variable symbol. We construct this symbol by appending the relation of the literal, an integer indicating the position of this relation in the query, and an integer indicating the position of the '-' symbol in the literal.

Example 2.3.
Consider the following queries:

```
q1(cn,cl) ← clone(cid,cn,ct,cl),clone(cid,cn,ct,cl);
q2(cn,cl) ← clone(cid,cn,ct,cl);
q3(cn,cl,gn) ← clone(cid,cn,-,cl), clone(cid,-,-,cl), contains(cid,gid),
   gene(gid,gn,-);
q4(cn,cl,gn) ← clone(cid,cn,-,cl), contains(cid,gid),gene(gid,gn,-);
```

We show that $q_1 \equiv q_2$ and $q_3 \equiv q_4$. q_1 has one literal twice; one is obsolete and can be removed, which yields q_2. In the second case, clone also appears twice. The self-join between clone in q_3 will produce the same result for all tuples (cn,cl) as the clone literal in q_4. For instance, h_1 is a containment mapping from q_2 into q_1, and h_2 is a containment mapping from q_4 into q_3:

h_1 = [cid→cid,cn→cn,cl→cl,ct→ct];
h_2 = [cid→cid,cn→cn,clone$_{13}$→clone$_{13}$,cl→cl,gid→gid,gn→gn, gene$_{13}$→gene$_{13}$];

Both mappings are bijective. In the first case, the reverse mapping is a containment mapping from q_1 to q_2. Hence, $q_1 \equiv q_2$. In the second case, we have mapped the clone literal from q_4 into the first clone literal from q_3. This is necessary since mapping it to the second literal would imply cn→clone$_{22}$, which conflicts with the requirement that every exported variable is mapped to an exported variable (condition (CM1) from Definition (D2.16)). The reverse mapping h_2' maps both clone literals from q_3 into the (only) clone literal from q_4:

h_2' = [cid→cid,cn→cn,clone$_{13}$→clone$_{13}$,cl→cl,clone$_{22}$→cn,clone$_{23}$→clone$_{13}$,
gid→gid,gn→gn,gene$_{13}$→gene$_{13}$,gene$_{14}$→gene$_{14}$];

■

Example 2.4.
Consider the following CS_S queries:

q_1(cn,cl) ← clone(cid,cn,ct,cl);
q_2(cn,cl) ← clone(cid,cn,ct,cl),cl<150;
q_3(cn$_1$,cn$_2$) ← clonelocation(mid,cid$_1$,-),clonelocation(mid,cid$_2$,-), clone(cid$_1$,cn$_1$,-,-), clone(cid$_2$,cn$_2$,-,-);
q_4(cn$_1$,cn$_2$) ← clonelocation(mid,cid$_1$,-),clonelocation(mid,cid$_2$,-), clone(cid$_1$,cn$_1$,-,c$_1$), clone(cid$_2$,cn$_2$,-,cl);

$q_2 \subseteq q_1$ since the conditions from q_2 imply true. q_2 will return only clones with are smaller than 150 KB, which is a set that is a subset of all clones, for any possible database.

q_3 searches for two clones that are placed on the same map. q_4 in contrast searches for two clones that are on the same map and also have the same length. The result of q_4 is always a subset of the result of q_3, since every valuation for q_4 is also a valuation for q_3, but not vice versa. Therefore, $q_4 \subseteq q_3$, which we can show with the following containment mapping h:

h = [mid→mid,cid$_1$→cid$_1$,clonelocation$_{13}$→clonelocation$_{13}$,cid$_2$→cid$_2$, clonelocation$_{23}$→clonelocation$_{23}$,cn$_1$→cn$_1$,clone$_{13}$→clone$_{13}$,clone$_{14}$→cl,
cn$_2$→cn$_2$,clone$_{23}$→clone$_{23}$,clone$_{24}$→cl];

No reverse containment mapping exists since this would imply cl→clone$_{14}$ and cl→clone$_{24}$.

■

Query equivalence is not always as easy to see as in the examples we considered so far. The following example is adapted from [AHV95], pp. 119.

Example 2.5.
Consider the following two queries:

q_1(x,y,z) ← rel(x$_2$,y$_1$,z),rel(x,y$_1$,z$_1$),rel(x$_1$,y,z$_1$),rel(x,y$_2$,z$_2$),rel(x$_2$,y$_2$,z);
q_2(x,y,z) ← rel(x$_2$,y$_1$,z),rel(x,y$_1$,z$_1$),rel(x$_1$,y,z$_1$),rel(x$_2$,y$_2$,z);

q_1 is equivalent to q_2 with the fourth occurrence of rel removed. Therefore, $q_1 \subseteq q_2$. We may prove $q_2 \subseteq q_1$, which implies $q_1 \equiv q_2$, with the following containment mapping h:

h = [x$_2$→x$_2$,y$_1$→y$_1$,z→z,x→x,z$_1$→z$_1$,x$_1$→x$_1$,y→y,y$_2$→y$_1$,z$_2$→z$_1$];

h maps: $r^1 \rightarrow r^1$, $r^2 \rightarrow r^2$, $r^3 \rightarrow r^3$, $r^4 \rightarrow r^2$, $r^5 \rightarrow r^1$ (superscription indicate the position of the literal in the query). The fourth occurrence of rel in q_2 is not used.

■

It does not suffice to consider only the bodies of queries to check containment or equivalence. The following is an example of two queries that are not equivalent, because they export different sets of variables.

Example 2.6.
Consider the following two queries:

q_1(cn,bp) ← clone(cid,cn,-,-),clonesequence(cid,sid),sequence(sid,bp);
q_2(cn,sid) ← clone(cid,cn,-,-),clonesequence(cid,sid),sequence(sid,bp);

There does not exist a CM that maps exported variables as required. For the same reason, q_3 and q_4 are not equivalent, although $q_3 \subseteq q_4$:

q_3(cn,bp) ← clone(cid,cn,-,-),clonesequence(cid,sid),sequence(sid,bp);
q_4(cn) ← clone(cid,cn,-,-),clonesequence(cid,sid),sequence(sid,bp);

■

2.3 Proving Query Containment

We study the problem of proving that a query q_1 is contained in query q_2 by computing a containment mapping from q_2 into q_1. To this end, we decompose the "query containment" problem into a set of "literal containment" problems. Containment mappings between single literals are then combined into containment mapping between queries. We shall use a similar decomposition strategy in Chapter 5.

2.3.1 Problem Decomposition

First, we define the decomposition of query containment into literal containment, and the composition of containment mappings between single literals to a containment mapping between queries. Then, we formally show that this approach is a sound and complete method for proving query containment.

Definition (D2.17)-(D2.18) (Matching and covering literal).
Let Σ be a schema.

(D2.17) Let l_1, l_2 be two literals for relations from Σ. l_2 *matches* l_1 iff:

- The relations of l_1 and l_2 are identical.
- There exists a symbol mapping from l_2 to l_1 that maps each constant in l_2 to the same constant in l_1.

(D2.18) Let $q_1, q_2 \in CQ_C^{\Sigma}$, and let l_1,l_2 be literals with $l_1 \in q_1$ and $l_2 \in q_2$. l_2 *covers* l_1 with mapping h, written $l_2 \geq_h l_1$, iff:

- l_2 matches l_1 with mapping h.
- h maps each exported variable from l_2 to an exported variable in l_1.
- $\mathrm{cond}(q_1, \mathrm{variables}(l_1)) \Rightarrow h(\mathrm{cond}(q_2, \mathrm{variables}(l_2)))$.

■

Remarks:

- We purposely use the order "l_2 matches l_1" to be consistent with the direction of containment mappings.
- l_2 can match l_1 with at most one symbol mapping. The fact that l_2 matches l_1 does not imply the reverse. Testing whether l_2 matches l_1 is possible in $O(arity(l_2))$.
- If l_2 covers l_1, then the mapping between them is unique.
- If l_2 covers l_1, we also say that l_1 is a *target* of l_2.
- Testing the first two conditions of *covers* is possible in time complexity $O(arity(l_2))$. Testing the third condition is linear for $q_1, q_2 \in CQ_S$, and polynomial for $q_1, q_2 \in CQ_C$ [LS93].

■

We use the notion of *covers* to decompose the problem of finding a containment mapping between two queries into the problem of finding *partial containment mappings* (PCM) between single literals of those queries.

Definition (D2.19)-(D2.20) (Partial and complete containment mapping).
Let $q_1, q_2 \in CQ_C^{\Sigma}$. Let L be a subset of the literals of q_2, and let $S = sym(L)$.

(D2.19) The mapping $h: S \mapsto sym(q_1)$ is a *partial containment mapping* from q_2 into q_1 iff h is a containment mapping from the query $<L, cond(q_2, S)>$ into q_1.

(D2.20) A partial containment mapping is a *complete containment mapping* if it is a containment mapping, i.e., L contains all literals of q_2.

■

Lemma (L2.8) (From covered literals to PCMs).

(L2.8) Let q_1,q_2 be two queries, and l_1, l_2 two literals with $l_1 \in q_1$, $l_2 \in q_2$ and $l_2 \geq_h l_1$. Then h is a PCM from q_2 into q_1.

■

The proof is straight-forward and omitted.

Suppose we want to test $q_1 \subseteq q_2$, and that we found PCMs from every literal of q_2 into q_1. To prove the containment, we have to show that those PCMs can be combined in such a way that their union is a containment mapping. This is not evident, because partial mappings can be incompatible, as shown in the following example.

Example 2.7.
Consider the following two queries. q_1 searches for clones that have their own name as alias:

```
q1(cid) ← clone(cid,cn,ct,cl),clonealias(cid,cn);
q2(cid) ← clone(cid,'yWXD1',ct,cl),clonealias(cid,'C15');
```

Suppose we want to test whether $q_2 \subseteq q_1$. Clearly, there exists a PCM from the first literal of q_1 into the first literal of q_2, and another PCM from the second literal of q_1 into the second literal of q_2. However, those mappings cannot be combined into a complete containment mapping from q_1 into q_2 because the variable `cn` is once mapped into the constant '`yWXD1`' and once into the constant '`C15`'.

■

Therefore, before we combine PCM, we have to check carefully is they are *compatible*.

Definition (D2.21)-(D2.23) (Compatibility and union of PCMs).
Let $q_1, q_2 \in CQ_C^{\Sigma}$ and h_1, h_2 be two PCMs from q_2 into q_1 defined over the symbol sets $S_1, S_2 \subseteq sym(q_2)$, respectively.

(D2.21) h_1 and h_2 are *reconcilable* iff $\forall\ s \in S_1 \cap S_2 : h_1(s) = h_2(s)$

(D2.22) The union of two reconcilable PCMs h_1 and h_2, written, $h = h_1 \cup h_2$, is the PCM h: $S_1 \cup S_2 \mapsto sym(q_1)$ with:

- $\forall\ s \in S_1,\ s \notin S_2 \quad : h(s) = h_1(s);$
- $\forall\ s \in S_2,\ s \notin S_1 \quad : h(s) = h_2(s),$
- $\forall\ s \in S_1 \cap S_2 \quad : h(s) = h_1(s) = h_2(s);$

(D2.23) h_1 and h_2 are *compatible*, written as $h_1 \sim h_2$, iff:

- h_1 and h_2 are reconcilable.
- Let $h = h_1 \cup h_2$. Then $cond(q_1, h(S_1 \cup S_2)) \Rightarrow h(cond(q_2, S_1 \cup S_2))$.

■

Remarks:

- Testing *reconcilability* of PCMs is possible in $O(max(|S_1|, |S_2|))$ for $q_1, q_2 \in CQ_C$.
- Union is commutative ($h_1 \cup h_2 = h_2 \cup h_1$) and associative ($h_1 \cup (h_2 \cup h_3) = (h_1 \cup h_2) \cup h_3$), assuming that the mappings h_3 and $h_2 \cup h_3$, and h_3 and $h_1 \cup h_2$, are reconcilable.
- Compatibility is symmetric ($h_1 \sim h_2 \Leftrightarrow h_2 \sim h_1$) and, together with the union operation, in one direction distributive ($h_1 \sim (h_2 \cup h_3) \Rightarrow (h_1 \sim h_2 \wedge h_1 \sim h_3)$). The reverse direction does not hold (see Example 2.8).
- Testing compatibility for CQ_S queries is $O(|S_1|+|S_2|)$. For CQ_C queries, it is polynomial in $|cond(q_1, S_1)| + |cond(q_2, S_2)|$.

■

Example 2.8.
To see that distributivity only holds in one direction, consider the queries:

$q_1(a_1,b_1,c_1,d_1) \leftarrow rel_1(a_1,b_1), rel_2(a_1,c_1), rel_3(a_1,d_1), c_1<d_1;$
$q_2(a_2,b_2,c_2,d_2) \leftarrow rel_1(a_2,b_2), rel_2(a_2,c_2), rel_3(a_2,d_2), c_2>d_2;$

Consider the PCMs $h_1 = [a_1 \rightarrow a_2, b_1 \rightarrow b_2]$, $h_2 = [a_1 \rightarrow a_2, c_1 \rightarrow c_2]$ and $h_3 = [a_1 \rightarrow a_2, d_1 \rightarrow d_2]$. Clearly, $h_1 \sim h_2$ and $h_1 \sim h_3$, but $h_1 \nsim (h_2 \cup h_3)$ since the implication "$c_1 > d_2 \Rightarrow c_1 < d_2$" is false. h_2 and h_3 are not compatible.

■

Before we use PCMs between literals to construct containment mappings between entire queries, we show that, whenever a set of PCMs is pairwise compatible, then each PCM of the set is also compatible to the union of the other PCMs. We prove this lemma for two reasons: First, it facilitates the following Lemma (L2.10). Second, it is necessary for an improvement of the algorithms described in Section 2.3 (see remark after Theorem (T2.16)).

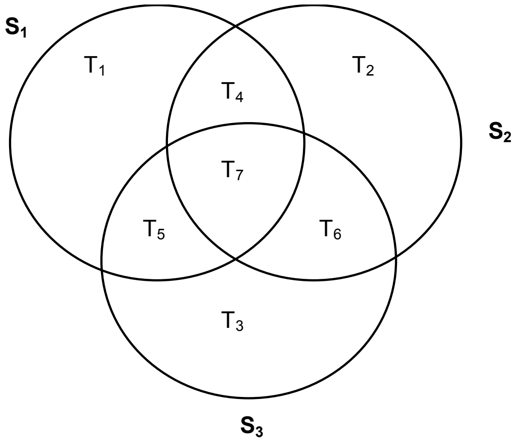


Figure 8. Decomposition of a set of conditions in seven subsets.

Lemma (L2.9) (Connecting partial mappings).
Let $q_1, q_2 \in CQ_C$ and let h_1, h_2, h_3 be three partial mappings from q_2 into q_1.

(L2.9) $h_1 \sim h_2 \wedge h_1 \sim h_3 \wedge h_2 \sim h_3 \Rightarrow h_1 \sim (h_2 \cup h_3)$.

■

Proof:
Let S_1, S_2, $S_3 \subseteq \mathrm{sym}(q_2)$ be the symbol sets for which h_1, h_2, h_3 are defined respectively. We show that the mapping $h = h_1 \cup h_2 \cup h_3$ is a containment mapping from q_2 into q_1. Since h_1-h_3 are PCMs, conditions (CM1) - (CM3) follow immediately. To show (CM4), let $S = S_1 \cup S_2 \cup S_3$. We divide S into seven disjoint subsets T_i, $1 \le i \le 7$, as illustrated in Figure 8. For the sets T_1-T_6, which only contain symbols appearing in at most two of the three partial mappings, (CM4) must hold since the mappings are pair-wise compatible to each other. For T_7, we must check those conditions in q_1 and q_2 that use a set of variables T with (a) $T \in S_1 \cap S_2 \cap S_3$ and (b) $T \notin S_1 \cap S_2$, $T \notin S_1 \cap S_3$, $T \notin S_2 \cap S_3$. However, no such condition exists since any condition has at most two variables.

■

Suppose we want to test $q_1 \subseteq q_2$. The previous definitions suggest that we can build complete containment mappings by chaining PCMs from the literals of q_2 into the literals of q_1. This is indeed the case, as shown by the following lemmas. Lemma (L2.10) shows that this approach is complete, i.e., whenever a containment mapping from q_2 into q_1 exists, then there exist compatible PCMs for each literal of q_2 into some literal of q_1. Lemma (L2.11) shows that the approach is sound, i.e., whenever there exist compatible PCMs for each literal of q_2 into some literal of q_1, then their union is a containment mapping from q_2 into q_1.

Lemma (L2.10)-(L2.11) (From partial to complete containment mappings).
Let $q_1, q_2 \in CQ_C^{\Sigma}$ and let $l_{i,j}$ be the j'th literal of q_i. Let $n_1 = |q_1|$ and $n_2 = |q_2|$.

(L2.10) Let h be a containment mapping from q_2 into q_1. Then there exist n_2 partial containment mappings $h_1, ..., h_{n2}$ with the following properties:

- h_j is a partial containment mapping from $l_{2,j}$ into some $l_{1,j'}$, $1 \le j \le n_2$, $1 \le j' \le n_1$;
- All h_j are pair-wise compatible to each other.

(L2.11) Suppose that there exist n partial containment mappings h_j from q_2 into q_1 such that each h_j maps $l_{2,j}$ into some $l_{1,j'}$. Furthermore, suppose that all these h_j are pair-wise compatible to each other. Then the union of all h_j's is a complete containment mapping from q_2 into q_1.

■

Proof:

- (L2.10) Let g_j be the mapping that is equal to h restricted to $sym(l_{2,j})$. g_j certainly exists and is a PCM from $l_{2,j}$ into $l_{1,j'}$ for some $j' \leq |q_1|$ since (CM3) must hold for h. Let g_l, g_k be two arbitrary such mappings, $l \neq k$. g_l and g_k must be reconcilable since their origins are both subsets of $sym(q_2)$ and h is a function. The union $g = g_l \cup g_k$ is a restriction of h to $S = sym(l_{2,l}) \cup sym(l_{2,k})$. We must prove:

  $$cond(q_1, g(S)) \Rightarrow g(cond(q_2, S))$$

  Since h is a containment mapping, the implication $cond(q_1, h(sym(q_2))) \Rightarrow h(cond(q_2, sym(q_2)))$ must hold. From this implication we can, simultaneously on both sides, remove in consecutive steps all conditions for each symbols $s \in sym(q_2) \setminus S$ without destroying the implication.

- (L2.11) If we can prove that the union of two compatible PCMs is again a (partial) containment mapping, then the theorem follows by induction.
  Consider two PCMs h_l, h_k from q_2 into q_1 with origins S_l and S_k and $h_l \sim h_k$. This implies that h_l and h_k are reconcilable. To show that $h_l \cup h_k$ is a containment mapping, it suffices to check that (CM4) holds for every $s \in S_l \cup S_k$. But exactly this is required in the definition of compatibility.

■

Now, we devise a method for testing whether $q_1 \subseteq q_2$.

Algorithm 1. Testing query containment.

Input: Two queries $q_1, q_2 \in CQ^{\Sigma}_{S}$.

Output: If $q_1 \subseteq q_2$: true, else false.

Algorithm: For each literal l_i of q_2, compute the set L_i of all target literals in q_1. Pick one literal of each L_i. Check if the corresponding PCMs are pair-wise compatible to each other. If yes, report true. If all combinations of covered literals have been tested unsuccessfully, report false.

■

The following theorems formally prove soundness and correctness of the previous algorithm. In the next section, we give examples for how the algorithm works.

Theorem (T2.12)-(T2.13) (Soundness and completeness of Algorithm 1).
Let q_1, q_2 be two queries against Σ.

(T2.12) $q_1, q_2 \in CQ^{\Sigma}_{C}$. If Algorithm 1 reports true, then $q_1 \subseteq q_2$.

(T2.13) $q_1, q_2 \in CQ^{\Sigma}_{S}$. Algorithm 1 reports true iff $q_1 \subseteq q_2$.

■

Proof:

- (T2.12): Following Lemma (L2.8), each element of L_i induces a PCM from l_i into some literal of q_1. If a combination of such elements is pair-wise compatible to each other, then their union is a complete containment mapping from q_2 into q_1, as shown in Lemma (L2.11). This proves $q_1 \subseteq q_2$.
- (T2.13): Assume that $q_1 \subseteq q_2$ with containment mapping h and that Algorithm 1 reports `false`. We prove that this leads to a contradiction.
  If h exist, then, according to Lemma (L2.10), there must exist PCMs for each literal of l_2 such that they all are pair-wise compatible. However, since Algorithm 1 has tested all possible combinations, we can conclude that no such mappings exist. This proves that $q_1 \not\subseteq q_2$ for CQ_s^Σ queries.

■

2.3.2 The Search Space of Query Containment

Algorithm 1 tests whether a query q_1 is contained in a query q_2. It proceeds in two steps. First, it computes all PCMs from literals of q_2 into literals of q_1. Each literal of q_2 is assigned a set L_i of target literals in q_1. Second, it enumerates the cartesian product of the elements of the different target sets. If all PCMs of one such combination are compatible, then q_1 is contained in q_2; otherwise, q_1 is not contained in q_2.

The second step of the algorithm is a search. We may visualise the search space as a tree, as shown in Figure 9. The i'th level of the tree represents the target set L_i of the i'th literal of q_2 (any literal of q_1 can be the target of several literals from q_2). Each inner node of the tree represents a bag of literals from q_1 taken from different target sets. Each path from the root to a leaf represents a bag of literals from q_1 that contains one target for each literal of q_2.

Since target literals are associated to PCMs (a $g_{i,j}$ in Figure 9 is the PCM from l_i into a literal of q_1), each leaf represents a *potential containment mapping* from q_2 into q_1, which is the union of the PCMs associated to the targets on the path to that leaf. However, if any two PCMs on a path are incompatible, their combination cannot yield a containment mapping. If such an incompatibility is detected, the subtree beneath this path can be pruned.

One can device several strategies to traverse the search space. We describe two such strategies. In Section 2.3.3, we present the *breadth-first algorithm* (BFA). In Section 2.3.4, we present the *depth-first algorithm* (DFA).

While traversing the search space, both algorithms build PCMs of increasing length, i.e., covering an increasing number of literals from q_2. Consider the transition from a node x on level i to a node y on level $i+1$. Let x be associated to the PCM h_1, obtained by building the union of the PCMs on the path to x. h_1 hence covers the first i literals of q_2. Let y be associated to the PCM h_2 from l_{i+1} into some literal of q_1. To check whether the transition from x to y is valid, we test whether $h_1 \sim h_2$. If yes, we associate $h_1 \cup h_2$ to y; if not, the subtree beneath y is removed.

To finish this section, we give some examples that illustrate this approach. In the following, "$q.rel^j$" denotes the j-th literal of query q, indicating that this is a literal for the relation `rel`.

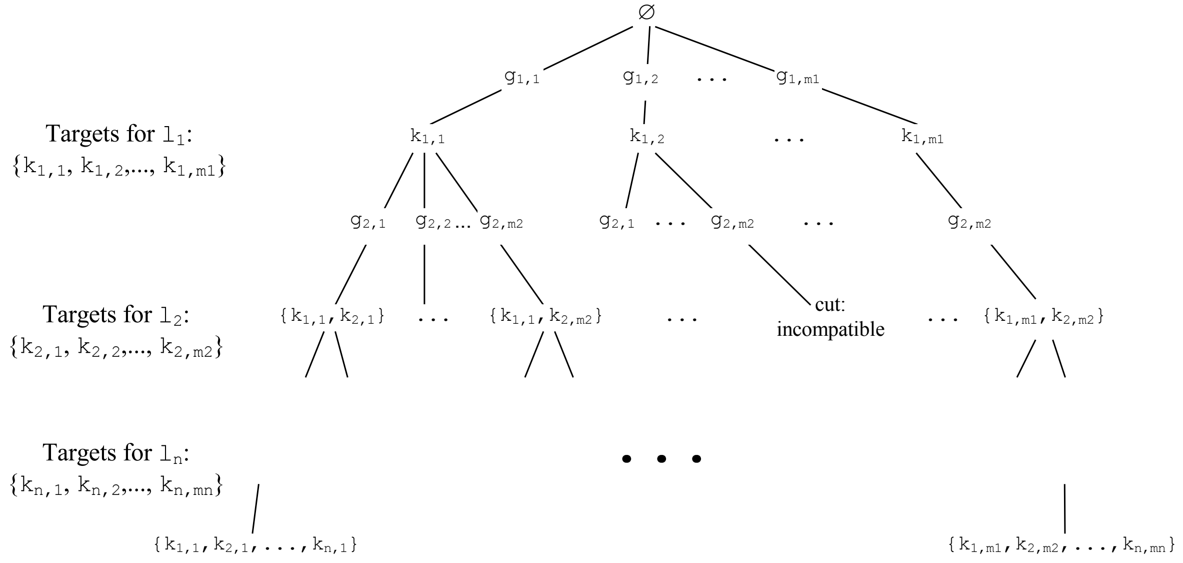


Figure 9. Search space for testing query containment.

Example 2.9.

Consider the following queries.

$q(a,b) \leftarrow rel_1 b),rel_2(b,c),c<100,b<c;$

$q_1(v,w,x,y) \leftarrow rel_1(v,w),rel_1(x,y),rel_2(y,v),v<500;$

$q_2(x,y) \leftarrow rel_1(x,'c'),rel_2('d',y);$

$q_3(v,w,x,y) \leftarrow rel_1(v,w),rel_2(x,y);$

$q_4(x,y) \leftarrow rel_1(x,y),rel_2(y,z),z<50,z<y;$

We examine containment of each q_i in q:

- $q_1 \not\subseteq q$: Clearly, $q.rel_1^1 \geq_{h1} q_1.rel_1^1$ and $q.rel_1^1 \geq_{h1} q_1.rel_1^2$ with $h_1 = [a \rightarrow v, b \rightarrow w]$ and $h_2 = [a \rightarrow x, b \rightarrow y]$. All those variables are exported. Furthermore, $q.rel_2^2$ matches $q_1.rel_2^3$, but $\nexists\, h_3: q.rel_2^2 \geq_{h3} q_1.rel_2^3$ because h_3 needs to map $[c \rightarrow v]$ – but $v < 500$ does not imply $v < 100$, which is $h_3(c < 100)$. q_1 contains no literal that is covered by $q.s^2$.
- $q_2 \not\subseteq q$: We find $q.rel_1^1 \geq_{h1} q_2.rel_1^1$ and $q.rel_2^2 \geq_{h2} q_2.rel_2^2$ with $h_1=[a \rightarrow x, b \rightarrow 'c']$ and $h_2=[b \rightarrow 'd', c \rightarrow y]$. h_1 and h_2 are not compatible, since $h_1(b) \neq h_2(b)$.
- $q_3 \not\subseteq q$: We find $q.rel_1^1 \geq_{h1} q_3.rel_1^1$ and $q.rel_2^2 \geq_{h2} q_3.rel_2^2$ with $h_1 = [a \rightarrow v, b \rightarrow w]$, $h_2 = [b \rightarrow x, c \rightarrow y]$. Again, h_1 and h_2 are incompatible since $h_1(b) \neq h_2(b)$. q_3 lacks the join.
- $q_4 \not\subseteq q$: We see $q.rel_1^1 \geq_{h1} q_4.rel_1^1$ and $q.rel_2^2 \geq_{h2} q_4.rel_2^2$ with $h_1 = [a \rightarrow x, b \rightarrow y]$ and $h_2 = [b \rightarrow y, c \rightarrow z]$. These two mappings are reconcilable, but not compatible, since $(z < 50 \wedge z < y) \nRightarrow (z < 100 \wedge y < z)$. Note that the conflict between the conditions $z < y$ and $y < z$ does not occur in single PCMs, but only in their union.

■

2.3.3 Breadth-First Algorithm

The *breadth-first algorithm* (BFA) (see Algorithm 2) for proving query containment traverses the search space described in the previous section in a breadth-first manner. To see that, let the body of q_2 consist of the literals $l_1,...,l_n$. In each step of the outer-most `foreach` loop (lines 2-15), the algorithm builds all possible PCMs from queries q_i into q_1, where q_i is "$l_1,...,l_i,\mathrm{cond}(q_2,\mathrm{variables}(l_1,...,l_i))$", $i \leq n$, i.e., the conjunctive query made of the first `i` literals of q_2 plus all conditions of q_2 that contain only variables occurring in those literals. The algorithm stops and reports `false` as soon as it finds a q_i for which no partial mapping exists (line 11-13).

In the following, we analyse Algorithm 2. Therein, we measure the number of times that the algorithm performs a compatibility check between two PCMs. This number correlates with the number of nodes in the search tree.

Theorem (T2.14)-(T2.16) (Analysis of the BFA).
Let $q_1, q_2 \in CQ_C$, $k_1 = |q_1|$, $k_2 = |q_2|$. For the average case analysis, we make the following assumptions. Let z, $0 \leq z \leq k_1$, be the average number of literals of q_1 covered by a literal of q_2. Furthermore, let p_{com}, $0 \leq p_{com} \leq 1$, be the probability that two arbitrary PCMs are compatible.

(T2.14) In the worst-case, the BFA performs $C^{wc}_{BFA} = \sum_{i=1}^{k_2} (k_1)^i$ compatibility tests.

(T2.15) In the average-case, the BFA performs $C^{ac}_{BFA} = z\sum_{i=0}^{k_2-1} \left(z^i (p_{com})^i\right)$ compatibility tests.

(T2.16) The complexity of the BFA is $O\left((k_1)^{k_2}\right)$.

■

Proof:
Let $loop_1$ be the outer-most loop (lines 2-15), $loop_2$ the second (lines 4-14) and $loop_3$ the inner-most (lines 6-10).

- (T2.14): In the worst case, we must assume that $|L|=k_1$ and that all compatibility tests are successful. $loop_1$ is traversed k_2 times. In its first run, $loop_2$ will be traversed once and $loop_3$ k_1 times. In the next round of $loop_1$, $loop_2$ is passed k_1 times, each time cycling k_1 times through $loop_3$. Line 7 is passed k_1 times in the first pass of $loop_1$, k_1^2 in the second, k_1^3 in the third, etc. The formula follows immediately.
- (T2.15): Let s_i be the size of R after the i'th traversing of $loop_1$. In the first run of $loop_1$, line 7 is passed z times and the test succeeds in p_{com} percent of the cases. Therefore, $s_1 = zp_{com}$. In the second run, line 7 is passed zs_1 times, and $s_2 = p_{com}s_1z = (p_{com})^2z^2$. We get:

$$C^{ac}_{BFA} = z + zp_{com}z + z^2(p_{com})^2z + \ldots = z\sum_{i=0}^{k_2-1}\left(z^i(p_{com})^i\right)$$

For $z = k_1$ and $p_{com} = 1$, this formula coincides with the worst-case complexity.

- (T2.16): The complexity of the algorithm depends on the worst case. The cost of computing union and compatibility of two PCMs is at most polynomial in the size of the mappings and therefore negligible. Accordingly, the complexity is:

$$O\left(\sum_{i=1}^{k_2}(k_1)^i\right) = O\left(k_2(k_1)^{k_2}\right) = O\left((k_1)^{k_2}\right)$$

Hence, the BFA is exponential in the size of q_2. This is what we expected due to the complexity of query containment.

■

Algorithm 2. Breadth-first implementation of Algorithm 1 (BFA).

```
Input: Two queries $q_1$, $q_2$;
Output: true, if $q_1 \subseteq q_2$, else false;

R = {[]};                                % Set of incrementally built mappings
foreach $l \in q_2$                      % Set of all literals of $q_2$
  L = {h |$\exists l' \in q_1 \wedge l \geq_h l'$};  % Set of partial mappings of literals
                                         % from $q_1$ covered by l
  foreach $g \in L$
    foreach $h \in R$
      R = R \ h;
      if compatible(h,g) then            % Try to combine h and g
        R = R ∪ (h ∪ g);
      end if;
    end for;
  end for;
  if R=∅ then                            % No mapping can exist
    return false;
  end if;
end for;
return true;                             % Every $h \in R$ is a CM from $q_2$ into $q_1$
```

Remarks:

- The worst-case makes quite extreme assumptions: (a) $|L| = k_1$ implies that all literals of q_1 and q_2 are from the same relation. (b) Mappings are never incompatible in an early stage, which would cut the search space drastically. (c) In the last pass of loop$_1$, all tests except the very last one fail, since, if a previous test succeeded, we could stop immediately (and report true).
- In general C^{ac}_{BFA} << C^{wc}_{BFA}. For instance, if both queries have 4 literals ($k_1 = k_2 = 4$), the worst-case predicts that compatibility has to be tested 424 times. If we however assume that each literal of q_2 covers only two literals of q_1 ($z = 2$), and that two PCMs have a 80% chance of being compatible ($p_{com} = 0.8$), we perform only 19 tests.
- Lemma (L2.9) shows that the following, slightly more efficient algorithm is also feasible. It avoids a great number of executions of compatible, but without removing the enumeration of an exponential number of combinations. We save some (cheap) compatibility tests, but we still have to visit each node in the search tree. The complexity of the algorithm remains the same.

  The modified algorithm would work as follows: First, compute sets L_i that contain all literals of q_1 covered by the i'th literal of q_2. Test compatibility between each pair of elements from different L's, and store the result in a matrix. Enumerate combinations of targets as in the BFA. However, do not compute compatible and the union in line 8 and 9 – instead, use the matrix to look-up the compatible result between the new PCM and all PCMs already contained in the current combination. If a complete mapping is finally reached, compute the union of all partial mappings.

Lemma (L2.9) proves that this is a valid approach. The number of executions of compatible is reduced to $O(|q_1|^2)$, and therefore this algorithm is more efficient than the BFA. However, the focus of this work is *query planning*, not *query containment*. Both problems are related, but not identical. The difference will become clear in Chapter 5, and we shall show that Lemma (L2.9) does not hold for query planning. Therefore, our planning algorithms will build upon the BFA.

■

2.3.4 Depth-First Algorithm

The breadth-first strategy is not optimal for testing query containment. To show that one query is contained in another it suffices to *find one containment mapping*, even though many may exist. Certainly, the BFA could be improved by stopping as soon as one complete mapping is found. But still, the BFA would compute all incomplete combinations of target literals (see Figure 9: the BFA visits all nodes of the levels 1 to n-1 before a complete mapping can be found). Using a depth-first strategy, there are good chances that a complete containment mapping is found early on, without expanding all branches of the search tree (in Figure 9, depth-first potentially computes only the left-most path).

Algorithm 3. Depth-first implementation of Algorithm 1 (DFA).

```
Input: Two queries $q_1$, $q_2$;
Output: True, if $q_1 \subseteq q_2$, else false;

R = new stack();
L = {l | l∈$q_2$};                % set of all literals of $q_2$
l = L.first();
H = {h | ∃l'∈$q_1$ ∧ l≥$_h$l'}       % set of PCMs into covered literals
R.push([], H, L\l);               % Start point
repeat
  (h,H,L) = R.pop();              % Takes last PCM constructed
  if H=∅ then                     % No mapping can survive this level
    return false;
  else
    g = H.first();
    R.push(h, H\g, L);            % Remains to be tested at this branch
    if compatible(h,g) then
      if L = ∅ then
        return true;              % All literals are mapped; finish
      else
        l = L.first();
        H' = {h | ∃l'∈$q_1$ ∧ l≥$_h$l'};
        R.push(h∪g, H', L\l);     % Expand that PCM next
      end if;
    end if;
  end if;
until R = ∅;
```

Algorithm 3 shows the depth-first algorithm (DFA) for proving query containment. Its central data structure is a stack `R` consisting of 3-tuples. Each such tuple corresponds to a node in the search tree. The first element of a tuple is the partial mapping for the subquery from the root of the tree to the current node (`h`). The second element is the set of children that have not been examined yet, i.e., the set of further potential extensions of the subquery (`H` and `H'`). The third element is the set of literals of q_2 that have not yet been considered in the subquery (`L`). `R` is ordered by the size of the third element in ascending order.

The algorithm first chooses an arbitrary literal `l` of q_2. It then expands the corresponding PCM with one of the possible PCM for the next literal (line 11). If both are compatible, there are two possibilities: Either we have already reached a leaf in the tree, which means that we can finish. The algorithm tests this by checking if the set `L` of literals from q_2 that have not yet been mapped to a target in q_1 is empty (line 14-15). Or we are at an inner node of the tree. In this case, we push a new node on the stack, corresponding to the union of the two PCMs (line 19). We must also push the actual node again if there are children that we did not consider yet (line 12). In the next `repeat` - loop, the new node is popped and expanded.

Consider Figure 9. The DFA will start by following the left-most path as long as possible, and backtrack whenever it fails. Therefore, it uses a depth-first strategy.

Theorem (T2.17)-(T2.19) (Analysis of the DFA).
Let $q_1, q_2 \in CQ_C$, $k_1 = |q_1|$, $k_2 = |q_2|$. For the average case analysis, we make the same assumptions as in Theorem (T2.15).

(T2.17) In the worst-case, the DFA performs $C_{DFA}^{wc} = \sum_{i=1}^{k_2} (k_1)^i$ compatibility tests.

(T2.18) In the average-case, the DFA performs: $C_{DFA}^{ac} \leq (k_2 - h) + \sum_{i=1}^{h} z^i$ compatibility tests. $h = \mathrm{ceiling}(\log_z(n_{avg}))$. n_{avg} will be defined in the proof.

(T2.19) The complexity of the DFA is $O(k_1^{k_2})$.

■

Proof:
The worst-case complexity of the DFA is the same as for the BFA. To see this, assume that all mappings in Figure 9 are compatible except $g_{n-1,m_{n-1}}$ and g_{n,m_n}. In this case, the DFA and the BFA explore the entire search tree. Therefore, Theorem (T2.17) follows from Theorem (T2.14), and Theorem (T2.19) follows from Theorem (T2.16).

Theorem (T2.18) postulates an upper bound indicating that the average case is by magnitudes better than the worst case, and also by far better than the average case for a breadth-first strategy. We prove this bound in two steps: (1) We estimate how many paths have to checked. (2) We estimate how many compatibility tests are necessary until the final path is reached.

(1) We compute the number n_{avg} of paths through the search tree we have to check in the average case. Recall that p_{com} is the probability that two arbitrary PCMs are compatible. The probability that a path in the search tree results in a complete mapping is hence $p = (p_{com})^{k2}$, since it entails that all mappings along the path must be compatible with each other. Let p_i be the probability that the `i`'th path is the first successful one, and let $q = 1-p$. Hence, $p_1 = p$. The chance that the second path is successful while the first path is not is:

$$p_2 = \overline{p_1}p = qp$$

for the third it is:

$$p_3 = q^2 p$$

For the k’th path, we get:

$$p_k = q^{k-1} p$$

Let X_i, be the random variable defined as :

$$X_i = \begin{cases} i : \text{the i'th path is the first successful path} \\ 0 : \text{else} \end{cases}$$

n_{avg} is equal to the sum over the expected values of the X_i, $E(X_i)$. Assuming uniform distribution, we get:

$$\sum_{1=1}^{\infty} E(X_i) = \sum_{i=1}^{\infty} iq^{i-1}p = \sum_{i=1}^{\infty} \left(iq^{i-1} - iq^{i}\right) = \sum_{i=0}^{\infty} q^i = \frac{1}{1-q} = \frac{1}{p}$$

The expected value of X_i is independent from the number of “tests” of our random variable, i.e., leaves in the search tree. Therefore, the sum goes from 1 to infinity. However, an expected value that is higher than the total number of leaves, which is z^{k2}, is meaningless[4]. Therefore, we get:

$$n_{avg} = \begin{cases} p^{-1} \text{ if } p^{-1} \le z^{k_2} \\ z^{k_2} \text{ else} \end{cases}$$

(2) We want to compute how many compatibility tests must be carried out before we reach a successful leaf. We only give an upper bound here. Consider Figure 10, and suppose that the node in the circle is the first successful leaf. To reach this point, we must have already carried out all the compatibility tests “left” of this node. As an upper estimation for their number, we use the number of tests left of the thick line plus the tests on the path to the root of this subtree. The total height of the entire tree is k_2, and it has z^{k2} leaves. For the i’th leaf, the height of the subtree is $h = \lceil \log_z(i) \rceil$. The total number c of comparisons in a tree of height h is:

$$c(h) = \sum_{i=1}^{h} z^i$$

This is exactly the formula for the worst-case complexity of the entire problem, i.e., for a query containment problem with $k_2 = h$ and $k_1 = z$. Therefore, we make similar “mistakes” again - not all tests are necessary, since branches might be cut before the lowest level is reached, and no branch “right” from the successful leaf is ever expanded (space between the dotted and the full line in Figure 10). The difference is that we, by definition, know that no leaf before the n_{avg}’th is successful.

The average number of compatibility tests is now bound by:

$$C^{ac}_{DFA} \le (k_2 - h) + \sum_{i=1}^{h} z^i \text{, with } h = \text{ceiling}(\log_z(n_{avg}))$$

where $k_2 - h$ is the number of tests necessary to reach the root of the subtree. Since n_{avg} will be higher for smaller p_{com}, the error increases the smaller p_{com} is.

■

[4] Imagine throwing a dice n times. What is the average number of throws until a “four” appears ? This value is independent of n. If we now ask how many times we have to throw the dice until a “four” appears, but know that we will anyway stop after the n’th throw, an expected value larger than n can never be realised.

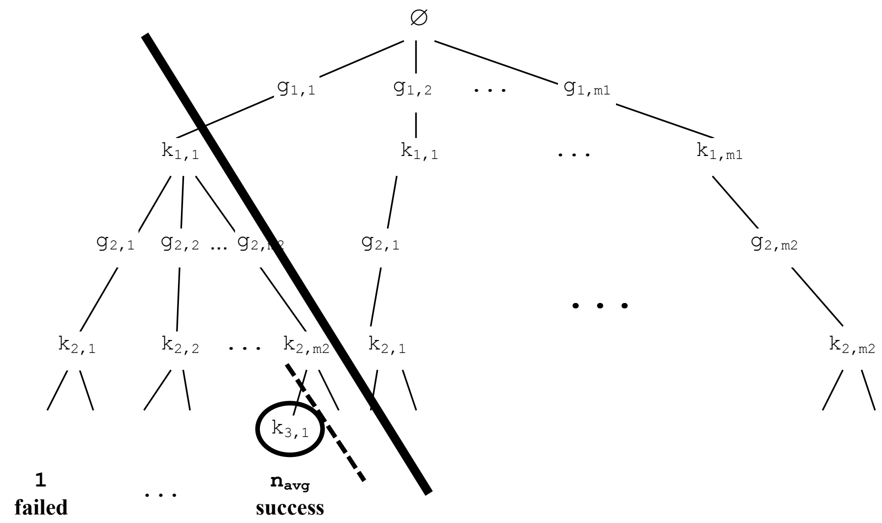


Figure 10. Search tree of height 3 with subtree used as upper bound.

2.3.5 Comparing BFA and DFA

To compare the performance of the DFA with the performance of the BFA, we give exemplary data in Table 2 for different values for z and p_{com}. We test $q_1 \subseteq q_2$ with $k_2 = |q_2|$ and $k_1 = |q_1|$. Figure 11, Figure 12, and Figure 13 plot average-case and worst-case values for varying instantiations of k_1, k_2, p_{com}, and z. Note that the assumption z=3 is already quite extreme, since it implies that every literal of q_2 appears, on the average, three times in q_1.

Consider Table 2. Since both C^{ac}_{BFA} and C^{ac}_{DFA} are independent from k_1 (which is however implicitly contained in the assumption on z) we mostly choose $k_1 = zk_2$. For the first two rows with z = 1, C^{ac}_{DFA} is not meaningful since only one leaf exists. In the third case, $C^{ac}_{DFA} > C^{ac}_{BFA}$ since the error in C^{ac}_{DFA} is very high for $p_{com} = 0.5$. In those rows, it is more reasonable to focus on the difference between C^{ac}_{BFA} and C^{wc}_{BFA}, which sheds some light on the on-first-sight frightening complexity of query containment.

k_1, k_2	z, p_{com}	p, n_{avg}, h	$C^{wc}_{BFA} = C^{wc}_{DFA}$	C^{ac}_{BFA}	C^{ac}_{DFA}
3,3	1,0.5	0.13,-,-	39	1.75	-
6,6	1,0.9	0.53,-,-	56000	4.69	-
6,3	2,0.5	0.13,7.69,3	258	6	14
12,6	2,0.9	0.53,1.89,1	3260000	82	7
12,6	3,0.99	0.94,1.06,1	3260000	1092	8

Table 2. Comparison of the worst case, average cost of BFA and average cost of DFA. Most figures are rounded. '-' indicate that the value is meaningless for the given parameters.

The last two rows prove our conjecture that a depth-first strategy outperforms a breadth-first approach. Depth-first is the better the greater is p_{com}, since successful leaves are found very early.

The depth-first and the breadth-first algorithm behave equally if $q_1 \not\subseteq q_2$. Proving that no containment mapping exists requires the exploration of the entire search space. This does not imply that the worst-case number of compatibility tests will be performed, since many branches will be pruned early due to incompatibility. Consider the third row of Table 2. The main cost reduction lies in the assumption about z. In the worst case, 258 leaves have to be tested; with $z = 2$, the tree has only $2^3 = 8$ leaves and 14 nodes, of which breadth-first in the average has to test 6, assuming $p_{com} = 0.5$. The forth row says that the search space has $2^6 = 64$ leaves and 127 nodes, of which BFA in average visits 82. DFA in contrast will only visit 7 in the average case. The fifth row shows that, although the search space of breadth-first has grown considerably, the increase in the average cost of the DFA value is not large; it is dominated by the effect of increasing p_{com} from 0.9 to 0.99.

The superiority of the DFA over the BFA is also highlighted in Figure 11-Figure 13. All three figures use a logarithmic scale for the y-axis. The steps in the lines for C^{ac}_{DFA} are artefacts introduced by the `ceiling` function in our estimation; recall that we only have an upper bound for C^{ac}_{DFA}, not an exact value.

In Figure 11, we use constant values for z and p_{com} and let k_1 grow from 4 to 30. We see that C^{ac}_{DFA} grows much slower than C^{ac}_{BFA}, which is still far better than C^{wc}_{BFA}.

In Figure 12, we keep k_1, k_2 and z constant and iterate p_{com} from 0.5 to 1. C^{wc}_{BFA} is constant in this and the following graph since it does only depend on k_1 and k_2. The error in C^{ac}_{DFA} is high for small values of p_{com}, leading to figures that are even worse than those for C^{ac}_{BFA}. C^{ac}_{DFA} shrinks with growing p_{com} since the probability that one of the first paths is already successful increases the higher the probability is that each branch (i.e., compatibility test) succeeds.

In Figure 13, we keep k_1, k_2 and p_{com} constant and vary only z. z has great impact on both C^{ac}_{DFA} and C^{ac}_{BFA} since it determines the width of the search tree. If we used $p_{com}=1$, C^{ac}_{DFA} would be constant since this implies that the first leaf is already successful; if we furthermore let z grow until 8, C^{ac}_{BFA} would eventually reach C^{wc}_{BFA} since this would exactly be the worst-case: all eight literals of q_1 are targets for each literal of q_2, and each compatibility test succeeds.

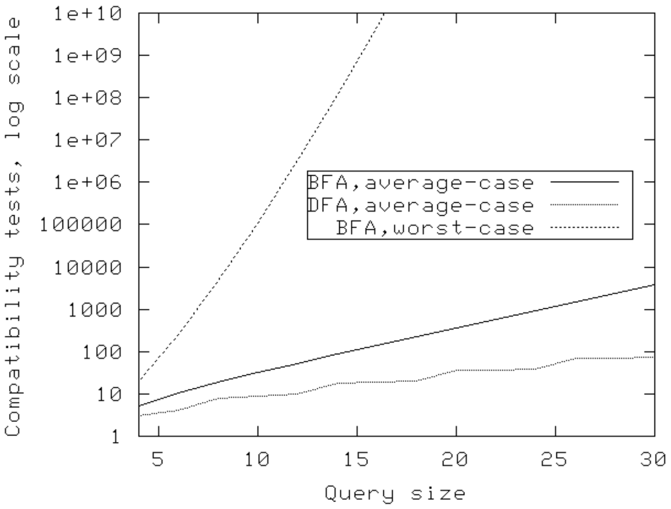


Figure 11. DFA versus BFA. Values: k_1=[2,30], k_2=k_1/2, p_{com}=0.8, z=2.

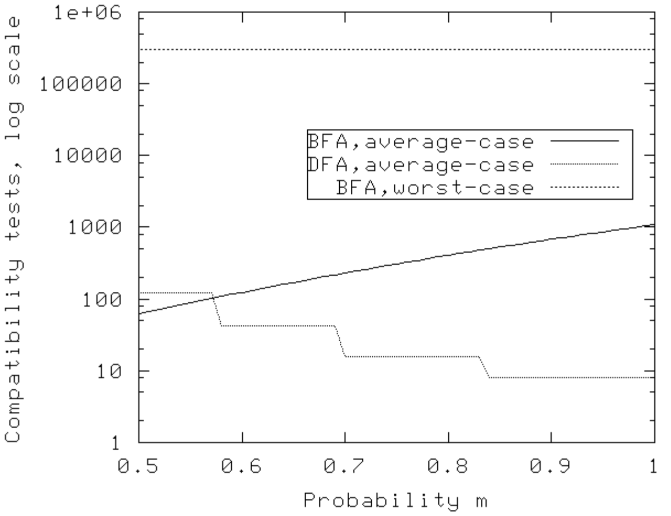


Figure 12. DFA versus BFA. Values: k_1=8, k_2=4, p_{com}=[0.5,1], z=3.

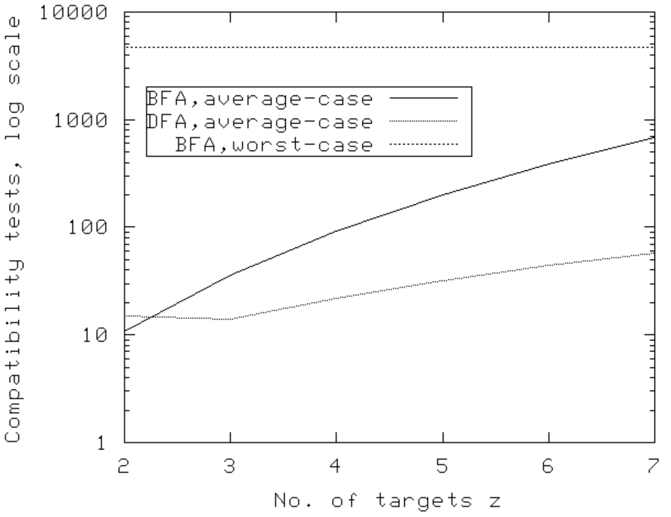


Figure 13. DFA versus BFA. Values: k_1=8, k_2=4, p_{com}=0.6, z=[2,7].

2.3.6 The "Frozen Facts" Algorithm

The DFA and the BFA test query containment by constructing containment mappings. Ramakrishnan et al. present a completely different method for checking containment of conjunctive queries in [RSUV89]. Although we shall not use this algorithm in the rest of this work, we describe and analyse it briefly since it is an interesting perspective on the nature of the containment problem.

To describe the algorithm, we first take a logic-oriented position. We rephrase the containment problem. Consider two queries $q_1, q_2 \in CQ_C$, $n_1 = |q_1|$, $n_2 = |q_2|$, against Σ:

$$q_1(E_1) \leftarrow rel_{1,1}(S_{1,1}), rel_{1,2}(S_{1,2}), \ldots, rel_{1,n1}(S_{1,n1});$$
$$q_2(E_2) \leftarrow rel_{2,1}(S_{2,1}), rel_{2,1}(S_{2,1}), \ldots, rel_{2,1}(S_{2,n2});$$

where E_1, E_2 and $S_{i,j}$ are vectors of symbols. Let $S_1 = (S_{1,1} \cup S_{1,2} \cup \ldots \cup S_{1,n1}) \setminus E_1$ and $S_2 = (S_{2,1} \cup S_{2,2} \cup \ldots \cup S_{2,n2}) \setminus E_2$. We may then abbreviate:

$$q_1(E_1) \leftarrow \phi_1(E_1, S_1);$$
$$q_2(E_2) \leftarrow \phi_2(E_2, S_2);$$

q_1 is contained in q_2 if every result computed by q_1 is also computed by q_2. Therefore, the containment problem can be solved by proving a logic formula.

Lemma (L2.20) (Query containment as logical implication).
Let $q_1, q_2 \in CQ_C$. Let E_1 (E_2) be the set of exported variables and S_1 (S_2) the set of non-exported variables of q_1 (q_2). Let $\phi_1(E_1, S_1)$ ($\phi_2(E_2, S_2)$) denote the body of q_1 (q_2).

(L2.20) $q_1 \subseteq q_2 \Leftrightarrow (\phi_2(E_2, S_2) \Rightarrow \phi_1(E_1, S_1))$.

■

The proof is straight-forward and omitted.

Hence, we can check $q_1 \subseteq q_2$ by proving a logical implication in first-order predicate logic. In principle, this method works for queries with arbitrary built-in predicates; however, the approach is restricted because implication is in general undecidable in first-order predicate logic.

Based on this background, the following method for testing query containment is not surprising. It basically implements a very simple theorem prover for the restricted class of logical formulas represented by conjunctive queries. Our description follows [Ull89], Chapter 14.5.

Algorithm 4. Frozen Facts Algorithm.

Input: Two queries $q_1, q_2 \in CQ^{\Sigma}$.

Output: If $q_1 \subseteq q_2$, then `true`; else `false`.

Algorithm: Consistently substitute each variable in q_1 with a fresh constant. Let `t` be the tuple corresponding to the head of q_1 after that substitution. Then build a new, empty instance `D'` of Σ. Insert into `D'` each literal $l \in q_1$ as tuple into the relation of `l`. Execute q_2 on `D'`. If `t` is in the result of q_2, then return true; else report `false`.

■

The algorithm takes as input two queries q_1 and q_2 against the same schema Σ. First, it creates an empty database `D'` for Σ (line 2). Then, it "freezes" q_1 (line 3). This means that it first replaces each variable in q_1 with a fresh constant. Then, for each literal of q_1, one tuple is inserted into `D'` into the extension of the corresponding relation. The attribute values of this tuple are the constants that appear in the literal in the frozen q_1. The algorithm finally computes q_2 on `D'`. If the frozen head of q_1 is in the extension of q_2, then it reports `true`; otherwise `false`.

Example 2.10.
Consider the following two queries against a schema Σ consisting of the two binary relations rel_1 and rel_2:

```
q₁(a,b) ← rel₁(a,b),rel₂(b,b);
q₂(x,y) ← rel₁(x,y),rel₂(y,z);
```

To check whether $q_1 \subseteq q_2$, we freeze q_1. Therefore, we consistently replace the variables of q_1 with constants using `[a→0, b→1]`. The frozen query is then:

```
q₁(0,1) ← rel₁(0,1),rel₂(1,1);
```

Next, we construct an empty database `D'` for Σ and insert the tuple `(0,1)` into rel_1 and `(1,1)` into rel_2. Finally, we compute q_2 on `D'`. Only one tuple, `(0,1)`, is obtained, using the valuation `[x→0, y→1, z→1]`. Since this is identical to the frozen head of q_1 the algorithm returns `true`, proving that q_1 is contained in q_2.

To check whether $q_2 \subseteq q_1$, we first freeze q_2 with `[x→0, y→1, z→2]`. This results in a database `D'` with tuple `(0,1)` in rel_1 and `(1,2)` in rel_2. Evaluating q_1 on `D'` yields an empty answer. This proves that $q_2 \not\subseteq q_1$.
■

Proofs for soundness and completeness of Algorithm 4 may be found in [Ull89] and [RSUV89]. The idea of the proof is the following: Let `R` be the result of q_2 on `D'`. First, suppose $t \notin R$. Then `D'` is a database for which q_1 computes `t`, but q_2 does not. `t` is hence a counterexample that proves $q_1 \not\subseteq q_2$. Now, suppose $t \in R$. Then the substitution of line 1 is a template for the valuation of q_1 on any instance `D` of `S`. Since $t \in R$, it follows that this template is also a valuation for computing q_2 on `D`. Hence, every tuple that is computed by q_1 is also computed by q_2. This implies $q_1 \subseteq q_2$.

Algorithm 4 may be implementing quickly using a RDBMS. One only has to generate tables according to the relations of q_1, insert as many tuples as q_1 has literals, and execute a SQL query corresponding to q_2 on this database. Finally, it has to be checked whether the result contains `t`.

Now, we show informally that the time complexity of this method is exponential in the size of q_2, as is the case for the DFA and BFA. We may safely assume that lines 1-4 have cost linear in the size of q_1. The exponential complexity is hidden in line 5. Assume again that `S` has only one relation, and let $k_1 = |q_1|$, $k_2 = |q_2|$. `D'` will contain k_1 tuples in a relation with name q_1. In the worst-case, q_2 will then compute the cartesian product of length k_2 of all tuples in q_1, which requires ${k_1}^{k2}$ combinations. Finally, checking whether $t \in R$ is linear in the size `R` (line 6); but $|R|$ is, in the worst case, again exponential in the size of q_2.

A great advantage of the frozen facts algorithm is that it is still sound and complete if q_2 is a *recursive query*. This is not the case for the DFA or BFA. The disadvantage of the method is that it is not directly applicable to CQ_S queries because conditions in q_1 disappear during the freeze-part.

2.4 Summary and Related Work

This chapter provided the necessary background for the query planning problem we shall consider in Chapter 5. We elaborated on algorithms for proving query containment. Therefore, we introduced a decomposition strategy for the problem of finding a containment mapping between two queries. We proved that this strategy yields sound and complete algorithms. We presented two such algorithms. The *breadth-first algorithm* uses a breadth-first strategy to traverse the search space combinations of containment mappings between literals, while the *depth-first algorithm* uses a depth-first strategy. We obtained average-case and worst-case boundaries for both algorithms. Our results show that the DFA is clearly superior for testing query containment because it, on average, finds a containment mapping much faster than the BFA. Finally, we described the *frozen facts algorithm* that proves query containment without constructing containment mappings.

In the following chapters we mostly build on the BFA. The reason is that the focus of this work is on *query planning* and not on *query containment*. We shall see in Chapter 5 that both problems are related, but not identical. An essential part of query planning is the construction of plans that are contained in a query. Therefore, query planning uses query containment in some form. However, for query planning the breadth-first and depth-first search strategy are equivalent because query planning anyway requires to find *all containment mappings* between two queries. The entire search space has to be explored, in which case the DFA and the BFA are equally appropriate. We shall use the BFA as the basis for query planning because its analysis is less complex. Note that we cannot use the frozen facts algorithm because it cannot cope with conditions in queries.

Related work.

The incentive to study query containment arose from *global query optimisation* [CM77]. Given a query q, global query optimisation tries to find a query q' that is equivalent to q but can be executed faster, for instance because it is shorter. In contrast to other optimisation techniques, global optimisation considers the entire query and not only local rearrangements of single operators [AHV95]. For instance, *query minimisation* [ASU79a] tries to find a query q' that is equivalent to q but has less literals.

Recently, query equivalence also became important for database vendors because of the possibility to optimise a query using *materialised views* [CKPS95; BDD+98]. The idea is the following: imagine that the result of a query q is materialised, i.e., it is stored as if it were a "normal" relation, together with q itself. Whenever q appears as subpart of another query, it is presumably better to reuse the materialised relation instead of recomputing q. Finding whether q is part of another query is essentially the containment problem. The complexity of this problem under different assumptions is studied in detail in [AD98].

Despite the great interest in query containment over the last 20 years, no article discusses algorithms in the same depth as this thesis does. It is also the first work that gives an average-case cost analysis of the query containment problem. Our results suggest that, in particular due to Theorem (T2.18), query containment is a tractable problem for most real-life queries.

Many researchers have focussed on finding classes of queries for which query containment is polynomial. For instance, Aho et al. describe *simple tableaux*, which roughly correspond to queries where non-exported variables do not appear at the same position in different queries [ASU79a]. The authors prove that equivalence for such queries is $O(n^4)$ in the length of the queries. Chekuri & Rajamaran analyse the complexity of query containment wrt. the *width of the query*, which is the highest number of occurrences of a relation in the query [CR97]. They show that containment is linear if each relation occurs at most once, quadratic if each relation appears at most twice, etc. This work extends previous results on queries with at most two literals of each relation published in [SY80].

In the following, we summarise extensions of the problem to other classes of queries than conjunctive ones. [Ull97] discusses some of them in more detail.

Queries with arithmetic comparisons.
As already mentioned in Theorem (T2.3), the existence of a containment mapping is only a sufficient, but not necessary condition for CQ_C queries, i.e., for queries with arbitrary arithmetic comparison predicates. The problem was studied by Klug [Klu88] for dense domains, such as real numbers, and by Levy et al. [LRU96] for integer numbers. van der Meyden proves the complexity given in Theorem (T2.5) [vdM92]. Furthermore, Levy & Sagiv describe a variation of the frozen facts algorithm that also works for queries with arithmetic comparisons and negation [LS93]. The difference is that they generate and test a *set of canonical databases* instead of only one.

Recursion.
Proving containment of a recursive query in a non-recursive query is shown to be double exponential in [CV92]. On the other hand, containment of a non-recursive query in a recursive query has the same complexity of containment between two conjunctive queries [RSUV89]. Finally, Shmueli proves that containment of a recursive query in another recursive query is undecidable [Shm93].

Disjunction.
Containment mappings may be used for queries containing disjunction without conditions. For two queries in normal disjunctive form the following holds:

$$q_1 \cup q_2 \cup \ldots \cup q_n \subseteq o_1 \cup o_2 \cup \ldots \cup o_m \leftrightarrow \forall q_i \ \exists o_j: \ q_i \subseteq o_j;$$

This is not true any more if conditions are allowed. Consider the following example:

```
u(x) ← rel(x), 10≤x, x≤20;
q1(x) ← rel(x), 10≤x, x≤15;
q2(x) ← rel(x), 15≤x, x≤20;
```

Clearly, $u \subseteq q_1 \cup q_2$ – any result computed by `u` will be computed either by q_1 or q_2. But $u \not\subseteq q_1$ and $u \not\subseteq q_2$. Hence, the existence of containment mapping is not a sufficient condition for queries including disjunction and conditions.

Negation.
[LS93] shows how the frozen facts algorithm can be extended to conjunctive queries including negation, and even for recursive queries with stratified negation [RU93].

Query containment as a view update problem.
Our tests for query containment depend on the search for a containment mapping. The frozen facts algorithm essentially searches a model for a database that could be a counterexample for containment. A third approach is suggested in [FTU98] (and also follows from [LS93]). The idea is to reduce the problem of query containment to the "*view update problem*". Consider a deductive database `D` and two queries q_1 and q_2. Insert into `D` the rule:

```
contained ← q1,!q2;
```

and try to refute `contained`. If the refutation succeeds, then there exist a that is computed by q_1 but not by q_2, and hence $q_1 \not\subseteq q_2$. This problem can be reformulated into: find inserts into or deletions from D such that the deduction, i.e., the view head, holds. This view update problem is currently an active research area because of its significance for updating database warehouses or in mobile computing.

Different semantics.

Our results are only valid for set semantics. Under *bag semantics*, two conjunctive queries are equivalent iff they are isomorphic [AHV95].

Sagiv introduces the notion of *uniform containment*, which differs from normal containment in that IDB predicates are considered to be materialised before the containment check [Sag88]. A sound, but not complete algorithm for uniform containment of DATALOG programs with stratified negation is presented in [ST96].

Finally, Aho et al. distinguish *weak from strong equivalence* [ASU79a]. Weak equivalence assumes that all relations are in fact views on one *universal relation* [Ull89]. No “dangling tuples” can exist. Consider the following queries:

```
q₁(x,y) ← rel₁(x,y),rel₂(y,z);
q₂(x,y) ← rel₁(x,y);
```

Under weak equivalence, both queries are equivalent, because for every value of y in relation rel_1, a corresponding value for y in relation rel_2 must exists. Under strong equivalence, these queries are not identical. We always consider strong equivalence in this thesis.

3. CONCEPTS OF MEDIATOR BASED INFORMATION SYSTEMS

While the previous chapter provided technical background, we now turn to the types of information systems in which these techniques shall be applied. In the following, we characterise *mediator based information systems* (MBIS). We describe the components of MBIS and discuss two topics in detail: strategies for the conceptual design of MBIS and languages for the specification of correspondences between heterogeneous schemas.

We start with a description of "*federated information systems*" (FIS) as the broadest class of integrated information systems. In Section 3.1 we present a catalogue of ten different criteria distinguishing different types of FIS. In Section 3.2 we discuss one of the ten criteria in more detail, namely development strategies for FIS. Two approaches can be distinguished: *top-down* and *bottom-up*. Top-down development starts from an independently designed global schema and relates data sources to this schema in a separate step. Bottom-up integration starts by analysing a given set of source schemas and infers the global schema from those using schema integration. Correspondingly, the predominating research topic in bottom-up developed systems is *schema integration*; the research focus in top-down developed systems is *query processing*. Both approaches seek solutions for closely related problems, i.e., bridging heterogeneity, but differ in how and when these problems are resolved.

In Section 3.3 we define our understanding of *mediator based information systems* (MBIS). Using our classification criteria, we characterise them as FIS that are tightly-coupled, read-only, and developed using the top-down strategy. We describe the two essential components of MBIS: *wrappers* and *mediators*. Wrappers deal with data model and technical heterogeneity, while mediators bridge structural and semantic heterogeneity.

Since MBIS are developed top-down, query processing faces the problem of translating queries that address in first place unrelated schemas: the mediator schema and a set of wrapper schemas. To guide this translation, MBIS must be aware of semantic correspondences between those schemas. The language used for specifying such correspondences is of great importance for MBIS, since it determines the types of conflicts that can be bridged and the types of queries that eventually can be answered. We distinguish between two classes of such languages: "*Local-as-View*" (LaV) and "*Global-as-View*" (GaV). We show properties of both classes. In particular, Section 3.4 shows examples where languages of either class fail. These failures motivated the design of the new correspondence specification language we shall introduce in Chapter 4.

3.1 From Federated to Mediator Based Information Systems

We consider as *federated information system* (FIS) any system for data integration that aims at providing a central point of access to a set of heterogeneous, autonomous, and distributed information systems. More precisely, the *data sources* that are to be integrated are potentially:

- *heterogeneous* wrt.
  - technical aspects: the operating system and hardware they run on, the protocol one has to use to access the source, access permissions, etc.,
  - interfaces: the language they support to query their content,
  - syntactic aspects: the format they use for representing data and query results,
  - data model,
  - semantic aspects: definition of terms, units of values, etc.,
  - structural aspects: the structure of the data representation,
- *autonomous* wrt. at what times they accept a query, from whom they accept a query, when they start a query execution, etc.,
- *worldwide distributed.*

Taking part in an integrating system requires that data sources compromise their autonomy at least to some degree. They must be accessible electronically, for instance, over the Internet, they must provide some description of their data, such as schemas and documentation, and they must be prepared to answer queries in some form, though possibly with delay.

3.1.1 Classification Criteria

Table 3 shows ten criteria that can be used to distinguish different types of FIS (see also [BKLW99]). We shortly illustrate three of them. The main purpose is it to pave the road for the characterisation of MBIS given in Section 3.3.

- A prominent criterion is whether or not the FIS offers a global schema. FIS without global schema are called *loosely coupled*, FIS with global schema are called *tightly coupled* [SL90]). In tightly coupled systems, users see only one schema and do not have to bother with different sources and their structures. Hence, a tightly coupled FIS inherently offers location, language, and schema transparency. In contrast, loosely coupled systems usually only offer language transparency: a user does not need to learn the query language of each source, but he still has to know their schemas.
- The degree of *semantic integration* that is offered by the FIS is more difficult to capture. A no-integration approach only collects results from sources, but does not relate them to other results. In general, this is assumed to be insufficient. A higher degree of integration is reached if results from different sources are merged if they correspond to the same real-world entities. This however requires the identification of identical objects, which can be difficult [Ken91]. Clearly, the usefulness of a system increases the better results are integrated, since it removes a great burden from the user and prevents him or her from drowning in “information overload”.
- Integrating systems that build on permanent *materialisation* of source data are commonly known as “*data warehouse*”, and attracted considerably interest in the last decade [HGM95; Wid95]. A warehouse has the advantage of being very fast in answering a query once the materialisation is complete, but it is always threatened by storing outdated data [GM95]. Furthermore, it suffers from the large amount of storage it requires to store the

data of all data sources. *Virtual integration* on the other hand requires almost no disk space and always retrieves the most current data - but query processing may be time consuming, especially in unreliable wide area networks such as the web.

	Criterion:	**Description:**
1	Integration of structured, semistructured or unstructured data sources	Unstructured data sources are, e.g., document collections; semistructured sources cannot be fully characterised through a schema, but have a structure [AMM97; Bun97].
2	Tight versus loose integration	Does the FIS offer a uniform global schema [SL90]?
3	Global data model	Relational, object-oriented, semantic, etc. [SCGS91]
4	Supported degree of semantic integration	To what extent are results from different sources integrated [Wie94]?
5	Level of transparency	Do users need to be aware of the location of sources, their content, their data model, their query language, their schemas, etc.
6	Query paradigm	Structured queries or information retrieval techniques.
7	Bottom-up versus top-down strategy	See Section 3.2.
8	Virtual or materialised integration	Are sources (partly) pre-materialised in the FIS, or are queries translated at query time [Wid95].
9	Read-only or read-and-write access	Functionality.
10	Requested access mechanisms	Are sources allowed to provide only restricted query mechanisms, e.g., binding patterns. [GMY99]

Table 3. Criteria for the characterisation of FIS.

3.1.2 Types of Federated Information Systems

Using the criteria given in Table 3, we precisely define our understanding of several “buzzwords” in data integration. A graphical representation of our classification may be found in Figure 14. The main purpose of this section is to highlight the differences between a MBIS and other approaches to the integration of information systems.

Distributed databases.
A distributed database system is a FIS containing only homogeneous data sources. Typically, distributed databases are *by design* distributed for performance or security reasons. Heterogeneity is suppressed as much as possible [OV99]. A distributed database is administered from a central point, i.e., components are not autonomous.

Most database vendors today offer distributed options for their RDBMS. In addition, some vendors offer *gateways* to integrate data sources running on a different RDBMS. We discuss gateways in more detail in Section 6.1.1.

Distributed databases do not provide a global schema. Location transparency is achieved through synonyms: By defining a synonym for a remote view, the administrator makes this view accessible as if it were a local relation. Semantic integration is not addressed specifically, but, in general, most operations of the query language will be supported, including aggregation. Write access is supported through special transaction protocols, such as 2-Phase Commit. Most products are based on relational databases. Criteria 7 and 8 do not apply since sources are not autonomous.

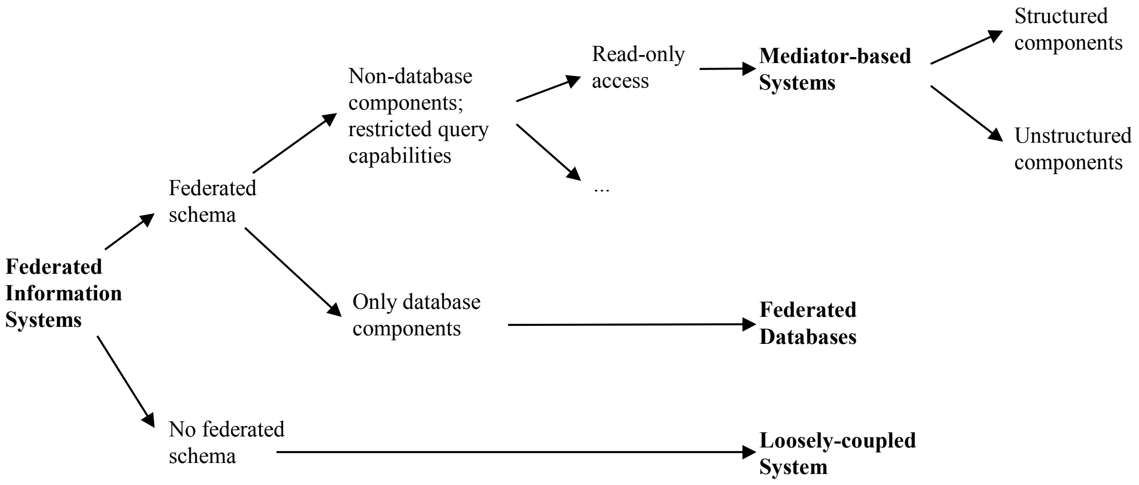


Figure 14. Characterisation of different architecture for data integration.

Loosely coupled systems: multidatabase query languages.

Loosely coupled systems offer a uniform multidatabase query language to access data in different sources , but do not have a global schema. Data sources must be structured and support unrestricted query access.

Examples of multidatabase query languages are MSQL [LMR90] and UniSQL [KCGS95]. Both are extensions of SQL. Syntactically, the essential add-on is the ability to prefix relation names with the name of the data source that stores this relation. The query processor uses this prefix to decompose the query and to route subqueries to appropriate data source [YM98]. The query processor also translates subqueries - but only handles syntactic differences, for instance between two different SQL dialects, and does not treat semantic or structural issues. Therefore, many integration tasks that one could expect to be transparently managed by the integration system are left to the user: semantic integration, query formulation, knowledge about content and structure of data sources, etc.

Note that the difference between loosely and tightly coupled systems is not clear. For instance, in the multidatabase query language CPL (Collection Programming Language, [DOTW97]), users may define *integrating views*, called “macros”, and use them as if they were global relations. If such views are made available to all users of the system and are defined in a careful way, their union amounts to a kind of global schema (see Figure 15).

Federated database systems:

The most prominent approach to database integration are federated database systems (FDBS) [HM85; SL90]. According to [SL90], a FDBS conceptually comprises five layers (see Figure 16). Dotted lines indicate where the different types of heterogeneity are resolved):

- The first layer consists of the schemas of the data sources.
- The second layer contains the schemas of the first layer transformed into the data model of the FDBS, the *canonical data model.*
- The third layer comprises those subsets of the component schemas that shall be integrated in the FDBS.
- The fourth layer consists of federated schemas that are composed from the export schemas. It is neither required that there is only one federated schema nor that federated schemas integrate export schemas completely. In most publications, federated schemas are derived by schema integration [Sch98].

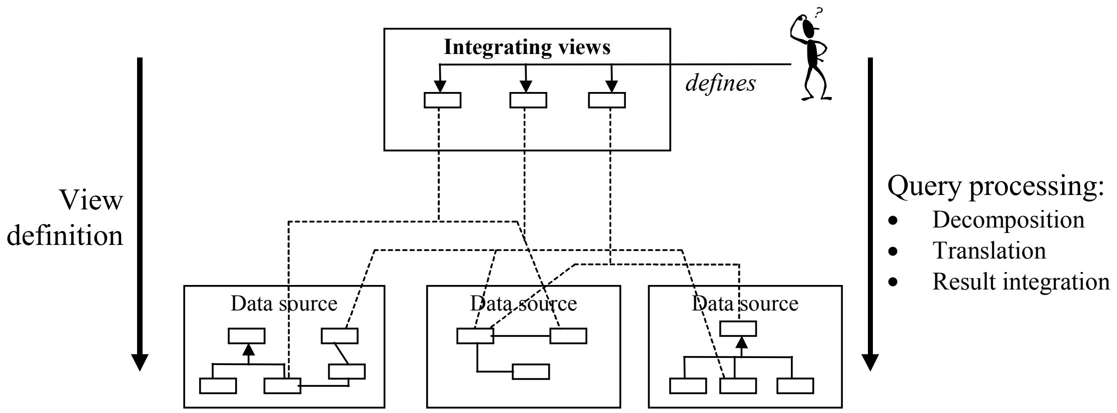


Figure 15. Loosely coupled integration system with integrated views defined by the user.

- The fifth layer defines application-dependent views on federated schemas. This layer is often omitted since its creation does not pose other problems than the creation of external schemas in a centralised RDBMS.

FDBS consider only full-fledged RDBMS as data sources. They provide a global schema and therefore offer a high level of transparency. Usually, write access is addressed but leaves many open research questions [Con97].

Mediator based information systems.
Mediator based information systems are tightly coupled, read-only, top-down developed and virtual systems that provide full transparency. Sources may be restricted in the types of access they support. The degree of semantic integration is not determined. MBIS are explained in detail in Section 3.3.

3.2 Development Strategies

Ideally, a centralised database is developed in the following manner: First, an analysis of the problem at hand leads to a requirements specification. Those requirements are expressed as views. Since different requirements from different users of the planned application lead to different views, a homogeneous database schema does not follow immediately. It needs to be derived from the views through *view integration* [NEL86].

Developing a tightly coupled FIS also faces the problem of finding a suitable global schema. However, in contrast to central database, FIS are not designed from scratch, but are built on top of existing systems. Taking this property into account, we identify two strategies for the conceptual design of the global schema:

- *Bottom-up*. The schemas of data sources are considered as views that must be integrated into one uniform, homogeneous, global schema. Schema integration is more complex than view integration since (1) possibly, it has to integrate very large schemas instead of single views, and (2) it has to consider the actual data in the sources and not only their schemas.

- *Top-down*: We design the global schema in exactly the same way as we would design a central database schema. This means that we start from global requirements, resulting in global views, and end with a homogeneous schema. Source schemas are not considered in this process, but have to be related to the global schema in a subsequent step.

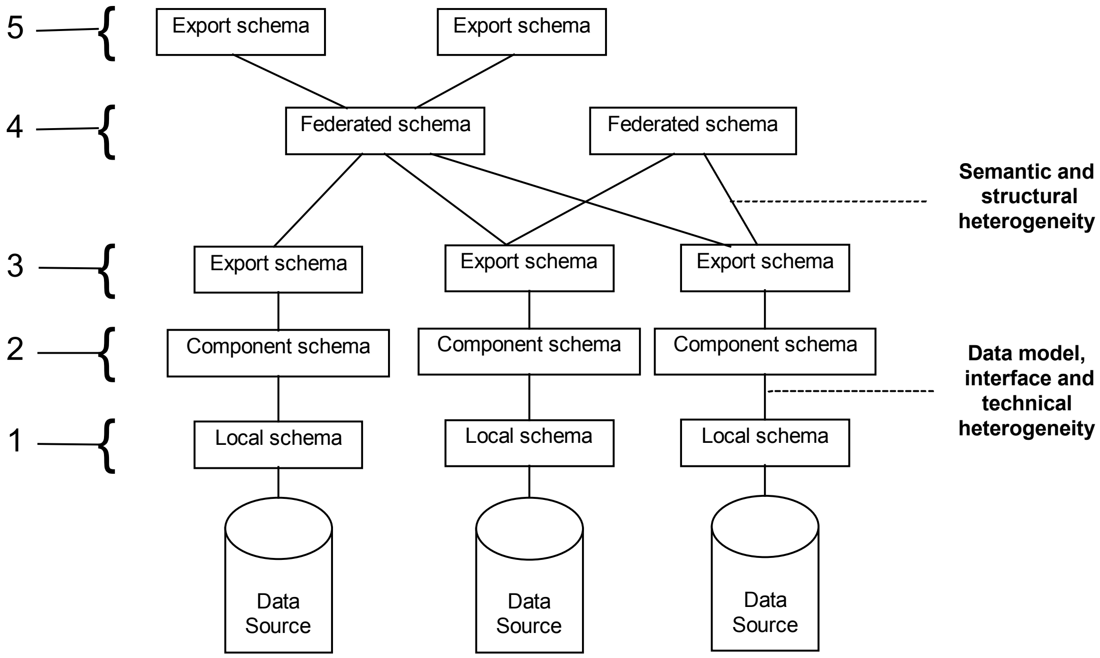


Figure 16. The 5-layer architecture of federated database systems.

The difference between both strategies is illustrated in Figure 17. In a top-down approach, the designer specifies *vertical schema correspondences*, i.e., correspondences between the global and the source schemas. In a bottom-up approach the designer defines *horizontal schema correspondences*, i.e., correspondences between different source schemas, which are used to derive the global schema. Since query processing requires the existence of vertical correspondences, those must be derived from the horizontal correspondences in an extra step. Surprisingly, many publications about schema integration do not even mention this requirement (examples are [SPD92; Sch98]).

The choice which strategy to use often cannot be chosen freely but is determined by the situation in which the new development is pursued. If, for instance, global write access is required, one has to consider source schemas carefully in the design of the system. This can be realised more comfortably with the bottom-up approach. On the other hand, if sources evolve frequently or often enter or leave the federation, then the top-down approach is clearly superior because it offers higher flexibility (see also Section 6.3).

3.2.1 Top-Down

The starting point for a top-down development of a FIS is a new information requirement that can be satisfied by some known data sources. Since accessing the sources in isolation is time-consuming and inefficient, the requirement for integrated access emerges.

Constructing integrated access starts from analysing the new information need through a requirements analysis. This analysis results in the specification of a schema that entails all necessary information structures to satisfy the requirements. Although it is certainly necessary to already consider at this stage whether there are any data sources that can provide the actual data at run-time, their structures, semantics and languages are in first place ignored.

Once the global schema is completed, data sources are incrementally plugged into the system. Each source is considered in isolation (see Figure 17 (b)). Its content is described relative to the global schema using a correspondence specification language (see Section 3.4). Correspondences are later used to decompose and translate queries.

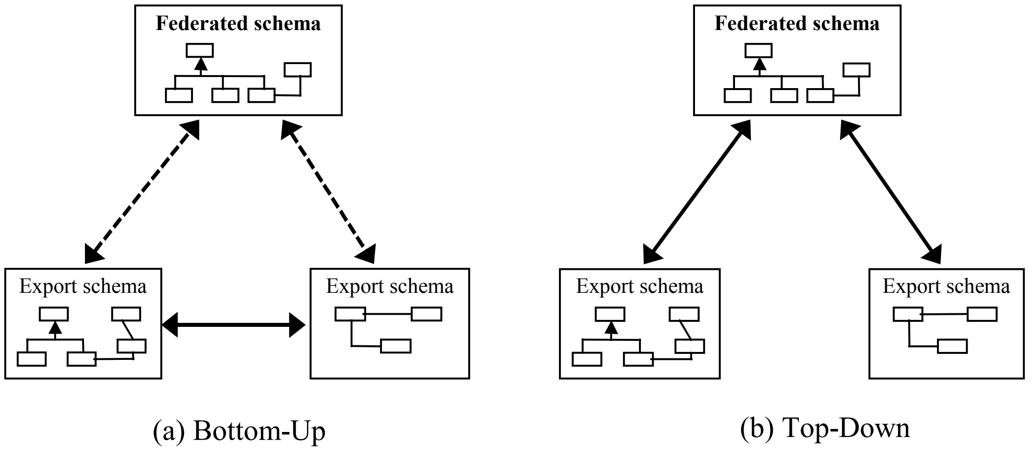


Figure 17. Design strategies in FIS. (a) Bottom-up. (b) Top down.
Dotted lines stand for derived correspondences, solid lines for specified correspondences.

Considering sources in isolation has advantages regarding the maintainability of the system [Les98b] (see also Section 6.3). It particularly allows for an easy plug-in / plug-out of data sources. Furthermore, changes in the structure of data sources can mostly be handled by solely changing the correspondence descriptions, keeping the global schema unaffected.

Top-down approaches however lead to a less coherent system since they introduce only weak bindings between sources and the FIS. Actually, in a web context, many systems are built such that sources are not even aware of their integration into a greater context. Those sources cannot inform the integrating system of any changes in their structure.

Typical scenarios that call for a top-down development are the following:

- Companies that develop global information services as products, for instance bargain finders, stock market data providers, or book finders.
- *Standards* for data access are always developed top-down. Examples include the ISO norm STEP [Sau98; SM98] for the automobile industry, and the domain specific services developed under the auspices of the *object management group* (OMG). An example for the latter is the emerging standard for access to genome maps described in [BLL+99; Muel99]. Standard usually only defines the access interfaces (OMG) or data structures (STEP); the mapping from the standard to an existing information system has to be developed independently.
- Integration approaches that base on *common ontologies* are also inherently top-down . The aim of an ontology is to give a comprehensive and coherent specification of the concepts, terms and constraints that are present in a certain domain [Gru93; GG95]. Once such an ontology is available at the FIS, the content of data sources are described in terms of the ontology, which makes them available through a common vocabulary [AHK96]. In a way, the ontology is used as a global standard.

3.2.2 Bottom-Up

A bottom-up approach to the construction of a FIS starts with the requirement to provide integrated access to a given set of data sources. This set is pre-defined and is not expected to change frequently. The scope of the FIS is essentially the “semantic union” of the content of these sources. The first problem is therefore the computation of this union, and its representation in a schema. This is achieved by *schema integration* (see Figure 17 (a)). We cannot describe the many facets of and approaches to schema integration in detail in this work. Interested readers are referred to [BLN86; PBE95; Con97; Sch98]. We include a short summary, following [BLN86].

Schema integration proceeds in four steps. In the first step (pre-integration) all schemas are transformed into a common data model. In the second step (schema comparison) the schemas are compared to each other to find and clearly identify the existing correspondences and conflicts. Conflicts that can be resolved by restructuring source schemas are resolved in the third step (schema conforming). The fourth step finally merges all schemas into one integrated schema. The resulting schema should be:

- complete, i.e., it should completely cover the source schemas,
- correct, i.e., it should be able to integrate data from any source without semantic distortion or loss,
- minimal, i.e., it should be as small as possible and
- understandable, i.e., it should be readable for human beings.

Although many authors investigate automatic approaches to schema integration, currently all methods are a mix of automatic and manual steps. A particular problem is that complexity and size of the resulting schema tends to grow with the degree of automation [Sch98]. However, large and complex schemas are difficult to understand, difficult to maintain, and difficult to query. A proper modularization of an integrated schema remains an open research problem.

Another problem is the static nature of schema integration: If any of the sources changes its schema, or a new source shall be integrated, the process must be repeated. Only few projects analysed the propagation of changes into an existing integrated schema, and they can only deal with a handful of special cases [Mot98; Kol99]. For instance, if an attribute that is uniquely assigned to a class and only occurs in one source is deleted, then this change can be propagated to the integrated schema by also deleting this attribute there.

There are cases where a bottom-up approach is the right choice. Especially if integration is performed as first step towards *migration*, i.e., if the FIS is constructed to support applications using the integrated schema while still keeping old applications alive that use the original schemas, then the four requirements mentioned above are vital.

3.3 Architecture of a Mediator Based Information System

A *mediator based information system* (MBIS) is a tightly integrated FIS that provides read-only access to a heterogeneous and dynamically changing set of data sources. The major components of a MBIS are *wrappers*, *mediators* , and the data sources themselves (see Figure 18). A *mediator* translates user queries into equivalent combinations of queries against wrappers, therein treating semantic and structural problems. Other types of heterogeneity, concerning for instance communication protocols or syntactical representation of data, must be resolved before a source can be integrated into a MBIS. To this end, sources are hidden underneath *wrappers*, which offer an interface that conforms to the requirements of mediators.

MBIS can integrate semistructured, unstructured, or structured data [AHK+95]. In this work we only consider structured MBIS and use the term “MBIS” as synonym for “structured MBIS”. In *structured MBIS*, mediators have a schema and only deal with structured data sources. User queries are posed against mediator schemas, but the data itself does not exist physically at one central place. Instead, the mediator collects appropriate information at query-time from the data sources.

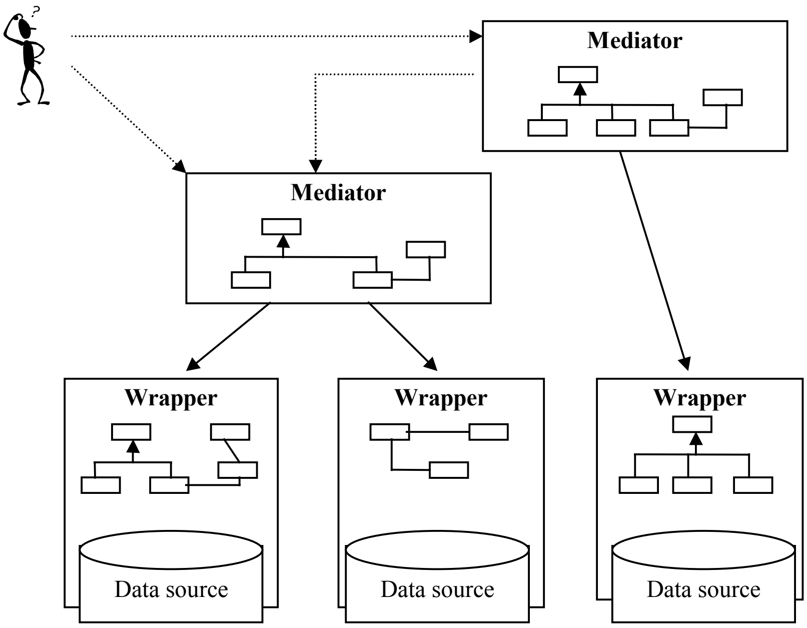


Figure 18. Mediator based information systems architecture.
Queries against mediator schemas (*user queries*) are dotted, queries against wrapper schemas (*wrapper queries*) are lines.

MBIS are developed top-down. Research in MBIS therefore concentrates on query processing. MBIS explicitly aim at integrating sources that are not accessible through a standard query language. Therefore, the query processor must be able to cope with restricted query capabilities.

A MBIS may comprise more than one mediator (see Figure 18). In this case, each mediator has its own schema, and a mediator may use other mediators as data sources. This implies that semantic and structural conflicts can appear at each level. For instance, semantic conflicts can occur between a wrapper interface and a mediator schema, and between each mediator that uses another mediator as wrapper. However, since mediators in general support the same type of interface as wrappers, the same mechanism should be applicable at each level.

We distinguish between *homogeneous MBIS* and *heterogeneous MBIS*. In a homogeneous MBIS, all mediators use the same data model and query language. Therefore, a wrapper is usable by any mediator of the MBIS. Technical and data model heterogeneity only appears at the bottom-most level, i.e., just above the physical access to data sources. In contrast, heterogeneous MBIS host different types of mediators. Accordingly, wrappers must provide different interfaces for different mediators, and mediators might need to be wrapped to be usable by other mediators.

In this work, we only consider homogeneous MBIS with one mediator based on the relational data model. We therefore simply speak of "the mediator" of a MBIS. However, our results apply equally well to homogeneous (relational) MBIS with more than one mediator.

3.3.1 Wrappers in MBIS

A wrapper transforms data represented in the data model of its "wrapped" data source into a representation in the data model of the mediator. Furthermore, it translates queries from the query language of the mediator into "queries" executable by the source. Wrappers are capable of describing their query capabilities and of passing this information to the mediator. Usually, it is within the responsibility of a mediator to generate only queries that are executable by a wrapper.

There are two parts in a wrapper that are of special interest for the mediator:

- Each wrapper has a (relational) *wrapper schema*. Data produced by the wrapper adheres to this schema, i.e., all data is structured according to this schema.
- Each wrapper is able to answer at least some explicitly *predefined queries* against its schema, resulting in a stream of tuples consisting of discrete attribute values.

As we shall see in Chapter 5, a mediator only uses the set of predefined queries to answer user queries. Wrapper schemas are mainly important for the selection of such queries. We shall give guidelines for the construction of wrappers in Section 6.1. In the meantime, we only highlight some points:

- Wrappers are source-specific. Their schema is chosen based on the data stored in the wrapped source and the interface that the wrapper uses to access the source. It is completely independent from the mediator schema. If a mediator shall use a wrapper, then semantic correspondences between the two schemas must be specified explicitly to enable the translation of queries.
- The physical location of a wrapper is not determined. Wrappers may be built at the site (host) of the data source, or at the site of the mediator, or at a third site. The mediator may, but need not, exploit this knowledge to achieve better performance by reducing network traffic.
- Although wrappers are source-specific, they often can be reused. For instance, a wrapper for a RDBMS can be used for other sources that uses the same RDBMS. Only the export schema and possible the gateway technology needs to be changed (see Section 6.1.1).
- A data source may be accessed through different wrappers in one MBIS. Reasons for this could be:
  - To exploit specific interfaces. Different wrappers may use different access mechanisms offering special features. For instance, a data source might provide some types of queries through a CORBA interface, and other queries through a form based web interface.
  - To increase maintainability through separation of concerns. Different wrappers for different parts of a source are advantageous if a source exports very complex data structures.

We now define a formal abstraction of a wrapper. In real-life, a wrapper is of course a piece of software, not a 3-tuple.

Definition (D3.1)-(D3.2) (Wrapper, executable wrapper query).

(D3.1) A *wrapper* `W` is 3-tuple $W = (\Sigma, \Omega, \chi)$. The components of `W` are:

- an export schema Σ,
- a set Ω of queries against Σ that can be executed by `W`, and
- the data source χ wrapped by `W`.

(D3.2) Let $W = (\Sigma, \Omega, \chi)$. A query `q` against Σ is *executable by W* iff:

- $q \in \Omega$, or
- $q = <q', C>$ where $q' \in \Omega$ and `C` is a set of conditions on variables of q' such that $\forall\ c \in C$: $sym(c) \subseteq export(q')$.

■

Remarks:

- Ω may intuitively be understood as a set of templates for queries executable by W. The actual set of executable queries is infinite as soon as Ω is non-empty, since there exists an infinite set of conditions to each query in Ω.
- Definition (D3.2) does not imply *how* an executable wrapper query is actually computed. In particular we do not request that the source itself can execute all necessary operations implied by conditions in C. For instance, if an executable wrapper query exports a variable v and C contains a condition c on v, then c may:
  - be pushed from the mediator through the wrapper into the source, or
  - be enforced inside the wrapper, or
  - be enforced inside the mediator after submitting the query without c.

  We do not determine which of those strategies shall be pursued if more than one is possible. In general it is considered to be advantageous to push conditions as much as possible into sources, since this firstly reduces the amount of data that has to be transmitted through the network. On the other hand, it may increase the amount of transmitted data in the long run since less data can be cached [HKU99].

■

Example 3.1.

Consider a wrapper W which is capable of executing the following query:

```
q(cn,cl) ← clone(cid,cn,ct,cl);
```

From this query, we derive, for instance, the following statements:

- q itself is executable.
- The query q_1(cn,cl) ← clone(cid,cn,ct,cl),cl<100 is executable.
- The query q_2(cn) ← clone(cid,cid,cn,cn,cl) is executable.
- The query q_3(cn,cl) ← clone(cid,cn,ct,cl),ct='YAC' is not executable because ct is not exported.

■

3.3.2 Mediators in MBIS

The main task of mediators is the translation of queries against their mediator schema into sets of executable wrapper queries. This process, i.e., query planning, is the focus of our work. Through query planning, a mediator must overcome structural and semantic heterogeneity between its schema and the available wrapper schemas. In this section we give a brief introduction into the problems and tasks that query planning faces.

Consider a user query u against the schema of a mediator. In general, answering u requires the combination of several wrapper queries. We call such a combination a *plan*; finding plans is called *planning*. A *correct plan* p for a user query u is a set of wrapper queries whose results, if combined appropriately, compute only correct answers to u. This does not imply that p computes all answers to u. The intuitive meaning of a "correct plan" is the following: A plan for a user query u is correct if it returns only tuples that conform to the intension of u, i.e., only tuples corresponding to real-world objects represented by u. This definition is usually put into operation by considering a plan as correct for u if it only produces tuples that a human would consider as correct answers to u. To judge the correctness of a plan, a human must fully understand the query, the mediator schema, and the semantics of the wrapper queries in the plan.

The situation is simpler in a centralised database. Consider a centralised database D and a query u against D. In this case, the plan for answering u is the query itself:

- The intension of u, i.e., the "intended" meaning of u, is determined by the intensions of the literals in u and any additional conditions.
- The extension of u is clearly defined, since a database D has exactly one instance at each moment in time, and this instance defines a unique extension for each relation that appears in u.

Answering a query is in this sense completely free of considering semantics. Intensions are encoded syntactically in the names of relations. A relational query may be understood as a program that can be executed directly[5]. The situation is completely different if we consider the distributed and heterogeneous nature of a MBIS. Since we pursue virtual integration, there is no instance for the mediator schema. A mediator is not a database, and the relations of a mediator schema have no extension.

To obtain a computable way of testing correctness, we have to bridge the gap between the different schemas that play a role inside a mediator: Every user query addresses the mediator schema, but executable queries always address wrapper schemas. This problem is approached by specifying *correspondences* between elements of wrapper schemas and elements of the mediator schema.

There exist different types of the correspondence. Each correspondence has an extensional and an intensional aspect, and can specify the connected elements to be either:

- disjoint, i.e., the elements have no overlap in their intension / extension,
- overlapping, i.e., the elements have some intensional /extensional overlap,
- equivalent, i.e., the elements have identical intension /extension, or
- subsuming, i.e., the intension /extension of one element is contained in the intension / extension of the other.

We shall approach this problem in Chapter 4 by introducing correspondences between queries, which intensionally specify equivalence or subsumption, and extensionally subsumption. More precisely, we shall search for each executable wrapper query q_w *a corresponding query* q_m against the mediator schema. q_m is corresponding to q_w if they either have the same intension, or if the intension of q_w is a subset of the intension of q_m (see Definition (D2.11)). In both cases we can be sure that all tuples computed by q_w are correct tuples for q_m. Consequently, the *extension* of q_w must be a subset of the extension of q_m.

In general, there exist more than one wrapper query corresponding to one mediator query. If there are three bookstores selling the same types of books, then the mediator relation book will have three corresponding queries addressing each a different data source. In contrast, the corresponding mediator query for a wrapper query will in general be unique. Assume there were two different mediator queries q_{m1}, q_{m2} corresponding to the same wrapper query. This implies that the intension of q_{m1} and q_{m2} are identical, and therefore the mediator schema is redundant. Redundancy cannot be completely avoided. For instance, if a wrapper query produces only the identifier of a certain class of objects, then corresponding attributes will be present in each relation of the mediator schema that is connected to this class through a relationship. However, such queries occur rarely in real-life applications.

Now, we formally define a mediator. Again this is a formal and idealised abstraction. In real-life, a mediator is a piece of software, not a 3-tuple.

[5] Usually there are many ways to execute a query. Classical query optimisation is about finding the cheapest.

Definition (D3.3) (Mediator).

(D3.3) A *mediator* M is a 3-tuple $M = (\Sigma, \Psi, \Gamma)$. The components of M are:

- the mediator schema Σ,
- a set Ψ of wrappers used by M, and
- a set Γ of correspondences.

■

Remarks:
In a MBIS with mediator $M = (\Sigma, \Psi, \Gamma)$, we distinguish three types of queries:

- Queries against Σ that are posed by a user are *user queries*. Answering user queries is the sole task of a mediator.
- Other queries against Σ are *mediator queries*. We shall use mediator queries for specifying schema correspondences in Chapter 4.
- Queries against the export schema of a wrapper $W \in \Psi$ are *wrapper queries*. In particular, we are interested in *executable wrapper queries* (see previous section).

■

3.4 Correspondence Specification Languages

In the previous section we described why mediators depend on correspondences between elements of different schemas. Such correspondences must be expressed in some language. We call such languages *correspondence specification language* (CSL). The choice of a CSL is very important for a MBIS, since it determines the types of schema conflicts that can be bridged. The integration of data sources that introduce a conflict that is not expressible in the chosen CSL is impossible without adapting either the schema of the problematic wrapper or the mediator schema [Les98b]. Schema adaptation breaks either the autonomy of the mediator or the autonomy of the wrapper, and hence impedes independent evolution.

Different CSLs, having different expressive powers, yield different algorithms for query planning. It is not possible to allow arbitrary rich languages, since planning may become undecidable. For instance, it is undecidable whether a recursive DATALOG program is equivalent to another recursive DATALOG program (see Section 2.4). If we allow global, recursive queries and correspondences with recursive mediator queries, we cannot decide if a plan computes only correct tuples for a user query.

Hull distinguishes two classes of CSL [Hull97]: Global-as-View (GaV) and Local-as-View (LaV). In this section we describe both approaches and show their shortcomings. Essentially, LaV relates a mediator query to a wrapper relation, while GaV relates a wrapper query to a mediator relation. Many projects use much simpler correspondences. For instance, the systems described in [CL93; MKSI96] allow only the specification of correspondences between single relations.

We highlight the properties of both approaches by examples. We assume a mediator M with schema Σ as given in Table 1, and two wrappers W_1, W_2, whose export schemas Σ_1 and Σ_2 are described in Table 4. W_1 stores only data about one particular type of clones (PACs) and is therein more restrictive than M. On the other hand it is more general, since it also has mapping data from other species than humans. W_2 is very simple. It uses object names as keys.

	Export schema	**Description**

Σ_1	c_map(mid,mapname,species,chromosome);
	PACS(cid,mid,clonename,clonelength);
		W_1 stores PAC mapping data from mammals such as human and mouse.

Σ_2	position(clonename,genename);	W_2 stores a list of gene names and clone names which contain them.

Table 4. Two fictive wrappers of a MBIS for human mapping data.

3.4.1 Global-as-View

The Global-as-View (GaV) approach allows the specification of correspondences between single relations of the mediator schema and views on wrapper schemas (see Figure 19 (a)). Possible conflicts must be resolved if more than one view corresponds to a mediator relation, indicating that the extension of this relation is scattered over many sources. Two solutions are possible:

1. We specify as many views correspondences as necessary, and the mediator uses a generic method for combining the results computed by the different views. This could be based on user-defined rules for object equivalence.
2. Only one correspondence is specified, addressing all relevant views at once. In this case, each correspondence may define its own way of how results from different wrappers are combined.

For instance, [PAG96] uses a method called ”*object fusion*”, which assigns ”semantic” keys to each object in each view. The mediator combines data that is getting assigned the same key at run-time, independently from which wrapper the data stems (method 1). [PAG96] calls method 2 ”*fusion by outer-joins*”.

GaV is a direct extension of a central database engine to the distributed case. It is used in many projects, such as TSIMMIS [GMP+97], DISCO [TRV96] or IRO-DB [FGL+98]. Query planning is straight-forward: given a user query u, each literal of u is replaced with its corresponding view definition (2), respectively the union of its corresponding view definitions (1). This expanded query is then decomposed and the resulting subqueries are shipped to the wrappers.

We now try to specify correspondences between Σ and Σ_1, Σ_2, respectively, using GaV rules. For clarity, we prefix relation names with their schema. We use the first of the two proposed methods for the integration of overlapping sources. Attributes for which no values are present in either schema are noted as “–“. We ignore potential problems that arise if mediator relations are not fully specified by a rule, i.e., if attributes are missing.

Our aim is to show that there are situations that cannot be handled by a GaV specification.

Example 3.2.
Defining Σ.map as a view on Σ_1.c_maps requires a condition to restrict the results to human maps. Σ.clonelocation, projected to its first two attributes, is intensionally equivalent to Σ_1.PACS:

Σ.map(mid,mn,-,-,ch) $\leftarrow$ Σ_1.c_map(mid,mn,sp,ch),sp='human';
Σ.clonelocation(mid,cid,-) $\leftarrow$ Σ_1.PACS(cid,mid,-,-);

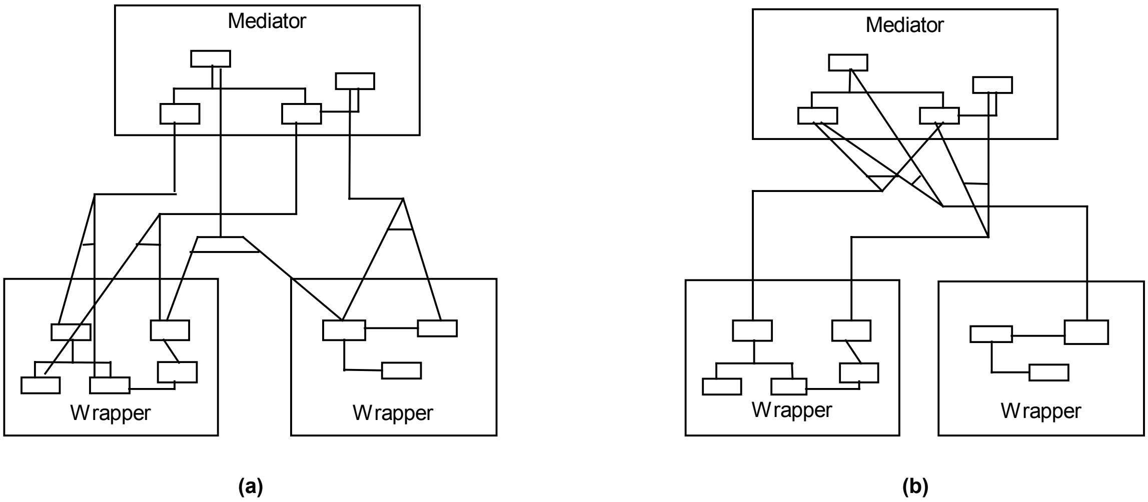


**Figure 19. Correspondence specification languages.
(a) Global-as-View. (b) Local-as-View. Angles indicate views.**

It is not possible to define Σ.`clone` in the same way. Σ.`clone` is more general than Σ_1`.PACS`. Defining the following view:

```
Σ.clone(cid,cn,-,cl) ← Σ1.PACS(cid,-,cn,cl);
```

could lead to erroneous results. A user query asking for clones of any type will still be answered correctly, since PACs are clones; but imagine a user query requesting data about YACs. The answer to this query must not include data from Σ_1`.PACS`. But exactly this happens if the rule is defined as above.

To see another problem, consider a user query requesting all PACs that are placed on the map with mid=500, together with the clone length:

```
q(cn,cl) ← Σ.clonelocation(mid,cid,-),Σ.clone(cid,cn,ct,cl),ct='PAC',
   mid=500;
```

`q` can directly be answered by using only Σ_1`.PACS`. However, this cannot be expressed if each relation of Σ is defined as a view , *in isolation* of all other relations. The mediator hence always must executes two queries to answer `q`. We shall see in Chapter 4 how we can describe Σ_1`.PACS` with only one rule such that answering `q` requires the execution of only one query.

Using GaV, we cannot describe Σ_2. The problem is that the information about genes and their containment in clones is spread over three different relations in the mediator schema, but stored in one relation in the wrapper. To formulate a user query for clones containing genes, we must join three tables:

```
q(cn,gn) ← Σ.clone(cid,cn,-,-),Σ.contains(cid,gid), Σ.gene(gid,gn,-);
```

Although the requested information is exactly what is stored in Σ_2`.position` stores, we cannot translate `q` using GaV rules. We cannot give a correspondence for Σ`.contains`. Σ_2 has no such relation and no such attributes.

■

3.4.2 Local-as-View

The Local-as-View approach (LaV) allows correspondences that relate a single relation of a wrapper schema to a view on the mediator schema (see Figure 19 (b)). This is exactly contrary to the GaV approach. In LaV, every mediator relation may appear in many views corresponding to different wrapper relations. Every such correspondence contributes to the total extension of the mediator relation.

LaV is a relatively new approach. It was first described in [TSI94], and is, for instance, used in the Information Manifold [LRO96b; LRO96a] and in Infomaster [DG97; GKD97]. It has its strength in environments with frequent evolution of sources, such as the web. However, query planning with LaV rules is considerable more complex than with GaV rules.

We write LaV rules with head and body in reverse order: "`body ← head`". We do this to be consistent with QCA (see Chapter 4). This notion emphasises the "flow of data", i.e., data is produced through data sources, represented as a tuple and then "pumped" into the mediator relations.

Example 3.3.
We first show that the problematic cases of the GaV approach can be solved using LaV. For instance, we can define Σ_1`.PACS` as a view on `Σ.clone` with a condition on `clonetype`:

```
Σ.clone(cid,cn,ct,cl),ct='PAC' ← Σ1.PACS(cid,-,cn,cl);
```

We can also avoid the splitting of rules for `Σ.clone` and `Σ.clonelocation` by using a join in the mediator query:

```
Σ.clonelocation(mid,cid,-),Σ.clone(cid,cn,ct,-),ct='PAC' ←
   Σ1.PACS(cid,mid,cn,cl);
```

However, we step into some subtle problems here. Consider a user query asking for clones. Results obtained from Σ_1 may be clones of any species; there is no guarantee that they are human clones. But non-human clones are intensionally wrong in the mediator schema. The same problem occurs if we try to describe Σ_1`.c_map`. The relation Σ_1`.c_map` stores tuples of all species and is hence not intensionally equivalent to or subsumed by Σ.map. To establish equivalence, we would need to add a condition on the wrapper site of the rule. We see that the problems with LaV are somehow symmetric to those of GaV. We discuss this "symmetry" between both approaches in the next section.

Using LaV, it is straight-forward to describe Σ_2:

```
Σ.clone(cid,cn,-,-),Σ.contains(cid,gid),Σ.gene(gid,gn,-) ←
   Σ2.position(cn,gn);
```

■

Query planning with LaV is the central theme of this work. Therefore, we give some examples that highlight the problems LaV planning faces. Of course, we cannot expect that a user query matches directly with the body of a LaV rule. User queries instead might use only parts of bodies of rules or require the combination of rules.

Example 3.4.
For the moment, we ignore the problem regarding species in Σ_1. Consider the following user queries:

```
q1(cn) ← Σ.clone(-,cn,ct,-), ct='PAC';
q2(cn) ← Σ.clone(-,cn,ct,-), ct='YAC';
q3(gn,cn) ← Σ.map(mid,-,-,-,ch),Σ.clonelocation(mid,cid,-),
   Σ.clone(cid,cn,-,-),Σ.contains(cid,gid),Σ.gene(gid,gn,-), ch='X';
```

q_1 asks for all PACs, q_2 for all YACs, q_3 asks for all genes of the X chromosomes. Σ_1 can be used to answer q_1, but not to answer q_2, since the condition `ct = 'YAC'` cannot be fulfilled by the data in this source. Such contradictions must be detected.

q_3 cannot be answered by one single wrapper query but only by a combination of wrapper queries. To see this, we examine the following conjunction of wrapper relations :

```
Σ1.c_map(mid,mn,-,ch),Σ1.PACS(cid,mid,cn,-),Σ2.position(cn,gn);
```

where each relation corresponds to a view on Σ as defined before. Hence, we can derive the semantics of this query in terms of the mediator schema by replacing the wrapper relations with their corresponding mediator queries. This yields (with the corresponding wrapper relation in brackets):

```
{Σ1.c_map} Σ.map(mid,mn,-,-,ch),
{Σ1.PACS} Σ.clonelocation(mid,cid,-),Σ.clone(cid,cn,ct,-),ct='PAC',
{Σ2.position} Σ.clone(cid,cn,-,-),Σ.contains(cid,gid),Σ.gene(gid,gn,-);
```

It is straight-forward to see that this *expanded query* is contained in q_3. There is no problem with the missing `ID`'s in Σ_2 since they are not requested by q_3. We conclude that the three wrapper relations, combined appropriately, compute correct answers for q_3.

■

3.4.3 Comparison

The differences between the GaV and the LaV approach are best described by analysing their respective perception of the integration problem.

In LaV, the mediator schema is assumed to be a stable structure, and sources are added and deleted as they appear to be useful for the mediator. This strategy is very well suited for the top-down development of MBIS. Constructing a LaV system necessarily starts by defining the mediator schema. In a second step, appropriate data sources are discovered and their correspondences to the mediator schema are defined. Not surprisingly, LaV approaches emerged in projects addressing web sources, where servers and data sources are typically unreliable and appear or disappear with high frequency.

In contrast, GaV starts from the need for integration of a given set of sources, such as different department databases of an enterprise. These sources are perceived as the more stable part of the MBIS. This leads to a bottom-up development strategy. Accordingly, all schema integration methods result in GaV rules.

A possible solution for the unresolved cases that occurred in our examples is to change the mediator schema or a wrapper schema whenever a problem occurs. For instance, integrating W_1 into a LaV-based MBIS could be accompanied by extending `Σ.clone` with a `species` attribute. However, changing schemas is not a good solution. If the mediator schema must be changed for any source that is plugged-in, maintenance will become a problem [Les98b]. All existing rules must be checked with every change. Furthermore, it is not clear how the mediator should react if a source is first added, leading to a schema change, and then deleted - should the change be rolled back? On the other hand, changes in the schema of wrappers are considered to be impossible. The schema of a wrapper closely reflects the structure of the underlying data source, and requesting changes in data sources conflict with the autonomy of the data source.

3.5 Summary and Related Work

In this chapter we first described different approaches to information integration. We identified ten criteria that distinguish different classes of such federated information systems. We in depth discussed different approaches to the conceptual design of tightly-coupled FIS. This criterion is a fundamental difference between MBIS and other approaches, such as federated databases. A bottom-up design depends on schema integration, a process that was recently characterised as “error-prone, static and difficult” [BBE98]. Navathe and Savasere state that

> *"[...] experience gained in building prototypes of heterogeneous systems [...] has shown that the process of schema integration is most likely to be a bottleneck in the realisation of full-scale distributed heterogeneous database systems. [...] despite of more than a decade of research in the area of schema integration, no methodology or tools have emerged that have proven to be usable for real-life integration problems."* [NS96]

We conclude that the types of systems we described in the introduction cannot be reasonably built with a bottom-up strategy.

MBIS are typically designed in a top-down fashion. They consider mediator and wrapper schemas as independent and concentrate on describing and exploiting relationships between those schemas. Therefore, MBIS presuppose only a loose coupling, which increases their maintainability and offers more flexibility wrt. the types of data sources that can be integrated. However, the loose coupling also requires more powerful mechanisms for query translation.

Then, we described components of a MBIS. Wrappers hide technical particularities of data sources and transform data and query into MBIS conform representations. Mediators answer queries against their schema by translating them into a combination of queries against wrappers. Therein, mediators must treat structural and semantic heterogeneity between different schemas. To perform this task, a mediator needs knowledge about the content and capabilities of wrappers. This knowledge is specified in schema correspondences. We compared two classes of such correspondences and concluded that both fail in certain cases. In the next chapter we shall define *query correspondence assertions*, which fare a synthesis between both approaches.

Related work.

Classification of FIS.
The classification of integrated information systems has been an open question for many years in the literature. The work of Sheth & Larson [SL90] is nowadays used as reference for federated databases and multidatabase systems.

There does not yet exist a similarly accepted reference for mediator based systems. Domenig & Dittrich recently gave a classification of what they call “*mediated query systems*” [DD99]. Their definition is very close to our definition of a MBIS. The criteria they use are split in two groups: *Functional features* are query language, query type (structured or unstructured), schema dependency, degree of structure of the sources, extensibility, and type of feedback. *Implementation features* are the applied architecture, the global data model, type of query processing, management of metadata, and the distinction between thin and thick wrappers. However, many of these criteria are only insufficiently described and the separation in the two groups remains unclear. Another set of criteria for the classification of FIS is given in

[ZHK96], concentrating on aspects of materialising mediators such as update strategy and update time.

Architecture and components of MBIS.

The architecture depicted in Figure 18 is used in many projects, such as TSIMMIS [GMP+97], SIMS [AHK96], and Information Manifold [LRO96a]. It was first described by Wiederhold in [Wie92]. Arens et al. describe several extensions, such as *facilitators* for the finding of data sources and *coordination and management services* for the management of a mediator [AHK+95].

Despite the ubiquitous application of the architecture, the terms "wrapper" and "mediator" are not consistently used in the literature. One may distinguish thick from thin wrappers. *Thick wrappers*, such as those used in TSIMMIS [GMP+97], not only handle data model and technical heterogeneity, but also leverage restricted query capabilities. The wrappers contain a complete query subsumption module to decide whether they can answer a query or not [VP97]. *Thin wrappers*, such as the ones we envisage, stick closely to the capabilities of the source and only perform syntactic operations. As noted in [DD99], thin wrappers better support extensibility of the system, since more functionality is treated in a declarative fashion.

Similarly, one can find different definitions of the term "mediator". Systems such as TSIMMIS [PGMU96], Aurora [YOL97] and MAGIC [Koe99] define a mediator as a single rule that completely encodes a program to compute the extension of one global concept. The integration part of such a system has two layers: A set of *homogenisation mediators*, each responsible for one global concept, and *integration mediators* that combine data from different homogenisation mediators [Koe99]. Homogenisation mediators are responsible for structural and semantic conflicts, while integration mediators only resolve conflicts at the instance level, i.e., data conflicts [YOL97].

Mediators in our definition contain the complete apparatus that is necessary to translate arbitrary queries against their mediator schemas. This model is also used in the Information Manifold [LRO96a] and in SIMS [AHK96]. Duties are clearly separated: Source specific wrappers treat technical and data model heterogeneity, and mediator handle structural and semantic heterogeneity. Every correspondence rule is, in some sense, a homogenisation mediator.

Correspondence specification languages.

Early proposals for correspondence specifications may be found in [SPD92; CL93]. Both allow only the specification of correspondences between single classes, but use a rich set of correspondence types. The fundamental difference and conceptual symmetry between GaV and LaV was first observed in [Hull97].

A particular expressive CSL, called "*model correspondence assertions*" and based on ODMG queries, was developed by Busse [Bus99]. However, this work does not take query planning into account. Another rich language is the "*well-founded object logic*" presented in [DKE94; DK97]. In their framework a rule connects two object-oriented queries, including recursion, disjunction and negation. However, each rule must be translated into a normal form before its usage. This translation is not possible for recursive queries and ill-defined for queries using projection. The normal form of a rule is essentially a GaV rule.

4. QUERY CORRESPONDENCE ASSERTIONS

One fundamental idea of this work is to combine LaV- and GaV rules into a new type of correspondences, called *query correspondence assertion* (QCA). As observed in the previous chapter, LaV and GaV naturally complement each other in the following way:

- Both approaches have their strength in different phases of query answering. LaV rules are a powerful language for the description of source content with respect to a mediator schema, while GaV rules are more geared towards query execution.
- Both approaches deal with different types of conflicts. LaV rules fail if wrapper schemas are more general than the mediator schema, while GaV rules fail if the mediator schema is more general than a wrapper schema.

A combination of LaV and GaV resolves the problems described in the previous chapter. QCAs are such a combination. QCAs use LaV techniques for those cases were GaV fails and vice versa. QCAs allow a concise and intuitive specification of schema correspondences. Furthermore, QCAs indirectly define the set of queries a wrapper can answer, i.e., the query capabilities of wrappers.

We describe the basic idea of QCAs in Section 4.1. Section 4.2 defines their syntax and semantics. In Section 4.3 we give a formal semantics for user query in a MBIS using QCAs, which is based on the construction of a fictive global database `D` through *materialisation* of QCAs. The answer to a user query `u` is then defined as the extension of a query `u'` in `D`, where `u'` is obtained from `u` by replacing each literal with a union of views corresponding to QCAs. This semantics introduces the possibility to prove formally properties of any method that computes answers to a user query: A method is *sound*, if it computes *only answers* that coincide with the semantics, and it is *complete*, if it computes *all answers* that coincide with the semantics of `u`. Such methods are the subject of Chapter 5. Section 4.4 finally introduces the concept of executable mediator queries, which will be important in the following chapter.

4.1 Basic Idea

QCAs are used for specifying schema correspondences. We assume a mediator schema and a set of wrapper schemas as given and initially unrelated. Furthermore, we know a set of executable queries against wrapper schemas. We use QCAs to define relationships between the result of such a wrapper query and the mediator schema. Therefore, although the general approach to the development of our MBIS is top-down, QCAs are in some sense specified bottom-up: They consider the executable queries given, and not so much the ‘typical’ user que-

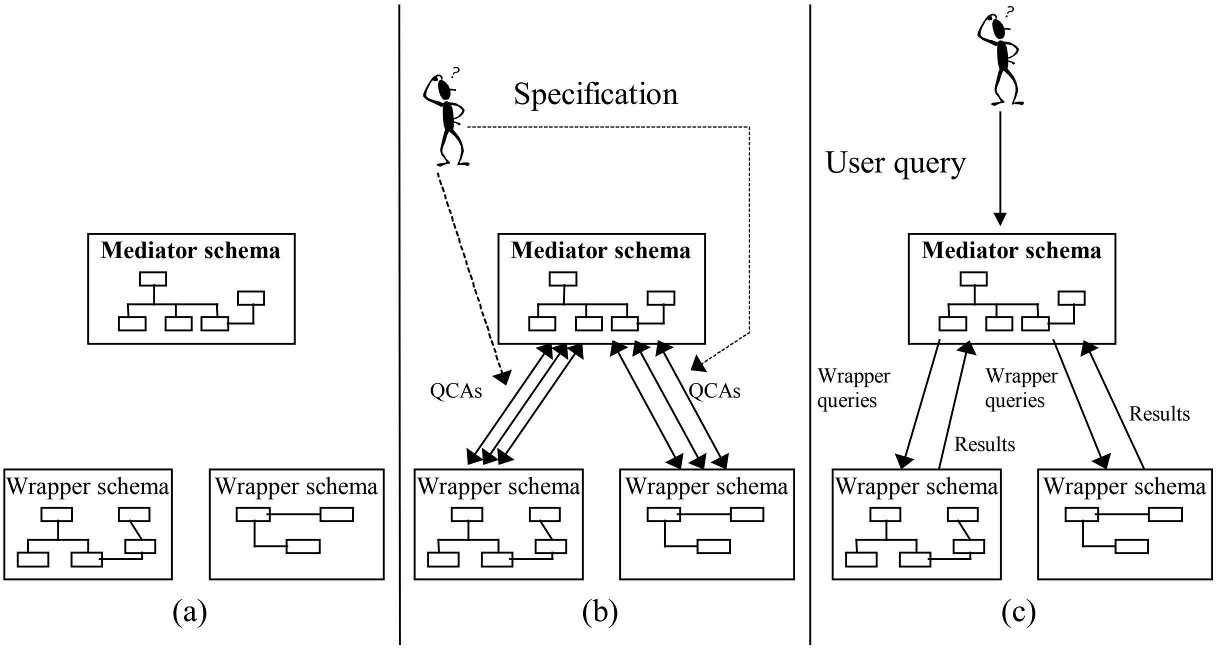


Figure 20. Generating a MBIS using QCAs.
(a) Mediator and wrapper schemas are designed independently. (b) QCAs are used to relate schemas through related queries. (c) At run-time, user queries are translated into executable plans using QCAs.

ries that shall be answerable (see Figure 20). In this section, we illustrate the ideas behind QCAs. A formal characterisation is given in the next sections.

A single QCA is a rule that defines an extensional and intensional correspondence between two queries. One query, the *mediator query*, is a query against the mediator schema. The other query, the *wrapper query*, is an executable query (see Definition (D3.2), page 53) against the export schema of a wrapper `W`. Both queries must have an identical head, i.e., export the same set of variable symbols and use the same head predicate.

Suppose that `mq` is a mediator query with head `v(E)` and body $\mathtt{mq}_{\mathtt{body}}$ and `wq` is the wrapper query addressing a wrapper `W` with body $\mathtt{wq}_{\mathtt{body}}$ and the same head as `mq`. A QCA connecting these queries has the following form:

$$\mathtt{mq}_{\mathtt{body}} \leftarrow \mathtt{W.v(E)} \leftarrow \mathtt{wq}_{\mathtt{body}};$$

The direction of the arrows indicate the actual flow of data: Physically, tuples of data values are computed by `W` by executing $\mathtt{wq}_{\mathtt{body}}$, then projected to `E`, and finally assigned to variables of $\mathtt{mq}_{\mathtt{body}}$. By executing the wrapper query, only exported variables of the mediator query are instantiated. A QCA may be considered as a *restructuring data pump*.

Semantically, the specification of such a QCA asserts the following to the mediator `M`:

- The *intension* of $\mathtt{v(E)} \leftarrow \mathtt{wq}_{\mathtt{body}}$ is a subset of the *intension* of $\mathtt{v(E)} \leftarrow \mathtt{mq}_{\mathtt{body}}$. I.e., the set of real world objects described through the wrapper query is a subset of the set of real world objects described through the mediator query.
- The *extension* of $\mathtt{v(E)} \leftarrow \mathtt{wq}_{\mathtt{body}}$ is a subset of the (only virtually existing) *extension* of $\mathtt{v(E)} \leftarrow \mathtt{mq}_{\mathtt{body}}$. I.e., the set of tuples computed by the wrapper query is a subset of the set of tuples computed by the mediator query, in every possible state of `W` and `M`.

A *QCA is a definition*; whether or not a QCA is a correct assertion is not checked and cannot be checked by the mediator. Furthermore, a QCA only asserts that each result of the wrapper query is a correct result for the mediator query; it does not assert that the result of the wrapper query is *the entire result* of the mediator query. If there exist, for instance, two wrappers $\mathtt{W}_1$

and W_2 that store name and length of clones and make them available through queries foo and bar, respectively, we may specify:

r_1: clone(-,cn,-,cl) ← W_1.v(cn,cl) ← foo(cn,cl);
r_2: clone(-,cn,-,cl) ← W_2.v(cn,cl) ← bar(cn,cl);

If there were no other QCAs containing a clone literal in their mediator query, then the extension of clone inside the mediator is defined as the union of the results of W_1.v and W_2.v.

It can be meaningful to have two QCAs addressing the same wrapper having the same mediator query but different wrapper queries. In this case, the wrapper stores information in different tables that are intensionally identical in the mediator context. Imagine a wrapper W that has one relation for clones available in-house (inhouse) and another relation for external clone data (external). This difference is irrelevant for a mediator with the schema given in Table 1, page 18. Therefore, we specify:

r_3: clone(-,cn,-,cl) ← W.v_1(cn,cl) ← inhouse(cn,cl);
r_4: clone(-,cn,-,cl) ← W.v_2(cn,cl) ← external(cn,cl);

Usually, mediator and wrapper queries are real queries and not only single literals. Remember the two wrappers W_1 and W_2 described in Section 3.4. Using QCAs, we may describe them as follows:

r_5: map(mid,mn,-,-,ch),clonelocation(mid,cid,-),clone(cid,cn,ct,cl), ct='PAC' ← W_1.v(mid,cid,mn,ch,cn,cl) ← c_map(mid,mn,sp,ch), PACS(cid,mid,cn,cl), sp='human';

r_6: clone(cid,cn,-,-),contains(cid,gid),gene(gid,gn,-) ← W_2.v(cn,gn) ← position(cn,gn);

The relationship between the mediator schema Σ and the schema Σ_1 of W_1 is defined through r_5. The difficulties we encountered with GaV rules are resolved: We constrain the applicability of the rule to queries asking for PACs, and we filter the tuples from W_1 with a condition on species. r_6 specifies the correspondence between Σ and Σ_2. Note how r_6 uses variables (mid and cid) in the mediator query that are never filled with values since W_2 does not provide them. But both variables are necessary to describe the relationship between the attributes mn and cn inside the mediator schema. Using r_6, we cannot answer a user query requiring values for cid or mid, but we can answer a query asking for genes that are contained in clones and that do not request ID values.

QCAs correspond to LaV and GaV rules in the following way:

- LaV rules are QCAs where wrapper queries are restricted to single literals.
- GaV rules are QCAs where mediator queries are restricted to single literals.

QCAs use the first option of GaV approaches to deal with semantically overlapping sources (see page 57). Therefore, QCAs are wrapper specific, which allows easy plug-in and plug-out of wrappers [Les98b]. For instance, removing a wrapper from a MBIS is logically performed by simply removing all QCAs that describe this source. If the second option had been chosen, all rules would have to be checked for occurrences of relations of that wrapper.

QCAs can only bridge conflicts that are expressible through conjunctive queries. For an example for a relationship that cannot be expressed through a QCA, imagine a wrapper W that has a relation with clones that are *not* YACs. An appropriate rule would be:

clone(cid,cn,ct,cl),cl≠'YAC' ← W.v(...) ← somehow(...);

However, this rule is not a QCA because the mediator query contains an inequality condition is hence not a conjunctive query. We can circumvent the inequality by defining one QCA per each other type of clones.

For another example, imagine a wrapper `W` exporting a relation storing gene and clone pairs where the gene *is not contained* in the clone. The following rule expresses this fact:

```
clone(cid,cn,-,-),NOT contains(cid,gid),gene(gid,gn,-) ← W.v(...) ← no-
   tin(...);
```

Again, this is not a QCA.

We may summarise the meaning of a QCA in the following way:

Intensional aspect.
A QCA asserts that the wrapper query is either intensionally identical or intensionally subsumed by the mediator query. The mediator may expect that any tuple returned from an execution of the wrapper query (which is executable) is intensionally correct for the mediator query (which is not directly executable).

Recall r_5. The mediator query was:

```
v(mid,cid,mn,ch,cn,cl) ← map(mid,mn,-,-,ch),clonelocation(mid,cid,-), clo-
   ne(cid,cn,ct,cl),ct='PAC';
```

The wrapper query was:

```
v(mid,cid,mn,ch,cn,cl) ←
   c_map(mid,mn,sp,ch),PACS(cid,mid,cn,cl),sp='human';
```

These two queries intuitively mean the same, i.e., they return name and ID of human maps together with PAC clones and their length contained in these maps. However, this intension is expressed completely differently in the mediator schema and in the wrapper schema.

Extensional aspect.
A QCA asserts that the set of tuples obtained from executing the wrapper query is a subset of the set of tuples *that are expected to be obtained* by executing the mediator query. Formally, we could write the following. Assume a database $D_M = (\Sigma_M, I^{\Sigma M})$ inside the mediator, and a database $D_W = (\Sigma_W, I^{\Sigma W})$ being wrapped by `W`. The QCA mq_{body} ← `W.v(E)` ← wq_{body} then asserts the following:

$$mq|_E(D_M) \supseteq wq|_E(D_W);$$

However, this definition is not meaningful since we have not yet defined what a global database could be. We shall get back to this issue in Section 4.3.

4.2 Syntax and Semantics of QCAs

We first define the syntax of QCAs. We distinguish two classes of QCAs: A “usual” QCA connects two conjunctive queries, whereas *enhanced QCAs* use a more expressive language for wrapper queries. Query planning, as described in Chapter 5, handles both classes equally well. To simplify further discussions, we also define a *normal form* for QCAs.

Next, we define the semantics of QCAs. From this point of view, a QCA is an assertion about the extensional and intentional relationship of the two connected queries.

Definition (D4.1)-(D4.3) (Syntax of QCAs, enhanced and simple QCAs).
Let $\mathtt{M} = (\Sigma_{\mathtt{M}}, \Psi, \Gamma)$ and $\mathtt{W} = (\Sigma_{\mathtt{W}}, \Omega, \chi)$ with $\mathtt{W} \in \Psi$. Let $\mathtt{v(E)} \leftarrow \mathtt{mq_{body}}$ be a $\mathtt{CQ}_{\mathtt{C}}^{\Sigma_{\mathtt{M}}}$ query and $\mathtt{v(E)} \leftarrow \mathtt{wq_{body}}$ be a $\mathtt{CQ}_{\mathtt{C}}^{\Sigma_{\mathtt{W}}}$ query executable by W.

(D4.1) The following formula is a *query correspondence assertion* r:

$$\mathtt{r:\ mq_{body}} \leftarrow \mathtt{W.v(E)} \leftarrow \mathtt{wq_{body};}$$

where:

- $\mathtt{v(E)} \leftarrow \mathtt{mq_{body}}$ is the *mediator query* of r, denoted as medq(r),
- $\mathtt{v(E)} \leftarrow \mathtt{wq_{body}}$ is the *wrapper query* of r, denoted as wrapq(r),
- W is the origin of r, denoted as origin(r), and
- E is the set of *exported variables* of r, denoted as export(r).
- $\mathtt{cond(medq(r),E)} \Leftrightarrow \mathtt{cond(wrapq(r),E)}.$

(D4.2) An *enhanced QCA* r is a QCA where the wrapper query is extended with a set of *attribute transformations* $\mathtt{t_i}$:

$$\mathtt{r:\ mq_{body}} \leftarrow \mathtt{W.v(E)} \leftarrow \mathtt{wq_{body}, t_1, ..., t_m;}$$

The $\mathtt{t_i}$ have the form: "$\mathtt{s = f(s_1, s_2, s_3, ...)}$" where f is a function symbol, $\mathtt{s \in E}$ and the arguments of f are either constants or variables from E.

(D4.3) A QCA is called *simple* if its mediator query is from $\mathtt{CQ}_{\mathtt{S}}^{\Sigma}$; it is called *complex* if its mediator query is from $\mathtt{CQ}_{\mathtt{C}}^{\Sigma}$.

■

Remark:
Attribute transformations are treated by the mediator as external functions that are computed after the execution of the wrapper query. Therefore, we require that both the arguments and the result of an attribute transformation is exported.

■

Enhanced QCAs are more powerful than normal QCAs since they may perform calculations with attribute values that are not computed inside a wrapper. In many cases, attribute transformations are necessary for bridging semantic and structural conflicts. Examples are:

- The conversion of values with different scales to mediate, for instance, between different currencies or different metric measurements.
- The mapping of object names to global IDs using, for instance, a lookup-table.
- The composition and decomposition of string values, for instance, to bridge between a mediator relation with one attribute for the author's first and one for author's last name, and a wrapper schema that has only one name field.
- The aggregation of values.

An enhanced QCA may be considered as consisting of two parts: the first part is a query that must be executed by the wrapper, and the second part is a post-processing of the query results through the attribute transformation functions performed inside the mediator.

In principle, a wrapper query could be any query that is computable by the wrapper. Wrapper queries need not be restricted to conjunctive queries. For instance, if a source is a RDBMS, any SQL query can be used as wrapper query of a QCA. However, in this work we restrict ourselves to conjunctive wrapper queries. The main reason is that non-conjunctive queries are much harder to optimise. In particular, multiple query optimisation (see Section 5.4.2), which relies on the identification of common subparts of wrapper queries, becomes much more complex for queries including negation and/or disjunction.

We have defined enhanced QCAs such that they are easy to execute (although harder to optimise) and at the same time offer sufficient expressive power for many applications. Enhanced QCAs are essential for systems that apply QCAs as specification language. However, the following sections and chapters are focussing on query planning. Normal and enhanced QCAs behave equally from this point of view. Therefore, we always assume normal QCAs if not specified otherwise.

Note that our definition excludes QCAs such as:

```
r1: clone(cid,cn,ct,cl),ct="YAC" ← W.v(cid,cn,cl) ←
   q_clone(cid,cn,ct',cl), ct="yeast artificial chromosome";
r2: clone(cid,cn,-,cl),cl<300 ← W.v(cid,cn,cl,ct) ←
   w_clone(cid,cn,cl), cl<500;
```

r_1 assigns the variable `ct`, which appears both in the mediator query and in the wrapper query, two different values. The wrapper query of r_2 produces tuples with a `cl` value smaller than 500 KB, but the QCA asserts that all `cl` values are smaller than 300 KB inside the mediator. Since our definition of QCAs requires that the conditions on exported variables in the wrapper query and in the mediator query imply each other, both QCAs are invalid. Conditions on variables that are not shared are not restricted.

We define two syntactical forms of QCAs, depending on the form of their queries. (see Definition (D2.7), page 16). Transforming QCAs from one form into the other form is straight-forward.

Definition (D4.4) (Normal and embedded form of QCAs).

(D4.4) A QCA `r` is in:

- *normal form* if both wrapper and mediator query are in normal form.
- *embedded form* if both wrapper and mediator query are in embedded form.

■

Remark:
In the following, we always assume simple, non-enhanced QCAs in normal form, if not specified otherwise.

■

After defining the syntax we turn to the semantics of a QCA. The semantics of a single QCA has two aspects: an intensional one and an extensional one. The following definition is based on the definition of extension and intension of queries and relations (see Definitions (D2.2), (D2.10), and (D2.11)).

Definition (D4.5) (Semantics of QCAs).
Let $M = (\Sigma_M, \Psi, \Gamma)$ and $W = (\Sigma_W, \Omega, \chi)$ with $W \in \Psi$. Furthermore, let `r` be a QCA with `origin(r) = W`.

(D4.5) `r` asserts the following to `M`:

- The intension of `wrapq(r)` is a subset of the intension of `medq(r)`.
- The extension of `wrapq(r)` is a subset of the extension of `medq(r)`.

■

Remark:
The extensional part of this definition has to be interpreted with caution, since a mediator schema has no extension. However, we shall virtually build up the extension of the mediator schema out of QCAs in the following section.

■

4.3 Semantics of User Queries in MBIS using QCAs

QCAs are used to specify correspondences between schemas through correspondences between queries. The purpose of QCAs is to guide the translation of user queries, i.e., to guide query planning. However, before we approach query planning, we must define the *semantics of user queries*, i.e., we must define what the answer to a user query against the schema of a mediator `M` should be.

To better understand the role of QCAs in MBIS, consider a rule-based expert system. The expert system is essentially a program consisting of single rules. The semantics of an execution of this program may, for instance, be defined top-down, i.e., by recursively chaining rules, or bottom up, i.e., by subsequently replacing rules with facts. In this sense, a QCA is a single rule, the mediator is a program, and a user query is an execution of this program.

In the following, we define a bottom-up semantics for the program execution. We interpret QCAs as “tuple generators” that virtually fill the mediator schema with real data, thus building a virtual database `D`. The answer to a user query `u` is then defined as the extension of a query `u′` against `D`, where `u′` is obtained from `u` using some transformations.

More precisely, we define the semantics of a user query `u` in two steps. In Section 4.3.1, we define the *virtual database* $\mathtt{D}_\mathtt{V}$ of a mediator `M`. $\mathtt{D}_\mathtt{V}$ is obtained by *materialising* all QCAs in `M`. In principle, the purpose of materialisation is to be able to compute the result of a wrapper query by executing the corresponding mediator query. If we had achieved this, we would have solved the problem of heterogeneity since the mediator would be able to access all data from a single homogeneous schema, i.e., the mediator schema.

However, we shall see that materialisation only partly succeeds in this goal, because it loses intensional information encoded in QCAs. Exploiting this information is the purpose of the second step, which we describe in Section 4.3.2. We define a *rewriting* of a user query `u` against a mediator schema into a query against $\mathtt{D}_\mathtt{V}$. This rewriting includes the intensional information lost during materialisation.

Unfortunately, our semantics is only meaningful for simple QCAs. It cannot cope with non-equality conditions connecting two literals, as they may occur in complex QCAs. We show why at the end of Section 4.3.2. This problem carries over to Chapter 5, where we describe sound and complete query planning algorithms for mediators with simple QCAs. Despite that the algorithms can be modified such that they also work with complex QCAs, finding a *proper semantics* for user queries in the presence of complex QCAs remains an open research problem.

4.3.1 Materialising QCAs

In this section, we define the *virtual database* $\mathtt{D}_\mathtt{V}$ of a mediator `M`. $\mathtt{D}_\mathtt{V}$ is obtained through *materialisation* of QCAs, i.e., it is constructed by executing all wrapper queries of QCAs and materialising the resulting tuples in a database with the schema of `M`. Our intention is to give the mediator a way of obtaining the result of *any wrapper query* using a uniform "vocabulary", i.e., the mediator schema.

First, we define what it means to materialise a single QCA. Let M = (Σ,Ψ,Γ) and r $\in \Gamma$. Consider an empty database D with schema Σ', which is obtained from Σ by extending every relation with an additional attribute qid (*QCA identifier*). We use qid to assign to each tuple the QCA which produced it.

Materialising r into D means to store all data obtained by wrapq(r) into D according to medq(r). Therefore, we first execute wrapq(r) without any variable bindings, which yields a set T of tuples. Next, we update D through the view medq(r). After this update, we want to be able to obtain T by executing medq(r) on D. To achieve this goal, every tuple of T must eventually be identical to an element of the cartesian product of the extensions of the literals of medq(r). This element must (a) fulfil the join conditions in medq(r), and must (b) fulfil the conditions of medq(r) (See Definition (D2.10), page 17).

We proceed as follows: We go through all tuples in T. For each such t $\in$ T, we go through all literals of medq(r). Let l be such a literal. We insert into the relation of l in D a tuple whose values are taken from t. However, recall that medq(r) might have non-exported variables. T contains only values for exported variables. To decide which values should be used for attributes of non-exported variables, we must consider two problems:

- medq(r) may contain a non-exported variable more than once, i.e., the non-exported variable carries a join.
  In order not to lose this join condition, we consistently replace each non-exported variable in medq(r) with the same, fresh value for each tuple of T. This value must not join with any other value in D.

- medq(r) may contain conditions on non-exported variables.
  Unfortunately, it is not possibly to express this information in the extension of a database. Therefore, we ignore conditions during materialisation and consider them in a second step, presented in Section 4.3.2.

After formally defining the materialisation of a QCA, we give an example.

Definition (D4.6) (Materialisation of QCAs).
Let M = (Σ,Ψ,Γ) and let r $\in \Gamma$ be in embedded form with wq = wrapq(r) and mq = medq(r). Let L be the set of literals of mq, F be the set of non-exported variables of mq, and E be the set of exported variables of mq. Let T be the set of tuples obtained by executing wq. For v $\in$ E and t $\in$ T, let t(v) denote the value of v in t. For v $\in$ F and t $\in$ T, let f(t,v) return an arbitrary but unique value. Furthermore, let D be an empty database with schema Σ' obtained from Σ by adding to each relation a new attribute qid at the first position.

(D4.6) The *materialisation of r* into D is defined as follows: For each pair (t,l) $\in$ T x L, where l is of the form rel(s_1, ..., s_k), we insert one tuple (s_1',...,s_{k+1}') into rel in D with the following values:

- s_1'=r.
- $s_i \in$ const $\Rightarrow$ $s_{i+1}' = s_i$.
- $s_i \in$ F $\Rightarrow$ s_{i+1}' = f(t,s_i).
- $s_i \in$ E $\Rightarrow$ s_{i+1}' = t(s_i).

■

Remark:
Materialising a QCA ignores conditions in mediator queries completely, i.e., also those on exported variables. This is reasonable since a QCA *asserts* that the conditions in the mediator query hold for all values computed by the wrapper query. If this assumption were removed, we could immunise a mediator against wrong values by checking all conditions for every tuple from T during the materialisation of a QCA.

■

The intuition behind the previous definition is the following: A QCA is essentially a restructuring of attribute values. The values computed by wq are distributed over the relations of Σ according to mq. One tuple of T results in one tuple for each occurrence of a relation in mq. However, a QCA does not define values for non-exported variables in mq. Any symbol s in mq that is not an exported variable may either be a constant, in which case this constant value is used, or it may be a non-exported variable. If a non-exported variable appears more than once, it expresses a join between literals of mq. Recall QCA r_6 from the previous section:

clone(cid,cn,-,-),contains(cid,gid),gene(gid,gn,-) ← W_2.v(cn,gn) ← position(cn,gn);

No values for cid or gid are obtained through the wrapper query, but the relationship of values for gn and cn is nevertheless clearly defined. To retain this relationship, we use unique values replacing cid and gid for each tuple computed by the wrapper query.

Example 4.1.
Consider the mediator schema given in Table 1, page 18. Imagine the following QCA r describing a wrapper for PAC data with aliases:

r: clone(cid,cn,'PAC',cl),clonealias(cid,al) ← W.v(cn,cl,al) ← pacs(cn,cl,al);

Let q = medq(r). Imagine W produces three tuples for this query: ('yWXD1',50, 'T18'), ('yWXD1',50,'T99'), and ('yWXD2',100,'T20'). Materialising r results in the following database:

clone	qid	cid	cn	ct	cl
	r	f_1	yWXD1	PAC	50
	r	f_2	yWXD1	PAC	50
	r	f_3	yWXD2	PAC	100

clonealias	qid	cid	al
	r	f_1	T18
	r	f_2	T99
	r	f_3	T20

f_1, f_2, and f_3 are fresh values that must not appear anywhere else in D.

■

In the previous example, we obtained the result of wrapq(r) from the materialisation of r by executing medq(r) on D. This *translation* is the reason for materialisation. However, the following example shows that executing medq(r) on D after materialising r *does not always* produce the result of wrapq(r).

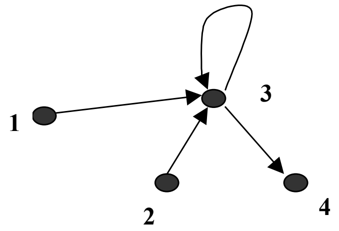


Figure 21. Graph induced by the result of the exemplary wrapper query.

Example 4.2.
Consider a mediator storing graphs: The mediator schema has only one binary relation, `edge`, storing edges between nodes. Nodes are represented by integer numbers. Assume we have a data source that stores a graph and that has a function that returns paths of length three. We model this through the following QCA `r`:

```
r: edge(a,b),edge(b,c) ← W.v(a,b,c) ← threewaypaths(a,b,c);
```

Let `wq` = `wrapq(r)` and `mq` = `medq(r)`. Suppose that executing `wq` returns:

W.v	from	via	to
	1	3	3
	2	3	4

If we materialise `r`, we obtain:

edge	qid	from	to
	r	1	3
	r	3	3
	r	2	3
	r	3	4

If we compute `mq` (ignoring `qid`) on this database, we do not obtain the result of `wq`, but:

edge	from	via	to
	1	3	3
	1	3	4
	3	3	3
	3	3	4
	2	3	3
	2	3	4

These are all paths of length three that must exist in a graph if the two paths reported by `wq` exist (see Figure 21). Thus, the result is justified.

■

The reason for this difference is that wrapper queries might be “incomplete”. Specifying a QCA only asserts that the tuples computed from the wrapper query are extensionally and intensionally contained in the mediator query, but does not guarantee that the result of the wrapper query adheres to the semantics of relational queries (see Definition(D2.10), page 17). Therefore, executing queries on the materialised result of a QCA may produce more tuples that the original wrapper query.

Materialising all QCAs of a mediator `M` yields the virtual database for `M`.

Definition (D4.7) (Virtual mediator database).
Let $M = (\Sigma, \Psi, \Gamma)$.

(D4.7) The *virtual database* D_V for M is obtained by materialising all QCAs in Γ.

■

4.3.2 Answering Queries using Materialised QCAs

We define the answer to a user query u against a mediator M. It is tempting to define the answer to u as the extension of u in the virtual database of M. However, this approach is not feasible. A QCA contains both an *intensional* and an *extensional aspect*. The materialisation of a QCA only reflects the extensional facet – the intensional information is lost.

Example 4.3.
Consider a mediator with the schema given in Table 1, page 18. Consider a wrapper W providing data about physical maps of the size of a single clone, i.e., its size is always smaller than 2 MB. W does not provide the length itself. We describe W through the following QCA r:

```
r: map(mid,mn,'physical',ms,chr),ms<2000 ← W.v(mid,mn,chr) ← some-
   how(mid,mn,chr);
```

Imagine wrapq(r) produces two tuples, (100,'map3',X) and (200,'map4',Y). Materialising r results in the following database D:

map	qid	mid	mn	mt	ms	Chr
	r	100	map3	physical	f_1	X
	r	200	map4	physical	f_2	Y

Note that the information about the size range of maps in W is lost. Since ms is not exported in r, wrapq(r) does not produce values for ms. Therefore, ms is replaced with fresh values (f_1 and f_2) during the materialisation. Now, consider the two user queries u_1 and u_2:

u_1(mid) ← map(mid,mn,-,ms,-),ms<3000;
u_2(mid) ← map(mid,mn,-,ms,-),ms<1000;

We cannot answer either of those queries using D since we cannot ensure the condition on ms. The fresh values are treated like null, i.e., any condition evaluates to false. However, if consider the original QCA r we realise that all tuples in D are correct for u_1 since r asserts that all map sizes are smaller than 2 MB.

However, we cannot compute tuples that are certainly correct answers for u_2. From the result of wrapq(r), we cannot filter tuples smaller than 1 MB because no length information is provided.

■

We solve this problem by capturing the intensional aspects of QCAs in *fragments*. We define one fragment for each literal of a mediator query of a QCA. Fragments include intensional information attached to their literal, i.e., conditions on variables appearing in their literal. Each fragment *induces a view* that relates the intensional information of the fragment to the according tuples, i.e., those that were obtained through the wrapper query.

Definition (D4.8)-(D4.9) (Fragments and induced views).
Let $\mathtt{M} = (\Sigma, \Psi, \Gamma)$ with only simple QCAs and let $\mathtt{D_V}$ be the virtual database for $\mathtt{M}$.

(D4.8) *A fragment* is a 4-tuple $(\mathtt{r},\mathtt{l},\mathtt{E},\mathtt{C})$ where:

- $\mathtt{r} \in \Gamma$.
- $\mathtt{l}$ is a literal from $\mathtt{medq(r)}$.
- $\mathtt{E} = \{\mathtt{e} \mid \mathtt{e} \in \mathtt{export(l)}\}$.
- $\mathtt{C} = \mathtt{cond(medq(r),variables(l))}$.

(D4.9) Let $\nu = (\mathtt{r},\mathtt{l},\mathtt{E},\mathtt{C})$ be a fragment, and let $\mathtt{l}$ have the form $\mathtt{rel(s_1,...,s_n)}$. The *view on D_V induced by ν* is the query:

$$\mathtt{q(qid,s_1,...,s_n)} \leftarrow \mathtt{rel(qid,s_1,...,s_n),qid=r;}$$

■

Remarks:

- An induced view exports all attributes of the corresponding relation of Σ', not only those that are exported in the corresponding mediator query.
- An induced view uses the QCA identifiers stored during the materialisation of QCAs to distinguish tuples computed by different QCAs. (see Definition (D4.6)). Thus, the semantic information stored in the fragment, which could not be represented in the virtual database, is associated to the right set of tuples.

■

The view on $\mathtt{D_V}$ induced by a fragment $\nu = (\mathtt{r},\mathtt{l},\mathtt{E},\mathtt{C})$ for a base relation $\mathtt{rel}$ computes the extension of $\mathtt{rel}$ in $\mathtt{D_V}$ derived from the materialisation of $\mathtt{r}$. In principle, we want to define the extensions of the literals of a user query $\mathtt{u}$ as the union over all induced views for the same base relation. However, we have to exclude from this union those views whose fragments have conditions that are contradicting with the conditions of $\mathtt{u}$.

Definition (D4.10) (Admissible fragments).
Let $\mathtt{M} = (\Sigma, \Psi, \Gamma)$ with only simple QCAs and let $\nu = (\mathtt{r},\mathtt{l},\mathtt{E},\mathtt{C})$ be a fragment. Let $\mathtt{u} \in \mathtt{CQ}_\mathtt{S}^\Sigma$ be a user query with a literal $\mathtt{k}$. Let $\mathtt{s_i}$ ($\mathtt{t_i}$) be the symbol at the $\mathtt{i}$'th position of $\mathtt{k}$ ($\mathtt{l}$).

(D4.10) ν *is admissible for* $\mathtt{k}$ iff:

- The relation of $\mathtt{l}$ is the same as the relation of $\mathtt{k}$.
- $\forall\, \mathtt{s_i}\mathtt{:}\ \mathtt{s_i} \in \mathtt{export(k)} \Rightarrow \mathtt{t_i} \in \mathtt{E}$.
- $\forall\, \mathtt{s_i}\mathtt{:}\ (\mathtt{cond(C,t_i)} \Rightarrow \mathtt{cond(u,s_i)}) \vee \mathtt{t_i} \in \mathtt{E}$.

■

Hence, a fragment ν is admissible for a literal $\mathtt{k} \in \mathtt{u}$ if it exports all required attributes and if the conditions of ν either imply the conditions of $\mathtt{u}$ (restricted to $\mathtt{k}$), or if "critical" attributes are exported in $\mathtt{l}$. In the latter case, the mediator may anyway filter the extension of the view induced by ν such that it adheres to the conditions of $\mathtt{u}$, independently of whether the conditions in ν and $\mathtt{u}$ conflict or not.

Using admissible views we now define the semantics of a user query against a mediator schema.

Definition (D4.11) (Semantics of a user query in MBIS).
Let $\mathtt{M} = (\Sigma, \Psi, \Gamma)$ with only simple QCAs and let $\mathtt{D_v}$ be the virtual database for `M`. Let $\mathtt{u} \in \mathtt{CQ}_{\mathtt{S}}^{\Sigma}$ with $\mathtt{n=|u|}$ and $\mathtt{E = export(u)}$. Let $\Lambda_{\mathtt{i}}$ be the set views induced by fragments admissible for the `i`'th literal of `u`. Furthermore, let `Q` be the set of all queries of the form "$\mathtt{q(E)} \leftarrow \nu_1, \nu_2, \ldots, \nu_n, \mathtt{cond(u)}$", where $\nu_{\mathtt{i}} \in \Lambda_{\mathtt{i}}$ and all symbols in $\nu_{\mathtt{i}}$, are replaced by the corresponding symbols in the `i`'th literal of `u`.

(D4.11) The *answer to* `u` *in* `M`, written `u(M)`, is defined as $\mathtt{u(M)} = \bigcup_{\mathtt{q \in Q}} \mathtt{q(D_v)}$.

■

Remarks:

- The replacement of symbols is well-defined since every view has exactly one literal.
- It does not matter if the literal of fragment has a join that is not present in a literal of the user query for which this fragment is admissible. Although the join is syntactically removed when the symbols are replaced, it is still "contained" in the data, i.e., in the materialisation of the QCA.

■

We demonstrate the previous definitions by the following example.

Example 4.4.
Consider the mediator schema as given in Table 1 (page 18) and two QCAs describing the interfaces to two wrappers $\mathtt{W_1}$ and $\mathtt{W_2}$:

$\mathtt{r_1}$: `clone(cid,cn,ct,cl),clonealias(cid,al),cl<150` ← $\mathtt{W_1}$`.v(cid,cn,ct,al)` ← `somehow(cid,cn,ct,al);`

$\mathtt{r_2}$: `clone(cid,cn,-,cl),cl<200` ← $\mathtt{W_2}$`.v(cid,cn,cl)` ← `somehow(cid,cn,cl);`

Assume the following tuples are obtained through $\mathtt{r_1}$ and $\mathtt{r_2}$:

$\mathbf{r_1}$	**cid**	**cn**	**ct**	**al**
	C1	yXWD1	PAC	HMI1
	C2	yXWD2	YAC	HMI2

$\mathbf{r_2}$	**cid**	**cn**	**cl**
	C1	yXWD1	80
	C2	yWXD2	50

First, we materialise both QCAs, resulting in the following virtual database.

clone	**qid**	**cid**	**cn**	**ct**	**cl**
	$\mathtt{r_1}$	C1	yWXD1	PAC	$\mathtt{f_1}$
	$\mathtt{r_1}$	C2	yWXD2	YAC	$\mathtt{f_2}$
	$\mathtt{r_2}$	C1	yWXD1	$\mathtt{f_3}$	180
	$\mathtt{r_2}$	C2	yWXD2	$\mathtt{f_4}$	50

clonealias	**qid**	**cid**	**al**
	$\mathtt{r_1}$	C1	HMI1
	$\mathtt{r_1}$	C2	HMI2

Note that the two sources have conflicting information about the clone 'yWXD1', since $\mathtt{W_1}$ claims its length to be smaller than 150 KB, but $\mathtt{W_2}$ has a clone length of 180 KB.

The fragments of $\mathtt{r_1}$ and $\mathtt{r_2}$ are:

ν_1`=(`$\mathtt{r_1}$`,clone(cid,cn,ct,cl),{cid,cn,ct},{cl<150});`

ν_2`=(`$\mathtt{r_1}$`,clonealias(cid,al),{cid,al},∅);`

ν_3`=(`$\mathtt{r_2}$`,clone(cid,cn,ct,cl),{cid,cn,cl},{cl<200});`

The induced views are:

q_1(cid,cn,ct,v_1) ← clone(qid,cid,cn,ct,v_1),qid=r_1;
q_2(cid,al) ← clonealias(qid,cid,al),qid=r_1;
q_3(cid,cn,v_2,cl) ← clone(qid,cid,cn,v_2,cl),qid=r_2;

where v_1 and v_2 are fresh variable symbols. Consider the following user queries:

u_1(a,b) ← clone(a,b,-,-);
u_2(a,b) ← clone(a,b,c,d),d<170;
u_3(a,b,c,e) ← clone(a,b,c,d),clonealias(a,e);

Both v_1 and v_3 are admissible for the only literal of u_1. v_1 is admissible for clone in u_2 because cl < 150 implies cl < 170. Therefore, it is irrelevant that a "true" cl value is missing. v_3 is admissible for clone in u_2 although the respective implication does not hold – but the critical variable cl is exported. v_3 is not admissible for the first literal of u_3 since it does not export values for cl. v_1 is admissible for the first literal of u_3, and v_2 is admissible for the second. The answers to u_1 – u_3 are hence defined as:

u_1(a,b) ← q_1(a,b,c,d) ∪ q_3(a,b,c,d);
u_2(a,b) ← q_1(a,b,c,d),d<170 ∪ q_3(a,b,c,d),d<170;
u_3(a,b,c,e) ← q_1(a,b,c,d),q_2(a,e);

The results are:

u_1	cid	cn
	C1	yWXD1
	C2	yWXD2
	C1	yWXD1
	C2	yWXD2

u_2	cid	cn
	C1	yWXD1
	C2	yWXD2
	C2	yWXD2

u_3	cid	cn	ct	al
	C1	yWXD1	PAC	HMI1
	C2	yWXD2	YAC	HMI2

We obtained "real" values for all required attributes, i.e., no artificially generated values of the virtual database appear in an answer.

■

Using the previous definition, we now define *soundness* and *completeness* of query planning algorithms.

Definition (D4.12) (Soundness and completeness of query planning).
Let M = (Σ,Ψ,Γ) with only simple QCAs.

(D4.12) Suppose a method computes the set T(u) of tuples as the result of a given user query u ∈ CQ_S^Σ. Then this method is:

- *sound* iff T(u) ⊆ u(M).
- *complete* iff T(u) ⊇ u(M).

■

The semantics of a query against a mediator with complex QCAs is not defined. We explain why this is the case by "climbing a ladder" of increasing complexity in the types of mediator queries that are allowed.

First, consider CQ mediator queries without projection, i.e., conjunctive queries without additional conditions that export all their variables. For such queries, a QCA does not encode intensional knowledge apart from the structural aspects reflected in the distribution of variables over relations. No conditions exist. Furthermore, no fresh values are required during the materialisation of QCAs since, in every QCA, all variables are exported. The semantics of a user query collapses to the result of executing that query on D$_V$.

The next interesting class of queries is CQ, i.e., including projections. To define a proper semantics, we must introduce fragments, and furthermore we must distinguish between admissible and non-admissible fragments.

The third step are CQ$_S$ queries, which are the focus of this work. Our definitions assumed such queries. Compared to CQ queries, we also need to consider conditions. Especially, we must take care for conditions on non-exported variables.

The most difficult class is CQ$_C$. CQ$_C$ queries may contain conditions that connect different literals apart from joins. Therefore, our approach cannot hold any more because conditions that cannot be assigned to a single literal are not expressed in fragments. Consider the following example:

u(cid$_1$,cid$_2$) ← clone(cid$_1$,-,-,cl$_1$),clone(cid$_2$,-,-,cl$_2$),cl$_1$<cl$_2$;

r: clone(cid$_1$,-,-,cl$_1$),clone(cid$_2$,-,-,cl$_2$),cl$_1$<cl$_2$ ← W.v(cid$_1$,cid$_2$) ← somehow(cid$_1$,cid$_2$);

To show that u can be answered from a materialisation of r we must show that condition cl$_1$ < cl$_2$ from u is implied by r. However, this is impossible by looking at each literal in isolation, since the condition spans two of them.

4.3.3 Consistent Sets of QCAs

The examples we gave so far used only small sets of QCAs. However, as soon as the number of QCAs in a mediator grows, the relationship between different QCAs will become less obvious. The semantics of the mediator could be blurred if the given set of QCAs can be *contradicting* or *inconsistent*. Therefore, we examine in the following if a set of QCAs can be such that no user query can ever be answered, for instance because a subset of the QCAs is contradicting.

We approach this question by the following examples.

Example 4.5.
Consider a wrapper W$_1$ being described through the following two QCAs:

r$_1$: clone(-,cn,ct,cl),ct='PAC' ← W$_1$.v$_1$(cn,cl) ← segments(cn,cl);

r$_2$: clone(-,cn,ct,cl),ct='YAC' ← W$_1$.v$_2$(cn,cl) ← segments(cn,cl);

These two rules may be considered contradicting: objects stored in segments cannot have both 'PAC' and 'YAC' as clonetype. If we cannot determine the clone type, we should rather ignore it, i.e., replace r$_1$ and r$_2$ with the rule:

r$_3$: clone(-,cn,-,cl) ← W$_1$.v$_3$(cn,cl) ← segments(cn,cl);

However, r$_1$ and r$_2$ could be interpreted as expressing that segments stores objects that are, on the mediator level, considered as both YACs *and* PACs, but nothing else. From this point of view, there is a big difference between r$_1$/r$_2$ and r$_3$: in the former case, the mediator will use W_1 for global queries for YACs or PACs, but not for BACs; in the latter case, W_1 will be used only for queries that do not export and not select upon ct. The query:

```
u(cn) ← clone(-,cn,ct,-),ct='YAC';
```

can be answered using r_2, but not using r_3.

■

Example 4.6.
Consider a wrapper W_2 described through the following QCAs:

r_3: clone(-,cn,-,cl),cl≤1000 ← $W_2.v_1$(cn,cl) ← segments(cn,cl);
r_4: clone(-,cn,-,cl),cl>1000 ← $W_2.v_2$(cn,cl) ← segments(cn,cl);

r_3 and r_4 seem to be contradicting. r_2 assures that clones from segments are longer than 1000 KB, while r_4 assures that they are smaller than 1000 KB. However, these two rules do not produce any harm: for a user query with a condition on clonelength only one will be used; for a user query without a condition on clonelength both will be used in first place, but the resulting plans are redundant, which will be detected in a later phase (see Section 5.4).

Nevertheless, both QCAs taken together are equivalent to a single rule without any condition on clonelength. It would be better to replace them.

■

Both QCAs do not lead to wrong results being computed by the mediator. However, it would be better to specify only one QCA. To detect such cases, we introduce the notion of *consistent QCAs* and of *consistent sets of QCAs*. The intention behind these definitions is to identify cases in which QCAs possibly conflict with each other.

We call a set of QCAs consistent if, whenever a wrapper query of one QCA is contained in the wrapper query of another QCA, also the mediator query of the first QCA is contained in the mediator query of the second. In other words: a set of QCAs is consistent if subset relationships between query extensions carry over from the wrapper schema to the mediator schema. This does, for instance, not hold for the two QCAs of Example 4.6.

Definition (D4.13)-(D4.14) (Consistent QCAs).
Let $M = (\Sigma, \Psi, \Gamma)$ and $r_1, r_2 \in \Gamma$ with $origin(r_1) = origin(r_2)$.

(D4.13) r_1 and r_2 are *mutually inconsistent* if:

$$wrapq(r_1) \subseteq wrapq(r_2) \wedge medq(r_1) \not\subseteq medq(r_2);$$

or:

$$wrapq(r_2) \subseteq wrapq(r_1) \wedge medq(r_2) \not\subseteq medq(r_1);$$

(D4.14) Γ is *consistent* if all its elements are mutually consistent.

■

Inconsistent sets of QCA may be considered as semantically suspicious. However, they do compromise the semantics of user queries, nor do they affect query planning as presented in the next chapter. In particular, a set of QCAs cannot be such that it crashes query planning, in the sense as a set of unsatisfiable axioms crashes logical deduction. We therefore do not require consistency of QCA sets for the rest of this work.

4.4 Executable Mediator Queries

In Section 3.3.1 we defined *executable wrapper queries*. We settled that each wrapper carries a set of query templates, and that it is able to execute any query that corresponds to such a template plus additional conditions. We argued that the result of such a query is always computable through post-processing, as long as the query template itself is executable.

In this section we define *executable mediator queries*. Recall that a mediator query is not the same as a user query. A mediator query is a query against the mediator schema that has a corresponding query associated. In contrast, user queries are arbitrary queries against the mediator schema. The mediator computes the answer to a user queries by executing appropriate wrapper queries. However, the question is what wrapper queries are appropriate. In the next chapter, we shall determine those sets by rewriting the user query into mediator queries.

Therefore, we have to make sure that those mediator queries are executable. Intuitively, a mediator query is executable if it has a corresponding, executable wrapper query. Consider a mediator with a single QCA `r`. Executing `wrapq(r)` results in a set of tuples that is a subset of the extension of `medq(r)`[6]. This does not imply that we can only execute `medq(r)`; instead, we can execute any query that can be computed on the result of `medq(r)`.

Example 4.7.
Let the following query `mq` be the mediator query of a QCA `r`:

```
W.v(mid,mn,ms,cid,cn) ← map(mid,mn,mt,ms),mt='physical', clonelocati-
   on(mid,cid,-),clone(cid,cn,-,cl),cl<100;
```

The following queries can be executed using only `r`:

```
u1(mid,mn,cid) ← map(mid,mn,mt,ms),mt='physical',clonelocation(mid,cid,-),
   clone(cid,cn,-,cl),cl<100;
u2(mid,mn,ms,cid,cn) ← map(mid,mn,mt,ms),mt='physical', clonelocati-
   on(mid,cid,-),clone(cid,cn,-,cl),cl<500,ms<100;
```

The result of u_1 can be obtained from the result of `mq` by removing some variables from the export list and by introducing a join on the attributes `mn` and `cn`. Computing this join is possible since both attribute values are exported. u_2 can be answered using `mq` since (a) `cl < 500` is implied by `mq` and (b) `ms < 100` can be enforced since `ms` is exported in `mq`.

In contrast, the following queries are not executable using `r`:

```
u3(mid,mn,ms,cid,cn) ← map(mid,mn,mt,ms),mt='physical', clonelocati-
   on(mid,cid,-),clone(cid,cn,-,ms),cl<100;
u4(mid,mn,ms,cid,cn) ← map(mid,mn,mt,ms),mt='physical', clonelocati-
   on(mid,cid,-),clone(cid,cn,-,cl),cl<50;
```

u_3 tries to join on `ms` and `cl`. This join cannot be computed because `cl` is not exported in `mq`. Therefore, the mediator cannot ensure this condition, nor can it be pushed to the source (see Definition (D3.2), page 53).

To compute u_4, we must enforce the condition `cl < 50`. This is not possible since the condition is neither implied by `mq` nor computable in the mediator – `cl` is not exported.

■

There are three transformations that can be applied to mediator queries of QCAs that do not affect executability:

- Adding new joins, as long as the joined variables are exported.

[6] Actually, it is the complete result if `M` has only a single QCA.

- Adding new conditions, as long as those conditions are either implied by existing conditions, or the affected variables are exported.
- Removing variables from the head.

Definition (D4.15)-(D4.16) (Variable renamings, query transformer).
Let $\mathtt{q} \in \mathtt{CQ}_{\mathtt{C}}^{\Sigma}$ for some schema Σ.

(D4.15) A *variable renaming* on $\mathtt{q}$ is a function $\alpha\mathtt{:}\ \mathtt{variables(q)} \mapsto \mathtt{variables(q)}$ that maps each variables either into itself or into a variable that is also the target of at least one other variable.

(D4.16) A *query transformer* σ for $\mathtt{q}$ is a pair $(\alpha,\mathtt{C})$, where:

- α is a variable renaming on $\mathtt{q}$, and
- $\mathtt{C}$ is a set of conditions involving only variables of $\mathtt{q}$.

■

Remark:
We use variable renamings to describe additional joins:

- If α is the identity then we write $\alpha = \mathtt{[]}$;
- If α maps two or more variables $\mathtt{v_1,...,v_n}$ to a variable $\mathtt{v}$, then α contains the mappings $\mathtt{[v_1 \rightarrow v,...,v_n \rightarrow v]}$.

■

Definition (D4.17) (Executable mediator query).
Let $\mathtt{M} = (\Sigma,\Psi,\Gamma)$ and $\mathtt{MQ}(\Gamma)$ be the set of all mediator queries in Γ.

(D4.17) A query $\mathtt{q} \in \mathtt{CQ}_{\mathtt{C}}^{\Sigma}$ is called an *executable mediator query* if there exists a query $\mathtt{q'} \in \mathtt{MQ}(\Gamma)$ and a query transformer $(\alpha,\mathtt{C})$ with:

- $\mathtt{body(q)} = \alpha\mathtt{(body(q'),C)}$.
- $\mathtt{export(q)} \subseteq \alpha\mathtt{(export(q'))}$.
- $\forall\ \mathtt{v_1,v_2} \in \mathtt{variables(q')},\ \mathtt{v_1} \neq \mathtt{v_2} \wedge \alpha(\mathtt{v_1}) = \alpha(\mathtt{v_2}) \Rightarrow \mathtt{v_1,v_2} \in \mathtt{export(q')}$.
- Let $\mathtt{C'} = \{\mathtt{c} \mid \mathtt{c} \in \mathtt{C} \wedge (\mathtt{cond(q')} \nRightarrow \mathtt{c})\}$. Let $\mathtt{V}$ be the set of variables appearing in $\alpha\mathtt{(C')}$. Then $\mathtt{V} \subseteq \mathtt{export(q')}$.

■

This definition captures all queries against Σ that can be answered by executing the mediator query of a QCA first and then filtering the result with additional constraints.

Example 4.8.
Continuing Example 4.7, we get:

$\mathtt{u_1} = \alpha_1\mathtt{(q,C_1)},\ \alpha_1 = \mathtt{[mn \rightarrow cn]},\ \mathtt{C_1} = \emptyset;$
$\mathtt{u_2} = \alpha_2\mathtt{(q,C_2)},\ \alpha_2 = \mathtt{[]},\ \mathtt{C_2} = \{\mathtt{cl < 500,\ ms < 100}\};$

■

4.5 Summary and Related Work

This section defined query correspondence assertions as a means to describe relationships between schemas. QCAs combine the advantages of previously published CSLs and avoid their pitfalls. Therefore, QCAs are more expressible than other approaches. In Section 6.2, we shall give an exhaustive list of possible conflicts between heterogeneous, relational schemas and demonstrate how they can be overcome with the help of QCAs. To our best knowledge, no other CSL could handle *all those conflicts* equally well.

We defined *syntax and semantics* of simple and complex QCAs. Basically, a QCA defines an *extensional and intensional* relationship between a query against the mediator schema and a query against the export schema of one wrapper. Intuitively, a QCA specifies the relationship between the ideal world (an integrated and complete database) and the real world (data scattered over many sources) wrt. a certain query.

Given a user query against the mediator schema, the mediator uses these relationships to decide whether or not the wrapper query of a QCA contributes to the answering of this query. We gave this decision a formal basis by defining the *semantics of user queries* in MBIS using QCAs. Only this definition enables us to judge algorithms for query answering wrt. their completeness and soundness, as we do in Chapter 5.

Furthermore, we discussed *consistency of QCAs* and of sets of QCAs. Intuitively, a set of QCAs is consistent if subset relationships between answer sets carry over from the wrapper schema to the mediator schema. We settled that even inconsistent QCAs do not break the semantics of user queries. However, they compromise understandability and maintainability of MBIS, and should hence be avoided.

Finally, we defined the notion of executable mediator queries. This definition will be necessary in the next chapter to decide which queries the mediator can use to answer user queries.

Related work.

Correspondence assertions.
We already discussed other CSLs than QCAs in Section 3.5. An interesting approach to define the semantics of correspondences is presented by Spaccapietra et al. [SPD92]. Their work is based on an entity-centred data model, the *generic data model* (GDM). Source schemas have to be transformed into equivalent GDM schemas. The relationship between two classes in different GDM schemas is captured through *correspondence assertions*. Such an assertion defines whether the extensions of the two connected classes are disjoint, overlapping, identical or in a subset-relationship. The semantics of a correspondence assertion is defined by a function that maps source-specific class extensions into sets of real-world objects (*real world semantics*). For instance, if two classes `X` and `Y` are defined to be disjoint, then the semantics of this definition is that $\texttt{RWS(X)} \cap \texttt{RWS(Y)}=\emptyset$.

The semantics of QCAs we gave in Section 4.2 was inspired by this approach. However, there are more differences than similarities: We use the relational data model, while Spaccapietra et al. use GDM; Spaccapietra et al. address schema integration, while we focus on query processing; Spaccapietra et al. allow assertions only to connect single classes, while we allow conjunctive queries on both sides; Spaccapietra et al. allow four different types of extensional relationships, while QCAs always assume a subset relationship. Particularly, Spaccapietra et al. assume correspondences between two source relations, and not between a mediator relation and a source relation as a QCA.

Restricted query capabilities.
Recently, a great number of projects have treated the problem of answering global queries in MBIS with sources having only *restricted query capabilities* [GMY99]. For instance, if a web site is used as a data source, then this source will typically allow only conditions on certain attributes, namely those that appear in a web search form. [VP97] describes algorithms that can decide upon the executability of complex restriction patterns specified as *regular grammars*. [HKWY97; KTV97] describe different methods of deciding *where* operations are optimally executed (in the mediator or in the wrapper) if wrappers support only certain functions. [YGMU99] addresses the problem of computing capabilities of mediators that use other mediators as sources.

Another type of query restrictions are *binding patterns*. Web sources typically require certain attributes to be bound. For instance, if we model a web source of a bookstore through a relation `book(author,title,publisher)`, then it is usually not possible to execute a query for all books without any variable bindings. Usually it is only possible to get a list of all books of a certain author, all books with a certain word in the title, etc. Query planning with binding patterns is considered in [RSU95] (see also 5.5).

Our model of query capabilities is currently very simple. We assume that each wrapper can answer every query used as wrapper query in a QCA. Executable wrapper queries (see Definition (D3.2)) are those that can be computed by only using the result of a wrapper query. They can, for instance, contain conditions on exported variables not present in a wrapper query. We do not require that the wrapper itself executes such conditions. This model is well suited for web sources that, typically, at most support conjunctions of conditions on attribute values [Hol99]. On the web, disjunction, negation, or complex conditions are virtually non-existing.

QCAs lack the ability to define binding patterns. This is a is restriction since binding patterns are ubiquitous on the web. However, a proper extension is straight-forward. It suffices to associate QCA with so-called *adornments* on exported variables. This technique is e.g. used in [RSU95; YGMU99]. However, query planning in the presence of binding pattern is considerable more complex than without binding pattern (see Section 5.5).

Relationship of QCAs to horn clauses.
In some sense, QCAs are a derivation of non-recursive, positive DATALOG rules. It is tempting to reduce the problem of query planning to the well-known field of Horn logic. Note however that a QCA is not a Horn clause, since the "head" of the rule, i.e., the mediator query, may consist of more than one literal and also may contain conditions.

If QCAs have mediator queries without conditions, one can break up one QCA into many Horn clauses with single-literal heads. Since the wrapper query cannot be broken up - it is only defined as an entity and we make no assumptions about parts of it - each of the emerging Horn rules must have the full wrapper query as body. This idea is pursued in the inverse-rule algorithms [Qia96; DG97] (see also Section 5.5). One difficulty with this approach are non-exported variables in head predicates. However, Duschka & Genesereth show that it is possible to first replace non-exported variables by skolemisation, and later get rid of the skolem-terms in the heads in a clean-up step [DG97]. However, the inverse rule algorithm does not work with conditions.

The Multiplex framework.
Motro gives a formal semantics for answers to global queries in a FIS scenario in [Mot95]. He proceeds as follows: First, he assume the existence of a virtual and consistent global database `D`. Then, he defines *derived databases* as databases whose relations are all defined as views on `D`. Assume two derived databases D_1 and D_2. Next, he defines views on these databases, lets say a view v_1 on D_1 and v_2 on D_2. Since v_i can be translated into a query against D_i, which

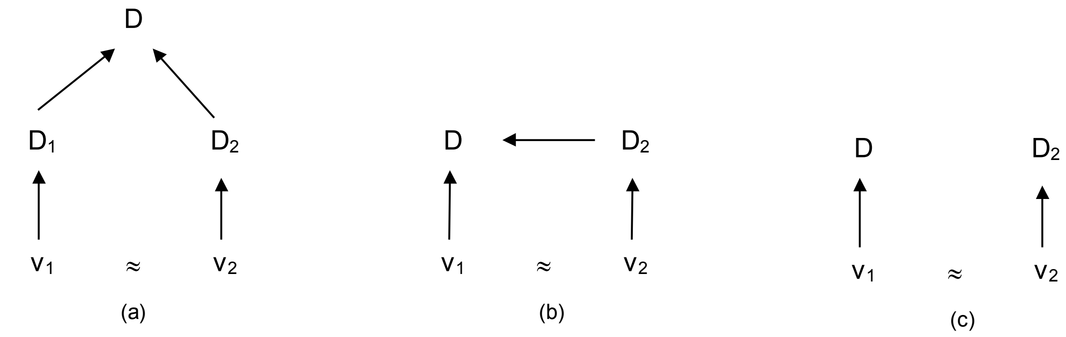


Figure 22. The Multiplex framework.
Arrows mean "defined as view". (a) General framework. (b) In database integration one database (D) is interpreted as global schema. (c) We argue that the definition of D_2 as a derived database is not a realistic assumption. D_2 must be considered as independent from D.

can be further translated into a query against D, he can formally test whether v_1 and v_2 are equivalent.

This method is applied to database integration by interpreting D_1 as global schema and D_2 as a source schema (see Figure 22). Mortro argues that he can then formally define equivalence between a view on the global schema and a view on the wrapper schema. He calls such equivalencies *schema mappings*.

A schema mapping is on first sight almost equivalent to a query correspondence assertion. However, we believe that Motro's foundation is not valid in a heterogeneous environment. The problem with Figure 22 (b) is that, in general, D_2 can not be described as being derived from the global database. Instead, any source can store data that is not present in the mediator schema. Hence, there is no a-priori way to relate v_1 and v_2. Actually, if D_2 were defined as views on D, the problem of finding correspondences is already solved.

Certain versus possible answers.

Grahne & Mendelzon analyse the semantics of answers to global queries in a LaV scenario in [GM99]. They distinguish between *sound* and *complete* views: a view is sound, if all tuples obtained through the view are correct; it is complete, if it contains all correct tuples, but possible also incorrect tuples. Allowing both types of views leads to a natural criterion for the consistency of a global database: For instance, if a sound view computes a tuple X that is not computed by a complete view, than no global database exists for which the view classification can hold.

According to this definition, QCAs always define sound views. [GM99] shows that no inconsistency in the above sense can occur if all views are sound (or all are complete), i.e., it is always possible to construct an instance of the global schema that is compatible with all view definitions.

The authors then define two bounds for answers to global queries: a *certain answer* is one that is contained in at least one sound view and all complete views. A *possible answer* is one that is contained in at least one complete view. The formal definition is based on tableaux techniques. In this sense, our definition is equivalent to certain answers, since we assume all views to be sound.

5. QUERY PLANNING USING QCAS

This chapter describes algorithms that allow a mediator to compute the answer to a user query without materialising all QCAs. Although a complete materialisation does make sense in some applications [ZHK96], it is complicated and inefficient if sources store large data sets or frequently change their content and do not support change sets. Even worse, source materialisation is often impossible. For instance, it is not possible to materialise a complete list of books sold by Amazon.com[7] because only queries for a certain author or for a certain keyword are permitted. Getting *all books* is not possible.

In this chapter, we device algorithms that answer a user query with only executing a minimal set of QCAs. The mediator selects this set by analysing the user query and the set of QCAs. For instance, any QCA whose mediator query does not contain any of the relations of the user query is certainly irrelevant because it cannot contribute to the result. We can answer `u` without executing such QCAs. Unfortunately, it is not easy to distinguish irrelevant from relevant QCAs.

In Section 5.1 we introduce *plans*. A plan as a combination of mediator queries. Plans are *correct* if they are contained in the user query, and they are *executable* if the mediator can enforce all joins and conditions of a plan. This leads to the definition of *query plans*, which are correct and executable plans together with a containment mapping from the user query into the plan (see Figure 23). We prove that every query plan computes only correct answers to a user query (wrt. the semantics defined in Section (D4.11), page 75). Furthermore, we prove that every answer to a user query is computed by some query plan. We conclude that any algorithm that computes all query plans is sound and complete. However, there usually exists an infinite number of query plans for a given user query. Fortunately, we can prove that it suffices to consider only a finite number of those (see Figure 24 for the structure of the important lemmas and theorems in this chapter).

Then, we move to query planning, i.e., the finding of query plans. We develop and analyse two algorithms. Section 5.2 describes the *generate & test algorithm* (GTA). The GTA generates all possible plans and tests each plan independently for correctness and executability. Hence, it breaks query planning into two phases: Plan generation and plan testing.

In Section 5.3 we develop the *improved bucket algorithm* (IBA). The IBA exploits two particularities of the role of query containment in query planning: First, one of the two queries in each containment test is fixed; this is the user query. Second, the other query in each test is composed from a set of known building blocks; these are the mediator queries of the QCAs. The IBA takes advantage of both properties by first computing suitable data structures and then *constructing* query plans.

We prove that both the GTA and the IBA are sound and complete. An analysis of their complexity reveals that the IBA is considerably more efficient than the GTA.

[7] See http://www.amazon.com

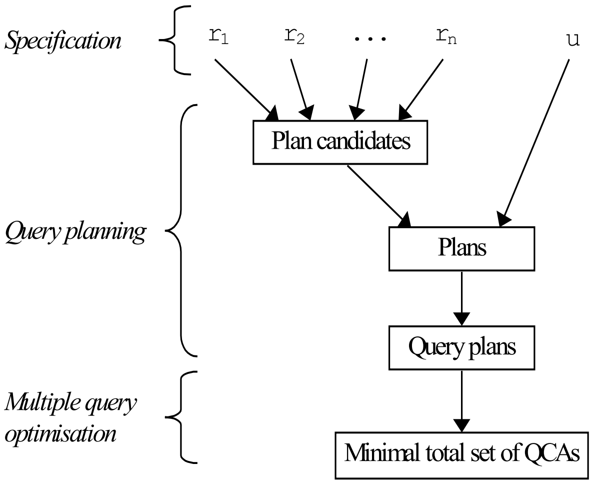


Figure 23. Complete planning process.

However, both algorithms cannot avoid *non-minimal plans*. A query plan is minimal if we cannot remove a QCA from it without changing its result. We prove that, if a query plan is not minimal, then there exists a minimal query plan producing the same result with less remote query executions. The GTA produces all minimal plans, but also non-minimal plans. The IBA does not necessarily find minimal plans, but only equivalent non-minimal plans. It remains an open question whether there exists an algorithm as efficient as the IBA that only produces minimal plans.

Computing the answers of a user query with minimal effort has actually three intertwined occurrences of "minimal":

- Each query plan in isolation should be minimal, i.e., not contain unnecessary QCAs.
- The total set of query plans should be minimal, i.e., not contain query plans that produce only results already obtained through another query plan.
- The total number of wrapper queries that eventually are executed should be minimal.

Avoiding non-minimal plans only solves the first problem. Different minimal query plans may still contain the same QCA, rendering them partially redundant.

In first place, a set of query plans is executed by executing all wrapper queries in any of the query plans. In Section 5.4 we address the problem of computing the same set of tuples with less executions of wrapper queries. Hence, we address all three meanings of "minimality". Given a user query u, we search algorithms that find the minimal set of wrapper queries that, if executed and combined appropriately, compute all and only answers to u. Our *multiple query optimisation* (MQO) first removes *redundant query plans*. In a second step it detects *redundant queries* across different query plans.

Remarks.

- Throughout this chapter, we only deal with simple user queries and simple QCAs. Where relevant, we describe implications of using complex queries.
- It is not guaranteed that all user queries are answerable. Depending on the set of QCAs of the mediator, for some user queries there might not exist a correct plan. Conforming to our query semantics, the answer is empty in such cases.
- The time necessary to answer a user query is determined by two factors: The total number of wrapper queries that have to be executed, and the amount of data that has to be transmitted as the result of wrapper queries. Trying to minimise both runs into a trade-off [PK98]: Assuming high network latency, it will often be faster to execute only a few queries even if the eventually transmitted data set is unnecessary large. In contrast, if the la-

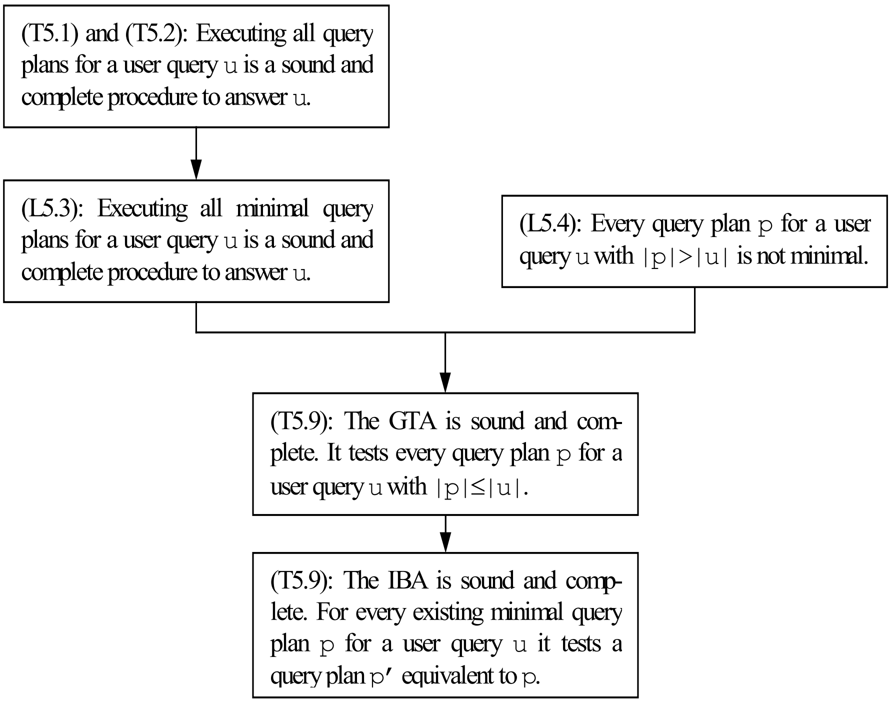

(T5.1) and (T5.2): Executing all query plans for a user query u is a sound and complete procedure to answer u.
(L5.3): Executing all minimal query plans for a user query u is a sound and complete procedure to answer u.
(L5.4): Every query plan p for a user query u with |p|>|u| is not minimal.
(T5.9): The GTA is sound and complete. It tests every query plan p for a user query u with |p|≤|u|.
(T5.9): The IBA is sound and complete. For every existing minimal query plan p for a user query u it tests a query plan p′ equivalent to p.

Figure 24. Structure of proofs.

tency of the network is low, it is reasonable to minimise the amount of data that is shipped, even at the cost of introducing new queries[8]. Finding the optimal balance between these two cost factors requires detailed knowledge about average execution time of queries, distribution of data over different sources, distribution of values in relations, etc. We leave considerations of such physical properties for future research.

- The most important criterion for the quality of a query optimisation algorithm is the total time it takes to achieve the answer to a query. The total time is composed of the time it takes to optimise the query, the time it takes to physically access the data, and the time it takes to perform the operations of the query on that data. In a central database, an optimiser has to balance the time it invests in computing better plans with the time that is necessary to execute a less optimal plan – if the latter is cheaper, further optimisation makes no sense. Since query optimisation algorithms are typically exponential in the size of the query, the optimiser often stops planning and starts executing the current best solution before the global optimum is computed, for instance by using heuristics to prune the search space.
  The situation is different in the highly distributed scenario that we consider. Given the unforeseeable delays that connections over wide-area networks experience frequently, it is a reasonable assumption that it pays off to spend much more time on query optimisation. In most cases, optimisation time will be marginal compared to the time necessary to query and obtain large amounts of data over Internet connections. Therefore, we do not balance the time necessary for query planning with the time necessary for query execution. Nevertheless, we are interested in efficient planning algorithms.

■

[8] The situation is similar to central databases, where a full table scan is preferable to the use of indices if more than approximately 7% of the data will finally be accessed [Juergens, 2000 #625].

5.1 Planning User Queries

We approach the problem of finding combinations of wrapper queries that together compute answers to a given user query. In Section 5.1.1 we define a *plan candidate* as a set of QCAs that *potentially* compute answers to a user query. A *correct and executable plan* is a plan candidate modified such that it certainly computes answers to a user query. Finally, a *query plan* is a correct and executable plan together with a containment mapping determining which of the exported variables of the plan are answers to the user query. We prove that any algorithm that finds all query plans for a given user query is sound and complete wrt. the semantics of user queries.

In Section 5.1.2 we analyse length restrictions on query plans. We introduce the notion of *minimal query plans* and show that every non-minimal query plan does not produce any result not already computed by some minimal query plan. We show that any query plan that is longer then the user query is certainly non-minimal. This helps us to prove that query planning algorithms only have to consider a finite number of query plans, although there exists an infinite number of plans. This result paves the ground for the following sections in which we shall develop algorithms that compute, more or the less efficient, all query plans for a given user query.

5.1.1 Plans and Query Plans

Definition (D5.1)-(D5.4) (Plan candidate, plan, plan expansion).
Let $\mathtt{M} = (\Sigma, \Psi, \Gamma)$ and $\mathtt{MQ}(\Gamma)$ be the set of all mediator queries of Γ.

(D5.1) A *plan candidate* π is a conjunction of queries $q_1, q_2, \ldots, q_n$, $q_i \in \mathtt{MQ}(\Gamma)$ with disjoint sets of variable symbols.

(D5.2) A *plan* $\mathtt{p}$ is a tuple (π, σ) where π is a plan candidate and $\sigma = (\alpha, \mathtt{C})$ is a query transformer (see Definition (D4.16), page 80). $\mathtt{p}$ is also written as:

$$\mathtt{p} = \alpha\texttt{<head}(q_1)\texttt{,...,head}(q_n)\texttt{,C>},\quad q_i \in \pi;$$

(D5.3) The length of a plan $\mathtt{p}$, written $|\mathtt{p}|$, is the number of queries that $\mathtt{p}$ contains.

(D5.4) Let $\mathtt{p} = \alpha\texttt{<head}(q_1)\texttt{,...,head}(q_n)\texttt{,C>}$. The *expansion* of $\mathtt{p}$, written $\Pi(\mathtt{p})$, is the query:

$$\mathtt{q}(e_1, e_2, \ldots, e_m) \leftarrow \alpha(\texttt{body}(q_1), \ldots, \texttt{body}(q_n), \mathtt{C});$$

where $\{e_1, e_2, \ldots, e_m\} = \bigcup_{v \in \exp ort(q_i), 1 \le i \le n} \alpha(v)$

■

Remarks:

- A plan is a plan candidate which is extended with conditions on variables and variable renamings. Renaming variables introduces joins, for instance between different mediator queries.

- The body of the expansion $\Pi(p)$ of a plan p is the conjunction of all bodies of all queries in p plus the additional conditions defined in p. The head of $\Pi(p)$ exports all exported variables of all queries in p.
- Applying a variable renaming to a query often equates exported variables. Definition (D5.4) automatically removes resulting duplicates from the head of a query.
- We usually abbreviate $\alpha\text{<}head(q_1),...,head(q_n),C\text{>}$ with $\alpha\text{<}q_1,...,q_n,C\text{>}$. Furthermore, if $p = (\pi,\sigma)$ and $\sigma = (\alpha,C)$, we also write $p = (\pi,\alpha,C)$.

■

Our definition of a plan expansion is similar to the conventional way of expanding relational views in a SQL query. However, plans never project out variables exported by any of the queries they contain.

Definition (D5.5)-(D5.7) (Executability and correctness of plans).
Let $M = (\Sigma,\Psi,\Gamma)$ with only simple QCAs. Furthermore, let $u \in CQ_S^{\Sigma}$ be a user query and $p = \alpha\text{<}q_1,...,q_n,C\text{>}$ be a plan.

(D5.5) p is *executable* iff:

- $\forall\, q \in p\!:\ \alpha(q, cond(C, variables(q))$ is executable.
- $\forall\, v_1,v_2\!:\ v_1 \in q_i \wedge v_2 \in q_j \wedge q_i,q_j \in p \wedge v_1 \neq v_2 \wedge \alpha(v_1) = \alpha(v_2) \Rightarrow (i = j) \vee (v_1 \in export(q_i) \wedge v_2 \in export(q_j))$.

(D5.6) *p is correct for u* iff:

- $\Pi(p) \subseteq u$.
- p is executable.
- No plan obtained by removing a renaming from α or a condition from C is correct for u.

(D5.7) Let $p = (\pi,\alpha,C)$ be a correct plan for u with $\pi = \{r_1, r_2, ..., r_n\}$. The *result* of p, written $p(M)$, is the extension of the following query in the virtual database obtained from materialising all QCAs in p:

$$head(p) \leftarrow \alpha(medq'(r_1),...,medq'(r_n),C);$$

where $medq'(r_i)$ is obtained from $medq(r_i)$ by (a) adding the qid attribute to every literal, and (b) joining all literals of the same QAC through qid.

■

Remarks:

- Recall Definition (D4.17). A plan p is hence executable if:
  - all conditions in C are either implied by the mediator queries in p or all variables they contain are exported in p, and,
  - whenever two variables are mapped onto the same variable in α, then either both variables are from the same QCA or both variables are exported.

  If both conditions hold then the mediator *can* execute the plan, independently of whether or not conditions and joins are pushed to the wrappers or computed inside the mediator.
- We require that a correct plan has a *minimal set* of variable renamings and additional conditions. Without this restriction, there always exist infinitely many correct plans as soon as there exist at least one. Consider the user query "$u(x,y,z) \leftarrow rel(a,y,z)$" and the

mediator query "q(a,b,c) ← rel(a,b,c)". Certainly, p = ({q},[],∅) is an executable plan and Π(p) ⊆ u. But also p′ = ({q},[x→y],∅) or p″ = ({q},[x→y, z→y],∅) are executable and their expansions are contained in u. However, both p′ and p″ are excluded by Definition (D5.6).

■

Query planning is about finding correct plans. The possibility to introduce query transformers complicates query planning compared to showing only query containment. However, query planning is incomplete without considering query transformers since many correct answers will not be produced. This is highlighted in the following examples.

Example 5.1.
Consider the following user queries and QCAs:

u_1(a,b) ← clone(a,b,-,-);
u_2(a,b) ← clone(a,b,c,d),d<100;
u_3(a,b,d,e) ← clone(a,b,c,d),clonealias(a,e);

r_1: clone(cid,cn,ct,cl),clonealias(cid,al),cl<150 ← W_1.v(cid,cn,ct,al) ←
somehow(cid,cn,ct,al);
r_2: clone(cid,cn,-,cl),cl<200 ← W_2.v(cid,cn,cl) ← somehow(cid,cn,cl);

Let q_1 = medq(r_1) and q_2 = medq(r_2). We analyse the following plans:

p_1(cid,cn,ct,al) ← []<q_1(cid,cn,ct,al)>;
p_2(cid,cn,ct,al) ← []<q_1(cid,cn,ct,al),cl<100>;
p_3(cid,cn,cl) ← []<q_2(cid,cn,cl),cl<100>;
p_4(cid_1,cn_1,ct_1,al_1,cid_2,cn_2,cl_2) ←
[cid_1→cid_2]<q_1(cid_1,cn_1,ct_1,al_1),q_2(cid_2,cn_2,cl_2)>;

The expansions of those plans are:

Π(p_1) = p_1′(cid,cn,ct,al) ← clone(cid,cn,ct,cl),clonealias(cid,al),cl<150;
Π(p_2) = p_2′(cid,cn,ct,al) ← clone(cid,cn,ct,cl),clonealias(cid,al),cl<150,cl<100;
Π(p_3) = p_3′(cid,cn,cl) ← clone(cid,cn,-,cl),cl<200,cl<100;
Π(p_4) = p_4′(cn_1,ct_1,al_1,cid_2,cn_2,cl_2) ← clone(cid_2,cn_1,ct_1,cl_1),clonealias(cid_2,al_1),cl_1<150,
clone(cid_2,cn_2,-,cl_2),cl_2<200;

The expansions of all four plans are contained in u_1. However, p_2 is not executable because cl is not exported in q_1. Only p_1, p_3 and p_4 are correct plans for u_1.

We next consider u_2. p_1 is not correct because $cl < 150 \nRightarrow cl < 100$. p_2 is not correct because p_2 is not executable. p_3 is correct; all necessary attributes are exported. p_4 again is not correct, although there exists a promising symbol mapping [a→cid_2,b→cn_2, d→cl_2,e→al_1]. But the condition on clonelength is not guaranteed. If we added the condition $cl_2 < 100$ to p_4, the resulting plan would be correct.

Finally, we test u_3. The only two candidates are p_1 and p_4 because only these contain a literal for clonealias. p_1 is not correct because it does not export cl. The only correct plan is p_4. p_4 joins the data from two different wrappers through the cid attribute.

■

Joins introduced by the query transformer do not necessarily combine data from different sources. They might also be necessary *inside a single mediator query*, as shown by the following example.

Example 5.2.
Consider the following QCA `r` and user query `u`:

```
u(x) ← rel(x,x);
r: rel(a,b) ← W.v(a,b) ← somehow(a,b);
```

The plan `[]<v(a,b)>` is not contained in `u`. But we can execute the plan `[a→b]<v(a,b)>`, which expands to `v(a) ← rel(a,a)`, and is a correct plan for `u`.

■

Query planning also has to consider that a correct plan can be contained in a user query in more than one way.

Example 5.3.
Consider the following QCA `r` and user query `u`:

```
u(x,y) ← rel(x,y);
r: rel(a,b),b<100,rel(c,d),d>100 ← W.v(a,b,c,d) ← somehow(a,b,c,d);
```

The plan `p=[]<v(a,b,c,d)>` is correct for `u`. This can be shown by two different containment mappings: h_1 = `[x→a,y→b]` and h_2 = `[x→c,b→d]`.

■

In the previous example there are two ways to compute answers for `u` using `p`: One will return all tuples of `r` where the second value is smaller than 100, and the other will return all tuples where the second value is bigger than 100. The two sets of tuples are disjoint, but, according to our semantics, both are valid answers to `u`.

We distinguish *plans* and *query plans* to capture such multiple containment mappings. A query plan is a correct plan together with a containment mapping.

Definition (D5.8)-(D5.10) (Query plan, result of a query plan).
Let M = (Σ, Ψ, Γ) with only simple QCAs, and let u $\in$ CQ_S^Σ be a user query.

(D5.8) A *query plan* ϕ for `u` is a tuple ϕ = `(p,h)` where:

- `p` is a correct and executable plan for `u`, and
- `h` is a containment mapping from `u` into Π`(p)`.

(D5.9) The length of ϕ = `(p,h)` with `p` = (π, h, C), written $|\phi|$, is $|\phi| = |\pi|$.

(D5.10) Let ϕ = `(p,h)` be a query plan for `u`. The *result* of ϕ in `M`, written ϕ`(M)`, is the extension of the following query:

```
h(head(u)) ← p(M);
```

■

Remarks:

- If `(p,h)` is a query plan for `u`, then `p` is a correct plan for `u` by definition.
- As shown in Example 5.3, one correct plan can yield different query plans.
- If ϕ = `(p,h)` with `p` = (π, α, C) we often abbreviate $\phi = (\pi, \alpha, C, h)$.

■

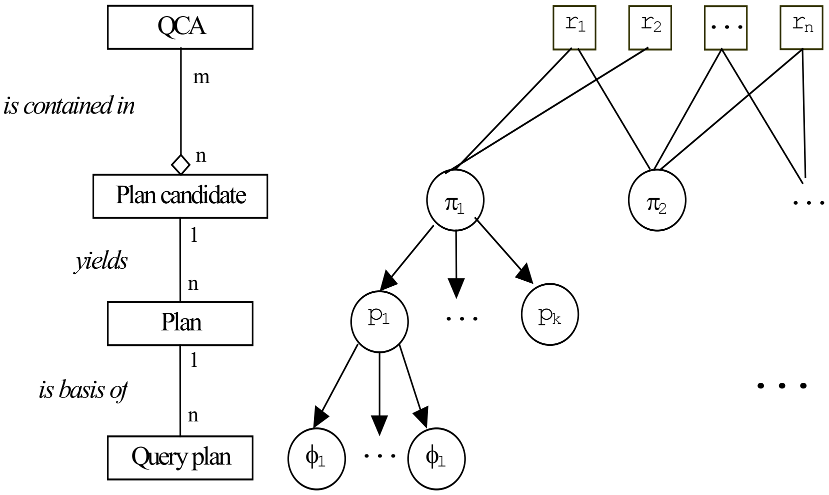


Figure 25. Relationship between QCAs, plan candidates, plans and query plans.

We distinguish carefully between plan candidates, plans and query plans (see Figure 25). A *plan candidate* is a conjunction of unrelated queries. A *plan* computes a set of tuples from which we can extract answers to u. How such answers are extracted is determined by the containment mapping. A *query plan* directly computes answers to u. There is a 1:n relationship between plan candidates and plans (differing in the query transformer), and a 1:n relationship between plans and query plans (differing in the containment mapping).

We can easily compute the result of a query plan from the result of its plan. We only have to chose the right values from the set of exported variables, as determined through the containment mapping. Therefore, computing the results of multiple query plans for the same correct plan requires to execute this plan only once. We discuss the issue of finding such cases in more detail in Section 5.4.2.

The following example intuitively shows that query planning produces the same answers as defined by the semantics of user queries. After the example, we shall formally prove this claim.

Example 5.4.
Consider a "graph database" as in Example 4.2 and the following user query and QCA:

```
u(x,y) ← edge(x,y);
r: edge(a,b),edge(c,d) ← W.v(a,b,c,d) ← somehow(a,b,c,d);
```

The plan p = []<v(a,b,c,d)> is correct for u. It yields two query plans, ϕ_1 = (p,h_1) and ϕ_2(p,h_2) with h_1 = [x→a,y→b] and h_2 = [x→c,y→d]. Assume that the wrapper query returns only one tuple (1,2,3,4). Materialising this tuple yields:

Edge	qid	a_1	a_2
	r	1	2
	r	3	4

The result of p is computed by the following query against this database:

```
p'(a,b,c,d) ← edge(qid,a,b),edge(qid,c,d),qid=r;
```

The extension of p′ is:

p′	a₁	a₂	a₃	a₄
	1	2	1	2
	1	2	3	4
	3	4	1	2
	3	4	3	4

The results of ϕ_1 and ϕ_2 are obtained by executing:

ϕ_1(M): u(a,b)← p′(a,b,c,d);
ϕ_2(M): u(c,d)← p′(a,b,c,d);

and are: ϕ_1(M) = {(1,2),(3,4)} and ϕ_2(M) = {(1,2)(3,4)}. The union of both is exactly the result prescribed by our semantics of u[9].

■

Now, we have all ingredients to define the relationship between query plans and answers to user queries. We prove that every query plan for a user query u computes only and all answers to u.

Theorem (T5.1)-(T5.2) (Soundness and completeness of query planning).
Let M = (S,Γ,Ψ) with only simple QCAs, and u ∈ CQ_S^Σ.

(T5.1) If ϕ is a query plan for u, then ϕ(M) ⊆ u(M).

(T5.2) Let t ∈ u(M). Then there exists a query plan ϕ for u with t ∈ ϕ(M).

■

Proof:

- (T5.1): Let ϕ = (p,h) with p = α<q_1,...,q_m,C>. We prove that the result of ϕ is contained in the result of u as given in Definition (D4.11) (see Figure 26). Let u have the form: u(E) ← l_1,...,l_n,cond(u). The result of the query plan, ϕ(M), is defined as the result of the query:

  h(head(u)) ← α(body′(q_1),...,body′(q_m),C);

  where body′(q_i) is obtained from body(q_i) by adding to each literal the same, fresh variable for the qid attribute. The expansion of the plan p underlying the query plan is:

  Π(p): p(e_1,...,e_l) ← α(body(q_1),...,body(q_m),C);

  and h is a containment mapping from u into Π(p). It follows that for each l_i ∈ u there exists a literal l_i′ ∈ Π(p) with h(l_i) = α(l_i′). Consider the fragments for those l_i′ (see Definition (D4.8), page 74). The fragment for every l_i′ must be admissible for l_i. If not, Π(p) could not be contained in u. Let v_i be the view induced by the fragment of l_i′, and let:

  p′(e_1,...,e_l) ← v_1,...,v_n;

  Now recall the definition of the result of a user query (see Definition (D4.11)). u(M) is:

  $$u(M) = \bigcup_{v_1 \in \Lambda_1, ...v_n \in \Lambda_n,} < v_1, v_2, ..., v_n, cond(u) >;$$

[9] This example shows that all our considerations only hold under set semantic.

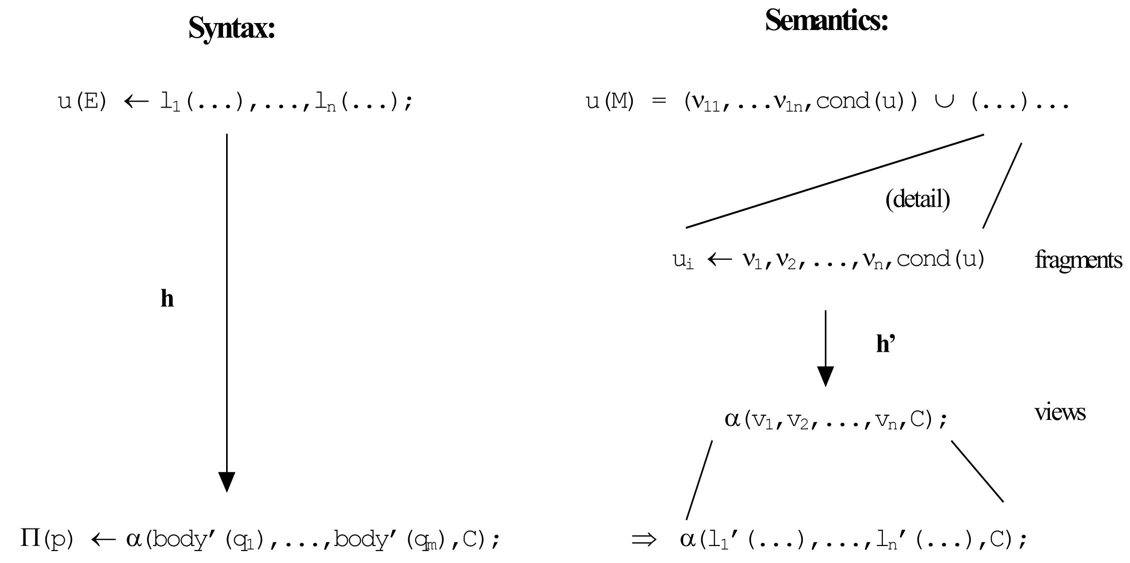

Syntax:
Semantics:
u(E) ← l1(...),...,ln(...);
u(M) = (ν11,...ν1n,cond(u)) ∪ (...)...
(detail)
ui ← ν1,ν2,...,νn,cond(u)
fragments
h
h'
α(ν1,ν2,...,νn,C);
views
Π(p) ← α(body'(q1),...,body'(qm),C);
⇒ α(l1'(...),...,ln'(...),C);

Figure 26. Illustration for the proof of soundness of query planning.
From the containment of the plan expansion in the user query we can conclude the containment of the extension of the plan expansion in the extension of the user query.

Notice that p' is one of the elements of that union – u(M) is computed out of all combinations of admissible fragments of any QCA in M, and p' is one combination of admissible fragments of a subset of the QCAs in M, namely those that are in p. Now first assume that cond(u), C and α are empty. It follows immediately that p' is contained in u(M). If cond(u) is not empty, then C would have been constructed such that C implies h(cond(u)), since ϕ is contained in u. On the other hand, α is constructed such that it follows the joins in u – the source of a variable renaming in α is always a join in u that is not present in p. Therefore, $\alpha(p',C) \subseteq u(M)$ with some mapping h'.
Furthermore, ϕ(M) is contained in α(p',C); the latter is constructed from the former by only removing literals – none of them can contain an exported literal by construction of p' – and removing literals can at most increase the number of tuples. By transitivity of query containment (see Lemma (L2.7)) it follows that $\phi(M) \subseteq u(M)$.

- (T5.2): Assume that $t \in u(M)$, and let D be the virtual database of M. t is the result of (at least) one conjunct of u(M). Let q be:

  $$\text{head}(u) \leftarrow \nu_1, \ldots \nu_n, \text{cond}(u);$$

  be the query that corresponds to this conjunct. Now consider the plan $p = [\,]\langle q_1, .., q_n \rangle$, where q_i is the mediator query of the QCA of which ν_i is a fragment. The materialisation of all QCAs in p yields a database $D' \subseteq D$. The extension of u in D' must also contain t.
  We construct a mapping h' and a query transformer (α', C') for p, yielding a query plan $p' = \alpha'\langle q_1, .., q_n, C' \rangle$, such that h' is containment mapping from q into $\Pi(p')$. If we can also show that p' is executable, then $\phi = (p', h')$ is a query plan for u. It then only remains to show that ϕ computes t.

Let h' be the mapping from q into $\Pi(p')$ that takes as targets for each $l_i \in q$ that literal of p' that corresponds to the fragment that induces v_i. Let $p' = \alpha'<q_1, ..., q_n, C'>$ with α' being a renaming implying all joins implied by u, and $C' = h'(cond(u))$.

- h' is be a containment mapping since all fragments are admissible[10].
- p' is executable. All 'critical variables', i.e., variables necessary to enforce conditions or joins not present in q, are exported in p since the fragments are admissible.
- $\phi = (p', h')$ computes t. The only remaining problem that could occur is a join in p', for instance between attributes a_1 and a_2, which is not present in q. But t is obtained through materialising all QCAs in p'. Therefore, t cannot contain different values for a_1 and a_2 in such a case.

■

It follows that every algorithm that finds *all query plans* for a given user query is sound and complete wrt. the semantics of user queries given in Definition (D4.11).

5.1.2 A Length Bound for Query Plans

The question remains whether it is possible to find all query plans. Obviously, if a plan $[]<v(a_1,b_1)>$ is correct for a user query u, then also $[]<v(a_1,b_1),v(a_2,b_2)>$, $[]<v(a_1,b_1),v(a_2,b_2),v(a_3,b_3)>$, etc. are correct plans, and each yields at least one query plan for u. Hence, there exists an *infinite number of correct plans* as soon as there exists at least one, and, accordingly, there exists an *infinite number of query plans*.

Fortunately, we can show that it suffices to consider only a finite number of query plans. We prove that any query plan that is longer than a certain bound cannot contribute any tuples not already obtained by smaller query plans. For the proof, we proceed in three steps:

- First, we define *minimality* of query plans.
- Then, we prove that any non-minimal query plan is subsumed by a minimal query plan.
- Finally, we prove that any query plan that is longer than the user query is not minimal.

Together, these results show that it is possible to devise sound, complete, *and terminating* algorithms for query planning.

Definition (D5.11) (Minimal query plans).
Let $M = (\Sigma, \Psi, \Gamma)$ with only simple QCAs and $u \in CQ_S^\Sigma$. Let $\phi = (\pi, h, \alpha, C)$ be a query plan for u.

(D5.11) ϕ is *minimal* iff $\nexists\ \phi' = (\pi', h'\alpha, C)$ for u with $\pi' \subset \pi$.

■

Our aim is to exclude query plans whose results are certainly contained in another, shorter query plan. The definition requires that a query plan can only be non-minimal if a shorter query plan with *identical query transformer and containment mapping* exists. The following example shows that this condition is necessary: If a query plan ϕ' is obtained from a query plan ϕ by removing from ϕ one or more mediator queries and adapting the transformation or the containment mapping, then ϕ might produce tuples that are not produced by ϕ'.

[10] This conclusions does not hold for queries with complex conditions.

Example 5.5.
Consider the following user query $\texttt{u}$, and QCAs $\texttt{r}_1$ and $\texttt{r}_2$ with $\texttt{q}_1 = \texttt{medq(r}_1\texttt{)}$ and $\texttt{q}_2 = \texttt{medq(r}_2\texttt{)}$:

$\texttt{u(x,y)} \leftarrow \texttt{rel}_1\texttt{(x,y),rel}_2\texttt{(x,y);}$

$\texttt{r}_1\texttt{: rel}_1\texttt{(a}_1\texttt{,b}_1\texttt{),rel}_2\texttt{(a}_1\texttt{,c}_1\texttt{)} \leftarrow \texttt{W}_1\texttt{.v(a}_1\texttt{,b}_1\texttt{,c}_1\texttt{)} \leftarrow \texttt{rel}_\texttt{x}\texttt{(a}_1\texttt{,b}_1\texttt{),rel}_\texttt{y}\texttt{(a}_1\texttt{,c}_1\texttt{);}$
$\texttt{r}_2\texttt{: rel}_2\texttt{(a}_2\texttt{,c}_2\texttt{)} \leftarrow \texttt{W}_2\texttt{.v(a}_2\texttt{,c}_2\texttt{)} \leftarrow \texttt{rel}_\texttt{z}\texttt{(a}_2\texttt{,c}_2\texttt{);}$

We consider the two plans $\texttt{p}_1$ and $\texttt{p}_2$:

$\texttt{p}_1\texttt{(a}_1\texttt{,c}_1\texttt{)} \leftarrow \texttt{[b}_1\texttt{→c}_1\texttt{]<q}_1\texttt{(a}_1\texttt{,b}_1\texttt{,c}_1\texttt{)>;}$
$\texttt{p}_2\texttt{(a}_2\texttt{,c}_2\texttt{)} \leftarrow \texttt{[a}_1\texttt{→a}_2\texttt{,b}_1\texttt{→c}_2\texttt{]<q}_1\texttt{(a}_1\texttt{,b}_1\texttt{,c}_1\texttt{),q}_2\texttt{(a}_2\texttt{,c}_2\texttt{)>;}$

which yield two correct query plans $\phi_1 = \texttt{(p}_1\texttt{, [x→a}_1\texttt{,y→b}_1\texttt{])}$ and $\phi_2 = \texttt{(p}_2\texttt{, [x→a}_2\texttt{, y→c}_2\texttt{])}$ for $\texttt{u}$. The set of QCAs used by ϕ_1 is identical to the set of QCAs used by ϕ_2 with one QCA removed. However, the query transformer of ϕ_1 is different from that of ϕ_2. We cannot conclude anything about the *relationship of the extensions* of ϕ_1 and ϕ_2. ϕ_1 returns all tuples that exist both in $\texttt{W}_1\texttt{.rel}_\texttt{x}$ and $\texttt{W}_1\texttt{.rel}_\texttt{y}$. In contrast, ϕ_2 returns all tuples that exist both in $\texttt{W}_1\texttt{.rel}_\texttt{x}$ and $\texttt{W}_2\texttt{.rel}_\texttt{z}$. Those sets might be disjoint, overlapping or identical. Both query plans are minimal.

■

The following lemma proves that any non-minimal query plan computes a result that is equal to or contained in a shorter query plan.

Lemma (L5.3) (Non-minimal query plans are redundant).
Let $\texttt{M} = (\Sigma,\Gamma,\Psi)$ with only simple QCAs, and $\texttt{u} \in \texttt{CQ}_\texttt{S}^\Sigma$.

(L5.3) If $\phi = \texttt{(p,h)}$ is a non-minimal query plan for $\texttt{u}$, then there exist a query plan ϕ' for $\texttt{u}$ with $\phi\texttt{(M)} \subseteq \phi'\texttt{(M)}$ and $|\phi'| < |\phi|$.

■

Proof:
Let $\texttt{p} = \alpha\texttt{<q}_1\texttt{,...,q}_\texttt{n}\texttt{,C>}$. Since ϕ is not minimal, we can remove at least one mediator query from $\texttt{p}$. Without loss of generality we remove the last, leading to the query plan $\texttt{p}' = \alpha\texttt{<q}_1\texttt{,...,q}_{\texttt{n-1}}\texttt{,C>}$. $\texttt{h}$ is a containment mapping from $\texttt{u}$ into $\Pi\texttt{(p}'\texttt{)}$ and also a containment mapping from $\texttt{u}$ into $\Pi\texttt{(p)}$. It follows that $\texttt{h}$ does not map any variable from $\texttt{u}$ into a variable appearing only in $\texttt{q}_\texttt{n}$. Hence, no such variable is exported in $\texttt{p}$, and $\phi\texttt{(M)}$ computes tuples consisting of the same set of variables as the tuples computed by $\phi'\texttt{(M)}$. Since $\texttt{p}'$ is obtained from $\texttt{p}$ by removing a query, $\texttt{p}'$ contains at most less literals, less joins, and less conditions than $\texttt{p}$, but never more. Therefore, $\phi\texttt{(M)}$ and $\phi'\texttt{(M)}$ can only differ in that $\texttt{p}$ filters out tuples.

■

The following theorem proves that any plan that consists of more QCAs than the user query has literals is non-minimal.

Lemma (L5.4) (Length bound on minimal query plans).
Let $\texttt{M} = (\Sigma,\Gamma,\Psi)$ with only simple QCAs, $\texttt{u} \in \texttt{CQ}_\texttt{S}^\Sigma$, and $\phi = \texttt{(p,h)}$ be a query plan for $\texttt{u}$.

(L5.4) If $|\texttt{p}| > |\texttt{u}|$ then ϕ is not minimal.

■

The proof follows from Lemma 3.5 in [LMSS95]. The idea is that, if u has k literals, then h needs at most k targets in p, because all symbols occurring in one literal must be mapped to one literal in p. If p is longer than k, then p must contain a QCA having no literal being a target in h. All such QCAs can be removed without affecting the result of the query plan.

Theorem (T5.5) (Length bound for query plans).
Let $M = (\Sigma, \Gamma, \Psi)$ with only simple QCAs, and let $u \in CQ_S^{\Sigma}$. Furthermore, let $t \in u(M)$.

(T5.5) There exists a query plan ϕ for u with $t \in \phi(M)$, and $|\phi| \leq |u|$.
■

The theorem follows immediately from Lemma (L5.3) and Lemma (L5.4).

We finish this section by applying our transformations to Example 4.2, page 72.

Example 5.6.
Consider the following user query u and QCA r (see also Figure 21, page 72):

u(x,y,z) ← edge(x,y),edge(y,z);
r: edge(a,b),edge(b,c) ← W.v(a,b,c) ← threewaypaths(a,b,c);

We assume that executing the wrapper query of r:

W.v	from	via	to
	1	3	3
	2	3	4

In Example 4.2 we showed that u(M) is:

edge	from	via	to
	1	3	3
	1	3	4
	3	3	3
	3	3	4
	2	3	3
	2	3	4

We now show that this answer is also obtained using query plans. Let q = medq(r). Since the size of the user query is two we must only consider query plans of length less than or equal to two. There are two such plan candidates, π_1 = {q} and π_2 = {q,q}, leading to the following (correct) plans:

p_1= []<q(a,b,c)>;
p_2= []<q(a_1,b_1,c_1),q(a_2,b_1,c_2)>;

and a great number of further correct plans, which have more joins and therefore cannot produce more results.

For p_1, there is only one query plan, ϕ_1. In contrast, there exist four query plans for p_2:

ϕ_1 = ({q},[x→a,y→b,z→c],[],∅);
ϕ_2 = ({q,q},[x→a_1,y→b_1,z→b_2],[a_2→b_1],∅);
ϕ_3 = ({q,q},[x→a_1,y→b_1,z→c_2],[b_2→b_1],∅);
ϕ_4 = ({q,q},[x→b_1,y→c_1,z→b_2],[a_2→c_1],∅);
ϕ_5 = ({q,q},[x→b_1,y→c_1,z→c_2],[b_2→c_1],∅);

The result of those query plans is defined as the extension of the following queries q_1 - q_5:

q_1(a,b,c) ← q(a,b,c);
q_2(a_1,b_1,b_2) ← q(a_1,b_1,-),q(b_1,b_2,-);

$q_3(a_1,b_1,c_2) \leftarrow q(a_1,b_1,-),q(-,b_1,c_2);$
$q_4(b_1,c_1,b_2) \leftarrow q(-,b_1,c_1),q(c_1,b_2,-);$
$q_5(b_1,c_1,c_2) \leftarrow q(-,b_1,c_1),q(-,c_1,c_2);$

The extensions are:

	from	**via**	**to**
q₁	1	3	3
	2	3	4
q₂		-	
q₃	1	3	3
	1	3	4
	2	3	3
	2	3	4
q₄		-	
q₅	3	3	3
	3	3	4

which is, under set semantics, the same result as prescribed by the semantics of u.

■

5.2 Generate & Test Algorithm

In this section we describe the *generate & test algorithm* (GTA) for query planning. We show that the algorithm finds all query plans for a user query within the previously proven length limitation and hence is sound and complete wrt. the semantics of user queries defined in Section 4.3. We analyse its worst-case and average-case time complexity with respect to the critical factors of query planning, i.e., the number of QCAs and the size of the user query.

The GTA proceeds in two steps: First, it generates all promising plan candidates, i.e., those that are not longer than the user query at hand. We devise an algorithm for plan enumeration in Section 5.2.1. For each plan candidate separately, the GTA tries to find query transformers and containment mappings that turn a candidate plan into a query plan. In Section 5.2.2 we show that these two problems, although they appear to be very different, can be combined in a single algorithm that has essentially the same structure as the BFA (see page 32). We prove that the resulting algorithm is sound and complete for query planning. In Section 5.2.3 we describe an implementation of the GTA and analyse its complexity. An improvement of the GTA based on pre-computation of so-called *buckets* is introduced in Section 5.2.4. Buckets are a first step towards the improved bucket algorithm presented in Section 5.3.

5.2.1 Candidate Enumeration

As shown in Theorem (T5.5), each minimal query plan for a user query u requires at most as many mediator queries as u has literals. If we enumerate all plan candidates up to that length and further consider find any query plan based on such candidates, we end up with a set of query plans for u that include all minimal ones.

Algorithm 5 generates all such plan candidates. In a plan candidate, the order of the mediator queries does not matter, and that mediator queries may appear more than once.

The algorithm assumes an arbitrary order on the set of mediator queries. This order is used to prevent multiple enumeration of candidates with the same mediator queries. The algorithm generates candidates of increasing length, starting from length one (line 2-4). It then continuously adds to each candidate π all queries with order number higher or equal to the order number of the query added lastly to π (line 5-15).

Algorithm 5. Enumerating all plan candidates.

```
Input: Ordered set $\{q_1,q_2,...q_n\}$ of mq's;
       User query u with $k=|u|$;
Output: Set P of all plan candidates for u;

1:  $P = \emptyset$;
2:  for i=1 to n {                   % Initial set of candidates, length=1
3:    $P = P \cup q_i$;
4:  }
5:  for i=2 to k {                   % Incrementally increase length
6:    $P' = \emptyset$;
7:    foreach $\pi \in P$ {
8:      $P' = P' \cup \pi$;
9:      m = lastQuery($\pi$);         % Returns index of last mq in $\pi$;
10:     for j=m to n {
11:       $P' = P' \cup \{\pi \cup q_j\}$;
12:     end for;
13:   end for;
14:   $P = P'$;
15: end for;
16: return P;                        % P is the set of all candidates plans
```

Example 5.7.
Imagine a user query of length three, and let $\{q_1, q_2, q_3\}$ be the set of mediator queries. Let P_i be the set of candidates of length i. Algorithm 5 first generates $P_1 = \{q_1, q_2, q_3\}$. Then it generates candidates of length two by adding only queries with higher or equal order number to any candidate in P_1. Therefore, $P_2 = \{(q_1,q_1), (q_1,q_2), (q_1,q_3), (q_2,q_2), (q_2,q_3), (q_3,q_3)\}$. Finally, P_3 is computed in the same manner: $P_3 = \{(q_1,q_1,q_1), (q_1,q_1,q_2), (q_1,q_1,q_3), (q_1,q_2,q_2), (q_1,q_2,q_3), (q_1,q_3,q_3), (q_2,q_2,q_2), (q_2,q_2,q_3), (q_2,q_3,q_3), (q_3,q_3,q_3)\}$. Because the user query has length 3, no more candidates must be constructed. The complete set of candidate plans is $P = P_1 \cup P_2 \cup P_3$.
■

We compute the number of candidates, i.e., $|P|$, in the following way. The number of combinations of length k of n different queries, C^n_k, where elements may occur multiple times and the order does not matter, is the number of "*combinations with repetitions*":

$$C^n_k = \binom{n + k - 1}{k}$$

In our case, we are interested in the number $C^n_{\leq k}$ of combinations of queries from $MQ(\Gamma)$ *up to the* length of the user query u. Let $k = |u|$ and $n = |MQ(\Gamma)|$. We get:

$$C^n_{\leq k} = \sum_{i=1}^{k} C^n_i = \sum_{i=1}^{k} \binom{n + i - 1}{i}$$

Using Stirling's Formula and $t = n+k-1$, we can estimate:

$$\left(\frac{t}{k}\right)^{k} \leq C^{n}_{\leq k} \leq \left(\frac{et}{k}\right)^{k}$$

which is $O((t/k)^{k}) \approx O(n^{k})$ for $n >> k$. The number of candidate plans is hence exponential in the length of the user query. We shall use this result in Section 5.2.3.

5.2.2 Finding Query Transformers

The main idea of the GTA is to enumerate a set of plan candidates and test for each candidate whether or not it can be turned into a query plan. For this test, it does not suffice to only prove query containment. Even if a plan candidate is not contained in the user query in first place, it can still yield query plans through appropriate query transformers.

We highlight the types of problems we face when we search suitable transformations by an example.

Example 5.8.
Consider the following user query u and mediator queries q_1–q_3:

$u(a,b) \leftarrow rel_1(a,b,b), rel_2(a,c), b<100;$

$q_1(x,y,z) \leftarrow rel_1(x,y,z), x=5;$
$q_2(u,v) \leftarrow rel_2(u,v);$
$q_3(u,v) \leftarrow rel_2(u,v), u=4;$

Consider the following two plans p_1 and p_2:

$p_1=[]<q_1(x,y,z), q_2(u,v)>;$
$p_2=[]<q_1(x,y,z), q_3(u,v)>;$

Neither p_1 nor p_2 are contained in u. However, we can execute the following plan:

$p_3=[u\rightarrow x, z\rightarrow y]<q_1(x,y,z), q_2(u,v), y<100 >;$

which expands to:

$\Pi(p_3): p_3'(x,y,v) \leftarrow q_1(x,y,y), q_2(x,v), y<100;$

$\Pi(p_3)$ is contained in u and executable. The result of p_3 may be computed by executing p_1 and then applying appropriate filter operations inside the mediator. Depending on the capabilities of the wrapper that offers q_2, we might also be able to execute p_3 by first executing q_1, then pushing x values as selection conditions into q_2, and only applying "$y < 100$" inside the mediator.

But we cannot find a suitable plan transformer for p_2: Joining q_1 and q_3 through x and z implies the condition "$x = 5 \wedge x = 4$", which is not satisfiable.

■

Informally, we can search a suitable query transformer for a given candidate plan π by carrying out the following steps:

1. Construct the expansion q of π and search a variable renaming α for q such that:
   - a symbol mapping g from u into α(q) exists that fulfils (CM1) - (CM3) (see Definition (D2.16), page 20), and
   - all joins that are implied by α are executable.

If no such α exists, π cannot be turned into a correct plan. Note that more than one variable renaming may exist. We must consider all.

2. To ensure (CM4), we check if we can add conditions $\mathtt{C}$ to $\alpha(\mathtt{q})$ such that:
   - there exists a containment mapping $\mathtt{h}$ from $\mathtt{u}$ into $\alpha(\pi,\mathtt{C})$, and
   - $\alpha(\pi,\mathtt{C})$ is executable and satisfiable.

If we have found such α, $\mathtt{C}$, and $\mathtt{h}$, then $\mathtt{p} = (\pi,\alpha,\mathtt{C})$ is a correct plan for $\mathtt{u}$, and $\phi = (\mathtt{p},\mathtt{h})$ is a query plan for $\mathtt{u}$.

To turn this informal procedure into an algorithm, we adapt our apparatus for testing breadth-first query containment (see Section 2.3.3) such that it automatically constructs plan transformers together with containment mappings. To this end, we define an *extended containment mapping (ECM)* as a PCM (see Definition (D2.19), page 25) with an additional query transformer. We shall prove that we can build *complete ECMs* out of *partial ECMs* in the same incremental fashion as with PCMs. This requires an adaptation of the definition of `union` and `compatibility`, taking into account that two in first place incompatible ECMs can be *made compatible*, for instance by adding a join.

Definition (D5.12)-(D5.15) (Extended containment mappings).
Let $\mathtt{M} = (\Sigma,\Gamma,\Psi)$ with only simple QCAs, $\mathtt{u} \in \mathtt{CQ}_{\mathtt{S}}^{\Sigma}$, and π be a plan candidate for $\mathtt{u}$.

(D5.12) Let $\mathtt{L}$ be a subset of the literals of $\mathtt{u}$, and $\mathtt{S} = \mathtt{sym(L)}$. Let $\mathtt{h}$ be a symbol mapping $\mathtt{h}: \mathtt{S} \mapsto \alpha(\mathtt{sym}(\pi))$. Then $\varepsilon = (\mathtt{h},\alpha,\mathtt{C})$ is an *partial extended containment mapping* (ECM) from $\mathtt{u}$ to π iff:

- $\mathtt{h}$ is a partial containment mapping from $\mathtt{u}$ into $\alpha(\pi,\mathtt{C})$, and
- $\alpha(\pi,\mathtt{C})$ is executable.

(D5.13) Let $\varepsilon_1 = (\mathtt{h}_1,\alpha_1,\mathtt{C}_1)$, $\varepsilon_2 = (\mathtt{h}_2,\alpha_2,\mathtt{C}_2)$ be two partial ECMs from $\mathtt{u}$ to π defined over $\mathtt{S}_1 \subseteq \mathtt{sym(u)}$ and $\mathtt{S}_2 \subseteq \mathtt{sym(u)}$, respectively. ε_1 and ε_2 are *reconcilable* iff, $\forall$ $\mathtt{s} \in \mathtt{S}_1 \cap \mathtt{S}_2$, one of the following conditions hold:

- $\alpha_1(\mathtt{h}_1(\mathtt{s})) = \alpha_2(\mathtt{h}_2(\mathtt{s}))$, or
- $\mathtt{h}_1(\mathtt{s}),\mathtt{h}_2(\mathtt{s}) \in \mathtt{export}(\pi)$

(D5.14) We define the *union operator* on reconcilable partial ECMs. Let $\varepsilon_1 = (\mathtt{h}_1,\alpha_1,\mathtt{C}_1)$, $\varepsilon_2 = (\mathtt{h}_2,\alpha_2,\mathtt{C}_2)$ be reconcilable, and let $\mathtt{S}^1 - \mathtt{S}^4$ be defined as follows:

- $\mathtt{S}^1 = \{\mathtt{s} \in \mathtt{S}_1 \wedge \mathtt{s} \notin \mathtt{S}_2\}$
- $\mathtt{S}^2 = \{\mathtt{s} \in \mathtt{S}_2 \wedge \mathtt{s} \notin \mathtt{S}_1\}$
- $\mathtt{S}^3 = \{\mathtt{s} \in \mathtt{S}_1 \cap \mathtt{S}_2 \wedge \alpha_1(\mathtt{h}_1(\mathtt{s})) = \alpha_2(\mathtt{h}_2(\mathtt{s}))\}$
- $\mathtt{S}^4 = \{\mathtt{s} \in \mathtt{S}_1 \cap \mathtt{S}_2 \wedge \mathtt{s} \notin \mathtt{S}^3\}$

Then $(\mathtt{h},\alpha,\mathtt{C}) = (\mathtt{h}_1,\alpha_1,\mathtt{C}_1) \cup (\mathtt{h}_2,\alpha_2,\mathtt{C}_2)$ is the extended partial containment mapping from $\mathtt{S}_1 \cup \mathtt{S}_2 \mapsto \alpha(\mathtt{sym}(\pi))$, where $\mathtt{h}$, α, and $\mathtt{C}$ are defined as follows:

- $\forall\ \mathtt{s} \in \mathtt{S}^1\!: \mathtt{h(s)} = \alpha_1(\mathtt{h}_1(\mathtt{s}))$
- $\forall\ \mathtt{s} \in \mathtt{S}^2\!: \mathtt{h(s)} = \alpha_2(\mathtt{h}_2(\mathtt{s}))$
- $\forall\ \mathtt{s} \in \mathtt{S}^3\!: \mathtt{h(s)} = \alpha_1(\mathtt{h}_1(\mathtt{s}))$
- $\forall\ \mathtt{s} \in \mathtt{S}^4\!: \mathtt{h(s)} = \alpha_1(\mathtt{h}_1(\mathtt{s}))$
- $\alpha = \alpha_1 \cup \alpha_2|_{\mathtt{S2}} \cup \{\mathtt{s}_1 \rightarrow \mathtt{s}_2 \mid \exists\ \mathtt{s} \in \mathtt{S}^4\!: \mathtt{s}_1 = \alpha_1(\mathtt{h}_1(\mathtt{s})) \wedge \mathtt{s}_2 = \alpha_2(\mathtt{h}_2(\mathtt{s}))\}$
- $\mathtt{C}$ is the minimal set of conditions such that:

$$\mathtt{C} \wedge \alpha(\mathtt{cond}(\pi)) \Rightarrow \mathtt{cond}(\mathtt{u},\mathtt{S}_1 \cup \mathtt{S}_2)$$

(D5.15) Let $\varepsilon_1 = (h_1, \alpha_1, C_1)$, $\varepsilon_2 = (h_2, \alpha_2, C_2)$ be two reconcilable partial ECMs, and let $\varepsilon = \varepsilon_1 \cup \varepsilon_2 = (h, \alpha, C)$. Let $S^1 - S^4$ be defined as above. ε_1 and ε_2 are *compatible*, written as $\varepsilon_1 \sim \varepsilon_2$, iff:

- $\alpha(\pi, C)$ is executable, and
- C is satisfiable.

■

Remark:
A query plan is identical to a complete ECM.

■

Unfortunately, some of the theorems for PCMs do not carry over to ECMs. For instance, the following counterexample shows that Lemma (L2.9), page 27, does not hold any more.

Example 5.9.
Consider the following query u and mediator queries q_1, q_2 and q_3:

$u(a,b,c) \leftarrow rel_1(a,b), rel_2(b,c), rel_3(b,c);$

$q_1(x) \leftarrow rel_1(x,x), x=5;$
$q_2(y,z) \leftarrow rel_2(y,z), z=7;$
$q_3(v) \leftarrow rel_3(v,v);$

For each mediator query we can construct one partial ECM ε_1, ε_2 and ε_3:

$\varepsilon_1 = \{[a\rightarrow x, b\rightarrow x], [], \emptyset\};$
$\varepsilon_2 = \{[b\rightarrow y, c\rightarrow z], [], \emptyset\};$
$\varepsilon_3 = \{[b\rightarrow v, c\rightarrow v], [], \emptyset\};$

Building their union leads to (the corresponding plan expansion is given behind each $\Rightarrow$):

$(\varepsilon_1\cup\varepsilon_2) = \{[a\rightarrow x, b\rightarrow x, c\rightarrow z], [y\rightarrow x], \emptyset\} \Rightarrow \langle rel_1(x,x), x=5, rel_2(x,z), z=7\rangle;$
$(\varepsilon_1\cup\varepsilon_3) = \{[a\rightarrow x, b\rightarrow x, c\rightarrow v], [v\rightarrow x], \emptyset\} \Rightarrow \langle rel_1(x,x), x=5, rel_3(x,x)\rangle;$
$(\varepsilon_2\cup\varepsilon_3) = \{[b\rightarrow v, c\rightarrow v], [y\rightarrow v, z\rightarrow v], \emptyset\} \Rightarrow \langle rel_2(v,v), v=7, rel_3(v,v)\rangle;$

The three ECMs are pairwise compatible, but $\varepsilon_1 \nsim (\varepsilon_2 \cup \varepsilon_3)$. This union expands to $\langle rel_1(x,x), x=5, rel_2(x,x), x=7, rel_3(x,x)\rangle$, which is not satisfiable.

■

As a consequence, we cannot test compatibility of ECMs in a pairwise fashion. We have to test compatibility of each new ECM we union with an existing ECM.

Algorithm 6. The generate & test algorithm (GTA).
Input: Mediator $M = (\Sigma, \Gamma, \Psi)$ with only simple QCAs, user query $u \in CQ_S^\Sigma$.
Output: Set of query plans for u such that their union computes $u(M)$.
Algorithm: Compute the set of plan candidates that are not longer than $|u|$. For each plan candidate π, compute all partial ECMs from the literals of u into the literals of the expansion of π. Build all combinations of those ECMs where each combination contains exactly one ECM for each literal of u. Let $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$ be such a combination. Check incrementally if $\varepsilon_1 \sim \varepsilon_2$, $\varepsilon_3 \sim (\varepsilon_1 \cup \varepsilon_2)$, ..., $\varepsilon_n \sim (\varepsilon_1 \cup \varepsilon_2 \cup \ldots \cup \varepsilon_{n-1})$. If these checks succeed for a plan candidate π and yield $\varepsilon = (h, \alpha, C) = \varepsilon_n \cup \varepsilon_1 \cup \varepsilon_2 \cup \ldots \cup \varepsilon_{n-1}$, then output $\phi = (\pi, \alpha, C, h)$.

■

Remarks:

- The GTA computes all minimal query plans based on a plan candidate. The 1:n relationship between plan candidates and query plans is hidden in the enumeration of all combinations of ECMs.
- Using containment mappings for testing query containment is only sound but not complete for $\mathtt{CQ_C}$ queries (see Theorem (T2.3), page 20). Therefore, query planning based on containment mappings can at best be sound, but not complete.
- A simple method to speed up Algorithm 6 is the following. A plan candidate π can only yield a query plan for a user query $\mathtt{u}$ if it contains each relation appearing in $\mathtt{u}$. We can filter out candidates that do not fulfil this condition be assigning each QCA $\mathtt{r}$ a bitvector $\mathtt{B_r}$ with one bit per relation of Σ. The bit representing `rel` is set if `rel` appears in `r`. In the same way we assign $\mathtt{B_u}$ to `u`. We must test a plan candidate π only if:

$$\mathtt{B_u} = \mathtt{B_u} \cap \left(\bigcup_{\mathtt{r} \in \pi} \mathtt{B_r} \right)$$

  where union and intersection are interpreted as bit-wise "`and`" and bit-wise "`or`", respectively. This test is a linear in the size of Σ.

■

Theorem (T5.6)-(T5.7) (Completeness and soundness of the GTA).
Let $\mathtt{M} = (\Sigma, \Psi, \Gamma)$ with only simple QCAs, and $\mathtt{u} \in \mathtt{CQ}_\mathtt{S}^\Sigma$.

(T5.6) The GTA is sound, i.e., every query plan it computes is a query plan for `u`.

(T5.7) The GTA is complete, i.e., it computes every minimal query plan for `u`.

■

Proof:

- (T5.6): Suppose the GTA has computed $\phi = (\mathtt{p}, \mathtt{h})$ with $\mathtt{p} = (\pi, \alpha, \mathtt{C})$ for a user query `u`. We must show that (a) `h` is a containment mapping from `u` into `p` and that (b) `p` is executable.
  (a) `p` contains a target literal for each literal of `u` since `p` is composed of such literals (the initial ECMs). `h` is a mapping into those target literals of `p`. Through the union of ECMs it is guaranteed that no variable is mapped to two different variables. Since all queries in `p` are in normal form they do not contain constants in literals. Finally, the conditions of `p` imply the conditions of `u` (including the moved-out constant conditions); if this were not the case, the ECMs could not be compatible. Therefore, `h` is a containment mapping from `u` into `p`.
  (b) By definition of QCAs, every query $\mathtt{q} \in \pi$ is executable. Furthermore, every initial ECM is by Definition (D5.12) executable. Finally, computability of two ECMs only holds if the union is executable. Therefore, `p` must be executable.

- (T5.7): Suppose $\phi = (\mathtt{p}, \mathtt{h})$ is a minimal query plan for `u` with $\mathtt{p} = (\pi, \alpha, \mathtt{C})$. Following Theorem (T5.5), π cannot be longer than `u`. Therefore, π is tested as plan candidate. The

GTA tests every possible combination of ECMs from the literals of u into literals of π, including the combination of targets used in h.

By Definition (D5.8), α and C are minimal in the sense that α does not introduce unnecessary joins and C does not unnecessarily restrict the result of ϕ. On the other hand, the initial ECMs for the targets of h in π have minimal query transformer (Definition (D5.12), and each union of two ECMs again leads to a minimal query transformer (Definition (D5.14)). Since minimality is unique (up to isomorphism of variable symbols), the pair (α,C) is computed.

■

The previous theorems do not imply that the GTA computes *only* minimal query plans. In fact, the GTA does compute non-minimal query plans. We shall come back to this issue in Section 5.4.

Consider the following, tempting idea to improve the GTA. Imagine that we have computed a query plan ϕ based on a plan candidate π with |π| < |u|. Is it then possible to remove any plan candidate π′ with π ⊆ π′ from further consideration? The answer is no, as the following example shows.

Example 5.10.
Consider the following user query u and mediator query q:

```
u(a,b) ← rel(a,b),rel(c,c);
q(x,y) ← rel(x,y);
```

p_1 = [x→y]<q(x,y)> is a correct plan based on the containment mapping [a→y,b→y, c→y]. But p_1 is not the best we can find: p_2 = []<q(x,y),q(z,z)> with containment mapping [a→x,b→y,c→y] never computes less tuples, but potentially more.

■

In the previous example we found a short plan p_1 and a longer plan p_2, which contains all queries of p_1. But p_2 is less restrictive then p_1 and will usually compute more tuples. This shows that we cannot stop to extend plan candidates even if we found a query plan.

5.2.3 Implementation and Complexity of the GTA

Our implementation of the GTA proceeds as follows. First, we use Algorithm 5 to enumerate all necessary plan candidates. For each of those candidates, we explore the search space of ECMs and unions of ECMs using a breadth-first search. Finding all query plans is not identical to containment testing, which only produces a “yes / no” answer. A plan can by contained in the user query in different ways, leading to different results. Since we have to compute all query plans, we must explore the *entire search space*. Therefore, a depth-first approach would not be beneficial.

The breadth-first search can directly use the BFA (Algorithm 2, see page 32). Only the functions compatible and union have to be replaced with functions ecm_compatible and ecm_union, respectively. Algorithm 7 shows ecm_compatible. We omit a description of ecm_union because it is mostly identical to ecm_compatible.

Recall Algorithm 2. It takes as input two queries. For the GTA, those will be the user query u and an expanded plan candidate π. Both must be in normal form, i.e., constants in literals must be replaced with fresh variables and a proper condition. The algorithm traverses the search space breadth-first, adding new literals to the current partial containment mapping in each loop. Consider an arbitrary node x in the search tree (see Figure 9, page 30). For the GTA, x represents a partial ECM $\varepsilon_L = (h_L, \alpha_L, C_L)$ for a subset L of the literals of u into a query $\alpha_L(Q, C_L)$, where Q is a subset of the literals of π represented by the path to x. Extending x by adding another ECM for some literal $l \notin L$ requires that we compute compatibility of the two ECMs through ecm_compatible and then build their union.

Algorithm 7. Compatibility of partial ECMs.

```
1:  Function ecm_compatible(ECM $\varepsilon_1$, $\varepsilon_2$, Query u,$\pi$) : boolean
      % $\varepsilon_1$ = ($h_1$,$\alpha_1$,$C_1$);
      % $\varepsilon_2$ = ($h_2$,$\alpha_2$,$C_2$);
2:    $\alpha$ = $\alpha_1 \cup \alpha_2$;                 % Union of renamings
3:    $h_1'$ = $\alpha_2(h_1)$;                   % Apply renamings
4:    $h_2'$ = $\alpha_1(h_2)$;
5:    V = {v | $v \in h_1' \wedge v \in h_2' \wedge h_1'(v) \neq h_2(v)$};
6:                                    % All variables with different targets
7:    foreach $v \in V$ {
8:      if $h_1'(v) \notin$ export($\pi$) then       % Both must be exported
9:        return false;
10:     end if;
11:     if $h_2'(v) \notin$ export($\pi$) then
12:       return false;
13:     end if;
14:     $\alpha = \alpha \cup \{[h_1'(v) \rightarrow h_2'(v)]\}$;    % Append new renaming, i.e., new join
15:   end for;
16:   h = $\alpha(h_1') \cup \alpha(h_2')$;
17:   if cond(<$\alpha$(p,$C_1$,$C_2$)>,image(h)) $\not\Rightarrow$ h(cond(u,org(h))) then
18:     if $\exists$C: cond(<$\alpha$(p,$C_1$,$C_2$),C>,image(h)) $\Rightarrow$ h(cond(u,org(h))) then
19:                                   % C is set of additional conditions
20:       if (sym(C) \ $\alpha$(export($\pi$))) $\neq \emptyset$ then
21:         return false;             % Conditions are not executable
22:       end if;
23:     else
24:       return false;
25:   end if;
26:   return true;
27: end function;
```

ecm_compatible takes as input two partial ECMs ε_1 and ε_2. Since both could map to literals of the same mediator query in π, the definition sets of their variable renamings could overlap, which can lead to interference. Therefore, we must first apply the variable renamings to each ECM (line 3-4).

Next, we check if the union of both ECMs requires *new renamings*. This is the case if one variable from the user query is mapped to two or more different variables in the ECM. If yes, we have to check if the new join is executable (line 5-12). At line 16 we have actually proven that ε_1 and ε_2 are reconcilable.

Finally, we check if we must add *new conditions* to remove conflicts in the containment mapping, and whether these conditions can be executed and are satisfiable. We first test if the appropriate subset of the conditions in u are not already implied by the appropriate subset of the conditions in π and the two sets of conditions in the ECMs (line 17). If not, we compute the minimal and satisfiable set C of conditions that is necessary to ensure the implication (line 18). If no such set exists, the two ECMs are not compatible. We finally have to check if all variables occurring in C are exported in p (line 20). If yes, then we have proven that the two partial ECMs are compatible.

Example 5.11.
Consider the following user query u and mediator queries q_1 and q_2:

$u(a) \leftarrow rel_1(a,a,b), rel_2(b,c,c), a<10;$

$q_1(x,y,z,u,v) \leftarrow rel_1(x,y,z), rel_2(u,x,v), v=5;$
$q_2(x,y,z,u,v) \leftarrow rel_1(x,y,z), rel_2(u,x,v), v=5, y=7;$

Analysing q_1, we find initial partial ECMs for the literal rel_1 of u: $\varepsilon_1 = ([a\rightarrow x, b\rightarrow z], [y\rightarrow x], \{x<10\})$ and for the literal rel_2 of u: $\varepsilon_2 = ([b\rightarrow u, c\rightarrow v], [x\rightarrow v], \varnothing)$. We must check if ε_1 and ε_2 are compatible.

We first compute the union of the two variable renamings and obtain $\alpha = [x\rightarrow v, y\rightarrow v]$. Applying this to h_1 and h_2 yields $h_1' = [a\rightarrow v, b\rightarrow z]$ and $h_2' = [b\rightarrow u, c\rightarrow v]$. In the next step we find that b is mapped to two different variables in h_1 respectively h_2. We get $V = \{b\}$ (see Algorithm 7). Because both target variables, z and u, are exported in q_1, we add $[z\rightarrow u]$ to α. If either z or u were not exported, we would immediately return false since the join between rel_1 and rel_2 required by u could not be achieved with q_1.

We next compute the union-mapping $h = [a\rightarrow v, b\rightarrow z, c\rightarrow v]$. We then check if we must add further conditions. The current ECM expands to: $<rel_1(v,v,z), rel_2(z,v,v), v<10, v=5>$. Since "$v < 10 \wedge v = 5 \Rightarrow v < 10$", no further conditions are necessary. Hence, ecm_compatible returns true. We can remove "$v < 10$" when we compute the union of both ECMs later.

Analysing q_2, we get the same initial ECMs. While checking their compatibility, we cannot find a suitable C because $C_1 \wedge C_2$ contains "$v = 5 \wedge v = 7$". This shows that the two ECMs are incompatible. We cannot answer u using only q_2.

■

In the previous example, there actually exist three more plan candidates: $\{q_1,q_1\}$, $\{q_1,q_2\}$ and $\{q_2,q_2\}$. For each of those, four combinations of ECMs have to be tested. For instance, for the second candidate those combinations are $\{q_1.rel_1, q_1.rel_2\}$, $\{q_1.rel_1, q_2.rel_2\}$, $\{q_2.rel_1, q_1.rel_2\}$, $\{q_2.rel_1, q_2.rel_2\}$. The first of those combinations is identical to the only combination obtained through the plan candidate $\{q_1\}$, and the last is identical to the only combination obtained through $\{q_2\}$. This shows that the GTA performs a great amount of redundant work. In the next section we describe an algorithm that saves much of this unnecessary computation.

Theorem (T5.8)-(T5.10) (Analysis of the GTA).
Let $\mathtt{M} = (\Sigma, \Gamma, \Psi)$ with only simple QCAs, and let $\mathtt{u} \in \mathtt{CQ}_\mathtt{S}^\Sigma$ with $\mathtt{k} = |\mathtt{u}|$, $\mathtt{n} = |\mathtt{MQ}(\Gamma)|$. Let $\mathtt{s}_{\mathtt{avg}} = \mathtt{avg}(|\mathtt{q}|,\ \mathtt{q} \in \mathtt{MQ}(\Gamma))$ be the average length of mediator queries in Γ. Furthermore, let $\mathtt{z}_\mathtt{i}$ be the average number of literals in candidates of length $\mathtt{i}$ covered by a literal of $\mathtt{u}$. Finally, let $\mathtt{p}_{\mathtt{com}}$, $0 \le \mathtt{p}_{\mathtt{com}} \le 1$, be the probability that two partial ECMs are compatible.

(T5.8) The GTA performs in the worst-case $\mathtt{C}_{\mathtt{GTA}}^{\mathtt{wc}}$ tests of compatibility of two ECMs:

$$\mathtt{C}_{\mathtt{GTA}}^{\mathtt{wc}} \approx \sum_{\mathtt{i}=1}^{\mathtt{k}} \left[\binom{\mathtt{n} + \mathtt{i} - 1}{\mathtt{i}} * \sum_{\mathtt{j}=2}^{\mathtt{k}} \left(\mathtt{i}\,\mathtt{s}_{\mathtt{avg}}\right)^{\mathtt{j}} \right]$$

(T5.9) The GTA performs in the average-case $\mathtt{C}_{\mathtt{GTA}}^{\mathtt{ac}}$ tests of compatibility of two ECMs:

$$\mathtt{C}_{\mathtt{GTA}}^{\mathtt{ac}} \approx \sum_{\mathtt{i}=1}^{\mathtt{k}} \left[\binom{\mathtt{n} + \mathtt{i} - 1}{\mathtt{i}} * \mathtt{z}_\mathtt{i} * \sum_{\mathtt{j}=0}^{\mathtt{k}-1} \left(\mathtt{z}_\mathtt{j}\mathtt{p}_{\mathtt{com}}\right)^{\mathtt{j}} \right]$$

(T5.10) The GTA has complexity $\mathtt{O}\left(\mathtt{e}^{\mathtt{k}} * \left(\mathtt{s}_{\mathtt{avg}}\right)^{\mathtt{k}} * \left(\mathtt{n} + \mathtt{k}\right)^{\mathtt{k}}\right)$.

■

Proof:
Essentially, we only have to combine the number of plan candidates with the complexity of the BFA. What remains to be considered is the additional complexity of computing the union and compatibility of ECMs instead of partial mappings. Note that both functions essentially do the same: checking compatibility of two ECMs entails computing and testing their union. Therefore, it suffices to consider only compatibility.

Applying a variable renaming to a mapping is linear in the number of variables in a mapping and hence negligible. The same holds for computing the union of two variable renamings. We therefore concentrate on the conditions in a ECM.

For $\mathtt{CQ}_\mathtt{S}$ queries, checking implication is simple. It suffices to define one interval per variable and translate conditions into interval operations [Kin99]. To compute the minimal set $\mathtt{C}$ of conditions that imply the appropriate conditions in $\mathtt{u}$, given two ECMs $\varepsilon_1 = (\mathtt{h}_1, \alpha_1, \mathtt{C}_1)$ and $\varepsilon_2 = (\mathtt{h}_2, \alpha_2, \mathtt{C}_2)$, we can proceed as follows: First, we assume $\mathtt{C}$ to be the set of all "appropriate" conditions from $\mathtt{u}$, which are the conditions that contains only variables from the definition sets of either $\mathtt{h}_1$ or $\mathtt{h}_2$. Then we remove from $\mathtt{C}$ all expressions implied by $\mathtt{C}_1 \wedge \mathtt{C}_2$, which is possible in time linear to $|\mathtt{C}_1| + |\mathtt{C}_2|$. For all variables of the remaining conditions we have to check if they are exported, which is again linear.

We conclude that the BFA using ECMs has the same complexity as the BFA using PCMs. Considering query transformers does not increase the complexity of the query planning problem. Based on this observation, we now prove the three theorems:

- (T5.8) is obtained in the following way: To find all query plans, it is necessary to consider all combinations of mediator queries up to the length of the user query. There exist $\mathtt{C}_\mathtt{i}^\mathtt{n}$ candidates of length $\mathtt{i}$, $1 \le \mathtt{i} \le \mathtt{k}$. For a candidate plan of length $\mathtt{i}$, the cost of testing containment in $\mathtt{u}$ is given by the worst-case complexity of the BFA, see Theorem (T2.14) (page 31). We estimate the number of literals in a candidate plan of length $\mathtt{i}$ as $\mathtt{i}\mathtt{s}_{\mathtt{avg}}$.

- (T5.9) is obtained similarly, using the average-case complexity of the BFA given in Theorem (T2.15) (page 31). The number of candidates remains the same, and we have to test them all. The difference to the worst-case analysis is that z_i models a more realistic assumption about the breadth of the search tree, and p_{com} models early pruning because of incompatible mappings.
- Theorem (T5.10) is obtained by using Stirling's Formula to obtain lower and upper bounds:
  - Upper bound:

$$\begin{aligned} C_{GTA}^{wc} &\le \sum_{i=1}^{k}\left[\left(\frac{e * (n + i - 1)}{i}\right)^{i} * k * (s_{avg} * i)^{k}\right] \\ &\le k * \left(\left(\frac{e * (n + k - 1)}{k}\right)^{k} * k * k^{k} * s_{avg}{}^{k}\right) \\ &= k^{2} * e^{k} * s_{avg}{}^{k} * (n + k - 1)^{k} \end{aligned}$$

  - Lower bound:

$$\begin{aligned} C_{GTA}^{wc} &\ge \sum_{i=1}^{k}\left[\left(\frac{(n + i - 1)}{i}\right)^{i} * (s_{avg} * i)^{k}\right] \\ &\ge \left(\frac{(n + k - 1)}{k}\right)^{k} * k^{k} * s_{avg}{}^{k} \\ &= s_{avg}{}^{k} * (n + k - 1)^{k} \end{aligned}$$

  The tightest complexity bound we can give is hence $O(e^{k} * (s_{avg})^{k} * (n + k)^{k})$. The lower bound shows that this estimation is relatively precise.

■

Remarks:

- For CQ_C queries, checking implication is not linear, but possible in $O(n^3)$, where n is the number of variables in the formulas [Ull89]. We have to perform this test for every appropriate condition from u to see whether it is necessary to include it into C or not.
- Values for z_i could be obtained as follows. Assume that the relations of Σ are uniformly distributed in $MQ(\Gamma)$. A candidate plan π of length i then contains an average of $i/|\Sigma|$ literals as potential targets for any literal of u. Let x be the probability that a literal from a candidate plan covers a literal from the user query, assuming that both literals are from the same relation. We can then estimate:

$$z_i = \frac{xi}{|\Sigma|}$$

■

While the examples given so far focused on "technically" difficult queries, we now give a more real-life example, showing the rather simple operations that *typically* have to be carried out during an execution of the GTA.

Example 5.12.
Consider a mediator M with the schema described in Table 1 (page 18), and three wrappers W_1, W_2 and W_3 that each offer only one query described by the QCAs r_1, r_2, and r_3:

r_1: clone(cid$_1$,cn$_1$,-,cl$_1$),cl$_1$<10 ← W$_1$.v$_1$(cid$_1$,cn$_1$,cl$_1$) ← COS-MIDs(cid$_1$,cn$_1$,cl$_1$);
r_2: clone(cid$_2$,cn$_2$,ct$_2$,cl$_2$),cl$_2$<100 ← W$_2$.v$_1$(cid$_2$,cn$_2$,ct$_2$) ← BACs(cid$_2$,cn$_2$,ct$_2$);
r_3: clone(cid$_3$,cn$_3$,-,-),contains(cid$_3$,gid$_3$),gene(gid$_3$,gn$_3$,-) ← W$_3$.v$_1$(cid$_3$,cn$_3$,gn$_3$,gid$_3$) ← interestingclones(cid$_3$,gid$_3$,cn$_3$,gn$_3$);

W$_1$ stores data about COSMIDs, which are clones smaller than 10 KB. The type of a clone is not exported. W$_2$ stores data about BACs, which are clones smaller than 100 KB, but does not export the actual clonelength. W$_3$ stores data about the relationship of clones and genes.

Now suppose the following three user queries. u$_1$ asks for clones smaller than 30 KB. u$_2$ searches for clones that contain the gene 'DMD', exporting the clone name and length. u$_3$ finally searches for clones having identical name and type:

u$_1$(cn) ← clone(-,cn,-,cl),cl<30;
u$_2$(cn,cl) ← clone(cid,cn,-,cl),contains(cid,gid),gene(gid,'DMD',-);
u$_3$(cn) ← clone(-,cn,cn,-);

For u$_1$, we enumerate all plans of length one: {r$_1$}, {r$_2$} and {r$_3$}. Testing containment of the expansion of {r$_1$} in u$_1$ yields the mapping h = [cn→cn$_1$,cl→cl$_1$]. Since "cl < 30" does not imply "cl < 10", we must add a filter condition to the plan, namely "cl < 10". Since cl is exported in r$_1$, the transformed plan is executable. Considering {r$_2$}, we also find the missing condition, but this plan cannot be repaired since cl is not exported. The same holds for {r$_3$}. Both {r$_2$} and {r$_3$} are hence discarded.

Next, we search plans for u$_2$. Although r$_3$ has all necessary literals it does not export cl, therefore it is cannot alone be the basis for a query plan. A promising plan candidate is π = {r$_1$,r$_3$}, i.e., picking cid, cn and cl from r$_1$ and joining with r$_3$ on cid to test the condition on the gene name. The expansion of π is:

$\Pi(\pi)$ = clone(cid$_1$,cn$_1$,-,cl$_1$),cl$_1$<30,clone(cid$_3$,cn$_3$,-,-),contains(cid$_3$,gid$_3$), gene(gid$_3$,gn$_3$,-);

This expansion is not contained in u$_2$, since any possible mapping needs to map cid onto both cid$_1$ (for the clone literal) and cid$_3$ (for the contains literal). However, by adding the executable renaming "cid$_1$→cid$_3$" we obtain a correct plan.

Finally, we examine u$_3$. The situation is similar as for u$_2$; {r$_2$} is not a correct plan, but can be turned into a correct plan by adding the renaming "cn$_2$→ct$_2$". The same renaming is not executable for u$_1$ since no clonetype values are obtained.

■

5.2.4 Computing Buckets

The GTA essentially uses the BFA for the traversal of the search space spanned by possible combinations of ECMs. To this end, the algorithm computes sets of possible targets for each literal of u. Currently, those sets are computed whenever an existing node in the search tree is extended. Since each level in the tree extends each node of an earlier level with ECMs covering the same literal of u, those sets of ECMs are unnecessarily computed multiple times. Precomputing and storing them is the main idea of the *bucket algorithm*, which was first presented in [LRO96b; LRO96a] (see Section 5.5 for a detailed discussion).

The necessary changes to the breadth-first search are minimal. Given a plan candidate π and the user query u, we first compute a bucket for every literal of u. If an ECM ε from a literal l of u into a literal k of π exists, then ε is put into the bucket for l. Upon extending a node in the search, the algorithm may use the appropriate bucket instead of recomputing the target set.

This modification does not improve the complexity of the GTA because it does not affect the complexity of the BFA, which did not take the computation of target sets into account since the cost is negligible. However, using buckets allows us to estimate the necessary amount of computation more precisely. Remember that for the computation of the complexity of the BFA we must assume that *every literal* of the candidate is a potential target for every literal of the query (see Section 2.3.3). Using buckets, we can refine this assumption.

Lemma (L5.11) (BFA worst-case analysis using buckets).
Let $\mathtt{M} = (\Sigma, \Psi, \Gamma)$ with only simple QCAs, $\mathtt{u} \in \mathtt{CQ}_{\mathtt{S}}^{\Sigma}$ and π be a plan candidate for $\mathtt{u}$. Let $\mathtt{B_i}$ be the bucket of the $\mathtt{i}$'th literal of $\mathtt{u}$.

(L5.11) In the worst-case, the BFA will perform the following number of compatibility tests (see Theorem (T2.14), page 31):

$$C_{BFA}^{wc} = \sum_{i=2}^{k} \left(\prod_{j=1}^{i} \left| B_j \right| \right)$$

■

Proof:
A bucket $\mathtt{B_i}$ contains all literals of the plan candidate that are covered by the $\mathtt{i}$'th literal of $\mathtt{u}$. The number of edges in the search tree depends on the number of children of a node in each level of the tree. Since $\mathtt{B_i}$ is exactly this number, the theorem follows from the proof of Theorem (T2.14).
■

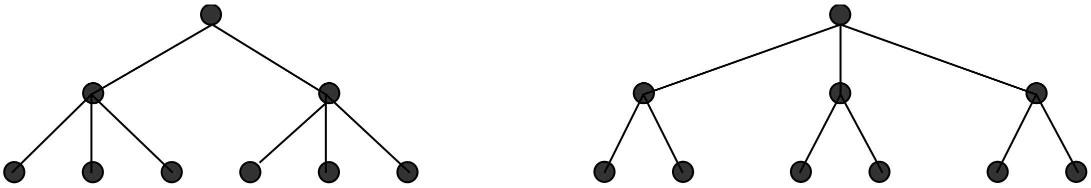


Figure 27. The order of buckets influences the number of leaves.

Remarks:

- Lemma (L5.11) leaves room for further optimisation: The number of compatibility tests depends on the number of edges in the search tree. But this number depends on *the order in which the buckets are visited.* Figure 27 illustrates that the number of inner nodes changes depending on this order, although the number of leaves remains constant. It follows that it is better to visit the buckets ordered by increasing size.
- There are two further optimisations:
  - We may compute "global" buckets filled with literals of all QCAs of Γ before we start enumerating plan candidates. Upon testing a candidate we can then construct "local", i.e., candidate specific, buckets by concatenating the buckets of the queries in the candidate.

- We may stop immediately if we found that at least one bucket is empty, because this implies that at least one literal of u is not covered by any literal of the plan.

- We do not give an average case analysis since the expected benefit is small compared to the necessary effort. In contrast to Theorem (T2.15), it is now more difficult to compute the height of a subtree for a given path.
- We emphasise again that the complexity of the BFA, and hence of the GTA, does not change. In the worst-case, all buckets are "full", i.e., contain any literal of any mediator query in Γ.

■

Soundness and completeness of the GTA is not affected. Precomputing buckets makes sense for CQ_S and for CQ_C queries. The gain in the latter case is higher than in the former. For CQ_S queries the cost of computing the set of all covered literals is very low. Computing this set for CQ_C queries is more expensive because the logical implication it involves is more complex.

5.3 Improved Bucket Algorithm

In this section we describe the *improved bucket algorithm* (IBA) for query planning. Recall that the GTA has two separate phases: Candidate generation and candidate test. The main idea of the IBA is to merge these steps, which results in a drastic reduction of complexity. The IBA first pre-computes buckets of *partial plans* considering all QCAs of the mediator. Then, it uses these buckets to purposely *construct* query plans instead of generating and testing candidates. No containment test is ever pursued.

The clear separation of enumeration and test phase makes the GTA intuitive and easy to capture. However, it also has drawbacks. Treating plan candidates as if they were completely unrelated does not exploit commonalties among them. Furthermore, the GTA does not recognise that the user query is the same in all tests. As a consequence, it performs many compatibility tests multiple times. Consider two plan candidates $\pi_1 = \{q_1, ..., q_n\}$ and $\pi_2 = \{q_1, ..., q_n, q_{n+1}\}$. All compatibility tests between ECMs for the literals of q_1 – q_n will be performed twice.

The IBA takes into account that in each test of a plan candidate (a) the candidate is composed of a set of building blocks (the mediator queries of the QCAs), and (b) the user query is always the same. The IBA first computes buckets considering *all mediator queries* instead of only those of a specific candidate plan. Then, it tests all combinations of elements of those buckets. Instead of selecting plan candidates based on their length, it constructs plans out of elements that are proven to be helpful. This approach saves the "outer loop" of the GTA: Plan enumeration and test are merged into one single pass. This merging yields the reduction in complexity. Another advantage is that the IBA only considers plans that are of the same length as the user query. This further reduces the amount of computation.

The price we have to pay for those benefits is that the IBA does not (and cannot easily) check whether a plan in construction is already a minimal query plan. In this case it could stop since adding further queries is useless – but the IBA only stops as soon as one element of each bucket is in the plan. Therefore, the IBA will generate non-minimal plans and miss minimal ones – although it does not miss correct results. Note however that the GTA has the same problem: It finds all minimal plans, but has no criterion to decide whether they are minimal, and therefore also generates longer plans. It remains an open question whether the generation of unnecessarily long plans can be avoided without plan minimisation, which is itself exponential in the length of the plan. In Section 5.4 we show how we can alleviate this problem by avoiding unnecessary execution of queries even in the presence of non-minimal query plans.

In Section 5.3.1 we explain the main idea of the IBA in detail and prove that it yields a sound and complete query planning algorithm. In Section 5.3.2 we describe an implementation. Its complexity is analysed in Section 5.3.3. Finally, in Section 5.3.4 we report on simulations that highlight the superiority of the IBA over the GTA.

5.3.1 Merging Candidate Generation and Test

The IBA constructs query plans by building sets of mediator queries such that every query in the set contains at least one specific literal of the user query. Every query is “responsible” for exactly one literal. It is ignored whether or not a query also contains other literals. It is also ignored whether or not a query is present multiple times in the set. All *instances* of a mediator query will be treated as if they stemmed from different QCAs. Proceeding in this way allows efficient query planning.

However, there are cases where ignoring the fact that a mediator query contains more than one literal of a user query compromises the completeness of query planning. Consider the following two examples.

Example 5.13.
Consider the following two user queries and one mediator query:

$u_1(x,y) \leftarrow rel_1(x,z),rel_2(z,y);$
$u_2(x,y) \leftarrow rel_1(x,z),rel_1(y,w);$

$q(a,b) \leftarrow rel_1(a,c),rel_2(c,b);$

We first examine how the GTA produces query plans for u_1. There are two plan candidates: $\pi_1 = \{q\}$ and $\pi_2 = \{q,q\}$. We first look at π_1. The two ECMs for the two literals of u are $\varepsilon_1 = ([x\to a, z\to c],[],\varnothing)$ and $\varepsilon_2 = ([z\to c, y\to b],[],\varnothing)$. ε_1 and ε_2 are compatible, and their union ε is $\varepsilon = (h, [],\varnothing)$, with $h = [x\to a, z\to c, y\to b]$.

For π_2, there are four ECMs: ε_1 and ε_2 as for π_1, and, for the second occurrence of q (with fresh variable symbols), $\varepsilon_3 = ([x\to a_1, z\to c_1, [],\varnothing)$ and $\varepsilon_4 = ([x\to a_1, z\to c_1],[], \varnothing)$. The combinations $\varepsilon_1 \cup \varepsilon_2$, $\varepsilon_1 \cup \varepsilon_4$, $\varepsilon_3 \cup \varepsilon_2$, and $\varepsilon_3 \cup \varepsilon_4$ have to be tested. $\varepsilon_1 \cup \varepsilon_2$ and $\varepsilon_3 \cup \varepsilon_4$ succeed, generating two equivalent query plans, and $\varepsilon_1 \cup \varepsilon_4$ and $\varepsilon_3 \cup \varepsilon_2$ fail because the necessary join “$c = c_1$” is not executable.

We apply our informal description of the IBA. We first compute buckets B_1 and B_2. Into a bucket B_i we put all ECMs that map the i-th literal of u into some literal of q. We treat each instance of a mediator query as distinct, which we achieve by using fresh variable names. We obtain $B_1 = \{([x\to a_1, z\to c_1],[],\varnothing)\}$ and $B_2 = \{([z\to c_2, y\to b_2],[], \varnothing)\}$. Both buckets contain only one element, so there exists only one combination. However, if we apply our definition of compatibility of ECMs, we find that the two ECMs are incompatible because the variables c_1 and c_2, required for the join in u, are not exported. This is wrong. We see that we must use only one instance of q to obtain a query plan for u_1.

Unfortunately, we cannot always use only one instance in such cases, as u_2 shows. Here, we get the buckets $B_1 = \{([x\to a_1, z\to c_1],[],\varnothing)\}$ for the first occurrence of rel_1 in u and $B_2 = \{([y\to a_2, w\to c_2],[],\varnothing)\}$ for the second. The two ECMs are compatible. We obtain the query plan $\phi = (<q,q>,[],\varnothing,[x\to a_1, z\to c_1, y\to a_2, w\to c_2]\}$. No query plan can be found using q only once.

■

These examples show that we have to be careful in the treatment of non-exported variables. Treating multiple copies of the same mediator query as distinct can introduce the necessity to enter joins into the query transformer that would not be necessary otherwise. We call such joins *artificial*. Artificial joins are not necessarily dangerous; they only affect query planning if they are not executable. If we consider a join between two literals of the same mediator query, two situations can occur:

- The join variables are exported. No problem occurs, since the join is executable.
- At least one of the join variables is not exported. In this case, the corresponding ECMs are not compatible. The IBA then has to check if it can proceed by *merging* the two instances of the mediator query into a single instance, thereby removing the artificial join.

We define *partial plans* to capture this particularity of the IBA.

Definition (D5.16)-(D5.19) (Partial plan, singleton partial plan).
Let $M = (\Sigma,\Gamma,\Psi)$ with only simple QCAs, and $u \in CQ_S$.

(D5.16) A *partial plan* for u is a triple $\varphi = (q,\varepsilon,\pi)$ with:

- q is a query formed from a subset L of the literals of u and all conditions of u using only variables appearing in L,
- π is a plan candidate for u, and
- $\varepsilon = (h,\alpha,C)$ is an ECM from q into $\alpha(\pi,C)$.

(D5.17) The *length of a partial plan* $\varphi = (q,\varepsilon,\pi)$, written $|\varphi|$, is the number of literals in q.

(D5.18) The *root of a partial plan* $\varphi = (q,\varepsilon,\pi)$, written $root(\varphi)$, is the set of literals from u which form q.

(D5.19) A *singleton partial plan* is a triple $\varphi = (q,\varepsilon,\pi)$ with $|q|=|\pi|=1$.

■

Partial plans in the IBA essentially take the role of ECMs in the GTA. We are only interested in partial plans that are obtained through the union of existing partial plans, starting from singleton partial plans. We call such partial plans *regular*. Regular partial plans are defined inductively over their length.

Definition (D5.20)-(D5.22) (Regular partial plans, union, compatibility).
Let $M = (\Sigma,\Gamma,\Psi)$ with only simple QCAs, and $u \in CQ_S$.

(D5.20) A partial plan of length n is *regular* if it is the union of a singleton partial plan φ_n and a regular partial plan φ with $|\varphi| = n - 1$ and $root(\varphi_n) \notin root(\varphi)$.

(D5.21) Let $\varphi_n = (q_n,\varepsilon_n,\pi_n)$ be a singleton partial plan and $\varphi = (q,\varepsilon,\pi)$ be a regular partial plan with $root(\varphi_n) \notin root(\varphi)$. Let φ be the union of $n-1$ regular partial plans $\varphi_i = (q_i,\varepsilon_i,\pi_i)$, $i \leq n-1$. Let $l_n = root(q_n)$ and $L = root(q)$. The *union* of φ_n and φ, written $\varphi_n \cup \varphi$, is defined as:

- If $\varepsilon_n \sim \varepsilon$ or $\pi_n \notin \pi$ then:
  $\varphi_n \cup \varphi = (<l_n,L,cond(u,variables(l_n,L))>,\varepsilon_n \cup \varepsilon,\pi_n \cup \pi)$;
- Otherwise:

- Let $C = \{\varphi_i \mid \pi_n = \pi_i \wedge \varepsilon_i \nsim \varepsilon_n\}$ be the set of partial plans that have to be merged with φ_n (C for *changed*)
- Let $L' = \{l \mid l \in L'' \wedge L'' = \mathrm{root}(\varphi_x) \wedge \varphi_x \in C\}$ be the set of literals of u that are mapped by partial plans of φ that have to be merged with φ_n.
- Let $\varphi_u = (q_u, \varepsilon_u, \pi_u) = \bigcup_{\varphi_i \notin R} \varphi_i$ and $\varphi_c = (q_c, \varepsilon_c, \pi_c) = \bigcup_{\varphi_i \in R} \varphi_i$. φ_u is the part of φ that remains unchanged, and φ_c is the part that has to be merged with φ_n.
- By definition of C, $\pi_c = \pi_n$. Compute the ECM ε' from $<l_n, L', \mathrm{cond}(u, \mathrm{variables}(l_n, L'))>$ into π_n such that the literals of L' are mapped to the same targets as in ε.

Now, $\varphi_n \cup \varphi = (<l_n, L, \mathrm{cond}(u, \mathrm{variables}(l_n, L))>, \varepsilon' \cup \varepsilon_u, \pi_n \cup \pi_u)$.

(D5.22) Let $\varphi_n = (q_n, \varepsilon_n, \pi_n)$ be a singleton partial plan and $\varphi = (q, \varepsilon, \pi)$ be a regular partial plan with $\mathrm{root}(\varphi_n) \notin \mathrm{root}(\varphi)$ .Let $\varphi' = (q', \varepsilon', \pi') = \varphi_n \cup \varphi$ with $\varepsilon' = (h', \alpha', C')$. φ_n and φ are *compatible*, written $\varphi_n \sim \varphi$, iff:

- $\alpha'(\pi', C')$ is executable, and
- C' is satisfiable.

■

Remarks:

- In a partial plan $\varphi = (q, \varepsilon, \pi)$, $\mathrm{root}(q)$ is always a set, while π is a bag. A partial plan maps a subquery of u into a plan candidate.
- If $\varphi = (q, \varepsilon, \pi)$ and $\varepsilon = (h, \alpha, C)$ we often abbreviate $\varphi = (q, h, \alpha, C, \pi)$.
- If a partial plan $\varphi = (q, \varepsilon, \pi)$ covers u completely, then q = u and $\phi = (\pi, \varepsilon)$ is a query plan for u.
- Definition (D5.21) contains the *merging* of partial plans. Figure 28 illustrates this definition. Merging is only necessary if the ECMs are incompatible, and it is only possible if the newly added mediator query is already contained at least once in the partial plan at hand, i.e., φ. Merging proceeds by first computing φ_u, which is φ without the critical mediator query π_n, and then computing an ECM from the critical subquery of u into π_n. The two resulting partial plans are then combined in the usual way, i.e., like ECMs. This procedure ensures a correct test for the executability of joins.

■

The difference between partial plans and ECMs is the following: While an ECM is a partial mapping from the user query into a plan candidate, a partial plan is a complete mapping from a subset of the user query into a plan candidate. We use partial plans to construct query plans, while ECMs are used to test a given plan candidate.

We now describe the IBA, using the previous definitions.

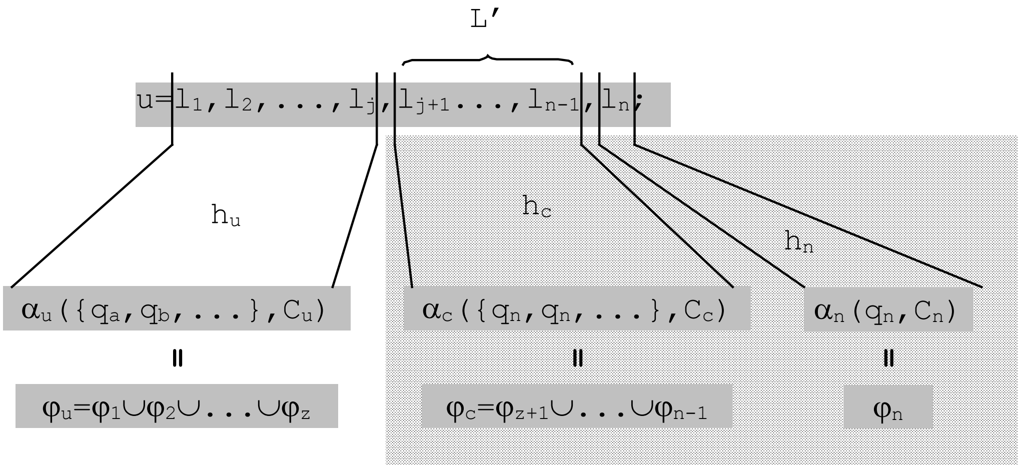


Figure 28. Merge-union of two partial plans.
The partial plan φ is decomposed into two partial plans φ_u and φ_c. φ_c consists of those partial plans contained in φ that must be merged with the new partial plan φ_n. The shaded area is eventually covered by ε '. The figure ignores conditions in u and assumes, without loss of generality, a certain order in the partial plans and in the literals of u.

Algorithm 8. The improved bucket algorithm (IBA).

Input: Mediator $M = (\Sigma,\Gamma,\Psi)$ with only simple QCAs, and user query $u \in CQ_S$.

Output: Set of query plans for u such that their union computes $u(M)$.

Algorithm: Compute all singleton partial plans for all literals l of u and sort them into buckets. Start with the elements of the smallest bucket and construct longer partial plans by incrementally adding singleton partial plans from new buckets. If a new element is incompatible with a growing partial plan, discard that branch. If finally $\varphi = (u,h,\alpha,C,\pi)$ is obtained as the union of one partial plan from each bucket, then (π,α,C,h) is a query plan for u.

■

We postpone examples that show how the IBA proceeds until the next section, because there we give a implementation of the union operation which facilitates its understanding.

We show that the IBA is complete and sound by proving that it computes only “correct” query plans, and that it computes an equivalent query plan for every existing query plan that is not longer than the user query.

Theorem (T5.12)-(T5.13) (Completeness and soundness of the IBA).
Let $M = (\Sigma,\Psi,\Gamma)$ with only simple QCAs, $Q = MQ(\Gamma)$, and $u \in CQ_S$.

(T5.12) The IBA is sound, i.e., every query plan ϕ is computes is a query plan for u.

(T5.13) The IBA is complete. If $\phi = (p,h)$ is a query plan for u with $|p| \leq |u|$, then the IBA will compute a query plan ϕ' such that $\phi'(M) = \phi(M)$.

■

Proof:

We prove both theorems by reduction to the GTA. Let $k = |u|$.

- (T5.12): Let $\varphi = (q,\varepsilon,\pi)$ be a regular partial plan computed by the IBA with $\varepsilon = (h,\alpha,C)$ and $n = |Q|$. φ is the union of singleton partial plans $\varphi_i = (q_i,\varepsilon_i,\pi_i)$, $i \leq n$, mapping the i'th literal of u into the query $\pi_i \in MQ(\Gamma)$. Furthermore, it must hold that $\varphi_1 \sim \varphi_2$, $(\varphi_1 \cup \varphi_2) \sim \varphi_3$, ... $(\varphi_1 \cup ... \cup \varphi_{k-1}) \sim \varphi_k$.
  The GTA tests each plan candidate π whose length is smaller than or equal to k. This includes $\pi = \{\pi_1, ..., \pi_k\}$. The GTA will, possibly among other mappings, find ε_1 as a partial ECM from the first literal of u into π, ε_2 for the second, etc. Since all those ECMs are compatible, the GTA will compute (π,α,C,h) as query plan for u. Since the GTA is sound, we can follow that also the IBA is sound.
- (T5.13): Let $\phi = (\pi,\alpha,C,h)$ be a query plan computed by the GTA. Hence, $\varepsilon = (h,\alpha,C)$ is a ECM from u into the plan candidate $\pi = \{\pi_1, ..., \pi_m\}$. Let π_n be the set of *necessary queries* in π, i.e., those that contain a literal that is used as a target in h. Let $n = |\pi_n|$. Since the GTA also computes non-minimal plans, it can be that $n < k$.

  (1) Let $n = k$, i.e., each π_i contains the target of exactly one literal of u. Without loss of generality, we can assume that π_i contains the target for the i'th literal l_i of u. Hence, there exists an ECM ε_i from l_i into π_i. It follows that $\varphi_i = (q_i,\varepsilon_i,\pi_i)$ is a singleton partial plans for q_i, which is the subquery from u consisting only of the literal l_i and appropriate conditions. Since the IBA tests all combinations of singleton partial plans, it will also test the combination $\varphi_1 \cup \varphi_2 \cup ... \cup \varphi_n$. All those partial plans are compatible because the corresponding ECMs are compatible. Therefore, the IBA finds ϕ.

  (2) Let $n<k$. We construct a plan $\phi' = (\pi',\alpha',C',h')$ that is equivalent to ϕ and show that the IBA will find ϕ'.
  Initially, let $\phi' = (\pi_n,\alpha,C,h)$. Since $n < k$, π_n must contain some mediator queries that contain a target for more than one literal of u. Let q be such a query containing the targets for the literals $l_1, ..., l_j$. There are two possibilities:
  - $l_1, ..., l_j$ are connected through one or more non-exported variables that are mapped to non-exported variables in q. Since ϕ is a query plan, the corresponding ECMs are compatible. Suppose that the IBA has constructed a partial plan φ for some literals of u, including some l_x, $1 \leq x \leq j$, and wants to add a singleton partial plan φ' for a literal l_y, $1 \leq y \leq j$, $y \neq x$. When building the union, the IBA will detect a conflict and compute φ_u without q, and ε' from l_x, l_y, and appropriate conditions, into q. It then computes their union, which must be compatible since ϕ is a query plan. The same happens for every singleton partial plan for a l_i, $i \notin \{y,x\}$. Eventually, the IBA computes exactly ϕ.
  - Otherwise, we can safely add to π_n $j-1$ instances of q. In h', each of these instances is the target of a different literal l_i, and α' is extended with appropriate joins for the new instances. Since those literals are not connected through non-exported variables, all those joins are executable and the corresponding ECMs are compatible. Since the IBA treats literals as stemming from different mediator queries, it will compute ϕ'.

  We repeat this for all queries as q. The resulting φ' is obviously equivalent to ϕ, since both contain the same mediator queries connected through identical joins.

■

Remarks:

- Neither the GTA nor the IBA is complete for CQ_C queries. In the previous proof, we only showed that for any query plan found by the GTA there will be an equivalent query plan found by the IBA. This does not imply that the IBA is complete for CQ_C.
- There can be only one partial plan φ from a subquery with literal l∈u into a literal k∈q. Therefore, each partial plan has a unique mapping attached. If there are more mappings from l into q (because the relation of l appears many times in q), then multiple singleton partial plans exist. Each is independently put into the bucket for l, and therefore all query plans are found.

■

The proof of Theorem (T5.13) highlights the major weakness of the IBA method: The solutions it finds are not guaranteed to be minimal.

Example 5.14.

Consider the following query u and QCA r. u asks for triples of children, parents, and grandparents. It does so be using a self-join on a global parent relation. r describes a wrapper W exporting such triples:

```
u(a,b,c) ← parent(a,b),parent(b,c);
r: parent(x,y),parent(y,z) ← W.v(x,y,z) ← p_grandparent(x,y,z);
```

Let q = medq(r). The IBA first generates one bucket per literal of u. Whenever a mediator query is put into a bucket, its variables are renamed – here, we simple add an index. The buckets are:

B_1={((<u'(a,b)←parent(a,b)>,[a→x_1,b→y_1],[],∅),{q}),

((<u'(a,b)←parent(a,b)>,[a→y_2,b→z_2],[],∅),{q})};

B_2={((<u'(b,c)←parent(b,c)>,[b→x_3,c→y_3],[],∅),{q}),

((<u'(b,c)←parent(b,c)>,[b→y_4,c→z_4],[],∅),{q})};

Next, it tries to union each pair of partial plans from B_1 x B_2. All unions succeed because u has no non-exported variable. Four query plans are generated, whose expansions are:

$p_1(x_1,y_1,y_3)$ ← $q(x_1,y_1,-),q(y_1,y_3,-)$;

$p_2(x_1,y_1,z_4)$ ← $q(x_1,y_1,-),q(-,y_1,z_4)$;

$p_3(y_2,z_2,y_3)$ ← $q(-,y_2,z_2),q(z_2,y_3,-)$;

$p_4(y_2,z_2,z_4)$ ← $q(-,y_2,z_2),q(-,z_2,z_4)$;

For instance, the first plan is obtained by combining the first two singleton partial plans of each buckets. Both are compatible, and their union requires the renaming $[y_1 \to x_3]$.

None of these plans is minimal. The IBA does not recognise this because it treats different occurrences of the same QCA as if they were different QCAs.

The GTA produces even more plans. It enumerates two candidate plans: π_1 = {q} and π_2 = {q,q}. Testing either requires a search tree with four leaves, since the two parent literals in u each have two potential targets in q. Four query plans are generated, corresponding to the four plans given above. This is exactly a *worst case*: all ECMs are compatible, and each literal of q covers every literal of u.

■

The example shows that both methods fail to produce an optimal result whenever a literal appears more than once in the user query and in a QCA. Note that, independent of any redundancy, all plans produced by either method are correct and executable. Furthermore, the computed answer to the query conforms to the semantics defined in Section 4.3.2, because, under set semantics, results of redundant plans are removed during duplicate elimination. However, redundancy leads to superfluous computations. We shall treat redundancy between plans and queries in Section 5.4.

5.3.2 Implementing the IBA

Algorithm 9 is an implementation of the union operation of partial plans. We shall show that it can be computed in linear time. The algorithm computes union and compatibility in one single function. Given two partial plans φ_1 and φ_1, it returns a tuple $(\mathtt{b},\varphi)$, where $\mathtt{b}$ is $\mathtt{true}$ iff $\varphi_1 \sim \varphi_2$. If $\mathtt{b} = \mathtt{true}$, $\varphi = \varphi_1 \cup \varphi_2$; otherwise, φ is undefined.

It starts by checking if the mediator query of the newly added φ_n is already contained in the partial plan φ. If this is not the case, no merging is possible and the two partial plans are united in the same way as two ECMs (line 4-10).

Otherwise, we go through all non-exported variables of $\mathtt{u}$ that are mapped both by $\mathtt{h}_n$ and $\mathtt{h}$ (line 11). Only those, called *critical* in the following, can participate in an artificial join that is not executable.

Next, we examine all partial plans φ_i of φ that map a critical variable. We collect in $\mathtt{C}$ all partial plans that must be merged (line 12-35). If the target of a critical variable $\mathtt{v}$ is exported both in φ_i and φ_n, we need not merge φ_i and φ_n. If $\mathtt{v}$ is exported in only one of the two, then the union must fail (line 31). The union also fails if $\pi_i = \pi$ but $\mathtt{h}_i$ maps $\mathtt{v}$ to variable that is an instance of a different variable than the target of $\mathtt{v}$ under $\mathtt{h}_n$ (line 26-27; the function `instance` is not given). If $\mathtt{v}$ is not exported in both, we add φ_i to $\mathtt{C}$ (line 29). Finally, $\mathtt{C}$ contains all partial plans that share at least one non-exported variable with $\mathtt{q}_n$.

Then, we test if the merge succeeds. We compute φ_C, the union of all partial plan of φ that are not in $\mathtt{C}$, and φ_U, the union of all partial plan of φ that are in $\mathtt{C}$ (line 37-38). In the next step, we consistently replace all variable symbols of different instances of $\mathtt{q}_n$ in φ_C with the appropriate variable symbol in φ_n, which yields $\varphi_{C'}$. φ_n and $\varphi_{C'}$ now use the same set of variable symbols (line 39; the function `replace` is not given).

This replacement can render some variable renamings obsolete. Furthermore, we have to check if the conditions of $\varphi_{C'}$ are still satisfiable. We must also check if the φ_n and $\varphi_{C'}$ are compatible, and if their union is compatible to the reminder of φ, i.e., φ_U. If this is the case, the merging was successful.

We examine examples highlighting some traps of computing the union of two partial plans.

Example 5.15.
For simplicity, we discuss only pairs of user queries and QCAs without further context.

$\mathtt{u(a,b)} \leftarrow \mathtt{rel_1(c,a),rel_2(c,b);}$
$\mathtt{q(x,y)} \leftarrow \mathtt{rel_1(z,x),rel_2(z,y);}$

This is the simple case of a single, non-exported variable that joins two relations. The IBA generates two partial plans:

$\varphi_1\mathtt{=(<u'(a)} \leftarrow \mathtt{rel_1(c,a)>,[c} \rightarrow \mathtt{z_1,a} \rightarrow \mathtt{x_1],[],} \varnothing \mathtt{,q);}$
$\varphi_2\mathtt{=(<u'(b)} \leftarrow \mathtt{rel_2(c,b)>,[c} \rightarrow \mathtt{z_2,b} \rightarrow \mathtt{y_2],[],} \varnothing \mathtt{,q);}$

The ECMs of those partial plans are not compatible because the join between two non-exported variables from different mediator queries is not executable.

The union computes the set of critical variables as $\mathtt{V = \{c\}}$. Since $\mathtt{c}$ is not exported in $\mathtt{u}$, we get $\mathtt{C} = \varphi_1$ and $\mathtt{U} = \varnothing$. Substituting all variables in φ_1 with the corresponding symbols from φ_2 leads to the partial plans $\varphi_{C'}\mathtt{=(<u'(a)} \leftarrow \mathtt{rel_1(c,a)>,[c} \rightarrow \mathtt{z_2,a} \rightarrow \mathtt{x],[],} \varnothing\mathtt{,}$

$q)$. $\varphi_{C'}$ and φ_2 are compatible. Their union is the final plan $(u,[c\rightarrow z_2,b\rightarrow y_2,a\rightarrow x],[],\varnothing,q)$.

$u(a,b) \leftarrow rel(c,a),rel(c,b);$
$q(x) \leftarrow rel(z,x);$

We proceed as in the previous example. Again, the critical variable is c. We obtain $\varphi_{C'} = (<u'(a)\leftarrow rel(c,a)>,[c\rightarrow z_2,a\rightarrow x],[],\varnothing,q)$. The final plan equates a and b, which is the only possible answer that can be computed using only q.

$u(a,b) \leftarrow rel(a,c),rel(b,d);$
$q(x) \leftarrow rel(x,k);$

In contrast to the previous example, we now have no join between the two literals of u. Therefore, there exists no critical variable. The union is computed using the union of the corresponding ECMs. The final plan is $(u,[a\rightarrow x_1,c\rightarrow k_1,b\rightarrow x_2,d\rightarrow k_2],[],\varnothing,\{q,q\})$ and (correctly) uses two instances of q.

$u(a,b) \leftarrow rel_1(a,b),rel_2(b,d),rel_3(a,d);$
$q_1(x,y) \leftarrow rel_1(x,y);$
$q_2(v,w) \leftarrow rel_2(v,z),rel_3(w,z);$

The three partial plans for the three literals of u are:

$\varphi_1=(<u'(a,b)\leftarrow rel_1(a,b)>,[a\rightarrow x_1,b\rightarrow y_1],[],\varnothing,q_1);$
$\varphi_2=(<u'(b)\leftarrow rel_2(b,d)>,[b\rightarrow v_2,d\rightarrow z_2],[],\varnothing,q_2);$
$\varphi_3=(<u'(a)\leftarrow rel_3(a,d)>,[a\rightarrow w_3,d\rightarrow z_3],[],\varnothing,q_2);$

The union of φ_1 and φ_2 is:

$\varphi=(<u'(a,b)\leftarrow rel_1(a,b),rel_2(b,d)>,[a\rightarrow x_1,b\rightarrow y_1,b\rightarrow v_2,d\rightarrow z_2],$
$[y_1\rightarrow v_2],\varnothing,\{q_1,q_2\});$

φ contains a join over the exported variables y_1 and v_2. Trying to extend φ with φ_3 finds the critical variable d. We further get $\varphi_U = \varphi_1$ and $\varphi_C = \varphi_2$. Substituting the variable symbols reveals that a single occurrence of q_2 suffices, and the final plan joining q_1 and q_2 through $[x_1\rightarrow w_3,y_1\rightarrow v_2]$ is found.

$u(a,c) \leftarrow rel_1(a,b),rel_2(b,c);$
$q(x,y,z) \leftarrow rel_1(x,y),rel_2(w,z);$

We find $V = \{b\}$ with $h(b) = y_1$ and $h'(b) = w_2$. Because the variables of which y_1 and w_2 are an instance are not identical and w_2 is not exported, the union fails. This is correct since no query plan exists.

$u(b,d) \leftarrow rel(a,b),b<10,rel(a,d),d>20;$
$q(y) \leftarrow rel(x,y);$

The two singleton partial plans are:

$\varphi_1=(<u'(b)\leftarrow rel(a,b),b<10>,[a\rightarrow x_1,b\rightarrow y_1],[],\{y_1<10\},q);$
$\varphi_2=(<u'(d)\leftarrow rel(a,d),d>20>,[a\rightarrow x_2, d\rightarrow y_2],[],\{y_2>20\},q);$

The critical variable is a. Variable substitution yields $\varphi_{C'} = (<u'(b)\leftarrow rel(a,b), b<10>,[a\rightarrow x_2,b\rightarrow y_2],[],\{y_2<10\},q)$. The IBA detects that these plans are not compatible, and the union fails.

■

Lemma (L5.14) (Complexity of the union of partial plans).
Let $M = (\Sigma,\Psi,\Gamma)$ with only simple QCAs, $Q = MQ(\Gamma)$, and $u \in CQ_S$ with $k = |u|$. Let φ be a partial plan for u of length $x < k$, and φ_n be a singleton partial plan for u. Furthermore, let a_{max} be the maximal arity of the relations in Σ.

(L5.14) Computing $\varphi_n \cup \varphi$ is possible in $O(x)$.

■

Algorithm 9. Computing the union of two partial plans.

```
function pp_union(query u, singleton_pp $\varphi_n$, pp $\varphi$) : (boolean, partialPlan);
% $\varphi_n=(q_n,\varepsilon_n,\pi_n)$, $\varepsilon_n=(h_n,\alpha_n,C_n)$;
% $\varphi=(q,\varepsilon,\pi)=\varphi_1\cup\varphi_2\cup...\cup\varphi_n$, $\varepsilon=(h,\alpha,C)$;
% $\varphi_i=(q_i,\varepsilon_i,\pi_i)$, $\varepsilon_i=(h_i,\alpha_i,C_i)$;
  L = root($\varphi_n$) $\cup$ root($\varphi$);
  q' = <L,cond(u,variables(L)>;
  if $\pi_n\notin\pi$ then                 % No merging possible
    if $\varepsilon_n\sim\varepsilon$ then
      return (true,(q',$\varepsilon_n\cup\varepsilon$,$\pi_n\cup\pi$));
    else
      return (false,null);
    end if;
  end if;
  V={v|v$\notin$export(u) $\wedge$ v$\in$org($h_n$)$\cap$org(h)};
  if V=$\emptyset$ then
    if $\varepsilon_n\sim\varepsilon$ then
      return (true, (q', $\varepsilon_n\cup\varepsilon$, $\pi_n\cup\pi$));
    else
      return (false,null);
    end if;
  end if;
  C=$\emptyset$;                            % Collect all $\varphi_i$ to be merged
  foreach v$\in$V
    foreach i:$\varphi_i\notin$C
      if v$\in$org($h_i$) then
        if h(v)$\notin$export($\pi_n$) then
          if $\pi_i\neq\pi$ then
            return (false,null);
          else if instance($h_i$(v))$\neq$instance(h(v)) then
            return (false,null);
          else
            C = C $\cup$ $\varphi_i$;
        else if $h_i$(v)$\notin$export($\pi_i$) then
          return (false,null);      % One exported, one not $\Rightarrow$ deadly
        end if;
      end if;
    end for;
  end for;
  U = {$\varphi_i$ | $\varphi_i\notin$C};                 % Complement of C
  $\varphi_C$ = ($q_C$, $\varepsilon_C$, $\pi_C$) = $\bigcup_{\varphi_i\in C}\varphi_i$ ;         % Part of $\varphi$ to be merged
  $\varphi_U$ = ($q_U$, $\varepsilon_U$, $\pi_U$) = $\bigcup_{\varphi_i\in U}\varphi_i$           % Part of $\varphi$ that remains
  $\varphi_{C'}$ = ($q_{C'}$, $\varepsilon_{C'}$, $Q_{C'}$) = replace($\varphi_C$, $\varphi_n$);% Replace all variable in $\varphi_C$ that are
                                          % instances of the same variable of q
                                          % with the according symbol in $\varphi_n$
  if (not satisfiable $\varphi_{C'}$) $\vee$ ($\varepsilon_{C'}\nsim\varepsilon_n$) $\vee$ (($\varepsilon_{C'}\cup\varepsilon_n$)$\nsim\varepsilon_U$)
    return (false,null);
  else
    return (true, (q', $\varepsilon_{C'}\cup\varepsilon_n\cup\varepsilon_U$, $\pi_n\cup\pi$));
  end if;
end function;
```

Proof:

Consider Algorithm 9. Line 2-18 are linear in the sizes of the plans. Since any critical variable has to be mapped both in φ_n and φ, V can at most contain as many variables as is the arity of the literal for which φ_n is a singleton partial plan, i.e., in the worst case $|V| = a_{max}$. For each such variable we check the x partial plans of which φ was constructed. In the worst case, C is computed in $O(a_{max}x)$.

Let $|R| = y$, $1 \leq y \leq x$. For φ_C, we must compute the union of $x-y$ partial plans, and for φ_U the union of y partial plans, i.e., x unions altogether. In all those unions no merging can occur, since such merges would already have been carried out in the construction of φ. This point is crucial to see that the recomputation of unions does lead to an exponential explosion in the algorithm. No merging ever has to be carried out twice, because the computation of φ_C and φ_U can use already merged partial plans, if they exist.

Therefore, computing those unions is as complex as computing the union of two ECMs, i.e., linear in the size of the ECMs. The remaining computations are also either unions of ECMs or the implication of conditions of CQ_S queries. Therefore, the complexity is $O(a_{max}x+x)$, i.e., essentially linear in x.

■

We now describe the complete implementation of the IBA, using Algorithm 9 to compute the union of two partial plans.

Algorithm 10. Implementation of the IBA.

```
Input: User query u, set Q of mediator queries;
Output: Set P of all query plans;

1:  foreach l∈u                        % Compute buckets for each literal of u
2:    $B_l$ = {(<l,cond(u,variables(l))>,h,$\alpha$,C,q) |$\exists$ l'$\in\pi$ $\wedge$ $\pi\in$Q $\wedge$ l$\geq_h\alpha$(l',C)};
3:    if $B_l=\emptyset$ then
4:      return $\emptyset$;                    % Empty bucket $\Rightarrow$ no query plan exists
5:    end if;
6:  end for;
7:  sort B's by size;                  % $B_1$ is now the smallest.
8:  P = $B_1$;                           % Start with smallest bucket
9:  for i=2...|u|                      % Go breadth-first through buckets
10:   foreach $\varphi\in$P                     % Partial plan of increasing length
11:     P = P \ $\varphi$;
12:     foreach $\varphi_n\in B_i$               % Try to combine with new elements
13:       (success,$\varphi'$) = pp_union($\varphi$, $\varphi_n$, u);
14:       if succeed then
15:         P = P $\cup$ $\varphi'$;
16:       end if;
17:     end for;
18:     if P=$\emptyset$ then                  % No partial plans remains
19:       return $\emptyset$;
20:     end if;
21:   end for;
22: end for;
23: return P;                          % $\forall$(u,h,$\alpha$,C,$\pi$)$\in$P: u$\rightarrow_h\alpha$($\pi$,C)
```

Algorithm 10 takes as input a user query u and a set Q of mediator queries. It starts by computing buckets, one for each literal of u (line 1-6). Into the bucket for literal l it puts all singleton partial plans for l in Q. If any of the buckets is empty, the algorithm stops immediately, since this implies that there is no target for the corresponding literal .

The algorithm then sorts the buckets by increasing size (see remark after Lemma (L5.11), page 109) and starts to construct longer partial plans. In each step of the loop between lines 9-21, it tries to expand each partial plan φ found so far with each element φ of the next bucket. If the union succeeds it is stored for further extension. If not, this branch of the search tree is cut.

5.3.3 Complexity of the IBA

We analyse the complexity of the IBA as implemented in Algorithm 10.

Theorem (T5.15)-(T5.17) (Analysis of the IBA).
Let $M = (\Sigma, \Psi, \Gamma)$ with only simple QCAs, $Q = MQ(\Gamma)$, and $u \in CQ_S$. Let $k = |u|$, $n = |Q|$, b_{avg} be the average size of a bucket, and p_{com}, $0 \le p_{com} \le 1$, be the probability that two partial plans are compatible. Furthermore, let $s_{sum} = \sum_{q \in Q} |q|$ be the number of literals in Q.

(T5.15) In the worst case, the IBA computes $C_{IBA}^{wc} = \sum_{i=2}^{k} (s_{sum})^i$ unions of partial plans.

(T5.16) In the average case, the IBA computes $C_{IBA}^{ac} = b_{avg} \sum_{i=0}^{k-1} (b_{avg} p_{com})^i$ unions of pp's.

(T5.17) The IBA has complexity $O((s_{sum})^k)$.

■

Proof:
Let B_i be the bucket of the i'th literal of u, and $b_i = |B_i|$.

- (T5.15): In the worst-case all unions succeed. The search space of the algorithm is very similar to that of the BFA (page 32): The elements of each bucket form one level of the search tree, and edges correspond to computations of pp_union. The only difference is that buckets are built from all queries in Q, not only of those of a specific plan candidate. Using Lemma (L5.11), we get:

$$C_{IBA}^{wc} = \sum_{i=2}^{k} \left(\prod_{j=1}^{i} b_j \right)$$

  Furthermore, we must assume that each literal of each QCA is a target for each literal of u. Therefore, the size of each bucket is worst-case $b_i = s_{sum}$, $i \le k$. Together, this yields:

$$C_{IBA}^{wc} = \sum_{i=2}^{k} \left(\prod_{j=1}^{i} s_{sum} \right) = \sum_{i=2}^{k} (s_{sum})^i$$

- (T5.16): We can directly use Theorem (T2.15) to estimate C_{IBA}^{ac}, replacing z with b_{avg}. Both parameters have the same function, i.e., they estimate the average width of a level in the search tree.

- (T5.17): We have to add the complexity of computing unions to the result of Theorem (T5.15). Note first that a union *either* fails (because a join with a non-exported variable cannot be executed) *or* requires the time given in Lemma (L5.14), page 119. This implies that, once we have a critical variable, the union either fails or leads in $O(a_{max}x + x)$ to a partial plan that is at most of the same size of the initial partial plan.
  Therefore, the cost of computing a union is bound by the maximal size of a query plan, i.e., k. In the worst case, a union therefore requires $O(a_{max}(k-1)+k-1)$. Together with Theorem (T5.15), the result follows.

■

Remarks:

- A value for b_{avg} can be estimated in the following way: Let l_{avg} be the average size of a query in Q, and suppose that all relations of Σ are uniformly distributed in all queries in Q. Let p_{cov}, $0 \leq p_{cov} \leq 1$, be the probability that there exist a singleton partial plan for a literal of u using a distinct literal of a mediator query. It follows that each bucket has an average number of elements:

$$b_{avg} = \frac{p_{cov} n l_{avg}}{|\Sigma|}$$

- A noteworthy difference between the worst-case estimations of the GTA and the IBA is that C^{wc}_{GTA} uses s_{avg}, while C^{wc}_{IBA} uses s_{sum}. The reason is the following: In the IBA, s_{sum} is the maximal, worst-case number of elements in a bucket. This determines the number of elements in the cartesian product of all buckets, and hence the number tests to be performed. In the GTA, each candidate plan is tested, and the cost of this test depends on the length of the plan. We use s_{avg} to estimate this length.

- The performance of the IBA crucially depends on the number of compatibility tests that fail. It is beneficial to detect incompatibilities as early as possible, since this removes costly branches of the search space. It is an interesting question how one can order buckets and elements of buckets such that early fails become more likely. Incompatibilities occur if necessary renamings cannot be executed because variables are not exported, or due to unsatisfiability of conditions. We can derive several heuristics out of this observation:

  - Start with buckets for literals that participate in many joins.
  - Enumerate buckets following the join structure of the user query.
  - Choose elements within a bucket with many conditions and few exported variables first.

  These heuristics conflict with the observation that choosing small buckets first reduces the number of inner nodes of the search space. We leave a thorough analysis of appropriate heuristics for future work.

■

In general, the IBA produces plans with the same length as the user query. Such plans are not necessarily minimal. Suppose we have found a partial plan φ for a subquery u' of u together with a compatible φ' for a literal l of $u \setminus u'$. Suppose that the mediator query q used by φ' already appears in φ and that that l is not joined to u' through a non-exported variable. It follows that the union of φ' and φ will contain q twice. We would like to know if this is really necessary, or if it suffices to use q only once.

Unfortunately, Levy et al. show that minimising a correct plan is in itself an NP-complete problem [LMSS95]. It can be solved by testing equivalence between the original plan and all plans obtained from the original plan by removing one or more queries.

However, having a non-minimal plan does not imply that we must execute unnecessarily many wrapper queries. In Section 5.4 we shall see how we can identify and superfluous queries and prevent them from being executed. This enables us to answer a non-minimal plan with the minimal amount of remote query executions.

5.3.4 Comparing GTA and IBA

The IBA outperforms the GTA by far. Figure 29 and Figure 30 illustrate this difference. The plotted formulas are given in Theorem (T5.8) and Theorem (T5.9) for the GTA and in Theorem (T5.15) and (T5.16) for the IBA. To make the different formulas comparable, we define the parameter x as the average number of times that an arbitrary relation appears in a QCA. A high value of x hence indicates that we can expect to have many potential targets for every literal of a user query.

Using x, we can modify our cost formulas for the IBA and GTA such that only five parameters remain: The size k of the user query, the number n of QCAs in Q, the average size s_{avg} of a QCA in Q, the probability p_{com} that two arbitrary QCAs are compatible during query planning, and x. Other parameters are calculated from those as follows:

- To compute C^{ac}_{GTA} (see Theorem (T5.9)), we need the average number z_i of literals in plan candidates of length i that cover a literal in u. We can estimate: $z_i = ix$, since a plan of length i contains i QCAs, which each contains a relation in the average x times.
- To compute C^{wc}_{IBA} (see Theorem (T5.15)), we need the total number s_{sum} of literal in Q. We can estimate $s_{sum} = ns_{avg}$, since s_{avg} is the average number of literals of a QCA and n is the total number of QCAs.
- To compute C^{wc}_{IBA} (see Theorem (T5.16)) we need the average size b_{avg} of a bucket for each literal of the user query. We can estimate $b_{avg} = nx$, since all n QCAs are considered as sources for the buckets of each literal, and each QCAs contains a relation in the average x times.

In Figure 29, we kept k, s_{avg}, x, and p_{com} constant and only varied n, i.e., the number of QCAs. We can see that for a user query of length four the average case behaviour of both algorithms is probably acceptable, although the IBA beats the GTA by a factor of approximately 100. Note that with $x = 0.2$ we assume that in a system with 100 QCAs each of the four literals of the user query has 20 QCAs in its buckets.

In Figure 30, we kept n, s_{avg}, x, and p_{com} constant and varied only k, i.e., the size of the user query. All formulas are exponential in k. Therefore, both figures have logarithmic scale on the y-axis. Query planning is therefore quite sensitive to large queries. However, for a query of six literals the number of compatibility tests is still below 10^6 if the IBA is used, but in the range of 10^8 for the GTA.

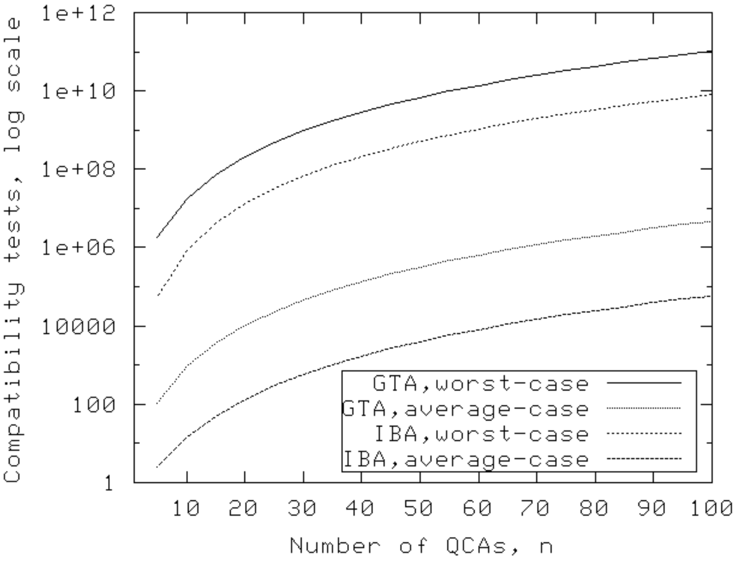


Figure 29. GTA versus IBA. Values: `k=4,n=[5-100],`s_{avg}`=3,x=0.2,` p_{com}`=0.7`.

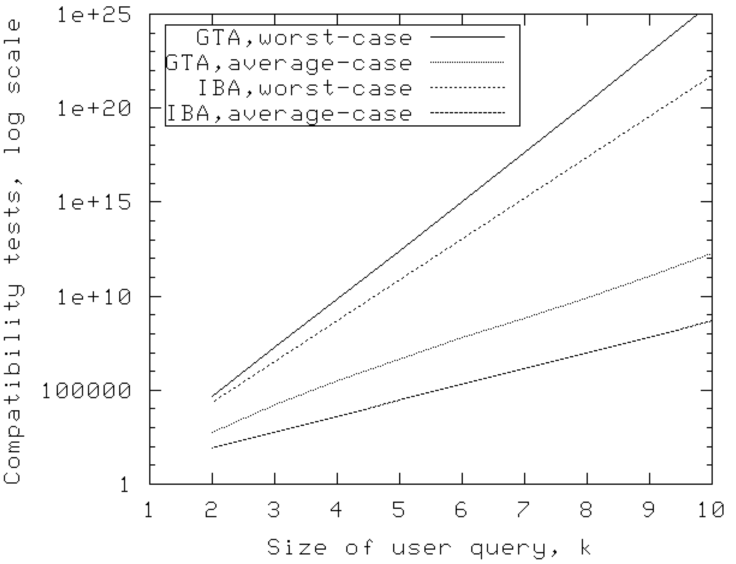


Figure 30. GTA versus IBA. Values: `k=[2-10],n=50,`s_{avg}`=3,x=0.2,` p_{com}`=0.7`.

5.4 Redundancy in Query Plans

The aim of query planning is to compute all answers to a given user query at minimal cost. In this work, cost is determined by the number of remote queries that have to be executed. Therefore, query planning must try to avoid executing remote queries whenever possible.

In the previous sections we have described algorithms that compute *query plans*, which generate answers to a user query `u`. Each query plan computes some answers for `u`; the *result* of `u` is defined as the union over the results of all query plans. We proved that both the IBA and the GTA are sound and complete, i.e., they generate a set of query plans whose union is the result of `u`. In this section we concentrate on post processing of query plans, aiming to compute the result of a query without executing all query plans, respectively all queries contained in different query plans.

The result of u can be obtained by executing each query plan and computing their union inside the mediator. Computing the result of a single query plan can be achieved by executing its wrapper queries one-by-one and applying necessary post-processing inside the mediator. However, this procedure is not optimal: It involves *redundant work* because the remote computations triggered by executing query plans are often overlapping.

We distinguish two types of redundancy in query plans:

- *Query plan redundancy*: A query plan ϕ_1 is redundant wrt. another query plan ϕ_2 if ϕ_1 computes only tuples that are also computed by ϕ_2.
- *Query redundancy*: The same wrapper query q may be involved in many query plans, or multiple times in one query plan. Often, we can execute q only once and reuse the result.

The mediator can save time by removing redundant query plans and by avoiding the repeated execution of redundant queries. The challenge is to *identify redundancy*. The algorithms we propose are based on query containment, since query containment allows us to derive statements about extensional relationships between queries based on the structure of the query only. Essentially, we argue that a query plan is redundant if it is contained in another query plan; and that a query is redundant if it is contained in another query.

From a broader perspective, one could argue that the detection of redundancy should be left to a query optimiser. The execution plan for a user query can be described as a query tree [Cha98], where leaves are wrapper queries (as the smallest, unbreakable unit of execution) and the inner nodes are either joins, unions, or selections. The root node is a union operation and has as many children as there are query plans.

The task of the optimiser is to find an optimal way of executing this tree. Typical optimisations are algebraic *reorderings* of the tree, for instance pushing unions through joins, and the identification of *common subexpressions*, i.e., identical subtrees [Jar85]. However, query optimisation is an exponential problem. Therefore, it is not reasonable to start optimising a large tree without applying as much as possible additional knowledge about the specific tree structure. The aim of this section is hence to “prepare” the query as good as possible before it will eventually be handed over to a query optimiser.

We analyse algorithms for finding redundant query plans in Section 5.4.1. We shall first clarify how plan redundancy can be detected. The main question is whether mediator queries or wrapper queries should be considered. In Section 5.4.2, we explore the detection of query redundancy through *multiple query optimisation*. We describe two methods: The first method is very efficient, but only removes redundant occurrences of a single QCA. The second method also detects redundancy between different QCAs describing queries against the same wrapper, based on certain assumptions about the extension of such wrapper queries. This approach subsumes the first method but is more complex.

Both methods aim at reducing the number of queries that have to be executed remotely. This does not necessarily coincide with a reduction in the required time it takes to answer queries. We finish this section with examples that highlight cases where our techniques are advantageous or fail, respectively.

5.4.1 Finding Redundant Query Plans

First, we define what it means for a query plan to be *redundant*. Our definition is simple: a query plan is redundant if it does not contribute any tuples to the answer to a user query that are not also obtained through another query plan. We then describe how the mediator can detect redundant plans.

Definition (D5.23) (Redundant query plan).
Let $\mathtt{M} = (\Sigma, \Psi, \Gamma)$ and $\mathtt{u} \in \mathtt{CQ_C}$. Let $\mathtt{P}$ be the set of all query plans for $\mathtt{u}$, and let $\phi \in \mathtt{P}$.

(D5.23) ϕ is *redundant wrt.* $\mathtt{u}$ iff $\bigcup_{\phi' \in \mathtt{P} \setminus \phi} \phi'(\mathtt{M}) = \mathtt{u}(\mathtt{M})$.

■

A first approach to the detection of redundant plans is to look at those plans whose result is contained in another plan. However, we have yet no definition of what it means that a *plan is contained in another plan*. We only defined when a plan is contained in a user query: if its expansion is contained in the user query (see Definition (D5.4)). However, plan expansions do not help in detecting redundant plans, as shown by the following example.

Example 5.16.
Consider a wrapper `W` providing data about RNA and non-RNA genes through two different queries[11]. Describing this with respect to the mediator schema of Table 1 (page 18), which does not distinguish between different gene types, requires two QCAs:

r_1: `gene(gid,gn,gd)` ← $\mathtt{W.v_1}$`(gid,gn,gd)` ← `RNAgenes(gid,gn,gd);`
r_2: `gene(gid,gn,gd)` ← $\mathtt{W.v_2}$`(gid,gn,gd)` ← `othergenes(gid,gn,gd);`

Now consider the user query `u`:

`u(gid,gn) ← gene(gid,gn,-);`

There exist two query plans $\phi_1 = (p_1, h_1)$ and $\phi_2 = (p_2, h_2)$, one using r_1 and the other one using r_2. We can see that $\Pi(p_1) \equiv \Pi(p_2)$, i.e., the two plan expansions are equivalent. However, we cannot infer that $\phi_1(\mathtt{M}) \equiv \phi_2(\mathtt{M})$.

■

A mediator query is merely a description of the intension of a wrapper query. It does not carry meaning about the extension of one mediator query with respect to other, possibly identical, mediator queries. We cannot even assume that a combination of mediator query and addressed wrapper is sufficient to derive extensional relationships. In Example 5.16, the same mediator query addressing the same wrapper is used in different QCAs and results in different extensions.

Therefore, we can only prove redundancy between query plans that use the same QCAs. The specific occurrence of a QCA in a query plan is uniquely characterised by the head of the mediator query and the query transformer of the query plan. Consider a query plan ϕ consisting of a single query $\mathtt{q}$, i.e., $\phi = (\mathtt{q}, \alpha, \mathtt{C}, \mathtt{h})$. By definition, all variables appearing in $\mathtt{C}$ or α must be exported, i.e., they also appear in the head of $\mathtt{q}$. If not, the query plan could not be executable. If ϕ comprises more queries, again, all variables in α or $\mathtt{C}$ must appear in the head of some of those queries.

Definition (D5.24) (Containment for query plans).
Let $\mathtt{M} = (\Sigma, \Psi, \Gamma)$ and $\mathtt{u} \in \mathtt{CQ_C}$. Let $\mathtt{P}$ be the set of all query plans for $\mathtt{u}$, and let $\phi_1, \phi_2 \in \mathtt{P}$, $\phi_i = (\pi_i, \alpha_i, \mathtt{C_i}, \mathtt{h_i})$. Let $\mathtt{q_i}$ be the query "$\mathtt{h_i(head(u))} \leftarrow \alpha_i(\pi_i, \mathtt{C_i})$" where we use the heads of the mediator queries in π_i as literals.

(D5.24) ϕ_1 *is contained in* ϕ_2, written $\phi_1 \subseteq \phi_2$, iff $\mathtt{q_1} \subseteq \mathtt{q_2}$.

■

[11] A RNA gene is a gene which is not translated into a protein, but only into RNA.

Example 5.17.
Consider the following query u and QCA r:

u(x,y) ← rel(x,y),rel(z,z);

r: rel(a,b) ← W.v(a,b) ← somehow(a,b);

Let q = medq(r). There are two query plans (the IBA would produce only the second):

ϕ_1=(<q(a,a)>,[b→a],∅,[x→a,y→a,z→a]);

ϕ_2=(<q(a_1,b_1),q(a_2,b_2)>,[b_2→a_2],∅,[x→a_1,y→b_1,z→a_2]);

We can see that one of the two query plans is redundant if we consider the query formed from interpreting query heads as literals:

ϕ_1(a,a) ← q(a,a);

ϕ_2(a_1,b_1) ← q(a_1,b_1),q(a_2,a_2);

■

Using plan containment, we can define a sufficient condition for plan redundancy.

Lemma (L5.18) (Redundant query plans are pointless).
Let M = (Σ,Ψ,Γ) and u ∈ CQ_C. Let P be the set of all query plans for u, and let $\phi \in$ P.

(L5.18) If P contains a query plan ϕ' with $\phi \subseteq \phi'$, then ϕ is redundant wrt. u.

■

Proof:
Our definition of containment for query plans directly uses the definition of the result of a query plan (see Definition (D5.10), page 90). If the result of ϕ is contained in the result of ϕ', then executing ϕ can, under set semantic, not produce any tuples not already produced by ϕ'. Therefore, ϕ is redundant wrt. u.

■

Remark:
The implication only holds in one direction. Following Definition (D5.23), we cannot find all redundant query plans without more detailed knowledge about data sources than encoded in QCAs. For instance, one source might offer the same data through different wrappers. The same extension will flow through different mediator queries into the mediator. Plans using those queries are potentially redundant, but this redundancy is not captured by our definitions.

■

We can prove plan containment using any of the algorithms described in Section 2.2. Since we must only show that one plan is contained in another plan (we are not interested in the particular containment mapping), it is better to use the DFA than the BFA.

Algorithm 11 finds all query plans that are contained in another query plan. It takes as input the set P of query plans and tests each pair $\phi,\phi' \in$ P for containment in both directions. The result is a set P′ of query plans such that no $\phi,\phi' \in$ P′, $\phi \neq \phi'$, contain the other.

Algorithm 11. Finding redundant query plans.

```
Input: Set P of query plans;
Output: Set P′ of query plans with |P′|≤|P|;

P′ = P;
foreach $\phi \in$P
  foreach $\phi' \in$P′
    if $\phi' = \phi$ then
      continue;
    else if $\phi \subseteq \phi'$ then
      P′ = P′ \ $\phi$;
      break;
    end if;
  end for;
end for;
return P′;
```

The algorithm manages two sets: the set P containing all query plans that have not yet been tested against all other query plans that are not yet found to be redundant, and the set P′ containing all query plans that have not yet been found to be redundant. No plan is tested against itself. Whenever a query plan is found to be redundant, it is removed from P′.

Example 5.18.
Consider four query plans ϕ_1, ϕ_2, ϕ_3, and ϕ_4, such that $\phi_2 \subseteq \phi_3$ and $\phi_4 \subseteq \phi_1$. In the first loop, the algorithm tests $\phi_1 \subseteq \phi_2$, $\phi_1 \subseteq \phi_3$, and $\phi_1 \subseteq \phi_4$. In the second loop, it first tests $\phi_2 \subseteq \phi_1$ and then finds $\phi_2 \subseteq \phi_3$. ϕ_2 is removed from P′. $\phi_2 \subseteq \phi_4$ is never tested. In the third loop, it tests $\phi_3 \subseteq \phi_1$, and $\phi_3 \subseteq \phi_4$. In the forth loop it finally finds $\phi_4 \subseteq \phi_1$. Therefore, ϕ_4 is removed from P′. The resulting set of plans is P′ = $\{\phi_1, \phi_3\}$. No containment test was performed twice.
■

Theorem (T5.19)-(T5.21) (Analysis of Algorithm 11).
Let $M = (\Sigma, \Psi, \Gamma)$ with only simple QCAs, and $u \in CQ_C$. Let P be the set of all query plans for u, and $k = |u|$, $n = |P|$. Furthermore, let p_{con}, $0 \le p_{con} \le 1$, be the probability that a query plan is contained in another, arbitrary query plan.

(T5.19) In the worst case, finding all query plans that are contained in another query plan requires $O(n^2)$ containment tests.

(T5.20) In the average case, finding all query plans that are contained in another query plan requires

$$\sum_{i=1}^{n} n_i * \min\left[\frac{1}{1 - p_{con}}, n_i\right], \quad n_i = n - i + 1 + \sum_{j=1}^{i-1} \left(p_{con}\right)^{n_j}$$

containment tests.

(T5.21) The complexity of finding all query plans that are contained in another query plan is $O(n^2 k^k)$.

■

Proof:

- (T5.19): In the worst case, no plan is contained in another plan. Testing each element of P with each other element of P requires $|P|^2 - |P|$ tests.
- (T5.20): Let $q = 1 - p_{con}$, and let n_i be the size of P′ in the i'th traversal of the main loop, i.e., $n_1 = |p| = n$. Using the same method as in the proof of Theorem (T2.18) (page 34), we can estimate that in the first traversal of the main loop in the average $\min(n_1, 1/q)$ containment tests must be performed before one succeeds and the inner loop is left. The probability that any of the n_1 tests succeeds, leading to the removal of ϕ from P′, is $1 - q^{n1}$ (1 minus the probability that all fail). The size of P′ after this first traversal is hence $n_2 = n_1 - 1 + q^{n1}$.

  In the i'th traversal, we have $\min(n_i, 1/q)$ tests and the size of P′ is:

  $$n_i = n_{i-1} - 1 + q^{n_i - 1}$$

  Since the main loop is passed n times, the formula follows.
- (T5.21): Following Theorem (T2.19) (page 34), the complexity of testing containment of a query q_1 in a query q_2 with $k_1 = |q_1|$, $k_2 = |q_2|$ is $O\left(k_1^{k_2}\right)$. Each query plan contains at most as many mediator queries as the user query has literals, i.e., at most k queries. Note the containment test for query plans interprets the query heads as literals and does not consider the expansions of query plans. Testing containment of a plan in another plan is hence $O(k^k)$. $O(n^2)$ such tests must be performed.

■

Remarks:

- It is beneficial to sort query plans in P such that early success is more likely, letting P′ shrink more quickly. For instance, in Example 5.18, the optimal order of containment tests would be to first try $\phi_2 \subseteq \phi_3$ and then $\phi_4 \subseteq \phi_1$. In the example, we performed eight tests, although an optimal enumeration required only four. Finding good enumeration strategies is an open research problem.
- Eventually, we are interested in avoiding remote query executions. Filtering out redundant plans is a means to achieve this goal. Each plan that is detected to be redundant can be removed. To remove as many queries as possible, it is therefore better to remove long plans instead of short plans. This is relevant in cases where two equivalent query plans with differing length are found. Depending on the order in which the containment tests are performed, either the shorter or the longer will be removed. We can guarantee that always the longer is removed by ordering P by decreasing size.

■

The two previous remarks are in conflict: If we want to remove as many queries as possible, we should sort P by size; if we want to remove redundant plans as quickly as possible, we need a different order.

So far, our definition of redundancy often fails to detect obvious redundancy, as we show in the next example. The reason is that containment was only defined for *query plans*, but not for plans. For instance, we cannot detect the redundancy that is present if one plan yields two different query plans. In such cases, it suffices to execute the plan once and obtain the results of the query plans from its result. But neither of the two query plans is redundant wrt. Definition (D5.23).

Example 5.19.
Recall Example 5.14, page 116:

u(a,b,c) ← parent(a,b),parent(b,c);

r: parent(x,y),parent(y,z) ← v_1(x,y,z) ← p_grantparent(x,y,z);

If we use the GTA to obtain query plans for u, we find the following plans:

$p_1(x,y,z) \leftarrow q(x,y,z)$;

$p_2(x_1,y_1,y_3) \leftarrow q(x_1,y_1,z_1),q(y_1,y_3,z_3)$;

$p_3(x_1,y_1,z_4) \leftarrow q(x_1,y_1,z_1),q(y_4,y_1,z_4)$;

$p_4(y_2,z_2,y_3) \leftarrow q(x_1,y_2,z_2),q(z_2,y_3,z_3)$;

$p_5(y_2,z_2,z_4) \leftarrow q(x_1,y_2,z_2),q(x_4,z_2,z_4)$;

If we check for redundant plans using our definition, we only find $p_1 \subseteq p_3$, i.e., only p_1 is redundant.

■

Recall that every plan (not every query plan) exports all variables that are exported in any of its mediator queries. In the example, the results of all plans can be computed by executing q only once. We see that a set of query plans can contain redundancy that cannot be detected on the query plan level. We treat such cases in the following section.

5.4.2 Multiple Query Optimisation in MBIS

As described previously, a query plan is in principle executed by executing all wrapper queries it contains, materialising the results in a temporary database, and finally executing the user query on this temporary database. However, if a QCA r appears twice in a query plan (or in different query plans), it is often possible to obtain the result of the query plan by executing the wrapq(r) only once. Other occurrences of r are redundant.

The optimisation discussed in the previous section removes query plans entirely; here, we show that how we can compute the results of different remaining query plans without having to execute all wrapper queries they contain.

Example 5.20.
Consider the following user query u and QCAs r_1, r_2, and r_3:

u(gid,gn,se) ← gene(gid,gn,-),genesequence(gid,sid,-),sequence(sid,se);

r_1: gene(gid,gn,gd) ← $W_1.v_1$(gid,gn,gd) ← genes(gid,gn,gd);

r_2: genesequence(gid,sid,-),sequence(sid,se) ← $W_2.v_1$(gid,se) ← sequences21(gid,se);

r_3: genesequence(gid,sid,-),sequence(sid,se) ← $W_2.v_2$(gid,se) ← sequencesX(gid,se);

W_2 supports two different queries: One for genes on the X chromosome and one for genes on chromosome 21. There are two query plans for u, expanding to:

p_1(gid,gn,se) ← $W_1.v_1$(gid,gn,-),$W_2.v_1$(gid,se);

p_2(gid,gn,se) ← $W_1.v_1$(gid,gn,-),$W_2.v_2$(gid,se);

To obtain the results of both plans it is not necessary to execute $W_1.v_1$ twice.

■

Finding and eliminating redundant parts in a set of queries is called *multiple query optimisation* (or *multi query optimisation*) [Jar85; SG90]. In a centralised database system, the goal

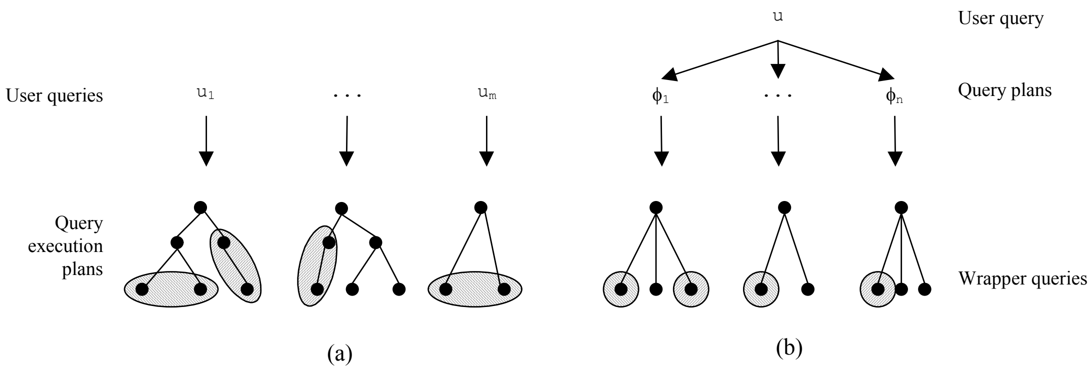


Figure 31. Multiple query optimisation. (a) In a central database. (b) In MBIS.

of a multiple query optimiser (MQO) is to find a query execution plan of minimal cost for a given set of queries. The problem can be decomposed into two sub-problems:

- The MQO needs to identify *common subexpressions*, i.e., subqueries that appear more than once in the set of queries [CD98].
- Assigning some cost to each operation (and therefore, indirectly, to each subexpression), the MQO needs to find a plan, i.e., a set of operations and an order on them, which computes the results of all queries at *minimal cost* [SSN94].

In MBIS, we have a single user query that is rewritten into a set of query plans. The difference between a MQO in a central database and in MBIS is illustrated in Figure 31. In centralised database systems, execution plans for a set of queries are compared and identical subexpressions are isolated. In MBIS, a *single query* results in multiple query plans potentially using common wrapper queries. This set of plans is the input to a MQO. The smallest, unbreakable "operation" in our context is the execution of single wrapper query because only entire wrapper queries are semantically defined (through a QCA), and only entire wrapper queries are guaranteed to be executable (see Definition (D3.2), page 53).

The two subproblems of MQO remain the same. We approach them as follows:

- We describe two approaches for the identification of common subexpressions. In Section 5.4.2.1, we describe an algorithm that identifies multiple occurrences of the *same wrapper query*. The algorithm is linear and already detects many typical cases of redundant queries.
  In Section 5.4.2.2, we introduce an algorithm that also detects redundancy between *different QCAs*. It is based on a modification of query equivalence (see Definition (D2.13), page 19). This algorithm is more powerful than the first one, but also more complex.
- We use a very simple cost model. We strive for the *maximal reduction in the number of remote query executions*. Therefore, our cost model can be described through the following formula. Let Q be the bag of all wrapper queries of any query plan for a user query u. The cost of answering u, c(u), is defined as:

$$c(u) = \sum_{q \in Q} c(q)$$

where the cost of a single wrapper query, c(q), is defined as:

$$\forall q \in Q : c(q) = \begin{cases} 1 : \texttt{q is executed} \\ 0 : \texttt{else} \end{cases}$$

In general, this does not coincide with *minimal query execution time*, as we shall discuss at the end of this section. The combination of our logic model with a detailed time-based cost model for distributed query optimisation, as described in [OV99], is beyond the scope of this work. Examples where our cost model is adequate and where not are given in Section 5.4.2.3.

5.4.2.1 Finding redundancy between identical QCAs.

At first, we assume that no selection or projection in a plan can be pushed to the wrapper. Under this assumption, a query plan is executed by executing (in isolation and possibly in parallel) each wrapper query as it is. Joins, constraints, and projections as derived from the user query are applied by the mediator. The order of executing wrapper queries does not matter. Furthermore, if two query plans are based on the same plan candidate, the result of both can be obtained by executing all queries of the candidate only once, since those query plans can only differ in the attributes they select or in additional conditions they require. Recall that a plan exports all attributes that are exported in any of the QCAs it contains (see Definition (D5.2), page 87).

We conclude that all occurrences of a QCA in any query plan are equivalent if no operations are pushed. If we have a set P of query plans, we must only execute the set Q of wrapper queries defined as:

$$Q = \bigcup_{q \in \phi, \phi \in P} q$$

The extensions of all query plans may be derived from those results. Q may be computed in time linear to the sum of the lengths of all query plans. Furthermore, the wrapper queries in Q can be executed in parallel.

However, ignoring the possibility to push operations often leads to non-optimal execution strategies, as shown in the following example.

Example 5.21.
Consider the following two QCAs r_1 and r_2 and user queries u_1 and u_2. u_1 asks for all map positions of clones smaller than 100KB, while u_2 asks for all map positions of a specific BAC clone.

u_1(cid,cn,mid,po) ← clone(cid,cn,-,cl),clonelocation(cid,mid,po),cl<100;
u_2(mid,po) ← clone(cid,cn,ct,-),clonelocation(cid,mid,po),ct='BAC',
  cn='yWXD1';

r_1: clone(cid,cn,-,cl),clonelocation(cid,mid,po),map(mid,mn,-,-,-) ←
  W_1.v_1(cid,cn,cl,mid,po,mn) ← WWWList(mid,mn,po,cid,cn,cl);
r_2: clone(cid,cn,ct,cl),ct='PAC' ← W_2.v_1(cid,cn,cl) ← PacList(cid,cn,cl);

We first plan u_1. There are two query plans:

ϕ_1=({medq(r_1)},[],{cl<100}},[cid→cid,cn→cn,cl→cl,mid→mid,po→po]);
ϕ_2=({medq(r_2),medq(r_1)},[cid_2→cid_1],{cl_1<100},[cid→cid_1,cn→cn_1,cl→cl_1,
  mid→mid_2,po→po_2]);

The first query plan uses only W_1, while the second gets all clones from W_2 that are smaller than 100 KB and then combines them with their map position in W_1. The extensions of both query plans can differ, since W_1 and W_2 may store different values for the length of a specific clone. Consider a clone whose length is above 100KB in W_1 and below 100KB in W_2. This clone will only be found by the second query plan. The extension of both plans can be computed by executing W_1.v_1 (without a condition on the clone length!) and W_2.v_1.

Now consider u_2, which requests information about a specific clone. There is only one query plan ϕ, which uses r_1. Using r_2 is not possible because of the conflict in ct. Clearly, ϕ can be executed by executing wrapq(r_1) and applying the condition on cn and ct in the mediator. However, this approach is not optimal since it requires to download much more data than necessary. If W_1 is capable of accepting equality conditions, it is more efficient to push the selection.

■

The question at hand is: When it is possible to combine pushing of operations with the efficient and simply optimisation model discussed so-far? We identify the following cases:

- *Pushing conditions* can be integrated as follows. Instead of removing all copies of a QCA except one, we only remove queries whose "footprint" is contained in the "footprint" of another instance of the same query. The *footprint of QCA* is the head of the QCA plus all conditions on variables appearing in this head, with variable renamings associated with the query plan being applied.
  Using this model, the conditions in u_2 (see Example 5.21) are pushed. Furthermore, we correctly detect that executing r_1 in the query plan ϕ_2 for u_1 is not necessary because the footprint $W_1.v_1$(cid,cn,cl,mid,po,mn),cl < 100 is contained in the footprint of the occurrence of r_1 in ϕ_1, i.e., $W_1.v_1$(cid,cn,cl,mid,po,mn). The necessary "containment tests" are linear since either query of each tested pair has only one literal.

- *Pushing joins* between different QCAs cannot be integrated as easily. Assume a plan with body "q_1(X,Y),q_2(Y,Z)". The idea would be to first execute q_1 (or q_2), obtain the values for Y and then push these values as conditions into q_2 (or q_1, respectively).
  This approach is not compatible with a MQO using query containment because we do not know any more in advance what wrapper queries are actually executed at run-time. Pushing joins requires a dynamic MQO.

- *Pushing projections* can be integrated as follows. Imagine a QCA r appearing multiple times, requiring different subsets of export(r). If r has no conditions, it suffices to execute r once, selecting the union of all required attributes in all occurrences of r. However, we have to be more careful if r contains conditions that shall be pushed, too. Imagine two occurrences r′ and r″ of a QCA r, selecting the attribute sets E_1 and E_2 and carrying different conditions. Suppose that there exists a symbol mapping from the footprint of r″ into the footprint of r′ that fulfils conditions CM2 - CM4 (see Definition (D2.16)). If $E_1 = E_2$ or $E_1 \subseteq E_2$, then only r″ needs to be executed. Otherwise, r″ needs to "piggyback" the exported attributes of r′, which means that r″ has to be modified such that more variables are exported. The necessary tests are again linear.

We highlight the problems of combining MQO with pushing of operations by the following examples.

Example 5.22.
Consider the following user query and QCA:

```
u(a,b,c) ← rel(d,a,b),rel(a,c,-),d='1';
r: rel(x,y,z) ← W.v(x,y,z) ← somehow(x,y,z);
```

There are two query plans:

ϕ_1=({medq(r)},[x→y],{y='1'},[d→y,a→y,b→z,c→y]);

ϕ_2=({medq(r),medq(r)},[x_2→y_1],{x_1='1'},[d→x_1,a→y_1,b→z_1,c→y_2]);

The footprint of the three occurrences of r are:

q_1: W.v(y,y,z),y=’1’;
q_2: W.v(x_1,y_1,z_1),x_1=’1’;
q_3: W.v(x_2,y_2,z_2);

If we push projections and joins, the actual queries are:

q_1: W.v(‘1’,’1’,z);
q_2: W.v(‘1’,y_1,z_1);
q_3: W.v(x_2,y_2,-);

Since $q_1 \subseteq q_2$, we conclude that q_1 is redundant. Furthermore, $q_2 \subseteq q_3$ – but we cannot answer q_2 by executing q_3 because q_3 does not export the attribute corresponding to z_1. Fortunately, we can modify q_3 to a query q_3':

q_3': W.v(x_2,y_2,z_2);

which is the only query we must execute remotely.

■

5.4.2.2 Finding redundancy between different QCAs.

In the previous section we presented a method to remove redundancy in plans that stem from multiple occurrences of a *the same QCA*. We now consider redundancy between *different QCAs*. Again, we first assume that no operations can be pushed and discuss extensions at the end of the section.

To compare the extensions of different wrapper queries, we make the assumption that the extension of a literal of a relation rel of the export schema of a wrapper W is identical in all appearances of rel in any wrapper query of W. This assumption is intuitive and correct if RDBMS are used as data sources, but has to be carefully checked if web sites are used.

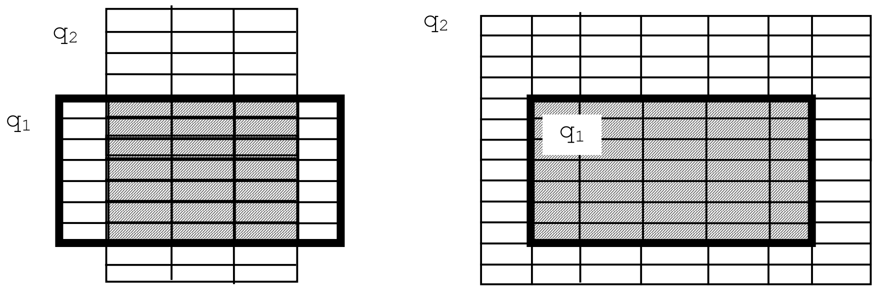
Based on this assumption we strive for a definition of *replaceability* of QCAs. Intuitively, a QCA r_1 is replaceable through another QCA r_2 if the result of r_1 can be computed only from the result of r_2. Hence, if r_1 is replaceable by r_2 and both appear in some query plan, it suffices to execute only r_2 remotely – the extension of r_1 can be computed from this result.

A first attempt to define replaceability formally is to require containment. However, the following examples motivate that containment is not enough. A next approach is to require containment of one QCA in a transformed and executable form of the other QCA. Requiring query transformers is necessary, but still not sufficient. Finally, we show that equivalence of two QCAs is too strong.

Example 5.23.
Consider the following wrapper queries:

W.v_1(cid,cn,ct) ← clone(cid,cn,ct,cl),cl<50;
W.v_2(cid,cn) ← clone(cid,cn,-,cl),cl<100;
W.v_3(cid,cn,ct) ← clone(cid,cn,ct,cl),cl<100;
W.v_4(cid,cn,ct) ← clone(cid,cn,ct,cl),cid<10,clonelocation(cid,mid,po),
  map(mid,-,mt,-),mt=’physical’;
W.v_5(cid,cn) ← clone(cid,cn,-,cl),cl<50;

Clearly, $v_1 \subseteq v_2$. But we cannot compute the extension of v_1 from the result of v_2 because v_2 does not export the clonetype. $v_1 \subseteq v_2$ only implies that the result of v_1, projected to the exported variables of v_2, is contained in the result of v_2. This does not suffices for replaceability (see Figure 32).



(a) (b)

Figure 32. Difference between containment and replaceability.
(a) $q_1 \subseteq q_2$: all rows of q_1 are contained in q_2, and all exported variables of q_2 are computed.
(b) $q_1 \sqsubseteq q_2$. The rows and columns for q_1 are contained in q_2. They must also be identifiable.

Next we see that $v_1 \subseteq v_3$. But, again, we cannot compute the extension of v_1 from the result of v_3 because `cl` is not exported, and therefore the condition in v_1 cannot be enforced on the result of v_3.

We next consider v_1 and v_4. We find that $v_4 \subseteq$ `[]<`v_1`(cid,cn,ct),cid<10>`. However, to obtain the result of v_4 we must consider all clones and then filter out all tuples describing clones that are not on any physical map. This filtering is not possible on the result of v_1 only.

All the problems discussed so far could be avoided by requiring equivalence of one wrapper query with the other wrapper query plus a query transformer. However, this requirement is too strong. Consider v_1 and v_5. Both wrapper queries are equivalent, except that `ct` is not exported in v_5. But nevertheless v_5 is obviously replaceable through v_1.

■

We can formally define replaceability.

Definition (D5.25) (Replaceable QCAs).
Let $M = (\Sigma, \Psi, \Gamma)$ and $u \in CQ_C$ Let $r_1, r_2 \in \Gamma$, $\text{origin}(r_1) = \text{origin}(r_2)$, be two QCAs and let $q_1 = \text{wrapq}(r_1)$ and $q_2 = \text{wrapq}(r_2)$.

(D5.25) r_1 is *replaceable by* r_2, written $r_1 \sqsubseteq r_2$, iff $\exists\ \alpha, C$:

- $\alpha(q_2, C)$ is an executable mediator query, and
- $\alpha(q_2, C) \subseteq q_1$, and
- $q_1 \subseteq \alpha(q_2, C)$, ignoring rule CM1 of Definition (D2.16), page 19.

■

Remark:
Every QCA is replaceable by itself. The definition therefore subsumes all cases discussed in the previous section. Checking replaceability is more general than comparing only query heads, but is also computationally more complex and makes additional assumptions that may not always hold.

■

Our definition of replaceability is independent of any query plan. Replaceable QCAs may, for instance, occur during the design of a MBIS if an administrator has certain "typical" global queries in mind. This administrator will naturally try to define QCAs according to these queries, even if they are overlapping or replaceable with other QCAs.

It is tempting to simply remove all replaceable QCAs from a mediator. But even replaceable QCAs can help the mediator to answer queries more efficiently, as shown in the following examples.

Example 5.24.
Consider the following QCAs describing two queries against a wrapper W. One query obtains information about STS's, which are a type of short PCR markers, and the other obtains information about EST's, which are a special kind of STS's. W stores both classes in a single relation together with a discriminating flag; in the mediator schema, the classes are stored in separate relations.

$\mathtt{r}_1\mathtt{:\ sts(stid,na,flag)} \leftarrow \mathtt{W.v}_1\mathtt{(stid,na,flag)} \leftarrow \mathtt{PCR\_marker(stid,na,flag);}$
$\mathtt{r}_2\mathtt{:\ est(esid,na)} \leftarrow \mathtt{W.v}_2\mathtt{(esid,na)} \leftarrow \mathtt{PCR\_marker(esid,na,flag),flag='true';}$

Clearly $r_2 \sqsubseteq r_1$ – the result of r_2 can be computed by filtering from the result of r_1 all tuples with $\mathtt{flag} \neq \mathtt{'true'}$. Formally, the conditions for replaceability are fulfilled with $\alpha = []$ and $C = \{\mathtt{flag='true'}\}$. But it is nevertheless meaningful to specify both QCAs. A user query asking for ESTs can be answered using r_2 only, shipping less data, while a user query asking for STS's and EST's can be answered using only r_1.

■

Proving replaceability of QCA includes the discovery of executable query transformers. For this purpose, we can reuse our query planning algorithms.

Algorithm 12. Testing replaceability of query plans.

Input: Two QCAs r_1, r_2.

Output: If $r_1 \sqsubseteq r_2$ then true, else false.

Algorithm: If $\mathtt{origin}(r_1) \neq \mathtt{origin}(r_2)$ then report false. Else, let $q_1 = \mathtt{wrapq}(r_1)$, $q_2 = \mathtt{wrapq}(r_2)$. Search a correct and executable plan $p = (\{q_2\},\alpha,C)$ for q_1. If no such p exists, report false. Else, test if $q_1 \subseteq \alpha(q_2,C)$, ignoring CM1 from Definition (D2.16). If yes, report true, else false.

■

The two conditions tested in Algorithm 12 are sufficient for complex QCAs and sufficient and necessary for simple QCAs. We omit a formal proof.

Lemma (L5.22)-(L5.23) (Complexity of testing replaceability).
Let Q be the set of all wrapper queries that must be executed to compute the answer to a user query u. Let $q_1,q_2 \in Q$ with $\mathtt{origin}(r_1) = \mathtt{origin}(r_2)$ and $q_1 = \mathtt{wrapq}(r_1)$, $q_2 = \mathtt{wrapq}(r_2)$. Furthermore, let $k_1 = |q_1|$, $k_2 = |q_2|$, and l_{max} be the maximal length of a wrapper query in Q.

(L5.22) Testing whether $r_1 \sqsubseteq r_2$ is $O\left((k_1)^{k_2} + (k_2)^{k_1}\right)$.

(L5.23) Finding all replaceable QCAs in Q is $O\left(|Q|^2 * (l_{max})^{l_{max}}\right)$.

■

Proof:

- (L5.22): We can use the IBA to test the first condition of Algorithm 12. The complexity of the IBA was given in Theorem (T5.17), page 122. We search a plan for q_1, hence $k = k_1$, only using q_2, hence $s_{sum} = k_2$. To test the second condition, we can use the DFA, whose complexity was given in Theorem (T2.19), page 34. Removing CM1 neither increases nor decreases the complexity of the DFA.
- (L5.23): In the worst case, the origin of all QCAs in Q is the same, all QCAs are different, none is replaceable, and they all have the same length l_{max}. We have to perform $|Q|^2 - |Q|$ replaceability tests. Together, we get:

$$O\left(\left(|Q|^2 - |Q|\right) * \left(\left(l_{max}\right)^{l_{max}} + \left(l_{max}\right)^{l_{max}}\right)\right) = O\left(|Q|^2 * \left(l_{max}\right)^{l_{max}}\right)$$

■

Remarks:

- We require Q to be a *set* (and not a bag) since it is reasonable to remove duplicates first, using the methods presented in the previous section.
- Replaceability only has to be tested between QCAs defined for the same wrapper. This greatly reduces the number of tests in real-life applications.
- Finding all redundant query plans is much faster than finding all query plans. For instance, assume that no QCA has a repeated literal. Then each replaceability test is linear in the size of the two QCAs because each bucket computed by the IBA contains at most one literal. In contrast, query planning can be arbitrarily expensive depending on how many times a relation appears in *any QCA*.

■

Replaceability of QCAs essentially requires the containment of one QCA in the other. Therefore, it is compatible with the pushing of conditions of the user query, i.e., we can safely ignore replaceable QCAs even if conditions are pushed to the wrappers. For pushing projections the same provisions must be made as for the removal of identical QCAs (previous section). Pushing joins again requires dynamic optimisation.

5.4.2.3 Assessing the cost model.

The optimisations we discussed in this section strive for the maximal reduction of queries that must be executed remotely. This is a heuristics for achieving the primary goal of query optimisation, i.e., minimising the time to answer a query. Under the assumption that latency is becoming the predominant factor in modern networks, this heuristic is reasonable. However, as soon as the difference in time it needs to transmit packages of varying size dominates latency, out cost model will not produce optimal results. An improved cost model, taking both transmission time of query results and network latency into account, is left to future work.

We finish this section by discussing one example where our MQO achieves very good results and one where it fails.

Example 5.25 (good).
Consider a mediator whose schema has only one relation, `parentOf(parent,child)`. A wrapper `W` stores relationships between persons and their grandparents. Consider the following query `u`, asking for all persons having grandchildren, and QCA `r`:

```
u(y) ← parentOf(y,x),parentOf(x,z);
r: parentOf(a,b),parentOf(b,c) ← W.v(a,b,c) ← grandParent(a,b,c);
```

Let q = medq(r). The IBA generates four query plans:

$\phi_1=(\{q,q\},[a_2\rightarrow b_1],\emptyset,[y\rightarrow a_1,x\rightarrow b_1,z\rightarrow b_2]);$
$\phi_2=(\{q,q\},[b_2\rightarrow b_1],\emptyset,[y\rightarrow a_1,x\rightarrow b_1,z\rightarrow c_2]);$
$\phi_3=(\{q,q\},[a_2\rightarrow c_1],\emptyset,[y\rightarrow b_1,x\rightarrow c_1,z\rightarrow b_2]);$
$\phi_4=(\{q,q\},[b_2\rightarrow c_1],\emptyset,[y\rightarrow b_1,x\rightarrow c_1,z\rightarrow c_2]);$

which expand to the plans:

$p_1(a_1) \leftarrow q(a_1,b_1,c_1),q(b_1,b_2,c_2);$
$p_2(a_1) \leftarrow q(a_1,b_1,c_1),q(a_2,b_1,c_2);$
$p_3(b_1) \leftarrow q(a_1,b_1,c_1),q(c_1,b_2,c_2);$
$p_4(b_1) \leftarrow q(a_1,b_1,c_1),q(a_2,c_1,c_2);$

None of these query plans is redundant. The MQO will find that all except one QCAs are replaceable, i.e., it will execute q only once. This is reasonable. The extensions of all four query plans can be computed on the extension of q.

■

Example 5.26 (bad).
Consider the same mediator schema as in the previous example, and a wrapper W′ having the same schema as the mediator. Consider the query u′ asking for the grandparents of a person called 'Peter', and QCA r′:

```
u'(y) ← parentOf(y,x),parentOf(x,z),z='Peter';
r': parentOf(a,b) ← W'.v(a,b) ← parentOf(a,b);
```

There is one query plan:

$\phi=(\{medq(r'),medq(r')\},[a_2\rightarrow b_1],\{b_2='Peter'\},[y\rightarrow a_1,x\rightarrow b_1,z\rightarrow b_2]);$

The MQO detects that both queries address the same wrapper and that the result of one (including the condition) is contained in the result of the other one (without condition). This is independent of whether the MQO assumes that the condition is enforced at the wrapper side or at the mediator side. The MQO will therefore execute q without any bindings. Imagine that W′ stores 100.000 tuples. The redundancy-free plan results in the execution of one remote query and the transmission of 100.000 tuples.

However, we can also execute wrapper queries sequentially. We can assume that each child has exactly 2 parents. The sequential execution strategy will therefore at most execute two queries: The first asks for the parent of Peter, resulting in the transmission of at most two tuples (assuming that only one person called 'Peter' exists). The second query then requests the parents of the parents of Peter, resulting in at most four tuples. Altogether, not more than six tuples are transmitted over the network.

■

5.5 Summary and Related Work

This chapter discussed algorithms for query planning and for the detection of redundancy in query plans. Our goal was to investigate methods to find the minimal set of remote query executions that is necessary to compute the complete answer to a given user query.

We formally introduced *plan candidates*, *plans*, and *query plans*. Plan candidates are simply sets of mediator queries; plans are plan candidates together with some transformations; finally, query plans are plans whose expansions are contained in a given user query.

We analysed the connection between query plans and the semantics of a user query. We proved that any algorithm finding *all query plans* for a user query `u` computes the answer to `u`. However, there potentially exists an infinite number of query plans for any user query. Fortunately, we could prove that most of those query plans do not compute helpful tuples. Precisely, we showed that any query plan that contains more mediator queries than the user query has literals is *non-minimal*, i.e., computes only tuples that are also computed by a minimal plan. This theorem allows query planning algorithm to consider only query plans up to the size of the user query.

We described and analysed two such algorithms: the *generate & test algorithm* and the *improved bucket algorithm*. Both algorithms are sound and complete for $\texttt{CQ}_\texttt{S}$ queries. A complexity analysis revealed that the IBA is considerably more efficient than the GTA. We confirmed this statement through simulations, in which the IBA behaved well for – on average – mediators with up to 50 and more QCAs and queries with up to six literals. To our best knowledge, the IBA is the most efficient algorithm for query planning published until today.

Next, we studied *redundancy* in query plans. Query planning essentially produces a set of query plans. Computing the union of the results of those plans does not necessarily require that each query plan is executed literally. We described three algorithm that detect and remove redundancy within and between query plans. These algorithms differ in their complexity and in the classes of redundancy they detect: The more complex the algorithm, the more types of redundancies are detected. However, even the simplest method is already capable of considerably accelerating the computation of the answer to a user query.

Related Work.

The basis for query planning in MBIS using an LaV approach was first studied in [LMSS95]. In this paper, Levy et al. formally analyse the problem of "answering queries using views": They assume the existence of a set of materialised views and study how they can answer arbitrary queries using only those views. They prove that, for conjunctive queries with only equality predicates, it suffices to consider only combination of views up to the length of the query (see Section 5.1.2). Using such a mechanism for data integration was first sketched by Tsatalos et al. in [TSI94]. In gained popularity through a seminal paper of Levy et al. published in 1996 [LRO96a].

Apart from our work, three algorithms for query planning have been published since: The *bucket algorithm* developed by Levy et al. in the Information Manifold project [LRO96a; LRO96b], the *query folding algorithm* published by Qian [Qia96], and the *inverted rules algorithm* developed in the Infomaster project by Genesereth et al. [GKD97]. A predecessor of the IBA was published in [Les98a]. In the following we describe each of those three algorithms and compare them to our results.

Information Manifold: The bucket algorithm.

The Information Manifold project (IM) was the first to apply a Local-as-View approach to the integration of heterogeneous data sources. The IM uses a subset of description logic CARIN [LR96] as data model for mediator schemas. This subset is equivalent to non-recursive, positive DATALOG extended with inclusion dependencies between relations.

In the IM data sources are represented through *capability records*. A capability record is a five-tuple that contains an intensional description of the content of the data source in terms of a conjunctive view on the mediator schema, and information about possible conditions on attributes. The IM assumes that every data source is described through one *interface program*, i.e., through one relation.

The IM uses the *bucket algorithm* (BA) for query planning. The algorithm is described in three different papers. [LRO96a] describes the Information Manifold project in general. The paper only sketches query planning but concentrates on whether or not a query plan can be executed considering the capabilities of the sources represented through the binding patterns. The planning algorithm is explained in more detail in [LRO96b], but without explaining the actual algorithm for containment tests. The reader is referred to [LR96], which discusses the "existential entailment problem" for CARIN. The relationship to the containment problem for conjunctive queries is not straight-forward.

Therefore, it is difficult to compare the BA with our algorithms *in detail*. Roughly, the BA proceeds as follows:

- It first constructs one bucket per literal of the user query. Into this bucket it puts every view that contains a literal with the same name and for which a most general unifier between both literals exists. It does not check if all necessary variables are exported, as the IBA does, although the authors recognise this as a problem in [LRO96a]. The authors also observed the problem of non-exported variables carrying conditions that cannot be enforced by the mediator, but do not address it in the following.
- In a second step, the BA enumerates the cartesian product of all buckets. Each element of this product is a combination of views. However, the BA has to test more view combinations then contained in the cartesian product of the buckets. Consider that a combination <q_1,q_1,q_2,q_2> was enumerated. To ensure completeness, the BA has to test the following combinations: <q_1,q_2>, <q_1,q_1,q_2>, <q_1,q_2,q_2>, and <q_1,q_1,q_2,q_2>, because the BA has no merging operation as the IBA has. The BA checks for each such combination whether it is contained in the user query or not. If yes, the combination is considered as correct.

This approach is equivalent to the GTA enhanced with a bucket generation strategy as discussed in Section 5.2.4. We there also showed that bucket construction alone *does not improve* the complexity of query planning. Furthermore, the BA does not consider the difference between plans and query plans, and especially the 1:n relationships between both.

Neither of the papers gives a precise complexity of the BA algorithm. The authors only state that it is exponential in the length of the user query. This confirms our Theorem (T5.10), assuming that every QCA is considered as a unique data sources.

The IM applies a *minimisation procedure* to each correct plan [LRO96a]. This procedure is exponential in the length of a plan. We discussed similar techniques in Section 5.4. We also included heuristics that are faster and already find many non-minimal plans. Recall that executing only minimal plans *does not guarantee* the execution of a minimal number of queries. The notion of minimality used in [LRO96a] does not take plan transformer into account, and it is not clear how this would affect their algorithms.

Query folding.

Qian [Qia96] analyses the problem of *query folding*. A query folding is a rewriting of a query using views, and hence a different term for the problem of answering queries using views. Qian calls a query folding

- *complete* if the rewriting uses only views and no other relations. For information integration, only complete foldings are interesting.
- *strong* if the rewriting is equivalent to the original query. In MBIS, planning algorithms usually produce only contained rewriting, since extensional relationships between different data sources are anyway unknown.

The *query folding algorithm* (QFA) presented in [Qia96] proceeds as follows. It first constructs a set of folding rules by "inverting" each view. This means that each view is translated in as many rules as the view has literals. The head of each of these rules is the literal of the view and the body is the original view head. Non-exported variables in a rule are skolemized, i.e., replaced by a function term using all exported variables of the view as arguments.

Given a user query u, the QFA constructs one set for each literal l of u, called the *label* of l. In the label of l it puts all rules where:

- the head of the rule unifies with l, and
- after unification, the body of the rule does not contain any skolem term and the head does not contain a skolem term for an exported variable of l.

Finally, the QFA constructs the cartesian product of all labels. Rules, i.e., their bodies, are connected using the *unification join* (u-join).

We describe the algorithm by an example. Consider the following user query and views:

```
u(ge,se,pr,or) ← gene(ge,se,pr),org(ge,or);
```

```
v1(ge,se,pr) ← gene(ge,se,pr);
v2(ge,se,or) ← gene(ge,se,-),org(ge,or);
```

The inverse rules are (note the skolemisation of the non-exported variable in v_2):

```
gene(ge,se,pr) ← v1(ge,se,pr);
gene(ge,se,f(ge,se)) ← v2(ge,se,or);
org(ge,or) ← v2(ge,pr,or);
```

The labels for the two literals of u are:

```
Lgene= {(ge,se,pr)|v1(ge,se,pr),
        (ge,se,f(ge,se))|v2(ge,se,or)};
Lorg = {(ge,or)|v2(ge,se,or)};
```

The second element of L_{gene} is deleted because it is not "proper": It contains a skolem term in the place of an exported variable. The two remaining label elements are then combined using the u-join. This yields the one and only query plan:

```
u(ge,se,pr,or) ← v1(ge,se,pr),v2(ge,se,or);
```

The QFA is somehow similar to the IBA. The labels are similar to the initial buckets, and the u-join essentially plays the role of the union of two partial plans. However, the QFA cannot deal with conditions in queries apart from equations. The QFA is, as the IBA, exponential in the size of u. A more detailed complexity analysis is not published.

Infomaster: the inverse rules algorithm.

A third algorithm for query planning, called the *inverted rules algorithm* (IRA), was developed in the Infomaster project [GKD97]. The IRA first generates inverse rules in the same way as the QFA. These rules, together with the user query, are interpreted as a logic program. This program can be cleaned from potentially occurring functional terms. The resulting DATALOG program can be executed, where inverse rules are executed by calling the appropriate source interface.

The idea of the IRA is enticingly simply. Notably it can also deal with recursive user queries [DG97] (but not with recursive views, since this problem is undecidable, see Section 2.4), and can be extended to consider functional dependencies and binding patterns [DL97].

The IRA is very similar to the QFA. The main difference is that it generates a more compact representation of the overall solution – the result is a DATALOG program instead of a union of query plans. As the QFA, the IRA cannot deal with built-in comparison predicates. A further disadvantage is that, in contrast to the BA and the IBA, the IRA has no way of detecting when views have to be executed multiple times and when not. For instance, if a view body consists of a join between two literals that also appears in the user query, then the resulting DATALOG program will execute this view twice – once for each of the two literals, represented through different inverse rules. The IBA also generates a query plan with two occurrences of the view, but we showed that such redundancies can be removed easily.

The IRA is polynomial in the sense that generating the DATALOG program takes only polynomial time [DG97]. However, executing the program, for instance by a top-down evaluation, will nevertheless need exponential time. Generating the program is the same step as computing buckets in the IBA – and executing the program amounts to a traversal of the search tree. Infomaster uses only conjunctive queries with equality. For such queries, partial plans are only rarely incompatible. However, as soon as more expressive queries are permitted, it is reasonable to first test entire plans (as the BA and IBA does), instead of immediately starting to execute views (as the IRA does).

Interestingly, the IRA has a restricted, built-in multiple query optimiser. Consider the following user query and views:

```
u(a,b) ← rel1(a,c),rel2(c,d);
```

```
v1(x,z) ← rel1(x,z);
v2(z,y) ← rel2(z,y);
v3(z,y) ← rel2(z,y);
```

The BA, as the IBA or the QFA, will find two query plans:

```
u(x,y) ← v1(x,z),v2(z,y);
u(x,y) ← v1(x,z),v3(z,y);
```

and, without recognising the redundancy, execute $\mathtt{v}_1$ twice. In contrast, the IRA will generate the following logic program:

```
u(a,b) ← rel1(a,c),rel2(c,d);
rel1(x,z) ← v1(x,z);
rel2(z,y) ← v2(z,y);
rel2(z,y) ← v3(z,y);
```

Executing this program requires to execute $\mathtt{v}_1$ only once.

This mechanism does however not find all redundancies. As already mentions, the IRA can, for instance, not decide whether or not a view that appears twice in a program indeed has to be executed twice.

More query planning algorithms.

Chaudhuri et al. describe an algorithm for using materialised views in a central database [CKPS95]. Accordingly, the authors are only interested in *equivalent* and not in contained rewritings. The main advantage of their algorithm is that it can be seamlessly integrated in a conventional dynamic programming query optimiser. However, it is not complete, i.e., it does not find all possible query rewritings. The reason is that every replacement of a subquery of the query with a view is immediately performed whenever the algorithm finds one. Other views that also cover parts of this subquery are not considered any more.

Grahne & Mendelzon investigate the semantics of LaV [GM99] (see also Section 4.5). They briefly sketch an algorithm conforming to their semantics. Due to the brevity of the description, no comparison is possible. However, the authors compare their approach to the BA. They claim that a “straight-forward implementation” of the BA has complexity $\mathtt{O}\left(\mathtt{n}^{\mathtt{k}}\mathtt{n}^{\mathtt{ns}_{\mathtt{max}}}\right)$. This result conflicts with our observation that the BA has the same complexity as the GTA, which is $\mathtt{O}\left(\mathtt{e}^{\mathtt{k}}\left(\mathtt{s}_{\mathtt{avg}}\right)^{\mathtt{k}}(\mathtt{n}\ +\ \mathtt{k})^{\mathtt{k}}\right)$ (see Theorem (T5.10)). The difference is tremendous, since their formula implies that the BA is exponential both in the size of the query and in the number of QCAs. Since the authors do not motivate their formula, we cannot explain the difference.

Extending user queries.

Extending the expressiveness of *user queries* with interpreted predicates is relatively simply. Consider, for instance, inequality predicates (a ≠ b). We can treat inequalities in the following way: We first remove all inequalities from the user query and generate query plans for this modified query as usual. After executing all query plans, we compute the original query on those results.

Note however that, by doing so, predicates are *never pushed* into the wrappers. Furthermore, it is not detected when a mediator query conflicts with such predicates. Consider for instance:

```
u(a,x) ← rel(x,y),x≠y;
q(a) ← rel(a,a);
```

The mediator executes q although the result of q is certainly not contained in the result of u.

User queries with disjunctions can be treated by first rewriting them into their disjunctive normal form and then planning each of the disjuncts separately. As for inequalities, this method is feasible, but not optimal, since disjunctions are never pushed.

Binding patterns.

Query planning in the presence of binding patterns was first analysed in [RSU95]. To include binding patterns, a couple of things have to be considered:

- The bound for the length of query plans does not hold any more. Kwok & Weld show that actually no length bound exists at all [KW96]. Query planning with binding patterns is hence inherently incomplete.
- Our semantics for MBIS is ill-defined in the presence of binding patterns since our notion of "executable query" is not sufficient any more (see Definition (D3.2), page 53).
- Consequently, the definition of "executable plans" also needs to be adapted (see (D5.5), page 88). A plan is only executable if there exists an ordering on its QCAs such that all binding patterns are respected. Finding such an ordering is simple. A polynomial algorithm is described in [LRO96a].

A detailed analysis of the impact of binding patterns on the search space of a query planner is given in [FLMS99].

Quality based query planning.

In many domains, the *quality of information* obtained from different data sources varies to a great extent. This leads to the situation that, within the many query plans that may be found for a user query, already a few will produce most of the relevant data. Avoiding the execution of query plans that obtain only qualitatively bad results can save considerable time without significantly lowering the quality of the overall result. Identifying "bad" query plans incorporates two sub-problems: measuring information quality, and using quality measurements in the query planner.

In joint work with Naumann and Freytag, we described a framework and algorithm for quality-driven query planning in [NLF99b]. We address the first problem by introducing *quality-annotated QCAs*. We describe a set of 12 different criteria that capture a domain specific definition of information quality, including for instance completeness, accuracy, and timelines. A vector of scores for those 12 criteria is attached to each QCA, measuring the quality of the information described through this QCA. During query planning, we consider information quality annotations in a two step procedure: In the first step we compute the set of all query plans. In the second step we sort query plans according to their quality, where the quality of a query plan is aggregated from the quality of its QCAs following the join structure of the plan. We therefore devise appropriate merge functions to combine quality annotations.

Other extensions to query planning.
Query planning with specialisation relationships between relations of the mediator schema is not more difficult than without. For the IBA, it is only necessary to adopt the bucket construction. For a bucket of a literal `l` of a relation `rel` in the user query, we not only have to consider literals for `rel` in mediator queries, but also literals for specialisations of `rel`. The complexity of the algorithm does not change. Specialisation relationships have already been mentioned in the Information Manifold project. A generalisation to arbitrary inclusion dependencies is formally analysed in [Gry98].

Recursive user queries and mediator schemas with functional dependencies are considered in [DG97; DL97]. Query planning with negation is described in [LS93]. [CNS99] treats materialised views involving aggregation. [LS97] discussed complex objects and query containment. Finally, [Mil98] investigates query containment and query planning in the presence of schematic variables, i.e., queries where relations or attribute names can be variables.

Multiple query optimisation.
Multiple query optimisation has received surprisingly little attention in the database research community. Only a handful of publications appeared in the last decade (e.g., [AR94; SSN94; CD98]; see also [Kin99] for a more detailed survey). MQO is usually carried out in two phases: (1) The detection of common subexpressions, and (2) the finding of the least expensive set (and order) of subexpressions such that all required queried are answered. Our approach described in Section 5.4 concentrates on the first of these two problems. Our cost model assumes that the less remote query executions are performed, the better will be the response time. This consideration was mainly motivated by the work of Johansson [Joh98], which shows that network latency dominates network bandwidth more and more.

Multiple query optimisation has received virtually no attention in information integration, although almost every algorithm in this context generates sets of highly redundant plans. Neither the BA nor the QFA algorithm consider MQO. As described above, the IRA implicitly carries out a limited form of multiple query optimisation. Apart from this work, this topic was, to our best knowledge, not discussed in a MBIS context before.

Friedman & Weld analyse *extensional relationships* between different data sources, which can also be considered as a form of redundancy [FW97]. Their idea is to define inclusion dependencies between wrapper queries explicitly, using so-called *local-completeness axioms*. With this mechanism one can, for instance, specify that a certain source is complete with respect to a certain mediator relation, implying that only this source should be used for the “filling” of the relation. All other sources can be disregarded.

The advantage of this approach compared to our methods is that more forms of redundancy can be detected. Local completeness axioms can also be used to model *preferences* between different sources, determining for instance that data from a source `A` should be considered before data from a (perhaps qualitatively worse) source `B`. The main disadvantage of local completeness axioms is that additional rules have to be specified. Especially in a web context, it is almost impossible to derive such rules or to control their correctness in the presence of change.

Another line of related work is *semantic caching* (see [KB94; DFJ+96]). Semantic caching means the caching of query results together with their definitions. Semantic caches may be considered as temporarily materialised views. It remains an open question whether or not the same effect as with MQO can be achieved through an intelligent usage of semantic caching.

6. METHODOLOGY

In the previous chapters we have discussed techniques for the construction of MBIS. In particular, we presented a language for the specification of correspondences between heterogeneous relational schemas, and we introduced algorithms that can translate queries against one schema into sets of queries against other schemas. These two contributions build the technological core of a MBIS.

However, developing a running system requires more than a technological basis. In this chapter, we discuss issues in the development and deployment of a MBIS. We do not examine the development of programs that constitute a MBIS, such as programming wrappers or implementing query planning. Instead, we assume such programs to be given and analyse their usage for the construction and maintenance of a concrete system.

The intention of this chapter is to show that the usage of MBIS using QCAs is a feasible approach to information integration. We highlight issues that support this claim. We do not attempt to give a complete development methodology for MBIS. Issues such as tool support, construction of user interfaces, or run-time management, are beyond the scope of this thesis. Interested readers are referred to [BS95; BE95; BBE98; KPS99].

The lifetime of a MBIS consists of four phases:

- In the *analysis phase*, the information requirements are identified and available data sources are characterised wrt. those requirements. If certain requirements remain uncovered, i.e., the necessary data is not available in any of the data sources, new data sources have to be identified and analysed.
- In the *design phase*, the mediator schema is designed based on the information requirements. Furthermore, the export schemas of wrappers are determined. If the requirements yield a large and complex mediator schema, is should be considered to break up that schema into a set of smaller schemas managed by separate mediators. In this case, the developer needs to determine which mediators use which data sources or mediators. Once the participating schemas and their relationships are established, suitable sets of executable queries against wrapper schemas (or mediator schemas in case of multiple mediators) are identified. Finally, QCAs are specified that describe those queries wrt. the schemas of the mediators that are supposed to use them.
- In the *implementation phase*, wrappers are developed that encapsulate data sources and that export the previously specified schemas. It has to be ensured that the previously identified queries are executable.
- In the *deployment phase*, user queries are answered through the mediator. Furthermore, the MBIS needs to react on various types of change, such as the addition or deletion of data sources or evolution in wrapper schemas. This is especially important in a web environment, where data sources can appear and disappear over night.

In the following, we discuss three critical issues in the design, implementation and deployment of MBIS based on QCAs. In Section 6.1 we describe properties of different types of data sources wrt. their integration in a MBIS. We examine relational databases, information systems accessed through a CORBA interface (CORBA based information systems, CBIS), and information systems accessed through the web (Web based information systems, WBIS). Our discussion covers the construction and reuse of wrappers as well as guidelines for the derivation of wrapper schema and wrapper queries.

Together, a wrapper schema and a set of wrapper queries form the interface between a mediator and a wrapper. The schema and the queries against this schema are of course strongly interdependent. This observation does not imply that either of both is superfluous. Despite their strong interconnection, both schema and queries are important for the description and integration of wrappers into MBIS. For instance, if a relational database is integrated, the natural point to look at is the schema of this database. Interesting queries may be derived in a second step, since all queries are executable. In contrast, if a WBIS is integrated, it is more natural to first look at the set of executable queries.

However, we illustrate in Section 6.1 that, for conceptual design, it suffices to consider only the export schema. Since wrapper schemas are developed independently from the mediator schema, conflicts are inevitable. In Section 6.2 we show how QCAs bridge different types of such conflicts, including semantic, structural and schematic conflicts. The section also includes a classification of possible conflicts between relational schemas.

In Section 6.3 we analyse one important aspect of the deployment phase of a MBIS: maintenance. In Section 3.2 we argued that a top-down development has advantages over a bottom-up development regarding maintainability. The reason is that the mediator schema and the wrapper schemas are not tightly connected, but only loosely coupled through declarative rules, i.e., QCAs. Due to this independence, many types of changes in a MBIS can be handled on the level of QCAs without affecting schemas. In Section 6.3 we confirm this statement by investigating five types of change in MBIS. For each, we describe the actions a mediator has to take for compensation.

6.1 Integrating Different Types of Data Sources

Typically, MBIS must deal with a large variety of different data sources: relational and object-oriented databases, web based information systems, files and email-collections, CORBA- or DCOM-based information systems, etc. One of the distinctive features of MBIS in contrast to FDBS is their ability to handle this heterogeneity (see Section 3.1).

In the following we discuss specific problems that are posed by three different types of data sources: relational databases, CORBA based information systems, and web based information systems. We aim at providing guidelines for the derivation of the logical interface between a wrapper for such sources and a mediator. The *logical interface* comprises the export schema of the wrapper and the set of executable queries against this schema – independent of its technical implementation.

In the literature, one can find many types of wrappers, ranging from as few lines of PERL code searching a web page using “grep”, to large software projects integrating a file-based document system in a transaction oriented CORBA architecture. In MBIS, wrappers can be distinguished by the amount of additional functionality they implement to facilitate access to the data source. Possibly add-ons are for instance:

- Implementation of additional search capabilities.

- Translation of languages, codes, vocabularies, and units.
- Session management and failure handling.

However, we assume that the goal is to provide "thin wrappers" quickly, i.e., we are interested in being able to provide wrappers with minimal effort. Thin wrappers only export functionality that is provided by the data source they wrap. We develop guidelines to support the development of thin wrappers.

6.1.1 Relational Databases

Integrating a relational database (RDB)[12] as data source into a MBIS is relatively easy because, in contrast to many other types of data sources, a relational database has a schema. The export schema of a wrapper for a RDB will usually be either identical to the schema of the RDB or a subset of it. Additional views inside the RDB can be used to facilitate the definition of QCAs, or to protect the MBIS from schema changes.

Furthermore, RDB are accessible through a query language such as SQL. All types of queries discussed in this work are supported. Therefore, all queries against the export schema of a RDB wrapper may be considered as executable.

Technically, RDBMS are accessible through a vendor-specific protocol, such as the "Oracle Call Interface". Vendor-independent *database gateways*, such as "Open Database Connectivity" (ODBC) from Microsoft or "Java Database Connectivity" (JDBC) from Sun, have been developed to shield a developer from those proprietary protocols by providing standardised methods, for instance for the execution of queries and the reception of query results. They do not hide syntactical heterogeneity, as for instance present in different SQL dialects. Therefore, a wrapper developed for one RDB, for instance using JDBC, may often be reused for all other RDB running on the same RDBMS. To reuse it for a different RDBMS, modifications will be necessary.

Providing true vendor-independent database access is the aim of *database middleware* [FRH98]. Examples are IBM's "DataJoiner" [CHKR98] or Oracle's "Oracle Transparent Gateway"[13]. Gateways hide location, network, and language heterogeneity within SQL dialects. For instance, the Oracle Transparent Gateway allows to access more than 30 different RDBMS using Oracle's SQL syntax. Using such gateways enables wrapper developers to write code that can be used on any RDBMS supported by the gateway – but also ties developers to the vendor of the gateway product.

More recently, database vendors also started to develop gateways that also access data that is not stored in RDBMS but, for instance, in structured files. This approach follows the idea of *universal access* (the database is able to access everything) as opposed to *universal storage* (the database is able to store – and access – everything) [BP98]. For instance, Microsoft's OLE DB (object link embedded – database) defines a standard interface for the access to any type of data based on the concept of *rowsets* [Mic98]. The same line is followed by IBM with the Garlic project [HKWY97]. Note that the integration of non-databases into such systems also requires the development of appropriate wrappers.

Gateways and database middleware may greatly facilitate the integration of data sources into MBIS. The usage of middleware components increases reusability of code and often achieves increased performance compared to hand-made wrappers. However, gateways and database middleware must not be confused with mediators. A mediator provides location, schema, and language transparency for users. Typical database middleware at most provides

[12] With use the abbreviation RDB for a concrete relational database, whereas the acronym RDBMS stands for a software system for the generation and administration of RDBs.

[13] See http://www.oracle.com/gateways

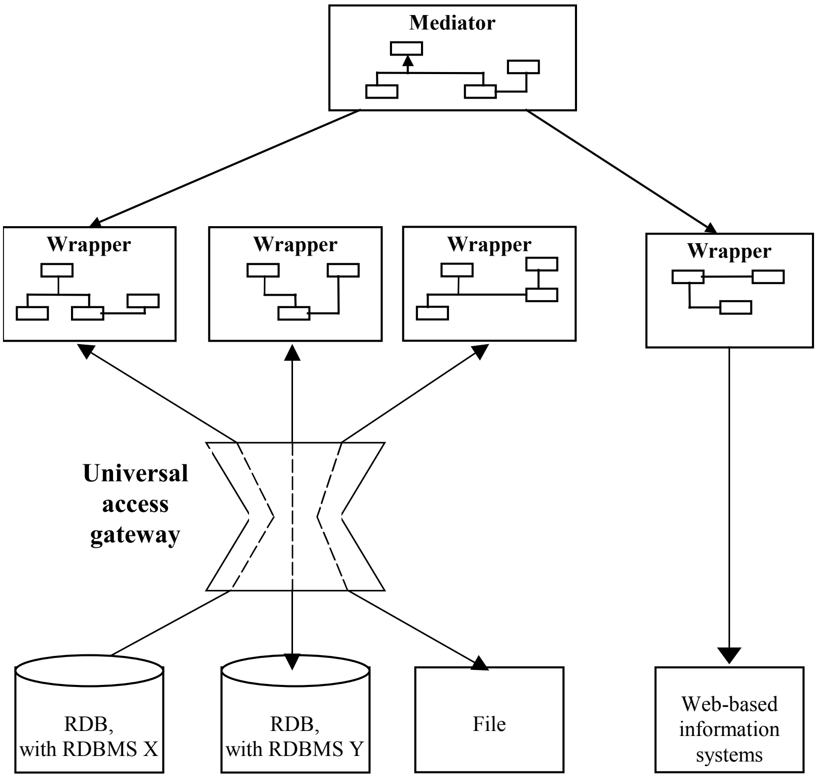


Figure 33. Gateways in MBIS.

location and language transparency, but never rewrites queries. Middleware does not translate between queries against different, heterogeneous schemas apart from the possibility offered by relational views. Furthermore, middleware does not automatically compute unions of results obtained from different data sources as query planning does. Middleware should therefore be considered not as replacement, but as support for MBIS (see Figure 33).

6.1.2 CORBA Based Data Sources

We do not intend to discuss CORBA exhaustively but only give a short summary here and then concentrate on their integration in a MBIS. Interested readers are referred to [WT94; Bak96; LTB98; ML99].

CORBA based information systems (CBIS) are accessed through an interface described in the "Interface Definition Language" (IDL) defined by the OMG[14] (see for instance [Bak96; OHE97]). CBIS interfaces are independent of the type of underlying system. To achieve this independence, CORBA servers are in fact object-oriented wrappers.

A CBIS interface comprises classes with attributes and methods[15]. There are several possibilities how the structure of databases, or of any structured data store, can be mapped to a CORBA interface [LTB98]. The simplest is to wrap the entire database into one class with one method, which takes arbitrary queries as strings and returns answers as some kind of structured string list. On the other extreme, every relation of a RDB is modelled through a proper interface class[16].

In between those two extremes lies the most common usage: Modelling only selected relations as interface classes, and offering a fixed set of parameterised queries with predefined

[14] See http://www.omg.org

[15] In OMG notation, every class is an interface, and attributes are called members.

[16] This approach faces the serious problem of representing results that are not tuples of a relation.

result type. This approach has the additional advantage that it is truly independent of the underlying data store. The previous two strategies either directly reflect a schema in the interface or require the client to have detailed knowledge about the schema for the formulation of queries. Domain standards defined by the OMG, such as in the field of life science research [LSR97], adopt this strategy to insulate clients from heterogeneous data providers.

To integrate a CBIS into a MBIS, we must synthesise an export schema and a set of queries. Consider the interface of a CBIS consisting of classes and methods that essentially encapsulate queries. One possibility is to represent each interface class and each method through a proper relation in the relational export schema of the CBIS wrapper.

The attributes of an interface class all become attributes of the according "class relations". However, there is no generic way to query such classes in CORBA. The only possibility is to access objects by their CORBA reference, which could be included as additional attribute in the relation. All queries must have a binding for that attribute, and no other bindings may be permitted. This approach faces two serious problems: First, CORBA references are transient if the server uses the *basic object adapter* [Muel99]. Transient references change between sessions, rendering them unusable in a global context. Second, CORBA references carry no meaning and are hence inadequate as global keys. We conclude that "class relations" are not useful.

Attributes of "query relations" are all parameters of the corresponding method. Methods in IDL are remote procedures with `input`, `output`, and `inout` parameters. All queries can be allowed that adhere to the binding pattern determined through the `input/output` status of the attributes. Clearly this simple 1:1 mapping cannot cope with complex attributes such a IDL `struct`'s or `sequence`'s. In such cases more elaborated mapping functions must be implemented.

Unfortunately, *IDL is not a data model* – methods are not declaratively described (as queries are) but programmed. It is therefore not possible to derive universally valid statements from the signature of a IDL method. On the other hand, in many CBIS, methods indeed correspond directly to fixed queries against a database backend server. A mapping from the method signature into a relation of the export schema is then straight-forward.

Compared to RDB, CBIS introduce the need for binding patterns on queries. A query that is translated into a method call

- must have bindings for attributes corresponding to `input` parameters,
- must not have bindings for attributes corresponding to `output` parameters, and
- may have bindings for attributes corresponding to `inout` parameters.

In Section 4.5 we showed that annotating QCAs with binding patterns does not pose conceptual problems. In Section 5.5 we discussed necessary modifications to a query planning algorithm considering binding patterns.

Technically, a CBIS is accessed by binding to an *object request broker*, which are available from different vendors. At the client site, first interface stubs are generated through an IDL compiler and second the desired functionality is programmed using those stubs. The interface and the implementation together form a *CORBA server*. Whether or not CORBA servers can be reused cannot be judged in general. First, it requires that the interface itself is reusable. This is, for instance, the case if the interface is a standard, or if it is highly generic. Second, it requires the use of widely accepted methods for the implementation, such as JDBC or ODBC for accessing the backend data store.

6.1.3 Web Based Data Sources

Web based information systems (WBIS) are accessed through a World Wide Web page or page collection (web site). WBIS interfaces are not designed to be used by programs, but by humans. Constructing wrappers for WBIS is therefore considerably harder than for RDB or CBIS. WBIS require wrappers that include complex and fault-tolerant parsers to pick the desired information from primarily design-oriented HTML pages. Also, such wrappers must translate declarative queries into a typically navigation-oriented site logic.

Therefore, it is reasonable to build wrappers for WBIS using a *wrapper specification language* (WSL). Typically, WSL have special elements for text parsing, such as regular expressions or grammar rules, and for HTTP operations, such as requesting a web page or submitting a HTML form. Existing wrapper languages are, for instance, JEDI [HFAN98] or W4F [SA99].

In [Hol99], the author describes the wrapper language WWScript, which is exemplarily considered in the following. The interface between a wrapper written in WWScript and a mediator is defined through a single export relation forming the export schema of the wrapper. This relation is a denormalised representation of the content of the data source similar to a *universal relation* [Ull89]. It can be derived through a analysis of the web sites. The attributes of that relation carry query capabilities, such as possible selection operations and binding patterns. Those query capabilities are determined by the forms offered by the data sources, since web forms often only allow the specification of conditions for some of the attributes. For instance, a search form of a library often only allows to search for authors and title, although the results will also include the publisher, the date of publication, etc. On the other hand, many source have several different search forms to support different types of queries and differently skilled users. For instance, the "Genome Database" [FLL+97] has one search form for every of its over 30 different object classes. Using WWScript, this can only be described through different wrappers.

The export schema of a wrapper written in WWScript is its export relation. The range of executable queries is determined through the query capabilities. QCAs can use any executable query as wrapper queries.

Internally, a wrapper consists of three layers (see Figure 34). The *communication layer* requests documents from a remote web server and receives HTML pages. The *extraction layer* parses those pages and extracts desired information. This information is transformed into tuples of the export relation in the *restructuring layer*.

Queries are transmitted to the wrapper in the form of bindings for attributes of the export relation. Such queries are pushed to the source by using the appropriate web forms. A WWScript developer can also choose to program manually additional selection operations inside the wrapper, for instance to filter out temporary tuples extracted from a web page. Consider the DBLP web site, which is a collection of publications in database research[17] (see also [Hol99]). The export schema of this site will be a relation with attributes for the authors, title, date and journal/conference of publications. The site offers two search forms: either searching the collection by author name or by publication title. To use those forms, the corresponding query capabilities must specify that either a author name or a title must be given but not both at a time. The wrapper can then use the proper form to answer the query. However, the WWScript developer could also choose to allow the specification of both an author name and a title at the same time. For such queries, the wrapper could use the author form to retrieve all publications of the searched author and scan this list for the given title, removing all non-matching entries.

[17] See http://dblp.uni-trier.de

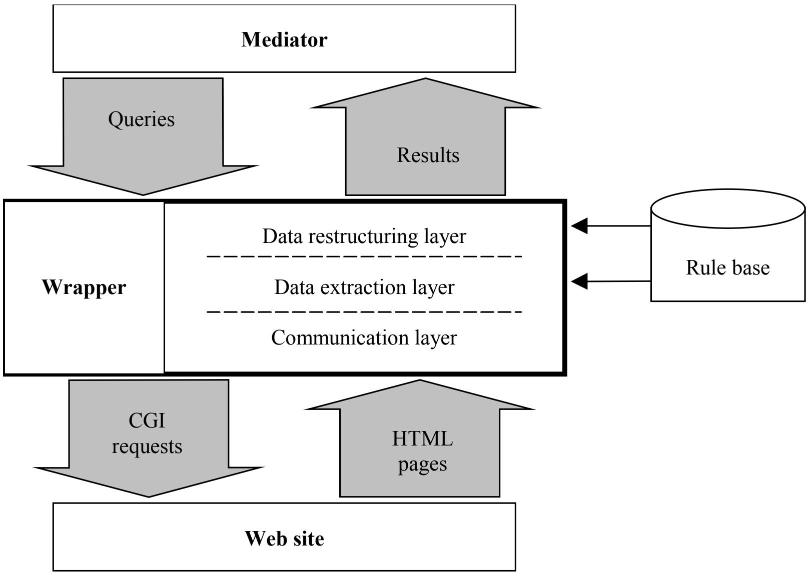


Figure 34. Architecture of a web wrapper.

Searching for publications for a given author and a given title is an example for a capability that is not offered by the source, but can easily be implemented by the wrapper. A more elaborate wrapper could even answer queries without any bindings, because DBLP has a web page that contains a list of all authors. This page can be used to access all publications in DBLP.

These examples show that a wrapper developer often has the choice of how powerful its wrapper shall be. Tork Roth et al. suggest in [TRS97] the reasonable strategy to first implement dumb wrappers quickly and then, if necessary, successively refine them by adding new functionality.

Technically, a WBIS is accessed through the wrapper, which uses HTTP. The two essential mechanisms is the request of HTML pages and the execution of so-called “CGI programs” (*common gateway interface*), which amounts to the execution of a program on the web server’s machine. CGI programs are used to execute queries triggered through the use of web forms.

Due to the high individuality and evolvability of web pages, it is difficult to reuse wrappers for WBIS [Fau00]. The design and structure of a web site is nowadays – commencing into the e-commerce area – considered as a distinguishing feature of companies giving competitive advantage or disadvantage. Therefore, no standardisation can be expected in the near future. However, the content of web sites could soon be represented through XML [MG98; SA99] rather than HTML, which would greatly facilitate the construction of wrappers (less parsing) and also increase their reusability.

6.2 Bridging Heterogeneity through QCAs

Query processing in MBIS is mainly hindered through the existence of various forms of heterogeneity in the participating systems. Heterogeneity results from autonomy, since autonomous systems naturally use different ways to achieve their particular goal. If we accept autonomy of data sources as prerequisite, heterogeneity is inevitable.

The previous section described ways to derive export schemas and executable queries from autonomous data sources. To make them available for a mediator, we must define the results of those executable queries in terms of the mediator schema. We use QCAs for this purpose. QCAs declaratively describe correspondences in the presence of heterogeneity.

In the following discussion we consider various conflicts between two relational schemas. We implicitly assume that every query against those schemas is executable. This is reasonable for the mediator side of a QCA and also justified for wrappers for RDB. However, for other types of data sources it is not obvious since we assumed throughout this work that only some queries against the export schema of a wrapper are executable. However, we have seen in the previous sections that for CBIS and for WBIS the export schema of a wrapper is *determined* by the set of executable queries – the interface to such sources actually is a set of queries. In both cases we essentially considered each executable query as a relation of the export schema. Therefore, each query against the export schema is, by construction, executable: at most, it can require the concatenation of different wrapper queries.

We start with a brief classification of types of heterogeneity. Then, we demonstrate the flexibility of QCAs by examples for semantic, structural, and schematic conflicts. MBIS are developed top-down, i.e., the first step is to design a mediator schema. However, QCAs are designed *bottom-up*: First, we identify executable wrapper queries. Then, we describe them through corresponding mediator queries.

6.2.1 Heterogeneity in MBIS

There exist numerous classifications of heterogeneity in database integration. Examples include [SPD92; KCGS95; KS96; VJB+97]. Usually they discern between:

- *Technical heterogeneity*: Data sources can differ in their hardware platform, operating system, query language, access mechanism, data representation, etc.
- *Data model heterogeneity*: Data sources can present data using an object-oriented, semantic, hierarchical, or relational data model [SCGS91].
- *Semantic heterogeneity*: Data may differ in the meaning of terms, and units, leading to synonyms and homonyms [Web82; SK93; VJB+97].
- *Structural heterogeneity*: Data can be stored in different structures, e.g. different degrees of relational normalisation [KCGS95; Sau98].
- *Schematic heterogeneity*: Schematic heterogeneity is a special case of structural heterogeneity. We speak of schematic heterogeneity if data is be represented by different concepts of the same data model, for instance attribute versus relation or value versus relation. [LSS93; Mil98].

For our work, we assume that technical and data model heterogeneity are resolved on the wrapper level (see Section 3.3.1). Therefore, QCAs only have to cope with the last three cases. We treat each of these types of conflicts in a separate section in which we also explain the nature of the conflict in more depth. Since we use the relational data model, only conflicts between relational schema are considered.

In real life applications, different types of heterogeneity usually appear in combination. It is usually difficult to break up such combinations into their "true" types of heterogeneity. No classification claims to define orthogonal classes of conflicts. Many conflicts are ambiguous; others are assigned to different conflict classes by different authors. For instance, [SPD92] defines semantic conflicts as conflicts in the extensions of classes; discrepancies in the representation of classes are called *descriptive conflicts*; and their examples of structural heterogeneity can be found as schematic heterogeneity in our classification.

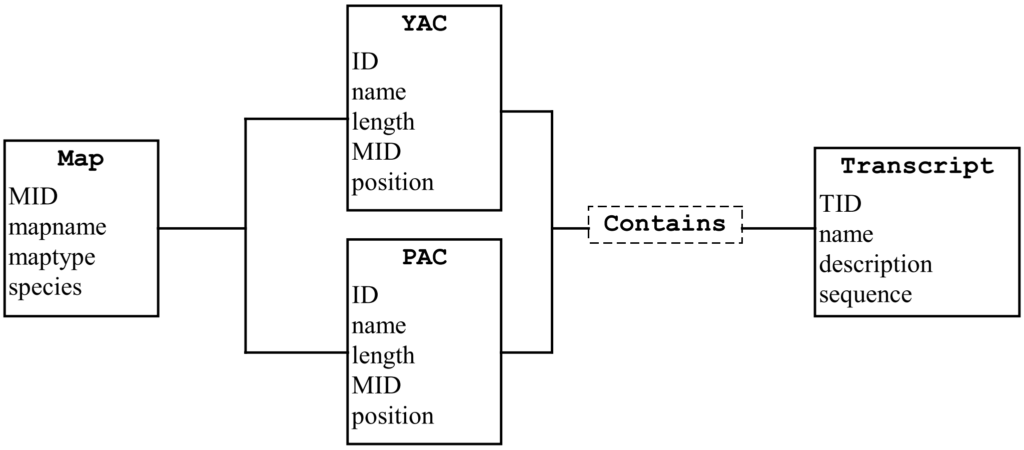


Figure 35. Exemplary wrapper schema.

For these reasons we do not aim to give a precise and complete characterisation of all possible conflicts between relational schemas. Instead, we show the potential of QCAs by examples. We use the mediator schema defined in Table 1, page 18, and the schema of a wrapper `W` defined in Table 5. A graphical representation is given in Figure 35. We denote the mediator schema with `S` and the wrapper schema with `T`. Relations of either are identified by prefixing them with either `S` or `T`.

```
map(mid, mapname, maptype, species);
PAC(id, name, length, mid, position);
YAC(id, name, length, mid, position);
contains(id, tid);
transcript(tid, genename, genedescription, sequence);
```

Table 5: Relational schema `T` of the fictive wrapper `W`. See also Figure 35.

In general, conflicts between different relational schemas can not be identified by considering only the schema definitions. One also needs a documentation clarifying the meaning of terms. In this sense, we shall give the semantics of `T` by explaining the conflicts with `S`.

We assume the following correspondences and conflicts between `S` and `T`:

- `T.transcript` has the same intension as `S.gene` (homonym, semantic conflict).
- Clones as stored in `S.clone` can occur in the tables `T.YAC` and `T.PAC` in `T` (structural and semantic conflict). The names of these two relations correspond to the `cloneType` attribute in `S.clone` (schematic conflict).
- `T.YAC` and `T.PAC` directly store the location of a clone on a map (through `mid` and `position`) (structural conflict). This implies a 1:n relationship between `T.YAC/T.PAC` and `T.map`.
- The length of `S.clone` is given in KB while the length of `T.YAC` and `T.PAC` are given in MB (structural conflict).
- Sequences for clones are not stored in `T`. Sequences for transcripts are stored in `T.transcript`. There is no separate `sequence` relation in `T` (structural and semantic conflict).

- The two maptype attributes have different meanings: in S, maptype describes the type of a map, i.e., the way it was constructed (e.g. physical, genetic, or radiation hybrid), while in T maptype describes the status of a map (e.g. atwork, confirmed, finished). All maps in T are physical maps in S (context problem, semantic conflicts).
- T.map has no mapSize and chromosome information in contrast to S.map. S.map lacks the species attribute (missing attributes, structural conflict). We assume that T stores data about arbitrary mammals (discerned through the species attribute), while S only addresses humans (context, semantic conflict).

In the following three sections, we classify the conflicts and give appropriate QCAs.

6.2.2 Semantic Heterogeneity

A data value alone has no meaning. Only the combination of a value and a description of what this value should mean is considered as information. The value of such a pair is its *extension*; the description is its *intension* and defines the set of allowed values, i.e., real-world objects. This perception is shared by many authors, and also used in the ANSI and ISO standards for "Information Recourse Dictionary Systems" (IRDS, see [ISO90; HL93]).

A semantic conflict only occurs in the description of data. In the case of relational schemas, such a "description" is in first place the name of a schema element, i.e., a relation or an attribute. This name must be accompanied by a textual documentation of what it stands for, since the meaning of names is highly context dependent and often ambiguous. A prominent German example is the term "Bank", which can be a financial institute or a resting place in a park. The documentation of a name must precisely define the semantics of any tuple (value) that is stored in a relation (attribute). The type of the schema element itself, i.e., the arity of a relation name or the domain of an attribute, is in contrast only a structural property of the information. Therefore, we consider type conflicts in Section 6.2.3.

It follows that semantic heterogeneity only occurs as conflicts in the *meaning of names*. A name has a denotation, which is a real world thing, and it stands for a concept, which determines the thing [Web82].

A similar triplet is used in the context of ontologies. An ontology is a "an explicit specification of a conceptualization" [Gru93]; for our purpose, it suffices to consider an ontology as an exact definition of terms within a certain domain, similar to a thesaurus or glossary. [VJB+97] uses the following terms: a (new) concept T (definiendum) of an ontology is characterised by a definition in terms of existing concepts of the ontology (the definiens) and an "ontological description" (concept description) in natural language. Projects that concentrate on the (semi-)automatic detection and removal of semantic conflicts therefore often use ontologies [Gru93; GMS94] as a formal and homogeneous basis for the definition of terms.

In our setting, the mediator schema takes the role of an ontology, although it clearly lacks certain properties that are usually assigned to "real" ontologies, such as a highly expressive language (for instance a description logic [Bor95; CPL97]) and the intention to model a domain completely . Nevertheless, the intuitive semantics of concepts of wrapper schemas have to be defined in terms of he concepts in the mediator schema. This does not imply any semantic reasoning: Semantic relationships, i.e., QCAs, are defined by humans.

We discuss the following cases of semantic heterogeneity:

- Synonyms in relation and attribute names.
- Homonyms in relation and attribute names.
- Relations with overlapping, but not identical meaning.
- Context propagation as a particular problem occurring with overlapping, but not identical scope in different schemas.

All our observation hold for attributes, relations, and queries, although we only give examples for the first two.

Synonyms.
Two relation (attribute) names are *synonyms* if the names are different, but express the same concept and hence have the same intension. In our example, the relations `T.transcript` and `S.gene` are synonyms, as are the attributes `name` in `T.YAC` and `clonename` in `S.clone`, restricted to clones of `clonetype "YAC"`.

Expressing this relationship in QCAs is straight-forward, especially since our query notation identifies attributes by position and not by name. Synonymous attributes are simply represented through the same variable in the mediator query and the wrapper query of a QCA they appear in. Synonymous relations are connected through a QCA expressing the intensional equivalence.

A QCA describing the relationship between `T.transcript` and `S.gene` is:

r_1: gene(gid,gn,gd) ← W.v_1(gid,gn,gd) ← transcript(gid,gn,gd,-);

Homonyms.
Two relation (attributes) names are *homonyms* if the names are identical, but express different concepts and hence have different intensions. In the example, the attributes `maptype` of `T.map` and `S.map` are homonyms, since the two attributes have different meanings.

Homonyms are also easy to capture through QCAs. The real problem in homonyms is not about names, but about the concepts that the names stand for. If two equal names, one in the mediator schema and one in the wrapper schema, stand for different concepts, then two cases are possible:

- either the concept that a name stands for exists in the other schema as well, probably assigned to a different name. Then we ignore the homonym and treat the synonym.
- or the concept that a name stands for does not exist in the other schema. Then we ignore it, since QCAs only connect intensionally identical relations and queries.

Consider the `maptype` attribute. The concept that `T.maptype` stands for, i.e., the processing status of a map, does not exist in `S`. Hence, we ignore `T.maptype`. Similarly, the concept of `S.maptype` does not exist in `T`. But we have additional knowledge that can substitute the missing concept with a constant:

r_2: map(mid,mn,mt,-,-),mt='physical' ← W.v_2(mid,mn) ← map(mid,mn,-,-);

Overlapping meaning of relations.
Two relations with overlapping, but not identical intensions are the most difficult type of semantic heterogeneity. For instance, `S.map` is similar to `T.map`, but not identical: `S.map` is more general since it comprises all types of maps, while in `T` only physical maps are intended. r_2 exactly handles this conflict. In this case, the difference between the intensions of the two `map` relations is clearly defined and can be leveraged by a condition. Unfortunately, this is nothing one can rely upon, because relations may have different intensions without representing this difference in the schema.

We structure our discussion in the following way. First, we assume that the discerning 'bit' is available in one of the schemas, i.e., expressible through a query. We distinguish three different cases for the relationship between the intensions of R_M and R_W. Next, we discuss the case that the 'bit' is not expressible through a query.

1. R_M and R_W are semantically overlapping, but not identical. The difference can be expressed through conditions in a QCA:

a) intension(R_M)$\supseteq$intension(R_W), i.e., R_W has a more restricted scope compared to R_M, which implies that R_W is a generalisation of R_M. An example is the map relation regarding maptype. Intensionally equivalent queries can be achieved through a condition in the mediator query.

b) intension(R_M)$\subseteq$intension(R_W), i.e., R_M is a specialisation of R_W. For instance, S addresses only human maps, while T stores maps about all mammals. Since the source does store the species of each map, we can include an appropriate condition in the wrapper query:

r_3: map(mid,mn,-,-,-) ← W.v_3(mid,mn) ← map(mid,mn,-,sp), sp='human';

Together with r_2 we can derive an improved QCA for map:

r_4: map(mid,mn,mt,-,-),mt='physical' ← W.v_4(mid,mn) ←
map(mid,mn,-,sp), sp='human';

c) intension(R_M)$\cap$intension(R_W) $\neq \emptyset$ and neither is contained in the other. This is a combination of cases (a) and (b). map is actually an example for exactly this. As another example, assume that S.map is restricted to physical, genetic and radiation hybrid maps, while a source Y has a map relation for physical, genetic and transcript maps. Expressing this in QCAs is a bit awkward, since we do not allow the IN operator in queries[18]. We have to define one rule for each of the common map types:

r_5: map(mid,mn,mt,-,-),mt='physical' ← Y.v_1(mid,mn,-) ←
Y_map(mid,mn,mt),mt='physical';
r_6: map(mid,mn,mt,-,-),mt='genetic' ← Y.v_2(mid,mn,-) ←
Y_map(mid,mn,mt),mt='genetic';

This makes all transcript maps from Y invisible for the mediator. This is necessary if the map types in S are only those given above.

There is a notable difference in the consequences of failing to specify such conditions. In case (a) no wrong data appears at the mediator. Actually, QCAs with subsuming queries are perfectly allowed. However, it is advantageous to "achieve" equivalence since otherwise we loose an optimisation possibility: Only queries asking for physical maps should use W. In case (b) the mediator will report wrong data because the QCA (without the condition) is wrong: the intension of the wrapper relation is neither equivalent to nor subsumed by the intension of the mediator relation. Since no species attribute is present in S, we cannot distinguish "good" tuples from "bad" tuples in M. Wrong data will be produced for any user query asking for a map of non-humans, including queries asking for all maps.

2. R_M and R_W are semantically overlapping, but not identical. The difference *can not* be expressed through conditions in QCAs:

a) intension(R_M) $\supseteq$ intension(R_W). For instance, a source might store only RNA genes, i.e., genes that are not translated into proteins. This property is not expressible in a query against S. We have two options:

- We extend S with a genetype attribute. This allows for pruning in user queries requesting genes of other types because we can conclude that R_W is not appropriate.

[18] Including an IN operator would be easy. One could either directly extend the algorithm for query planning or translate such a QCA in one QCA for each element of the IN set.

- We ignore the problem. This is not harmful if we assume that the mediator schema does not mind the difference between RNA and non-RNA genes. Since no `genetype` attribute exists in `S`, a user cannot specify a condition on it in a query. No wrong data occurs, since all tuples from R_W are correct for R_M.

b) $\texttt{intension}(R_M) \subseteq \texttt{intension}(R_W)$. Our mediator addresses only human maps. consider a source `X` having maps of all mammals *without* storing the species. This problem cannot be resolved by extending the schema of the mediator – it requires a condition in the wrapper query, not in the mediator query. But we cannot change the schema and content of a data source. Therefore, the problem unavoidably results in wrong data. `M` cannot decide upon the species of maps from `X`. There are two solutions: either the R_W is ignored, or the intension (not only and not necessarily the schema) of the mediator is extended.

c) $\texttt{intension}(R_M) \cap \texttt{intension}(R_W) \neq \varnothing$ and neither is contained in the other. This is a combination of (a) and (b). All problems discussed in (b) apply.

Context propagation.
Our discussion about relations with overlapping but not identical intensions has hidden a tricky problem that is not immediately visible but can complicate the specification of QCAs. We devote a separate section to it.

Recall case 1. from the list above. We expressed the difference between the intension of the two relations by means of a condition in either the mediator query or the wrapper query. However, the requirement that the difference must be expressible by a query does not imply *where* the discerning information is present in the schema. The difference in the intension between `T.map` and `S.map` does not only apply to maps, but carries over to clones and transcripts – actually, it affects the entire schema. r_1 is therefore wrong since it potentially retrieves transcripts of all mammals and not only of humans. The mediator is not able to filter out erroneous tuples obtained through r_1. Therefore, a correct QCA needs to join clones and transcripts with `map` in the wrapper query:

```
r₇: gene(gid,gn,gd) ← W.v₅(gid,gn,gd) ← map(mid,-,-,sp),
    YAC(cid,-,-,mid,-),contains(cid,gid),transcript(gid,gn,gd,-),sp='human';
r₈: gene(gid,gn,gd) ← W.v₆(gid,gn,gd) ← map(mid,-,-,sp),
    PAC(cid,-,-,mid,-),contains(cid,gid),transcript(gid,gn,gd,-),sp='human';
```

For every relation of `T`, one must include a join to `T.map` in the wrapper query to ensure the correctness of the QCAs. The same situation occurs the other way round in case (1.a) for mediator queries. It is a problem of *context* [GMS94; KS98]: QCAs have to ensure that the same context is used in both queries. We required that this context is expressible by a query, but we cannot restrict the length of such a query.

Although this is a problem, we notice the following:

- The problem does not occur in all cases of overlapping intensions. For instance, the intensional difference in the `maptype` attributes does not propagate to clones or transcripts. The difference only affects `map`, and not the entire schema.
- The problem only affects the specification of QCAs. It does not affect query planning. As shown in Section 5.4, the mediator will remove possible redundancies in plans that follow from QCAs that need large queries to ensure the right context.

6.2.3 Structural Heterogeneity

In contrast to other authors, we consider only a relatively small group of conflicts as structural in nature. For instance, Kim et al. have a much broader understanding of structural conflicts [KCGS95], including synonyms and homonyms in relation names.

We assume that two classes are semantically identical if they have the same name. However, different schemas may represent classes with identical intension differently. We consider the following cases:

- Missing or additional attributes in relations.
- Different attribute types and units.
- Different attribute positions.
- Decomposed attribute values (horizontal aggregation).

Missing or additional attributes in relations.
Two relations intending the same concept can represent this concept with different sets of attributes. Imagine `S.map` and `T.map` would both address physical maps of mammals. Still, `S.map` lacks the `species` attribute, while `T.map` lacks `maplength` and `chromosome`.

Handling such conflicts in QCAs is straight-forward using projections.

Different Attribute units.
Attributes having the same intention can have different types, such as real versus integer, or can express their value using different units, such as different currencies, or can have different numerical precision. In the example, the length of clones is once stored in kilobases and once in megabases. Such conflicts require a attribute value transformation as possible in enhanced QCAs (see Definition (D4.2)).

However, it is not always possible to find transformation functions. Imagine one attribute a_1 storing grades in a range between 1-15 and another attribute a_2 representing grades in six levels from ‘good’ to ‘bad’. While a transformation from a_1 to a_2 is feasible, no unambiguous transformation from a_2 to a_1 exists.

A QCA for `T.YAC` including the transformation is:

r_9: `clone(cid,cn,ct,cl),ct=’YAC’ ← W.v`$_7$`(cid,cn,cl) ← YAC(cid,cn,l,-,-),`
`cl=l*1000;`

Type conversions, for instance from string to integer, may also achieved through transformation functions.

Different attribute positions.
Attributes storing the same information do not necessarily appear at the same position in different schemas. Different positions within the same relation are trivial and handled by using the same variable. The situation is more complex if attributes are placed in different relations.

Imagine a relational schema generated by translation from an entity-relationship model with cardinality constraints. A 1:N relationship between two entities in the entity-relationship model will usually be mapped into a foreign-key constraint between two relations. A N:M relationship will be mapped into a separate bridge table. In both cases, the scope of the entities can be identical, but the structural representation in the relational schema is not.

Another source of such conflicts are different *degrees of normalisation* [EL94]. This case is especially important if data sources are addressed that are not based on the relational data model. For instance, wrappers for web based information systems often export their data through a highly de-normalised relations because a web page is not formatted according to its logical structure, but to human readability (see Section 6.1.3).

In our example, `T` does not have a `sequence` relation but directly incorporates sequences in the `transcript` relation. A QCA can bridge this as follows:

```
r10: gene(gid,gn,gd),genesequence(gid,sid,-),sequence(sid,bp) ←
   W.v8(gid,gn,gd,bp) ← transcript(gid,gn,gd,bp);
```

The attribute for which the variable `bp` stands for in the mediator query of $\texttt{r}_{10}$ has the same meaning as the attribute for which the variable `bp` stands for in the wrapper query. The join with `gene` is important since there is no counterpart for `bp` in `T` for sequences of clones.

The relationship between maps and clones is similar. We need two QCAs, one for `PAC` and one for `YAC` (see Section 6.2.4 for a further analysis of why we need two QCAs):

```
r11: map(mid,mn,mt,-,-),clonelocation(mid,cid,po),clone(cid,cn,ct,cl),
   mt='Physical',ct='YAC' ← W.v9(mid,mn,cid,cn,cl,po) ← map(mid,mn,-,sp),
   YAC(cid,cn,le,mid,po),sp='Human', cl=le*1000;
r12: map(mid,mn,mt,-,-),clonelocation(mid,cid,po),clone(cid,cn,ct,cl),
   mt='Physical',ct='PAC' ← W.v10(mid,mn,cid,cn,cl,po) ← map(mid,mn,-,sp),
   PAC(cid,cn,le,mid,po),sp='Human', cl=le*1000;
```

Decomposed attribute values.
The information content of an attribute in one schema may correspond to the information content of multiple attributes in another schema. Consider two schemas for product sales, one storing net prices and the other storing gross prices and VAT separately. Or imagine two schemas storing the income of employees paid according to the projects they work in. One schema could only stores the total income of each person while the other might have a separate value for each project (see Table 6).

Schema 1 stores total income per person	person$_1$(name,age,totalIncome)
Schema 2 has separate values for each projects. The total income is the sum over the income in each project.	person$_2$(name,age,p$_1$,p$_2$,p$_3$,p$_4$,p$_5$,p$_6$)

Table 6: Two schemas with decomposed attribute values.

If schema 1 were the mediator schema, we may specify the relationship as follows:

```
r13: person1(na,ag,ti) ← X.v(na,ag,ti) ← person2(na,ag,p1,p2,p3,p4,p5,p6),
   ti=p1+p2+p3+p4+p5+p6;
```

We need enhanced QCAs to express a relationship that entails horizontal aggregation of values. Note however that no QCA can be formulated if schema 2 were the mediator schema, since we cannot deduce project-specific incomes from the total income. There is no bijective transformation function.

6.2.4 Schematic Heterogeneity

A schematic conflict is present if the model element, i.e., relation, attribute, and value, that is used to represent a concept is different in two schemas. Such conflicts cannot be bridged through relational views [KLK91], although their occurrence is common-place [Mil98]. Consequently, most multidatabase query languages (see Section 3.1.2) allow variables that range not only over tuples as in SQL, but also over database names, relation names and attribute names. The concrete values are read at run time from a data dictionary. Lakshmanan et al. call such languages “syntactically higher order, but semantically first order” [LSS93] since they, on a first view, range over predicates; but they can be reduced to first-order queries, respectively relational queries, since they always address a given and finite schema.

We use the three schemas given in Table 7 as examples for the discussion of schematic heterogeneity. All three schemas store data about the research areas of faculty members.

Schemas:	Instances:

S_1: faculty(name,research_area);	faculty(Smith,DBIS);
	faculty(Smith,LP);
	faculty(Kim,RDBS);
	faculty(Kim,LP);
	faculty(Wayne,SE);

S_2: research(area,smith,kim,wayne);	research(DBIS,yes,no,no);
	research(LP,yes,yes,no);
	research(RDBS,no,yes,no);
	research(dbis,no,no,yes);

S_3: smith(research_area);
kim(research_area);
wayne(research_area);
	smith(DBIS);
	smith(LP);
	kim(RDBS);
	kim(LP);
	wayne(SE);

Table 7: Three schemas that all store the same information but are schematically conflicting.

In the three schemas, the concept “faculty member” (represented by its last name) is modelled with different elements of the relational model:

- in S_1 it is the value of the attribute `name`,
- in S_2 it is the name of the attributes `smith`, `kim` and `wayne`, and
- in S_3 it is the name of the relations `smith`, `kim` and `wayne`.

Imagine that all relations of Table 7 exist in a local database. Furthermore, suppose we want to define a view `interest` that merges the research areas of all faculty members in any of the three representations. In SQL, this requires the union of seven subqueries In DATALOG, it requires the following rules:

```
interest(n,r) ← faculty(n,r);
interest(n,r) ← research(r,s,-,-),s=’yes’,n=’Smith’;
interest(n,r) ← research(r,-,s,-),s=’yes’,n=’Kim’;
interest(n,r) ← research(r,-,-,s),s=’yes’,n=’Wayne’;
interest(n,r) ← smith(r),n=’Smith’;
interest(n,r) ← kim(r),n=’Kim’;
interest(n,r) ← wayne(r),n=’Wayne’;
```

The problem is that the query is *data-dependent* with respect to the schemas of S_2 and S_3: the query must change if the data in the database content changes, not only if the schema changes. In a heterogeneous environment with independently created schemas this occurs frequently, since for all real-life schemas there exists queries that are data-dependent[19].

Between any two schemas `S` and `T` there exist six different schematic conflicts:

- `S` models a concept as relation, `T` as attribute name.
- `S` models a concept as relation, `T` as attribute value.
- `S` models a concept as attribute name, `T` as relation

[19] The “canonical schema” [Vassalos, 1997 #474] consisting out of the two relations `attribute(rel_name,att_name)` and `tuple(rel_name,tup_id,att_name,value)` has no data-dependent queries.

- S models a concept as attribute name, T as attribute value.
- S models a concept as attribute value, T as relation.
- S models a concept as attribute value, T as attribute name.

We describe each of these six possibilities through QCAs by taking at a time one of the three schemas S_1 - S_3 as S and the other two as wrapper schemas.

1. S=S_1:

   a) Integrating S_2:

   faculty(n,r),n='Smith' ← $W_2.v_1$(r) ← research(r,s,-,-) s='yes';
   faculty(n,r),n='Kim' ← $W_2.v_2$(r) ← research(r,-,s,-) s='yes';
   faculty(n,r),n='Wayne' ← $W_2.v_3$(r) ← research(r,-,-,s), s='yes';

   We need as many QCAs as there are tuples in faculty that have a corresponding value in S_2. Furthermore, we must know all the tuples in advance.

   b) Integrating S_3:

   faculty(n,r),n='Smith' ← $W_3.v_1$(r) ← smith(r);
   faculty(n,r),n='Kim' ← $W_3.v_2$(r) ← kim(r);
   faculty(n,r),n='Wayne' ← $W_3.v_3$(r) ← wayne(r);

   As in the previous case, we need to know the tuples of W_3 in advance. Note that in neither of the two cases we have to add QCAs if only the *values* in the sources change; adding a new faculty requires a schema change in both W_2 and W_3.

2. S=S_2:

   a) Integrating S_1:

   research(r,s,-,-), s='yes' ← $W_1.v_1$(r) ← faculty(n,r), n='Smith';
   research(r,-,s,-), s='yes' ← $W_1.v_2$(r) ← faculty(n,r), n='Kim';
   research(r,-,-,s), s='yes' ← $W_1.v_3$(r) ← faculty(n,r), n='Wayne';

   The situation here is reverse to case 1a). All problems mentioned there apply here as well.

   b) Integrating S_3:

   research(r,s,-,-), s='yes' ← $W_3.v_1$(r) ← smith(r);
   research(r,-,s,-), s='yes' ← $W_3.v_2$(r) ← kim(r);
   research(r,-,-,s), s='yes' ← $W_3.v_3$(r) ← wayne(r);

   The same problem as for 2a) occur.

3. S=S_3:

   a) Integrating S_1:

   smith(r) ← $W_1.v_1$(r) ← faculty(n,r), n='Smith';
   kim(r) ← $W_1.v_2$(r) ← faculty(n,r), n='Kim';
   wayne(r) ← $W_1.v_3$(r) ← faculty(n,r), n='Wayne';

   Here, we need as many QCAs as there are relations in the *mediator schema*. We do not depend on the number of tuples in a source. If we assume that the mediator designer had good reasons to break up its schema into one relation per person, then this situation would probably not be considered as a problem. Anyway, one can imagine more comfortable ways to express it, especially if the number of different persons is high.

   b) Integrating S_2:

   smith(r) ← $W_2.v_1$(r) ← research(r,s,-,-),s='yes';
   kim(r) ← $W_2.v_2$(r) ← research(r,-,s,-),s='yes';
   wayne(r) ← $W_2.v_3$(r) ← research(r,-,-,s),s='yes';

   The same thoughts hold as in case 3a).

Using QCAs in the presence of schematic conflicts can be cumbersome. First, we get large sets of almost identical QCAs. Second, in some cases we can specify QCAs only if we know the extension of a wrapper in advance. These problems can be partly removed by introducing *schema variables* into QCAs, i.e., variables in the place of relations or attribute names. Schema variables are used in many multidatabase query languages [KLK91; LSS96].

Even if appropriate QCAs are specified, user queries may still be complicated. Suppose the mediator has schema S_2 and a user asking for the names and research areas of all faculty members. This request cannot be formulated in a single query, but requires three separate queries – the request is data-dependent:

```
interest('Smith',r) ← research(r,s,-,-), s='yes';
interest('Kim',r) ← research(r,-,s,-), s='yes';
interest('Wayne',r) ← research(r,-,-,s), s='yes';
```

6.3 MBIS in the Presence of Change

A MBIS connects existing and independent subsystems. As such, MBIS have to cope with evolution in those subsystems: Data sources may undergo schema revisions, new sources have to be added, requirements to the MBIS itself may change, etc. MBIS are especially prone to changes due to the high autonomy of the integrated data sources [Les98b]. For instance, WBIS are often integrated into MBIS without being notified. Consequently, those sources do respect the requirements of the MBIS. If the WBIS changes, the MBIS is not notified.

For these reasons, MBIS must pay special attention to their ability to cope with continuous change [KS99] during their deployment phase. In Section 3.2 we discussed two development strategies for FIS, bottom-up and top-down, and argued that top-down developed systems are more flexible and better support maintenance. In the following we underpin this chaim by discussing different scenarios of change.

We consider data sources as completely autonomous. Furthermore, we assume that structural changes are not handled inside a wrapper. Note that in other contexts, wrappers are actually introduced to shield "upper" layers from changes in the underlying system. For instance, the main idea of CORBA standards is to keep the interface stable even if the underlying information system experiences heavy restructuring. In contrast, we assume thin wrappers (see Section 6.1), which implies that wrappers reflect, and not compensate, most changes in the underlying sources. The only components of the MBIS we can influence are the mediator and the set of QCAs.

We distinguish three classes of change wrt. the reaction they provoke (see Figure 36):

- Changes that can *only* be counteracted through changes in the mediator schema. We call such changes *schema affecting*. Schema affecting changes are highly undesirable. Stable mediator schemas are important to (1) protect applications (or other mediators) that use those schemas, and to (2) prevent the necessity to change QCAs describing different wrappers.
- Changes that can be counteracted by adding, deleting or modifying QCAs. Changing QCAs is inexpensive compared to changing schemas. It is the general idea of declarative specification methods, such as QCAs, to move as much knowledge as possible into rules since it is easier to change rules than to change programs.
- Changes that can be ignored, for instance because they can easily be treated inside the wrapper. An example is a layout change in a WBIS.

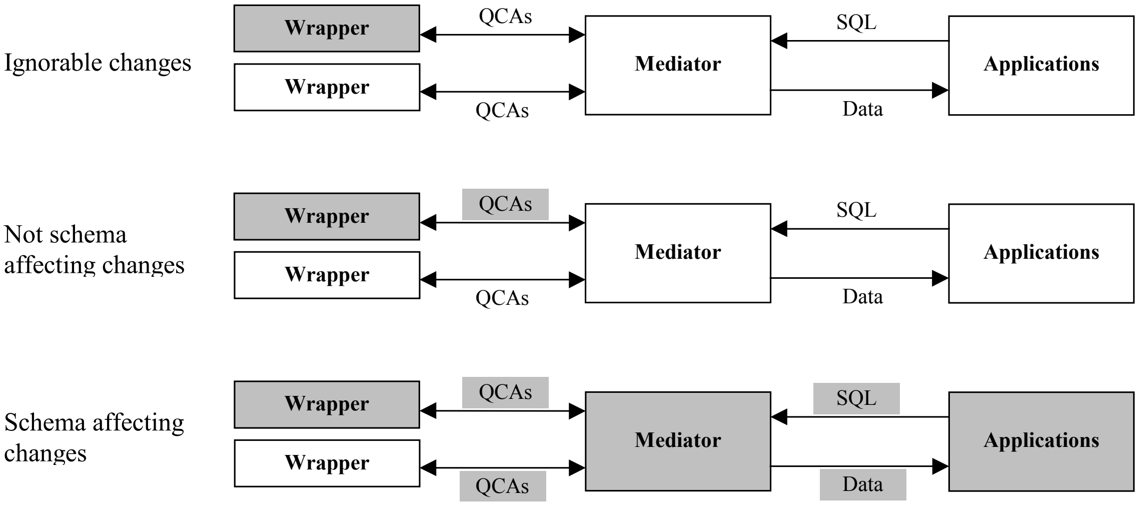


Figure 36. Three classes of change in MBIS.
Shaded areas are undergoing change.

Note that the ability to distinguish between the first two classes of change discerns top-down from bottom-up approaches: Since bottom-up approaches define the scope of the mediator schema as the "union" of the scopes of its data sources, virtually all changes in sources are schema affecting.

Changing wrapper schemas.
Structural or semantic changes in an underlying data source provokes changes in the export schema of a thin wrapper. For instance, schema evolution in a RDB leads to changes in the wrapper schema. If the data source is a WBIS, changes in the site structure or the set of available forms will have the same effect.

Possible changes in an export schema are:

- *Deleting or adding attributes or relations*. Adding or deleting export schema elements with an intension that is not present in the mediator schema is not schema affecting. Adding or deleting schema elements that have a semantic counterpart in the mediator schema can be offset by changing or adding/deleting QCAs. A special case is the deletion of a schema element that stands for a concept that is not present in any other wrapper schema. If the mediator administrator chooses to only delete the appropriate rule, any user queries involving this concept becomes unanswerable. If the concept is also present in another wrapper, only the extension of the query changes. To avoid unanswerable queries, the developer might chose to also modify the mediator schema, making the initial change schema affecting.
- *Altering the intension of schema element*. Altering the intension of a schema element may be considered as first deleting that element and then adding an element with the new intension. This type of change is hence subsumed by the previous discussion.
- *Altering attribute types*. Changes in the types of attributes is rarely schema affecting because usually type transformations can be included in enhanced QCAs. An example for a change that could become schema affecting is the following: Consider a mediator attribute storing school grades in the range 1-10 and an equivalent wrapper attribute. If the wrapper changes to a range of 1-6, no "fair" mapping can be specified. The mediator administrator can either change the mediator schema or use an "unfair" mapping.

Many cases can be handled without changing the mediator schema. Nevertheless, a mediator administrator is free to change a mediator schema deliberately if, for instance, important new and previously unavailable data is becoming available through a change in a data source.

Adding wrappers.

MBIS are systems that fulfil a domain-specific information requirement. Since in many domains new data providers are constantly appearing, MBIS should be able to integrate them rapidly in order to maintain a comprehensive view on its domain.

Integrating new data sources requires the implementation of an appropriate wrapper and the specification of its export schema. As discussed in Section 6.1, this can mean everything from simply loading a new schema from the data dictionary of a RDBMS to implementing complicated rules to parse flat-files or web pages. In some cases existing wrapper functionality can be reused.

Once the export schema is fixed, we can essentially deal with every element of that schema as if it were a new schema element of an existing wrapper. Since a QCA is a correspondence between the mediator and one single wrapper, introducing a new wrapper does not affect QCAs describing other wrappers.

If we consider the mediator schema as fixed – because it completely covers the information need it serves – then new sources will always only add new extensions – more data – but not new intension. Adding a new data source then only requires adding new QCAs. It is not schema affecting.

Deleting wrappers.

In the same way as new sources appear, existing sources may cease to exist. Another reason for the removal of a wrapper from a MBIS can be the appearance of a "better" data source – one that stores a superset of the data that the old source stores. For instance, in the area of molecular biology there are numerous *integrated databases* that completely contain the content of other data sources. Integrating the new source renders other sources superfluous. Also think of mirrors: A MBIS will probably only integrate one of a set of mirror sites – usually the nearest one[20].

Deleting wrappers is semantically equivalent to deleting all its schema elements. In most cases, deleting a wrapper simply requires to remove all QCAs describing that wrapper. Deleting QCAs is not schema affecting.

Unavailability of sources.

Sources may become temporarily unavailable due to network failures or server crashes. For the time that a source is absent, query plans involving queries against this source cannot be executed. All other query plans remain unaffected.

Unavailability of source will first be detected by its wrapper(s) who will notify the mediator. The mediator could react by flagging the appropriate QCAs as unusable for query planning, or simply by ignoring the problem. The former method has the advantage that query planning is accelerated, but also requires additional functionality in the implementation of the mediator.

Temporal unavailability of a wrapper is not schema affecting.

Changing requirements.

Not only data sources can change during the lifetime of a MBIS. Also the requirements to the MBIS can change, which will be reflected in modifications of the mediator schema.

[20] It might be reasonable to integrate more then one of a set of mirrored sources to flexibly react on system and network failures.

Such a change will naturally be schema affecting, and has the same impact on the MBIS as a database schema evolution on applications using that database. Formerly missing concepts must be covered through QCAs, which triggers the search for and addition of new data sources. Removed concepts can lead to the removal of QCAs, and to the removal of now unnecessary data sources. Essentially, the entire set of QCAs needs to be revised.

6.4 Summary and Related Work

In this section we focussed on three important problems during the lifetime of a MBIS: The creation of wrappers for new data sources, the semantic description of wrapper schemas wrt. the mediator schema in the presence of heterogeneity, and the effects of change on a running MBIS.

We discussed each of these problems by identifying a set of typical scenarios and describing their treatment. For the creation of wrappers we considered the three most important types of data sources: relational databases, CORBA based information systems and web based information systems. We consider those types as representatives for many more types of data sources, because

- many of the ideas for RDBMS carry over to other types of database systems, such as object-oriented or hierarchical databases,
- techniques for the integration of CBIS also apply to other types of object-oriented middleware such as DCOM,
- the integration of flat-files or file collections can use exactly the methods we described for WBIS.

One result of this analysis was the emphasis of the tight coupling between the export schema of a wrapper and its set of executable queries. Based on this coupling we showed that in some cases it is sufficient to only consider export schemas for the characterisation of wrappers.

In Section 6.2 we described the power of QCAs in bridging heterogeneity. Therefore, we classified types of heterogeneity in relational schemas and gave examples for the three important classes: semantic, structural and schematic conflicts. We showed that QCAs are indeed a powerful method to describe the correspondences between schemas carrying semantic and structural conflicts.

Finally, we investigated the ability of MBIS based on QCAs to react on five different types of change that a real-life MBIS will presumably face all to often. We saw that most changes in data sources can be counteracted by only changing QCAs, leaving the mediator schema unaffected.

Related work.

Wrapping RDB.
Wrapping relational databases is common technology supported by numerous products. A comparison of three different database middleware products can be found in [FRH98]. De Ferreira & Hergula therein consider criteria such as performance, transaction management, and language independence. The relationship between MBIS and database middleware is discussed by Mattos et al. [MKTZ99]. They argue that database middleware technology is currently moving towards MBIS by integrating more and more of the necessary functionality.

This is certainly true, although no product has yet tried to attack the problem of query rewriting, which is necessary for the provision of true schema transparency.

In contrast to Mattos et al., Leyman argues that the emerging technology for federated databases, and hence also for MBIS, is *message-orientation* [Ley99], not database middleware. In his view, a message oriented federation of databases is not based on the mediation of queries, but uses a *publish-subscribe schema*: Any database joining the federation has to subscribe to updates of other members, and also provide the possibility to notify other members from internal updates. Thus, copies of selected information are distributed and automatically kept consistent using a *message broker* with persistent message queues as middleware. Note that the problem of heterogeneity is not solved by the broker. A broker merely works in a store-and-forward mode, while coping with heterogeneity requires to translate and interpret messages.

Wrapping CBIS.

As for RDB, there are many publications discussing the construction of CORBA servers. Two that especially focus on the construction of wrappers for databases are [Bak96] and [DDO98]. Different possibilities to represent a relational database in an IDL interface are discussed in detail in [LTB98]. In [JL99], Jungfer & Leser present a semi-automatic method for the generation of CORBA wrappers based on a declarative mapping language between relational schemas and IDL.

The integration of CBIS into MBIS is especially interesting because of the existence and expected proliferation of CORBA domain standards. If such standards were commonly used, the treatment of heterogeneity would actually be moved to the data providers, making the construction of data integration systems much simpler [ML99]. Similar movements are underway in other middleware technologies, such as STEP and EXPRESS [HST99].

Wrapping WBIS.

Renown projects that aim at facilitating the construction of web wrappers are the “World Wide Web Wrapper Factory” (W4F, [SA99]) and JEDI ([HFAN98]). Both are based on the definition of a powerful language. In contrast to WWScript, they do not export structured data but *semistructured data*. They can therefore not be easily used in a structured MBIS. Furthermore, both are developed completely independent of any mediator systems, leading to deficiencies in the formulation of a clean interface. Several projects in this field, including JEDI and W4F, are compared in detail in [Hol99].

Maintenance of MBIS.

Maintaining MBIS has not been considered much in the the literature, probably because most projects are only short–termed and, to the best of our knowledge, until to date no commercially used system has emerged.

Maintaining integrated schemas in the presence of evolution of source schemas is considered in [Kol99]. The results indicate that it is rarely possible to keep the integrated schema stable, which underlines our claim that independent schemas, and hence top-down developed MBIS, are more flexible than tightly-coupled schemas, and hence FDBS.

7. DISCUSSION

We give a summary of the main contributions of this thesis in Section 7.1. Future research directions, both including possible improvements on the methods presented in this work and directions in data integration in general, are described in Section 7.2. Finally, Section 7.3 draws conclusions gained from the research reported in this work.

7.1 Summary

In this work we presented a solution to the problem of answering queries in tightly integrated, structured federated information systems.

In Chapter 2 we paved the technical ground by introducing the relational data model and conjunctive, relational queries. We presented the fundamental concept of query planning, i.e., *query containment*, and discussed algorithms for query containment. The main findings of this chapter are the *detailed complexity analysis* for these algorithms. In particular, we gave the first *average case analysis* for query containment, proving that the problem, although exponential in nature, is efficiently solvable in the average case.

In Chapter 3 we characterised *mediator based information systems* and compared them with other approaches to data integration. MBIS provide access to a collection of heterogeneous information systems through a central, homogeneous schema. They have two key components: *Mediators* manage the central schema and are responsible for answering queries against this schema. Data sources are encapsulated by *wrappers*, which shield the mediator from technical and data model heterogeneity. The main task of the mediator is the translation of queries formulated against one schema (the mediator schema) into queries against other schemas (the wrapper schemas).

The main problem in query translation is the heterogeneity between different schemas. The essential idea to the treatment of this heterogeneity is the encoding of knowledge about conflicts in rules that connect *corresponding queries*. In Chapter 4 we introduced *query correspondence assertions* as a powerful language for such rules. We also defined the semantics of a user query in MBIS that are based on QCAs.

The main contribution of Chapter 4 are QCAs, which are more powerful than previous approaches to the specification of schema correspondences. Furthermore, this is the first work to give a *satisfying, declarative semantics* for the type of correspondences we are considering. This semantics allows us to prove properties of query answering algorithms in Chapter 5.

Algorithms that find all and only correct answers to a user query were presented in Chapter 5. The algorithms are based on the generation of *query plans*. A query plan is a combination of executable queries against wrappers. After defining a correctness and a minimality criterion on query plans, we formally proved that (a) there exists only a finite number of correct and minimal query plans for each user query, and that (b) algorithms finding all correct and minimal query plans are sound and complete wrt. the previously defined semantics. We then described and analysed two concrete query planning algorithms: the *generate & test algorithm* (GTA) and the *improved bucket algorithm* (IBA). Furthermore, we considered possible optimisations to the set of all correct plans by investigating different forms of *redundancy* in and in between query plans.

Chapter 5 contains the main technical contributions of this thesis. We showed that the IBA has *considerable better complexity* than previously published algorithms for the types of queries we consider. Furthermore, redundancy in query plans for information integration was to our best knowledge not analysed adequately before.

In Chapter 6 we focussed on methodological issues. We discussed three steps in the life cycle of a MBIS. First, we discussed the *construction of wrappers* for different types of data sources. Wrappers make data sources accessible through an export schema and a set of executable queries. To integrate a wrapper into a MBIS, those queries must be described wrt. the mediator schema through QCAs. QCAs therein have to *bridge schema heterogeneity*, and we showed the power of QCAs by examples that handle structural, semantic and schematic conflicts. Finally, we investigated the flexibility of a MBIS wrt. the ability to cope with change. Analysing five different change scenarios, we demonstrated that MBIS using QCAs can handle many types of change by changing rules, while schemas and applications remain unaffected. This is achieved through the high degree of independence between schemas in MBIS, which in turn is only possibly through the high expressiveness of QCAs.

7.2 Future Research Directions

The method and algorithms presented in this thesis form a powerful basis for MBIS. However, many problems that are important for the success of MBIS have not been addressed. We shortly highlight some of those issues as potential starting points for future research.

Tool support.

The design and implementation of MBIS should be supported by tools, such as QCA editors. Furthermore, it is important to facilitate the finding of QCAs through automatically derived suggestions, for instance by using a domain-specific thesaurus for the detection of correspondences between schema elements. First steps towards this goal are described in [Koe99].

Another important tool should test the *sufficiency* of a set of QCAs for a given mediator schema. The query planning algorithms only work successfully if appropriate QCAs are available. If, for instance, a user query contains a relation that is not present in any QCA, then no plan can be found. Avoiding such frustrating failures requires to test whether the set of QCAs at hand can provide answers to any query against the mediator schema. This test is not trivial, given the many ways how mediator queries can be incompatible to each other.

Query capabilities.

The expressiveness of QCAs regarding the modelling of query capabilities of sources can be improved. For instance, our current query planner cannot decide whether or not a selection can be pushed to a wrapper because this knowledge is simply not expressible in QCAs. Furthermore, binding patterns were frequently mentioned in this work but not considered in the planning algorithms.

However, we are faithful that such extensions do not pose conceptual problems. Various approaches have been published [VP97; CGM99] and could be incorporated into the IBA.

Design rules for QCAs.
In many cases, especially if the data source is a RDB, the mediator developer has considerable freedom in the design of QCAs. In general, QCAs should be as tight as possible to avoid unnecessary query plans. For instance, if it can be observed that a data source has a restricted range of values for a certain attribute, this range should be incorporated into all mediator queries using this attribute. Queries selecting values outside this range will then avoid this data source.

An interesting question is whether mediator queries of QCAs should be kept small, i.e., contain only one or a few literals, or should be generally large. The promise of large mediator queries is that they build large subplans in one step. However, this promise does not hold since the planning algorithm will break up every mediator query into single literals anyway. Therefore, short mediator queries are probably preferable. However, a detailed analysis is still lacking.

Result presentation.
We have not discussed issues of the presentation of results. For instance, users often want to be source-aware, i.e., want to know the source of the information they receive [LBM98]. The biggest problem however is probably the semantic integration of results from different query plans because it requires the identification of real-life objects in the absence of globally valid keys. This infamous problem, called "*object fusion*" [PAG96] or "*record linkage*" [Nei99], is extremely complex and has even been called "the breakdown of the information model in multi-database systems" [Ken91]. Identity is a application dependent concept: For instance, a gene present in different organisms will be considered as identical for an application trying to predict its protein structure, but different for an application analysing gene expression profiles.

Approaches to the solution of the object identification problem either use statistical methods or *identity functions* for the construction of global keys out of other object properties.

Mediators that use mediators.
For simplicity, we restricted ourselves in this work to MBIS with a single mediator. However, MBIS integrating data from a complex domain are certainly better manageable if they use specialised mediators, each responsible for a different sub-domain.

Such networks of mediators raise interesting challenges. For instance, we cannot simply consider a subordinated MBIS as a RDB, being capable of answering any query against its schema, since some queries might have no plan. Furthermore, the same information could be reported to a superordinate mediator through two different subordinate mediators accessing the same source. Removing such artificial duplicates is one problem; avoiding the redundant execution of the same query in different mediators is another.

Yernerni et al. report an interesting step into this direction [YGMU99]. They describe algorithms that automatically derive the query capabilities of a mediator given the capabilities of the wrappers it uses. These algorithms can equally be applied to mediators that use other mediators.

7.3 Conclusions

The focus of research in information integration is currently changing. Previous approaches concentrated on the integration of a given set of well-structured databases with the purpose of having an integrated access to exactly those databases. In contrast, information integration in the Internet age is about providing a certain type of information to a user, independently of which information sources are used. Examples of the new type of integration services are companies that sell information integrated from autonomous web sites, interfaces that provide researcher with experimental results produced and managed in hundreds of laboratories, and bargain finders that harvest hundreds of web sites to find the cheapest offer for a certain good.

Common properties of these scenarios are:

- Integration is provided by a third party. A service for the location of cheap flight tickets on the Internet will not be offered by a ticket selling agents.
- The task of integration is to satisfy a source-independent information requirement. A customer of the ticket finder service does not care which original data sources are searched; he or she is interested in a cheap ticket.
- The data sources remain completely autonomous and evolve independently of the integration. Ticket issuing agents typically do not ally with bargain finders.

Despite the growing importance of this new wave in information integration, few successful solutions are known that are not ad-hoc, hard-coded ‘hacks’. We believe that this has two reasons: First, information integration is difficult. The main source of difficulty is heterogeneity and independent evolution, which both are consequences of autonomy. Second, virtual information integration is prone to bad performance. It is inherently inefficient compared to homogeneous, monolithic systems because it involves the execution of remote methods or queries. Therefore, virtual integration is almost defenceless to bandwidth limitations.

This thesis contributes solutions to the challenges posed to information integration by unsatisfying performance, heterogeneity, and change. We presented a formalism for the semantic description of data sources that bridges heterogeneity. We described an algorithm that efficiently operates on this formalism to answer queries. We showed how query answering can be achieved with a minimal set of remote query executions. Finally, we demonstrated that this approach allows for stable interfaces in a world of continuously changing data sources.

Although we carefully tailored our algorithms towards high performance, virtual integration of distributed data sources remains time-consuming. Compared to a central database, virtual integration suffers from two problems: First, query planning is more complex. Second, execution of remote queries is more costly than accesses to local disks.

However, our results are not only helpful for those types of systems. First, query planning need not necessarily be carried out at query-time. Frequent user queries can be precompiled, which renders the cost of query planning almost irrelevant. Precompilation may also include a manual post-optimisation of a computed set of query plans using knowledge that cannot be specified in a declarative fashion. The final set of query plans is stored and can be executed immediately if the appropriate query is issued.

Second, using declarative schema correspondences and flexible query planning algorithms also pays off if data is stored locally. For instance, one might periodically replicate heterogeneous data sources on local disks. Integrating the data in a homogeneous database might be prohibitive because rebuilds are extremely costly but continuously necessary because of frequently changing data and schemas. In such a case, we may use QCAs and query planning to provide integrated access without building a central database – and without remote query executions.

REFERENCES

[AD98] Abiteboul, S. and O. M. Duschka (1998). Complexity of Answering Queries using Materialized Views. *17th ACM Symposium on Principles of Database Systems*, pp. 254-263, Seattle, WA.

[AHK+95] Arens, Y., R. Hull, R. King, et. al. (1995). Reference Architecture for the Intelligent Integration of Information; Version 2.0 (DRAFT). DARPA - Defence Advanced Research project Agency; Program on Intelligent Integration of Information (I3), Report.

[AHK96] Arens, Y., C.-N. Hsu and C. A. Knoblock (1996). Query Processing in the SIMS Information Mediator. In *Advanced Planning Technology*. A. Tate, pp. 61-69, AAAI Press, Menlo Park, California.

[AHV95] Abiteboul, S., R. Hull and V. Vianu (1995). Foundations of Databases. Addison-Wesley Publishing Company, Reading, Massachusetts.

[AMM97] Atzeni, P., G. Mecca and P. Merialdo (1997). Semistructured and Structured Data in the Web: going back and forth. *SIGMOD Record,* 26(4): 16-23.

[AR94] Alsabbagh, J. R. and V. V. Raghavan (1994). Analysis of Common Subexpression Exploitation Models in Multiple-Query Processing. *10th Int. Conference on Data Engineering*, pp. 488-497, Houston, Texas.

[ASU79a] Aho, A. V., Y. Sagiv and J. D. Ullman (1979). Efficient Optimisation of a Class of Relational Expressions. *ACM Transactions on Database Systems,* 4(4): 435-454.

[ASU79b] Aho, A. V., Y. Sagiv and J. D. Ullman (1979). Equivalence among Relational Expressions. *SIAM Journal of Computing,* 8(2): 218-246.

[Bak96] Baker, S. (1996). CORBA and Databases - Do you really need both ? *Object Expert,* May 1996.

[BBE98] Bouguettaya, A., B. Benatallah and A. K. Elmagarmid (1998). Interconnecting Heterogeneous Information Systems. Kluwer Academic Publishers, Boston, Dordrecht, London.

[BDD+98] Bello, R. G., K. Dia, A. Downing, J. Feenen Jr, W. D. Norcott, H. Sun, A. Witkowski and M. Ziauddin (1998). Materialized Views in ORACLE. *24th Conference on Very Large Database Systems*, pp. 659-664, New York.

[BE95] Bukhres, O. and A. K. Elmagarmid, Eds. (1995). Object-Oriented Multidatabase Systems: A Solution for Advanced Applications. Prentice Hall.

[BKLW99] Busse, S., R.-D. Kutsche, U. Leser and H. Weber (1999). Federated Information Systems: Concepts, Terminology and Architectures. Technische Universität Berlin, Forschungsberichte des Fachbereichs Informatik 99-9.

[BLL+99] Barillot, E., U. Leser, P. Lijnzaad, C. Cussat-Blanc, K. Jungfer, F. Guyon, G. Vaysseix, C. Helgesen and P. Rodriguez-Tome (1999). A Proposal for a Standard CORBA Interface for Genome Maps. *Bioinformatics,* 15(2): 157-169.

[BLN86] Batini, C., M. Lenzerini and S. B. Navathe (1986). A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys,* 18(4): 323-364.

[Bor95] Borgida, A. (1995). Description Logic in Data Management. *IEEE Transactions on Knowledge and Data Engineering,* 7(5): 671-682.

[BP98] Blakeley, J. A. and M. J. Pizzo (1998). Microsoft Universal Data Access Platform. *ACM SIGMOD Int. Conference on Management of Data 1998*, pp. 502-503, Seattle, Washington.

[BS95] Brodie, M. L. and M. Stonebraker (1995). Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach. Morgan Kaufmann Publishers, Inc., San Francisco.

[Bun97] Buneman, P. (1997). Semistructured Data. *16th ACM Symposium on Principles of Database Systems*, pp. 117-121, Tuscon, Arizona.

[Bus99] Busse, S. (1999). A Specification Language for Model Correspondence Assertions. Technische Universität Berlin, Forschungsberichte des Fachbereichs Informatik 99-8.

[CD98] Chen, A. F.-C. F. and A. M. H. Dunham (1998). Common Subexpression Processing in Multiple-Query Processing. *IEEE Transactions on Knowledge and Data Engineering,* 10(3): 493-499.

[CGM99] Chang, C.-C. K. and H. Garcia-Molina (1999). Mind your Vocabulary: Query Mapping across Heterogeneous Information Sources. *ACM SIGMOD Int. Conference on Management of Data 1999*, pp. 335-346, Philadelphia.

[Cha98] Chaudhuri, S. (1998). An Overview of Query Optimisation in Relational Systems. *17th ACM Symposium on Principles of Database Systems*, pp. 34-43, Seattle, Washington.

[CHKR98] Carey, M. J., L. M. Haas, J. Kleewein and B. Reinwald (1998). Data Access Interoperability in the IBM Database Family. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Interoperability,* 21(3): 4-11.

[CKPS95] Chaudhuri, S., R. Krishnamurthy, S. Potamianos and K. Shim (1995). Optimizing Queries with Materialized Views. *11th Int. Conference on Data Engineering*, pp. 190-200, Los Alamitos, CA.

[CL93] Catarci, T. and M. Lenzerini (1993). Representing and Using Interschema Knowledge in Cooperative Information Systems. *Journal for Intelligent and Cooperative Information Systems,* 2(4): 375-399.

[CM77] Chandra, A. K. and P. M. Merlin (1977). Optimal Implementation of Conjunctive Queries in Relational Databases. *9th ACM Symposium on Theory of Computing*, pp. 77-90.

[CNS99] Cohen, S., W. Nutt and A. Serebrenik (1999). Rewriting Aggregate Queries using Views. *18th ACM Symposium on Principles of Database Systems*, pp. 155-166, Philadelphia.

[Con97] Conrad, S. (1997). Föderierte Datenbanksysteme: Konzepte der Datenintegration. Springer Verlag, Berlin, Heidelberg, New York.

[CPL97] Cadoli, M., L. Palopoli and M. Lenzerini (1997). Datalog and Description Logics: Expressive Power. *6th Workshop on Database Programming Languages*, pp. 281-298, Estes Park, Colorado.

[CR97] Chekuri, C. and A. Rajaraman (1997). Conjunctive Query Containment Revisited. *6th Int. Conference on Database Theory; LNCS 1186*, pp. 56-70, Delphi, Greece.

[CV92] Chaudhuri, S. and M. Y. Vardi (1992). On the Equivalence of Datalog Programs. *11th ACM Symposium on Principles of Database Systems*, pp. 55-66, San Diego, CA.

[DBBV00] Discala, C., X. Benigni, E. Barillot and G. Vaysseix (2000). DBcat: a catalog of 500 biological databases. *Nucleic Acids Research,* 28(1): 8-9.

[DD99] Domenig, R. and K. R. Dittrich (1999). An Overview and Classification of Mediated Query Systems. *SIGMOD Record,* 28(3).

[DDJ+98] De Michelis, G., E. Dubois, M. Jarke, F. Matthes, J. Mylopoulos, M. P. Papazoglou, K. Pohl, J. Schmidt, C. Woo and E. Yu (1998). Cooperative Information Systems: a Manifesto. In *Cooperative Information Systems*. M. P. Papazoglou and G. Schlageter, pp. 315-363, Academic Press, San Diego.

[DDO98] Dogac, A., C. Dengi and M. T. Özsu (1998). Distributed Object Management Platforms. *Communications of the ACM,* 41(9): 95-103.

[DFJ+96] Dar, S., M. Franklin, B. Jonsson, D. Srivastava and M. Tan (1996). Semantic Data Caching and Replacement. *22nd Conference on Very Large Databases*, pp. 330-341, Bombay, India.

[DG97] Duschka, O. M. and M. R. Genesereth (1997). Answering recursive queries using views. *16th ACM Symposium on Principles of Database Systems*, pp. 109-116, Tuscon, Arizona.

[DK97] Davidson, S. and A. S. Kosky (1997). WOL: A Language for Database Transformations and Constraints. *13th Int. Conference on Data Engineering*, pp. 55-65, Birmingham, UK.

[DKE94] Davidson, S., A. S. Kosky and B. Eckman (1994). Facilitating Transformations in a Human Genome Project Database. *3rd International Conference on Information and Knowledge Management*, pp. 423-432, Gaithersburg, Maryland.

[DL97] Duschka, O. M. and A. Y. Levy (1997). Recursive Plans for Information Gathering. *15th International Joint Conference on Artificial Intelligence*, pp. 778-784, Nagoya, Japan.

[DOTW97] Davidson, S., G. C. Overton, V. Tannen and L. Wong (1997). BioKleisli: a digital library for biomedical researchers. *Int. Journal on Digital Libraries,* 1(1): 36-53.

[EL94] Elmasri, R. and S. B. Navathe (1994). Fundamentals of Database Systems. Benjamin / Cummings Publishing Company Inc., Redwood City, CA.

[EP90] Elmagarmid, A. K. and C. Pu (1990). Special Issue on Heterogeneous Databases (editors). *ACM Computing Surveys,* 22(2).

[Fau00] Faulstich, L. (2000). The HyperView Approach to the Integration of Semistructured Data. Freie Universität Berlin. Ph.D. Thesis.

[FGL+98] Fankhauser, P., G. Gardarin, M. Lopez, J. Munoz and A. Tomasic (1998). Experiences in Federated Databases: From IRO-DB to MIRO-Web. *24th Conference on Very Large Database Systems*, pp. 655-658, New York.

[FLL+97] Fasman, K. H., S. Letovsky, P. W. D. Li, R. W. Cottingham and D. Kingsbury (1997). The GDB Human Genome Database Anno 1997. *Nucleic Acids Research,* 25(1): 72-80.

[FLMS99] Florescu, D., A. Y. Levy, I. Manolescu and D. Sucia (1999). Query Optimisation in the Presence of Limited Access Patterns. *ACM SIGMOD Int. Conference on Management of Data 1999*, pp. 311-322, Philadelphia, USA.

[Fre91] Frenkel, K. A. (1991). The Human Genome Project and Informatics. *Communications of the ACM,* 34(11): 40-51.

[FRH98] de Ferreira Rezende, F. and K. Hergula (1998). The Heterogeneity Problem and Middleware Technology: Experiences with and Performance of Database Gateways. *24th Conference on Very Large Database Systems*, pp. 146-157, New York.

[FS98] Faulstich, L. and M. Spiliopoulou (1998). Building HyperNavigation Wrappers for Publisher Web-Sides. *2nd European Conf. on Digital Libraries; LNCS 1513*, pp. 115-134, Heraklion, Krete.

[FTU98] Farre, C., E. Teniente and T. Urpi (1998). Query Containment Checking as a View Updating Problem. *9th Int. Conf. on Database and Expert Systems Applications*, pp. 310-321, Vienna, Austria.

[FW97] Friedman, M. and D. S. Weld (1997). Efficiently Executing Information-Gathering Plans. *15th International Joint Conference on Artificial Intelligence*, pp. 785-791, Nagoya, Japan.

[GG95] Guarino, N. and P. Giaretta (1995). Ontologies and Knowledge Bases: Towards a Terminological Clarification. In *Towards Very Large Knowledge Bases*. N. J. I. Mars, pp. 25-32, IOS Press, Amsterdam.

[GKD97] Genesereth, M. R., A. M. Keller and O. M. Duschka (1997). Infomaster: An Information Integration System. *ACM SIGMOD Int. Conference on Management of Data 1997*, pp. 539-542, Tuscon, Arizona.

[GM95] Gupta, A. and I. S. Mumick (1995). Maintenance of Materialized Views: Problems, Techniques and Applications. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing,* 18(2): 3-18.

[GM99] Grahne, G. and A. O. Mendelzon (1999). Tableau Techniques for Querying Information Sources through Global Schemas. *7th Int. Conference on Database Theory*, pp. 332-347, Jerusalem, Israel.

[GMP+97] Garcia-Molina, H., Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. D. Ullman, V. Vassalos and J. Widom (1997). The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems,* 8(2): 117-132.

[GMS94] Goh, C. H., M. E. Madnick and M. D. Siegel (1994). Context Interchange: Overcoming the Challenges of Large-Scale Interoperable Database Systems in Dynamic Environments. *3rd International Conference on Information and Knowledge Management*, pp. 337-346, Gaithersburg, Maryland.

[GMY99] Garcia-Molina, H. and R. Yerneni (1999). Coping with Limited Capabilities of Sources. *8th GI Fachtagung: Datenbanksysteme in Büro, Technik und Wissenschaft*, pp. 1-19, Freiburg, Germany.

[Gru93] Gruber, T. R. (1993). A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition,* 5(2): 199-220.

[Gry98] Gryz, J. (1998). Query Folding with Inclusion Dependencies. *14th Int. Conference on Data Engineering*, pp. 126-133, Orlando, Florida.

[HFAN98] Huck, G., P. Fankhauser, K. Aberer and E. Neuhold (1998). JEDI: Extracting and Synthesizing Information from the Web. *6th Int. Conf. on Cooperative Information Systems*, pp. 32-43, New York.

[HGM95] Hammer, J., H. Garcia-Molina, J. Widom, W. Labio and Y. Zhuge (1995). The Stanford Data Warehousing Project. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing,* 18(2): 41-48.

[HKU99] Haas, L. M., D. Kossmann and I. Ursu (1999). Loading a Cache with Query Results. *25th Conference on Very Large Database Systems*, pp. 351-362, Edinburgh, UK.

[HKWY97] Haas, L. M., D. Kossmann, E. L. Wimmers and J. Yang (1997). Optimizing Queries across Diverse Data Sources. *23rd Conference on Very Large Database Systems*, pp. 276-285, Athens, Greece.

[HL93] Habermann, H.-J. and F. Leymann (1993). Repository. R. Oldenbourg, München, Wien.

[HM85] Heimbigner, D. and D. McLeod (1985). A Federated Architecture for Information Management. *ACM Transactions on Office Information Systems,* 3(3): 253-278.

[Hol99] Holzheuer, C. (1999). Wrappergenerierung für WWW Datenquellen. Technische Universität Berlin. Diploma Thesis.

[HST99] Härder, T., G. Sauter and J. Thomas (1999). The intrinsic problems of structural heterogeneity and an approach to their solution. *The VLDB Journal,* 8(1): 25-43.

[Hull97] Hull, R. (1997). Managing Semantic Heterogeneity in Databases: A Theoretical Perspective. *16th ACM Symposium on Principles of Database Systems*, pp. 51-61, Tuscon, Arizona.

[ISO90] ISO and IEC (1990). Information Technology - Information Resource Dictionary System (IRDS). International Standard, ISO/IEC 10027.

[Jar85] Jarke, M. (1985). Common Subexpression Isolation in Multiple Query Optimization. In *Query Processing in Database Systems*. W. Kim, D. S. Reiner and D. S. Batory, pp. 191-205, Springer Verlag, Berlin, Heidelberg, New York, Tokyo.

[JL99] Jungfer, K., U. Leser and P. Rodriguez-Tome (1999). Constructing IDL Views on Relational Databases. *11th Conference on Advanced Information Systems Engineering; LNCS 1626*, pp. 255-269, Heidelberg, Germany.

[Joh98] Johansson, J. M. (1998). Towards a More Accurate Network Response Time Model for Distributed Systems Design. *8th Workshop on Information Technology and Systems*, pp. 177-186, Helsinki, Finland.

[JP99] Jeusfeld, M. A. and M. P. Papazoglou (1999). Information Brokering. In *Information System Interoperability*. B. Krämer, M. P. Papazoglou and H.-W. Schmidt, pp. 265-302, John Wiley, New York.

[Karp94] Karp, P. D. (1994). Report of the Workshop on Interconnection of Molecular Biology Databases. SRI International Artificial Intelligence Center, Stanford, California. Technical Report, SRI-AIC-549.

[Karp95c] Karp, P. D., Ed. (1995). 2nd Meeting on Interconnection of Molecular Biology Databases. Electronic Proceedings, available at http://www.ai.sri.com/people/pkarp/mimdb.html, Cambridge, UK.

[KB94] Keller, A. M. and J. Basu (1994). A Predicate-Based Caching Scheme for Client-Server Database Architectures. *The VLDB Journal,* 5(1): 35 - 47.

[KB98] Klusch, M. and W. Benn (1998). Intelligente Informationsagenten im Internet. *Künstliche Intelligenz,* 3/98: 8-17.

[KCGS95] Kim, W., I. Choi, S. Gala and M. Scheevel (1995). On Resolving Schematic Heterogeneity in Multidatabase Systems. In *Modern Database Systems*. W. Kim, pp. 521-550, ACM Press, Addison-Wesley Publishing Company, New York.

[Ken91] Kent, W. (1991). The Breakdown of the Information Model in Multi-Database Systems. *SIGMOD Record,* 20(4): 10-15.

[Kim95] Kim, W., Ed. (1995). Modern Database Systems. The Object Model, Interoperability and Beyond. ACM Press, Addison Wesley Publishing Company, New York.

[Kin99] Kinne, O. (1999). Multiple Query Optimisation in verteilten, heterogenen Informationssystemen. Technische Universität Berlin. Diploma Thesis.

[KLK91] Krishnamurthy, R., W. Litwin and W. Kent (1991). Language Features for Interoperability of Databases with Schematic Discrepancies. *ACM SIGMOD Int. Conference on Management of Data 1991*, pp. 40-49, Denver, Colorado.

[Klu88] Klug, A. (1988). On Conjunctive Queries Containing Inequalities. *Journal of the ACM,* 35(1): 146-160.

[Koe99] König-Ries, B. (1999). Ein Verfahren zur Semi-Automatischen Generierung von Mediatorspezifikationen. INFIX Verlag.

[Kol99] Kolmschlag, S. (1999). Schemaevolution in föderierten Datenbanksystemen. Shaker Verlag, Aachen. Ph.D. Thesis.

[KPS99] Krämer, B., M. P. Papazoglou and H.-W. Schmidt, Eds. (1999). Information System Interoperability. John Wiley, New York.

[KS96] Kashyap, V. and A. Sheth (1996). Semantic and Schematic Similarities between Database Objects: A Context-Based Approach. *The VLDB Journal,* 5(4): 276-304.

[KS98] Kashyap, V. and A. Sheth (1998). Semantic Heterogeneity in Global Information Systems: The Role of Metadata, Context and Ontologies. In *Cooperative Information Systems*. M. P. Papazoglou and G. Schlageter, pp. 139-178, Academic Press, San Diego.

[KS99] Kutsche, R.-D. and A. Sünbül (1999). A Meta-Data Based Development Strategy for Heterogeneous, Distributed Information Systems. *3rd IEEE Metadata Conference*, Bethesda, Maryland.

[KTV97] Kapitskaja, O., A. Tomasic and P. Valduriez (1997). Dealing with Discrepancies in Wrapper Functionality. INRIA: Institute National de Recherche en Informatique et en Automatique. Technical Report, 3138.

[KW96] Kwok, C. T. and D. S. Weld (1996). Planning to Gather Information. University of Washington, Department of Computer Science & Engineering. Technical Report, UW-CSE-96-01-04.

[LBM98] Lee, T., S. Bressan and S. Madnick (1998). Source Attribution for Querying Against Semistructured Documents. *1st Workshop on Web Information and Data Management, in conjunction with CIKM'98*, Washington, D.C.

[Les98a] Leser, U. (1998). Combining Heterogeneous Data Sources through Query Correspondence Assertions. *Workshop on Web Information and Data Management, in conjunction with CIKM'98*, pp. 29-32, Washington, D.C.

[Les98b] Leser, U. (1998). Maintenance and Mediation in Federated Databases. *8th Workshop on Information Technology and Systems*, pp. 187-196, Helsinki, Finland.

[Ley99] Leymann, F. (1999). A Practitioners Approach to Database Federation. *4. Workshop Föderierte Datenbanken*, Berlin, Germany.

[LLRC98] Leser, U., H. Lehrach and H. Roest Crollius (1998). Issues in Developing Integrated Genomic Databases and Application to the Human X Chromosome. *Bioinformatics,* 14(7): 583-590.

[LMR90] Litwin, W., L. Mark and N. Roussolpoulos (1990). Interoperability of Multiple Autonomous Databases. *ACM Computing Survey,* 22(3): 267-293.

[LMSS95] Levy, A. Y., A. O. Mendelzon, Y. Sagiv and D. Srivastava (1995). Answering Queries Using Views. *14th ACM Symposium on Principles of Database Systems*, pp. 95-104, San Jose, CA.

[LR96] Levy, A. Y. and M.-C. Rousset (1996). CARIN: A Representation Language Combining Horn Rules and Description Logics. *12th European Conference on Artificial Intelligence*, pp. 323-327, Budapest, Hungary.

[LRO96a] Levy, A. Y., A. Rajaraman and J. J. Ordille (1996). Querying Heterogeneous Information Sources Using Source Descriptions. *22nd Conference on Very Large Databases*, pp. 251-262, Bombay, India.

[LRO96b] Levy, A. Y., A. Rajaraman and J. J. Ordille (1996). Query-Answering Algorithms for Information Agents. *13th AAAI National Conf. on Artificial Intelligence*, pp. 40-47, Portland, Oregon.

[LRTL93] Lee, A. J., E. A. Rundensteiner, S. W. Thomas and S. Lafortune (1993). An Information Model for Genome Map Representation and Assembly. *2nd International Conference on Information and Knowledge Management*, pp. 75-84, New York.

[LRU96] Levy, A. Y., A. Rajaraman and J. D. Ullman (1996). Answering Queries Using Limited External Processors. *15th ACM Symposium on Principles of Database Systems*, pp. 227-237, Montreal, Canada.

[LS93] Levy, A. Y. and Y. Sagiv (1993). Queries Independent of Updates. *19th Conference on Very Large Databases*, pp. 171-181, Dublin, Ireland.

[LS97] Levy, A. Y. and D. Suciu (1997). Deciding Containment for Queries with Complex Objects and Aggregations. *16th ACM Symposium on Principles of Database Systems*, pp. 20-31, Tuscon, Arizona.

[LSR97] LSRDTF (1997). Mission, Working Groups and Documents of the Life Science Research Domain Task Force. WWW Page, http://www.omg.org/lsr.

[LSS93] Lakshmanan, L. V. S., F. Sadri and I. N. Subramanian (1993). On the Logical Foundation of Schema Integration and Evolution in Heterogeneous Database Systems. *2nd Int. Conf. on Deductive and Object-Oriented Databases*, pp. 81-100, Phoenix, Arizona.

[LSS96] Lakshmanan, L. V. S., F. Sadri and I. N. Subramanian (1996). SchemaSQL: A Language for Interoperability in Relational Multidatabase Systems. *22nd Conference on Very Large Databases*, pp. 239-250, Bombay, India.

[LTB98] Leser, U., S. Tai and S. Busse (1998). Design Issues of Database Access in a CORBA Environment. *Workshop on Integration of Heterogeneous Software Systems*, pp. 74-87, Magdeburg, Germany.

[LWG+98] Leser, U., R. Wagner, A. Grigoriev, H. Lehrach and H. Roest Crollius (1998). IXDB, an X Chromosome Integrated Database. *Nucleic Acids Research,* 26(1): 108-111.

[MG98] Meltzer, B. and R. J. Glushko (1998). XML and Electronic Commerce: Enabling the Network Economy. *SIGMOD Record,* 27(4): 21-24.

[Mic98] Microsoft Cooperation (1998). OLE DB/ADO: Making Universal Data Access a Reality. White Paper.

[Mil98] Miller, R. J. (1998). Using Schematically Heterogeneous Structures. *ACM SIGMOD Int. Conference on Management of Data 1998*, pp. 189-200, Seattle, Washington.

[MKSI96] Mena, E., V. Kashyap, A. Sheth and A. Illarramendi (1996). OBSERVER: An Approach for Query Processing in Global Information Systems based on Interoperation across Preexisting Ontologies. *4th Int. Conf. on Cooperative Information Systems*, pp. 14-25, Bruessels, Belgium.

[MKTZ99] Mattos, N. M., J. Kleewein, M. Tork Roth and K. Zeidenstein (1999). From Object-Relational to Federated Databases. *8th GI Fachtagung: Datenbanksysteme in Büro, Technik und Wissenschaft*, pp. 185-209, Freiburg, Germany.

[ML99] Müller, H. and U. Leser (1999). Integration durch Standards: Erfahrungen mit CORBA in Life Science Research. *4. Workshop Föderierte Datenbanken*, pp. 89-102, Berlin, Germany.

[Mot95] Motro, A. (1995). Multiplex: A Formal Model for Multidatabases and Its Implementation. George Mason University. Technical report, ISSE-TR-95-103.

[Mot98] Motz, R. (1998). Propagation of Structural Modifications to an Integrated Schema. *2nd East European Symposium on Advances in Databases and Information Systems; LNCS 1475*, pp. 163-174, Poznan, Poland.

[Muel99] Müller, H. (1999). Realisierung eines einheitlichen Zugriffs auf molekularbiologische Genomkarten unter Verwendung von CORBA. Technische Universität Berlin. Diplomo Thesis.

[Nei99] Neiling, M. (1999). Datenintegration durch Objekt-Identifikation. *4. Workshop Föderierte Datenbanken*, pp. 117-143, Berlin, Germany.

[NEL86] Navathe, S. B., R. ElMasri and J. A. Larson (1986). Integrating User Views in Database Design. *IEEE Computer,* 9(1): 50 - 62.

[NLF99b] Naumann, F., U. Leser and J. C. Freytag (1999). Quality-driven Integration of Heterogeneous Information Systems. *25th Conference on Very Large Database Systems*, pp. 447-458, Edinburgh, UK.

[NS96] Navathe, S. B. and A. Savasere (1996). A Schema Integration Facility using a Object-Oriented Data Model. In *Object-Oriented Multidatabase Systems - A Solution for Advanced Applications*. O. Bukhres and A. K. Elmagarmid, pp. 105-128, Prentice Hall, Eaglewoods Cliffs.

[OHE97] Orfali, R., D. Harkey and J. Edwards (1997). Instant CORBA. Wiley Computer Publishing, John Wiley and Sons Inc.

[OV99] Oezsu, M. T. and P. Valduriez (1999). Principles of Distributed Database Systems. Prentice Hall, Inc., New Jersey.

[PAG96] Papakonstantinou, Y., S. Abiteboul and H. Garcia-Molina (1996). Object Fusion in Mediator Systems. *22nd Conference on Very Large Data Bases*, pp. 413-424, Bombay, India.

[Pap94] Papadimitriou, C. H. (1994). Computational Complexity. Addison-Wesley, USA.

[PBE95] Pitoura, E., O. Bukresh and A. K. Elmagarmid (1995). Object Orientation in Multidatabase Systems. *ACM Computing Survey,* 27(2): 141-195.

[PGMU96] Papakonstantinou, Y., H. Garcia-Molina and J. D. Ullman (1996). Medmaker: A Mediation System Based on Declarative Specifications. *12th Int. Conference on Data Engineering*, pp. 132 - 141, New Orleans, Louisiana.

[PK98] Patterson, D. A. and K. K. Keeton (1998). Hardware Technology Trends and Database Opportunities. *ACM SIGMOD Int. Conference on Management of Data 1998*, Seattle, Washington.

[Pri96] Primrose, S. B. (1996). Genomanalyse. Spectrum; Akademischer Verlag, Heidelberg.

[PS98] Papazoglou, M. P. and G. Schlageter, Eds. (1998). Cooperative Information Systems - Trends and Directions. Academic Press, San Diego.

[Qia96] Qian, X. (1996). Query Folding. *12th Int. Conference on Data Engineering*, pp. 48-55, New Orleans, Louisiana.

[Rob92] Robbins, R. J. (1992). Challenges in the Human Genome Project: Progress Hinges on Resolving Database and Computational Factors. *IEEE Engineering in Medicine and Biology,* March 1992: 25-34.

[Rob95] Robbins, R. J. (1995). Information Infrastructure for the Human Genome Project. *IEEE Engineering in Medicine and Biology,* 14(6): 746-759.

[RSU95] Rajaraman, A., Y. Sagiv and J. D. Ullman (1995). Answering Queries using Templates with Binding Patterns. *14th ACM Symposium on Principles of Database Systems*, pp. 105-112, San Jose, CA.

[RSUV89] Ramakrishnan, R., Y. Sagiv, J. D. Ullman and M. Y. Vardi (1989). Proof-Tree Transformation Theorems and their Applications. *8th ACM Symposium on Principles of Database Systems*, pp. 172 - 181, Philadelphia.

[RU93] Ramakrishnan, R. and J. D. Ullman (1993). A Survey of Research on Deductive Database Systems. *Journal of Logic Programming,* 23(2): 125 - 149.

[SA99] Sahuguet, A. and F. Azavant (1999). Building Light-Weight Wrappers for Legacy Web Data-Sources using W4F. *25th Conference on Very Large Database Systems*, pp. 738-741, Edingurgh, UK.

[SAC+79] Selinger, P. G., M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price (1979). Access Path Selection in a Relational Database Management System. *ACM SIGMOD Int. Conference on Management of Data 1979*, pp. 23-34, Boston, MA.

[Sag88] Sagiv, Y. (1988). Optimising DATALOG Programs. In *Foundations of Deductive Databases and Logic Programming*. J. Minker, pp. 659-698, Morgan Kaufmann, Los Altos.

[Sau98] Sauter, G. (1998). Interoperabilität von Datenbanksystemen bei struktureller Heterogenität. Infix, Sankt Augustin.

[SCGS91] Saltor, F., M. Castellanos and M. Garca-Solaco (1991). Suitability of Data Models as Canonical Models for Federated Databases. *ACM SIGMOD Record,* 20(4): 44-48.

[Sch98] Schmitt, I. (1998). Schemaintegration für den Entwurf föderierter Datenbanken. Infix Verlag, Sankt Augustin.

[SG90] Sellis, T. K. and S. Ghosh (1990). On the Multiple-Query Optimization Problem. *IEEE Transactions on Knowledge and Data Engineering,* 2(2): 262-266.

[Shm93] Shmueli, O. (1993). Equivalence of DATALOG Queries is Undecidable. *Journal of Logic Programming,* 15: 231-241.

[SK93] Sheth, A. and V. Kashyap (1993). So Far (Schematically) Yet So Near (Semantically). *Proc. IFIP TC2.6 DS-5 Conference on Semantics of Interoperable Databases*, pp. 283-312, Lorne, Victoria, Australia.

[SL90] Sheth, A. and J. A. Larson (1990). Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases. *ACM Computing Survey,* 22(3): 183-236.

[SM98] Sellentin, J. and B. Mitschang (1998). Data-Intensive Intra- & Internet Applications - Experiences Using JAVA and CORBA in the World Wide Web. *14th Int. Conference on Data Engineering*, pp. 302-311, Orlando, Florida.

[SPD92] Spaccapietra, S., C. Parent and Y. Dupont (1992). Model Independent Assertions for Integration of Heterogeneous Schemas. *The VLDB Journal,* 1(1): 81-126.

[SSN94] Shim, K., T. K. Sellis and D. Nau (1994). Improvements on a Heuristic Algorithm for Multiple-Query Optimization. *Data and Knowledge Engineering,* 12(2): 197-222.

[ST96] Staudt, M. and K. v. Thadden (1996). A Generic Subsumption Testing Toolkit for Knowledge Base Queries. *7th Int. Conf. on Database and Expert Systems Applications; LNCS 1134*, pp. 834-844, Zurich, Switzerland.

[SY80] Sagiv, Y. and M. Yannakakis (1980). Equivalence among Relational Expressions with the Union and Difference Operators. *Journal of the ACM,* 27(4): 633-655.

[TRS97] Tork Roth, M. and P. M. Schwarz (1997). Don't scrap it, wrap it! A Wrapper Architecture for Legacy Data Sources. *23rd Conference on Very Large Database Systems*, pp. 266-275, Athens, Greece.

[TRV96] Tomasic, A., L. Raschid and P. Valduriez (1996). Scaling Heterogeneous Databases and the Design of DISCO. *16th Int. Conference on Distributed Computing Systems*, pp. 449-457, Hong Kong.

[TSI94] Tsatalos, O. G., M. H. Solomon and Y. E. Ioannidis (1994). The GMAP: A Versatile Tool for Physical Data Independence. *20th Conference on Very Large Databases*, pp. 367-378, Santiago de Chile, Chile.

[Ull89] Ullman, J. D. (1989). Principles of Database Systems and Knowledge-Based Systems. Volume II: The New Technologies. Computer Science Press, Rockville.

[Ull97] Ullman, J. D. (1997). Information Integration using Logical Views. *6th Int. Conference on Database Theory; LNCS 1186*, pp. 19-40, Delphi, Greece.

[vdM92] van der Meyden, R. (1992). The complexity of Querying Indefinite Data about Linearly Ordered domains. *11th ACM Symposium on Principles of Database Systems*, pp. 331-345, San Diego, CA.

[VJB+97] Visser, P. R. S., D. M. Jones, T. J. M. Bench-Capon and M. J. R. Shave (1997). An Analysis of Ontological Mismatches: Heterogeneity versus Interoperability. *AAAI 1997 Spring Symposium on Ontological Engineering*, Stanford, USA.

[VP97] Vassalos, V. and Y. Papakonstantinou (1997). Describing and Using Query Capabilities of Heterogeneous Sources. *23rd Conference on Very Large Database Systems*, pp. 256-265, Athens, Greece.

[Web82] Weber, H. (1982). On the Unambiguous Use of Names in Data Base Design. *2nd Int. Conf. on Data and Knowledge Bases*, pp. 358-403, Jerusalem, Israel.

[Wid95] Widom, J. (1995). Research Problems in Data Warehousing. *4th International Conference on Information and Knowledge Management*, pp. 25-30, Baltimore, Maryland.

[Wie92] Wiederhold, G. (1992). Mediators in the Architecture of Future Information Systems. *IEEE Computer,* 25(3): 38-49.

[Wie94] Wiederhold, G. (1994). Interoperation, Mediation and Ontologies. *Int. Symposium on 5th Generation Computer Systems; Workshop on Heterogeneous Cooperative Knowledge-Bases*, pp. 33-48, Tokyo, Japan.

[WT94] Wells, D. L. and C. W. Thompson (1994). Evaluation of the Object Query Service Submissions to the Object Management Group. *IEEE Quarterly Bulletin on Data Engineering,* 17(4): 36-45.

[YGMU99] Yerneni, R., H. Garcia-Molina and J. D. Ullman (1999). Computing Capabilities of Mediators. *ACM SIGMOD Int. Conference on Management of Data 1999*, pp. 443-545, Philadelphia, USA.

[YM98] Yu, C. and W. Meng (1998). Principles of Database Query Processing for Advanced Applications. Morgan Kaufmann.

[YOL97] Yan, L. L., M. T. Oezsu and L. Liu (1997). Accessing Heterogeneous Data Through Homogenization and Integration Mediators. *5th Int. Conf. on Cooperative Information Systems*, pp. 130-139, Kiawah Island, North Carolina.

[ZHK96] Zhou, G., R. Hull and R. King (1996). Generating Data Integration Mediators that Use Materialization. *Journal of Intelligent Information Systems,* 6(2/3): 199-222.

[ZO93] Zhang, X. and M. Z. Oezsoyoglu (1993). On Efficient Reasoning with Implication Constraints. *2nd Conf. on Deductive and Object-Oriented Databases*, pp. 236-252, Phoenix, Arizona.

APPENDIX

List of Figures

List of Definitions

List of Algorithms

List of Lemmas and Theorems

List of Abbreviations

BA	Bucket algorithm for query planning
BAC	Bacterial artificial chromosome, type of clone
BFA	Breadth-first algorithm for proving query containment
CGI	Common gateway interface
CBIS	CORBA based information system
CM	Containment mapping
CSL	Correspondence specification language
DFA	Depth-first algorithm for proving query containment
ECM	Extended containment mapping
FDBS	Federated database system
FIS	Federated information system
GaV	Global-as-View correspondence specification language
GDM	Generic data model
GTA	Generate & test algorithm for query planning
HGP	Human genome project
IBA	Improved bucket algorithm for query planning
IDL	Interface definition language
IRA	Inverted rules algorithm for query planning
IM	Information Manifold
IRDS	Information Resource Dictionary Systems
KB	Kilo bases = $1*10^3$ base pairs
LaV	Local-as-View correspondence specification language
MB	Mega bases = $1*10^6$ base pairs
MBIS	Mediator based information system
MQO	Multiple query optimiser
OMG	Object management group
PCM	Partial containment mapping
QCA	Query correspondence assertion
QFA	Query folding algorithm for query planning
RDB	Relational database
RDBMS	Relational database management system
WBIS	Web based information system
WSL	Wrapper specification language
YAC	Yeast artificial chromosome (type of a clone).

List of Symbols

Symbol	Explanation	First occurrence
var	Set of variable symbols.	13 ff
const	Set of constant symbols.	13 ff
$\texttt{rel}_E$	Set of relation symbols.	13 ff
att	Set of attribute symbols.	13 ff
Σ	Schema.	13
rel	Relation.	13
$\lvert\Sigma\rvert$	Size of a schema Σ.	13 ff
arity(rel)	Arity of a relation rel.	13 ff
$\texttt{I}^{\Sigma}$	Instance of a schema Σ.	13 ff
$\texttt{D} = \{\Sigma, \texttt{I}^{\Sigma}\}$	Database with schema Σ and instance $\texttt{I}^{\Sigma}$.	13 ff
l	Literal.	13 ff
$\texttt{I}^{\Sigma}\rvert_{\texttt{rel}}$	Extension of rel in $\texttt{D}=(\Sigma, \texttt{I}^{\Sigma})$.	14
$\texttt{rel}_Q$	Set of query symbols.	14
q	Query.	14
$\lvert q\rvert$	Size of q.	15 ff
head(q)	Head of q.	15 ff
body(q)	Body of q.	15 ff
export(q)	Exported variables of q.	15 ff
sym(q)	Symbols of q.	15 ff
variables(q)	Variables of q.	15 ff
constants(q)	Constants of q.	15 ff
$\texttt{CQ}_c^{\Sigma}$	Queries with complex conditions against schema Σ.	15
$\texttt{CQ}_s^{\Sigma}$	Queries with simple conditions against schema Σ.	15
cond(q)	Conditions of query q.	15 ff
cond(q,v)	All conditions involving only variable v in query q.	15 ff
v	Valuation function.	16
q(D)	Extension of query q in database D.	17
$q_1 \equiv q_2$	Query equivalence.	19
$q_1 \subseteq q_2$	Query containment; q_1 is contained in q_2.	19
h	Symbol mapping or containment mapping.	20
org(h)	Origin of mapping h.	20 ff
img(h)	Image of mapping h.	20 ff
$l_2 \geq l_1$	Literal l_2 covers literal l_1.	24
$h_1 \cup h_2$	Union of two reconcilable mappings.	26
$h_1 \sim h_2$	Containment mappings h_1 and h_2 are compatible.	26
$\texttt{W} = (\Sigma, \Omega, \chi)$	Wrapper W, with Σ: wrapper schema; Ω: set of query templates against Σ; χ: data source addressed by W.	53
$\texttt{M} = (\Sigma, \Psi, \Gamma)$	Mediator M, with Σ: mediator schema; Ψ: set of wrappers used by M; Γ: set of QCAs used by M.	56

$mq \leftarrow W.v(E) \leftarrow wq$	QCA describing wrapper query $v(E) \leftarrow wq$ from wrapper W through mediator query $v(E) \leftarrow mq$.	67
$export(r)$	Exported variables of QCA r.	67 ff
$origin(r)$	Wrapper of QCA r.	67 ff
$medq(r)$	Mediator query of QCA r.	67 ff
$wrapq(r)$	Wrapper query of QCA r.	67 ff
α	Variable renaming.	80
$\sigma = (\alpha, C)$	Query transformer σ with variable renaming α and set C of conditions.	80
$\pi = \{q_1, ..., q_n\}$	Plan candidate.	87
$p = (\pi, \alpha, C)$	Plan with plan candidate π and query transformer (α, C).	87
$\phi = (p, h)$	Query plan. p is correct and executable for user query, h is containment mapping from user query into p.	90
$\phi = (\pi, \alpha, C, h)$	2. notation for a query plan $\phi = (p, h)$ with $p = (\pi, \sigma), \sigma = (\alpha, C)$.	90
$\varepsilon = (h, \alpha, C)$	Extended containment mapping with mapping h and query transformer (α, C).	100
$\varphi = (q, \varepsilon, \pi)$	Partial plan for subquery q of u with ECM ε and plan candidate π.	112
$\varphi = (q, h, \alpha, C, \pi)$	2. notation for partial plan.	112