Object/Relational Query Optimization with Chase and Backchase

Lucian Popa

University of Pennsylvania,
Department of Computer and Information Science,
200 South 33rd Street, Philadelphia, PA 19104, USA
E-mail: lpopa@gradient.cis.upenn.edu

Dissertation

Advisor: Val Tannen
Abstract

Traditionally, query optimizers assume a direct mapping from the logical entities modeling the data (e.g., relations) and the physical entities storing the data (e.g., indexes), each physical entity corresponding precisely to one logical entity. This assumption is no longer true in non-traditional applications (object-oriented and semi-structured databases, data integration), which often exhibit a mismatch between the logical view and the actual storage of data. In addition, there is an increased amount of redundancy, even at the logical level, that can greatly enhance optimization opportunities, if exploited. To deal with all this, we propose a novel architecture for query optimization, in which physical optimization is leveraged at the level of query rewriting. As a consequence, the other important aspect of query optimization, semantic optimization (that takes advantage of the redundancy at the logical level), can be naturally incorporated. The optimizer can then make global decisions based on both semantic and physical knowledge, leading to plans of higher quality than those obtainable by a traditional two-level approach.

The main idea is to describe the relationship between physical and logical schemas by constraints, with the same syntactic form as the semantic constraints describing the logical schema. Many physical structures such as indexes, materialized views, access support relations, GMAPs, etc. can be captured in this way. The search space for query plans is then defined and enumerated in a novel way: First, the input query is rewritten by chase with constraints into a "universal" plan that integrates all the relevant physical and logical structures. In a second phase (backchase), minimal plans are produced by eliminating, exhaustively, the various combinations of redundancies from the universal plan.

We proved the completeness of the method for "path-conjunctive" queries, views and constraints. This class is expressive enough to handle complex objects and dictionaries (modeling OO classes and index-like structures). It has the same properties regarding containment, chase, constraint implication, rewriting with views, that hold for the conjunctive relational case. Therefore, it is a natural candidate for further theoretical and practical development of query optimization in complex environments.

We have implemented our method and examined how far we can push it in terms of complexity of schemas and queries. We employed our optimization framework in two main sets of experiments. In the first one, we measured the performance of the chase/backchase as a procedure for enumeration of minimal plans. No cost information is required in this case. Since the size of the universal plan can often become large, we developed "stratification" techniques that work by reducing the enumeration problem to several subproblems each with a smaller universal plan. This resembles the dynamic programming approach of traditional optimizers. The experimental results demonstrate that the method is practical, i.e feasible and worthwhile. In the second case, we combined the chase/backchase optimization with a cost-based pruning strategy, in order to avoid the enumeration of all minimal plans. The experimental results show a considerable improvement in performance over the first situation. The cost-based version of the chase/backchase optimizer is shown to be practical even when no stratification is possible.
# Contents

1 Introduction .................................................. 4  
1.1 Motivation .................................................. 4  
1.2 Examples of Desirable Optimizations ......................... 8  
1.3 Approach and Contributions .................................. 15  
1.4 Overview of the Dissertation ................................ 21  

2 Optimization with Chase and Backchase: The C&B Approach .... 23  
2.1 Chase: Checking Query Equivalence .......................... 24  
2.2 Backchase Minimization ...................................... 25  
2.3 Physical Structures as Constraints ............................ 28  
2.4 Chase: Discovery of Relevant Physical Sources ............... 31  
2.5 Examples of C&B Enumeration ................................ 31  

3 The Theory of Path-Conjunctive Chase .......................... 38  
3.1 The Path-Conjunctive Language ............................... 39  
3.2 A Canonical Instance Construction ........................... 42  
3.3 Trivial dependencies and query containment ................. 48  
3.3.1 Trivial EGDs ............................................ 48  
3.3.2 Homomorphisms of tableaux ............................. 49  
3.3.3 Trivial EPCDs and query containment .................... 52  
3.4 The PC Chase .............................................. 55  
3.5 Terminating Chase .......................................... 59  
3.6 Chase with Full EPCDs ..................................... 62
3.6.1 Chase with Full EPCDs: Termination ................................................. 63
3.6.2 Chase with Full EPCDs: Complexity Analysis. ................................. 64
3.6.3 Chase with Full EPCDs: Confluence and Semantic Invariance .......... 66
3.7 Non-Terminating Chase ................................................................. 69

4 A Completeness Result for the C&B Optimization ................................. 75
4.1 Preliminary Definitions. .................................................................. 75
4.2 Bounding Chase Theorem ................................................................ 76
4.3 Complete Subquery Enumeration ..................................................... 81

5 Feasibility of the C&B Enumeration ..................................................... 88
5.1 Feasibility of the Chase ................................................................. 89
5.2 Feasibility of the Backchase ......................................................... 89
  5.2.1 On-line Query Fragmentation (OQF) ........................................... 90
  5.2.2 Off-line Constraint Stratification (OCS) ...................................... 93
5.3 The Architecture of the Prototype .................................................. 95
5.4 Experiments .................................................................................. 96
  5.4.1 Experimental configurations ..................................................... 96
  5.4.2 Feasibility of the Chase: Experiments ........................................ 97
  5.4.3 Feasibility of the Backchase: Experiments ................................. 98
  5.4.4 The Benefit of Optimization ..................................................... 102

6 Mixing the Chase and Backchase with Cost-Based Optimization ........ 104
6.1 Physical Plans for Physical Queries ................................................ 106
  6.1.1 $PC^c$ Restriction .................................................................. 106
  6.1.2 Decomposition into Atomic Fragments ..................................... 107
  6.1.3 Physical Plans ........................................................................ 109
  6.1.4 Decomposition into Atomic Fragments: Formal Details .............. 116
6.2 A Cost Model for Nested Sets and Dictionaries ................................ 118
  6.2.1 Cardinality Information ......................................................... 119
  6.2.2 Estimating Cardinalities of Physical Plans. .............................. 120
6.2.3 Estimating Sizes ................................................................. 121
6.2.4 A Simple Model for Cost-Evaluation of Physical Plans. ............................. 122
6.3 Global Dynamic Programming .................................................. 125
6.4 Bottom-Up Backchase with Cost-Based Pruning .................................... 128
6.5 Experimental Results ............................................................... 131
6.6 Conclusion .............................................................................. 143

7 Conclusion and Future Work ................................................................ 145
  7.1 Related Work ........................................................................... 145
  7.2 Summary of Contributions ......................................................... 147
  7.3 Limitations of Our Approach ..................................................... 147
  7.4 Future Work Items .................................................................... 148
Chapter 1

Introduction

1.1 Motivation

Traditional query optimization. One of the main reasons for the success of commercial relational DBMSs is physical data independence. This allows a user to write a query in a high-level, declarative language such as SQL, on a logical view of the data (we call it logical schema), without worrying about how will the query be executed on the underlying physical database implementation. It is then the important role of the query optimizer to find the physical access paths that are relevant to the query and to find an optimal execution strategy by combining the access paths. The role of the optimizer in a database system becomes even more important when considering that the volume of data manipulated these days is very large and often distributed over a network, and a bad execution plan can be quite costly.

The typical flow of a query in an optimizer is shown in figure 1.1. In a first phase the logical query is normalized and transformed, if possible, into a simpler but equivalent query. This phase is usually done by rewriting the original query in a rule-based fashion [PHH92] and includes transformations such as unnesting of queries having nested subqueries in the from clause. The resulting query (still a declarative query) is then passed to the cost-based optimizer, responsible for finding a good physical plan, expressed as an operator tree, that specifies: physical access paths for each relation, join order, algorithms for join, etc. Conceptually we distinguish between finding what are the usable physical access paths and finding how to combine them into an operator tree, thus the separation of the cost-based optimizer into the two modules in the figure. However, in a traditional optimizer the two are performed simultaneously because finding what are the relevant physical access paths is rather trivial: for each relation that occurs in the logical query, we can scan the relation, scan an index for that relation, or lookup an index if there is an appropriate selection condition. We say that the physical schema is the set of all relations and all indexes available in the system. There are many possible equivalent physical plans combining the physical schema elements and the optimizer must find one that minimizes some cost function.

Thus, there are three main components of a cost-based optimizer: search space, cost model and search strategy. The search space is determined by the relevant physical schema elements, the physical algorithms available and the join orderings. Physical algorithms are usually chosen from a small set of algorithms known to perform well: merge-join, hash-join, index-join, etc. We are then left, as the main component of the search space, with the various join orderings. One can limit here the search space by considering only a subset of all orderings, for example, all left-linear trees as opposed to all bushy trees. However, the search space is usually at least exponential in the size of the original query.

The strategy used then to explore the space of alternative plans can be: exhaustive search combined with pruning
(for example, dynamic programming [SAC+79]), rule-based [GCD+94] in which a specific set of rules can guide the optimizer in choosing a plan, heuristic, randomized [IW87] etc. With the exception of the first strategy, the guarantee that the resulting plan is optimal is lost and the focus is on obtaining a good enough plan in a reasonable amount of time. For the first strategy, the cost model plays an essential role in pruning the search space by not considering partial plans that have higher costs than other equivalent partial plans previously explored. This is the main idea behind the dynamic programming algorithm.

**An extension: materialized views.** There are several extensions to this basic model of optimization that have been considered in the literature. Both can be viewed as additional modules that can work on top of the traditional cost-based optimizer. The first direction tries to make the data independence concept even more flexible, by allowing more complicated physical storage schemas in addition to the base relations and indexes: relational materialized views [LMS95, Lev, CKPS93], join indexes [Val87], access support relations for OO databases [KM90a, KM90b], sources with limited capabilities [RSU95, LR96, FLMS99], GMAPs [TS96] etc. In all these cases, the elements of the physical schema are defined as queries (views) over the logical schema. The advantage is that the logical schema is then fixed while the physical schema can be easily changed in order to achieve better storage, faster access, or simply in order to take advantage of previously answered queries. Finding what physical schema elements can be used to answer the input logical query becomes then non-trivial. The problem is the one of rewriting, many times in the same declarative language, from a query over the logical schema to one over the physical schema. The general architecture of the optimizer, shown in figure 1.2, includes then a separate module responsible to finding such rewritings. There can be many alternative rewritings (we call them physical queries or candidate plans), and each can then be passed to a cost-based module that finds the best physical plan. The global optimal plan is then the best among all these physical plans. Finding the rewritings and applying the cost-based optimization are combined, when possible, for efficiency reasons. Note that, in principle, the rewriting module takes over a part of the functionality of the traditional cost-based module: finding the relevant indexes (when indexes are expressible as views [TS96]).

**Another extension: semantic optimization.** The second main direction that tries to extend the basic model of optimization is one that uses semantic knowledge about the logical schema: key constraints, func-
Figure 1.2: Query optimizer architecture using materialized views.

tional dependencies, inclusion constraints such as foreign key constraints, inverse relationship constraints in OO schemas, etc. Semantic optimization [CGK+99, GGM97, LS95, CGM90, FRV96, CD92] means finding rewritings, on the same logical schema, that are equivalent under the existing constraints with the original query. Typical transformations include join elimination, join introduction, predicate elimination or predicate introduction. These rewritings are, as in the previous case, described as declarative queries, and the hope is that when passed to the cost-based optimizer the resulting plans are better than the one obtained by examining only the input query. The architecture of such system is shown in figure 1.3.

Figure 1.3: Semantic query optimizer architecture.

**Drawbacks; our motivation.** There are several issues and limitations regarding the two extensions mentioned above. First, the two techniques were considered mainly in isolation so far. It is often the case that the use of semantic constraints, even as simple as key constraints or foreign key constraints, enables the use of an index or of a materialized view. In the absence of such interaction between the two techniques, important opportunities for optimization can be missed. The work that tries best to take advantage of this interaction, in a context that
is more general than a relational one, is the work of [TS96]. Their GMAPs are physical access structures that can express index-like structures and OO classes and the rewriting algorithm for GMAPs takes advantage of integrity constraints. However, their use of such constraints was limited to that of inclusion constraints, they did not have any theoretical characterization of the search space (i.e. no completeness), and the language was limited to SPJ (select-project-join) queries in which only one occurrence of a given relation is allowed. Moreover by using SPJ queries or, for that matter, any relational language, one is not able to directly express the fast lookup capabilities of indexes.

Second, the search space for rewritings as opposed to the search space for physical plans in traditional optimization, is of a different nature and not fully understood so far. The transformations required to rewrite the input query are non-trivial and they often change radically the elements of the input query. A first step to define a search space for rewritings in the context of answering queries with materialized views was made in [LMSS95]. There it was proved than any "minimal" rewriting is bounded in size by the size of the original query, thus providing a complete procedure for enumerating minimal rewritings. However the language considered there, relational conjunctive queries, cannot express indexes, and the consequence is that optimality is lost: one can find an example in which the rewriting with the best plan is not a minimal one. A more important limitation is that it was not clear how their approach can be mixed with arbitrary semantic constraints. For semantic optimization a clean definition for a search space does not even exist, to the best of our knowledge. Most of the transformations used in semantic query optimization were rather ad-hoc, i.e. heuristics. We believe that having a clear characterization of a search space, that is amenable to mathematical reasoning, is of both theoretical and practical interest. On the more theoretical side, we are interested in a search space that is "complete", i.e. it contains the optimal solution. On the practical side, we have to know what is the search space (even if it is a subset of the complete one) in order to design efficient algorithms for exploring it.

A third and important issue that was an obstacle in the systematic implementation of such extensions was the high complexity involved, thus the danger of spending more time on optimization than on execution itself. However, we believe that the new, complex, applications that emerged with the growth of the Web need optimizations of higher complexity. In mediator-based data integration systems there is less knowledge about the physical capabilities of the sources (such as indexes of a relational database, and thus a complete cost-based physical optimization is not always possible) but more high-level semantic knowledge about the relationships between the sources. Being able to exploit such knowledge to discover alternative information pathways that can be used to answer a query can have dramatic improvements in execution time.

This dissertation's answers to the drawbacks. We study a new optimization framework that integrates in a coherent and uniform way the techniques used for physical data independence and semantic optimization. Our target data model and query language extend the relational ones by being able to deal with more complex data, in particular complex objects such as nested sets, OO classes and various kinds of index-like structures. We give a precise characterization of the search space for alternative rewritings in this integrated framework, that allows us to investigate properties such as completeness, and also allows us to design efficient methods for exploring the search space. The framework can then be used in two ways:

- as a complementary module to the traditional optimization, which can be plugged in an optimizer between the normalization and the cost-based modules, see figure 1.4(a). In this situation, our module performs an enumeration of alternative rewritings. At the end, each candidate plan must be evaluated by the cost-based optimizer, which chooses the best physical plan. In this context, we study several enumeration strategies (that are independent of cost) and we show experimentally that they are practically feasible.

- mixed with cost-based optimization. In this situation, see figure 1.4(b), the cost-based optimizer is used to evaluate the cost (and produce a physical plan, in the process) of each explored physical query. Based on this information, the search space can be pruned and a complete enumeration of candidate plans is avoided. The improvement over the performance of a complete enumeration is significant. The end result is a full-
fledged optimization framework, practical (feasible and worthwhile), integrating in a single module, four main components of optimization:

1. Semantic optimization
2. Use of traditional access structures (indexes)
3. Use of materialized views
4. Cost-based optimization

For both OO and relational

The systematic interaction between all of the above components allows the optimizer to generate plans of considerably higher quality than that of plans generated in previous approaches. The main goal of this thesis is to show, first, how this interaction can be achieved and, second, that the whole approach is feasible.

1.2 Examples of Desirable Optimizations

We show in this section, via examples from both the relational and OO world, that the interaction, during optimization, between use of indexes, use of materialized views and semantic optimization can lead to a large variety of alternative query plans. This in turn increases the potential for finding a fast execution plan. The rest of the chapters will then be dedicated to demonstrating how to include such capability into an optimizer. Before showing the examples (we will often refer to them later), we give an account of the main features of the language for queries and constraints that we use in this dissertation. An important restriction of this language (path-conjunctive queries and constraints), used in all our theoretical results, will be given later, in Chapter 3.

Language. We use throughout this dissertation the well-known syntax of ODMG/ODL and ODMG/OQL [Cat97] extended with a few constructs for both logical and physical schema and queries. As in ODL, we denote by Set(T)
the type of finite\(^1\) homogeneous sets of elements of type \(T\). We use \textbf{struct}(\(k_1 : T_1, \ldots, A_n : T_n\)) to denote record types. Record and set types can be nested in arbitrary ways (i.e. complex values are captured in our framework).

We consider a standard set of base types such as \texttt{bool, string}, etc.

In addition, we consider \textbf{dictionary types} denoted, as in ODL, by \texttt{Dict}(\(T_1, T_2\)). Such a dictionary type is the type of all dictionaries (finite functions) with keys of type \(T_1\) and entries of type \(T_2\). OQL already has \texttt{M}\{\(k\)}, the \textbf{lookup} operation that returns the entry corresponding to the key \(k\) in the dictionary \(M\), provided that \(M\) is defined \(^2\) for \(k\). In practice, for dictionaries with set-valued entries, one often assumes the existence of a non-failing lookup operation that returns the empty set rather than failing when \(k\) is not defined for \(M\). We denote this operation by \texttt{M}[\{A\}]. To this we add the operation \texttt{dom} \(M\) that returns the \textbf{domain} of the dictionary \(M\), i.e., the set of keys for which \(M\) is defined. Dictionaries are an essential component of our language and allow to express OO classes with extents and index-like structures, together with their operations: oid dereferencing and index lookup. We will detail this in example 1.2.1, to come shortly.

To summarize what we have so far, the following are the \textbf{path expressions} of our language:

\[
P ::= \texttt{R} \mid c \mid x \mid \texttt{dom} P \mid P [k] \mid P_2 [P_1] \mid P_2 [P_1]\]

Here we denote by \texttt{R} schema names (they can be of set type such as relation names or of dictionary types), \(c\) stands for constants at base types, while \(x\) denotes variables. For us a schema is usually a collection of names with their types.

\textbf{Queries.} We adopt the OQL \texttt{select-from-where} syntax for set-valued queries, which for the core of this dissertation have the following form:

\[
Q ::= \texttt{select} \quad \textbf{struct}(A_1 : P_1, \ldots, A_n : P_n) \\
\textbf{from} \quad P_1 x_1, \ldots, P_m x_m \\
\textbf{where} \quad C_1 \text{ and } C_2 \text{ and } \ldots \text{ and } C_k
\]

The semantics of such a query is the usual one, with one main difference: here we assume set semantics and not bag semantics (in OQL we would have to explicitly write the keyword \texttt{distinct} to obtain the same effect). The reason for restricting to set semantics will become apparent when we discuss later the chase.

Thus, the expression in the \texttt{select} clause is always a record and moreover we require that the expressions appearing as components of the output record are path expressions (i.e. not queries). Thus we do not allow for nested queries inside the \texttt{select} clause. We call the bindings \(P_1 x_1, \ldots, P_m x_m\) in the \texttt{from} clause \texttt{vars}. For a scan \(P_i x_i\) we call \(P_i\) a generator and it must be a path expression of set type. Thus, there cannot be any nested queries in the \texttt{from} clause. This is only for simplicity of exposition, in general we could allow queries with nested queries as generators and these could be normalized into queries that have just path expressions as generators. Such normalization is a polynomial time rewriting and the details of it can be found in [PT99]. On the other hand, nested queries in the \texttt{select} clause cause significant difficulties in checking (in a complete way) equivalence of queries and as far as we know this is an open problem [LS97]. We believe that our method can be extended to handle in a sound but not complete way such queries and we discuss possible extensions in Chapter 7. Finally, the \texttt{where} clause is a conjunction of predicates \(C_i\) where each \(C_i\) is of the form \(P_i \texttt{ op} P_h\) with \texttt{op} being one of \(\{=, <, =, >, <, >\}\). However, for our completeness results we will need to rule out the inequality operators and consider only equalities in the \texttt{where} clause. We do not consider negation nor union (but we discuss the possibility of extension in Chapter 7).

\textbf{Dictionary-valued queries.} Let \(Q\) and \(Q'\) two expressions, as introduced before (i.e path expressions or select-from-where queries). In addition, \(Q\) must be of set type while \(Q'\) is allowed to mention the free variable \(x\). Then

\(^1\)Except in section 3.7 where we discuss non-terminating chase and infinite models.

\(^2\)Otherwise, lookup will fail. We will be careful to avoid this in the case of path-conjunctive queries, see chapter 3.
\[ \text{dict } x \text{ in } Q \Rightarrow Q'(x) \]

is an expression which, when evaluated, produces a dictionary having as domain the result of \( Q \) and associating to each element \( x \) in the domain the value of \( Q'(x) \).

**Constraints.** Logical assertions of the following form are used to describe constraints:

\[ \forall (x_1 \in P_1) \ldots \forall (x_n \in P_n) \left[ B_1(\bar{x}) \Rightarrow \exists (y_1 \in P'_1) \ldots \exists (y_m \in P'_m) B_2(\bar{x}, \bar{y}) \right] \]

Here \( P_i \) and \( P'_i \) are path expressions while \( B_1 \) and \( B_2 \) are conjunctions of equalities between path expressions. We use the notation \( B_1(\bar{x}) \) to denote that \( B_1 \) may depend on variables \( x_1, \ldots, x_n \). Remark that these are not first-order logical formulas because the quantifications are bounded and each \( P_i \) (or \( P'_i \)) may be not only relation names but paths that depend on variables previously bounded. When the data model is restricted to be relational, then this class of constraints becomes the same as that of *tuple-generating dependencies (TGDs)* of [BV84b] or *embedded implicational dependencies* of [Fag82]. In the case when the existentially quantified part is missing (i.e. the formula inside the universal quantification is an implication of conjunctions of equalities) the above generalizes *equality-generating dependencies (EGDs)* of [BV84b], a class that includes functional dependencies.

In Chapter 3 we will consider a restriction of this language, restriction that we call *path-conjunctive (PC) queries* and *embedded path-conjunctive dependencies (EPCDs)*, for which we define the chase in its most general form and prove the completeness results of Chapter 3 and Chapter 4. The path-conjunctive language is also the language used in the current implementation of our prototype.

**Example 1.2.1 (Mixing OO and relational, semantic optimization and use of indexes.)** Consider the logical schema in figure 1.5. It is written following mostly the syntax of ODL, the data definition language of ODMG, extended with referential integrity (foreign key) constraints in the style of data definition in SQL. It consists of a class **Dept** whose objects represent departments, with name, manager name, and **DProjs**, the set of names of all the projects done in the department. It also consists of a relation **Proj** whose tuples represent projects, with name, customer name, **PDept**, the name of the department in which the project is done, and the budget associated, **Budg**.

```plaintext
Proj: Set<Struct{
  string PName;
  string CustName;
  string PDept;
  string Budg;
}>

class Dept{
  string PName;
  attribute string DName;
  string CustName;
  relationship Set<string> DProjs
  string Budg;
  inverse Proj(PDept);
  primary key PName;
  attribute string MgrName;
  foreign key PDept
  foreign key DProjs
  references Dept::DName;
  references Proj(PName);
  relationship PDept
  inverse Dept::DProjs;
}
```

Figure 1.5: The Proj-Dept schema in extended ODMG

The internal representation of this logical schema is different and more precise (see below). **Proj** is represented as it is, i.e. a schema name, **Proj**, with the same type as the ODL type. However, the class **Dept** is translated as a dictionary. In our approach an OO class must have an extent and is represented as a dictionary whose keys are the oids, whose domain is the extent and whose entries are records of the components of the objects. To maintain the abstract properties of oids we do not make any assumptions about their nature and we invent fresh new base types for them (see **Doid** for **Dept**; we abused the notation a little by choosing for the dictionary the same name as the class). This representation actually corresponds to the usual semantics of OODB constructs [AK89].
Dept: Dict(DId, Struct{string DName;
Set(string) DProjs;
string MgrName})
Proj: Set(Struct{string PName; string CustName;
string PDept; string Budg})

To complete the translation of the extended ODMG schema of figure 1.5 into our logical schema representation we need to represent, in addition to the schema names, the referential integrity (RIC), inverse relationship, and key constraints. Here they are:

(RIC1) \( \forall (d \in \text{domDept}) \forall (s \in \text{Dept}[d].DProjs) \exists (p \in \text{Proj}) s = p.PName \)

(RIC2) \( \forall (p \in \text{Proj}) \exists (d \in \text{domDept}) p.PDdept = \text{Dept}[d].DName \)

(INV1) \( \forall (d \in \text{domDept}) \forall (s \in \text{Dept}[d].DProjs) \forall (p \in \text{Proj}) \)
\[ s = p.PName \rightarrow p.PDdept = \text{Dept}[d].DName \]

(INV2) \( \forall (p \in \text{Proj}) \forall (d \in \text{domDept}) \)
\[ p.PDdept = \text{Dept}[d].DName \rightarrow \exists (s \in \text{Dept}[d].DProjs) p.PName = s \]

(KEY1) \( \forall (d \in \text{domDept}) \forall (d' \in \text{domDept}) \) [\( \text{Dept}[d].DName = \text{Dept}[d'].DName \rightarrow d = d' \)]

(KEY2) \( \forall (p \in \text{Proj}) \forall (p' \in \text{Proj}) \) [\( p.PName = p'.PName \rightarrow p = p' \)]

Consider also the following OQL query that asks for all project names, with their budgets and department names, that have a customer called "CitiBank":

\[
\text{select distinct struct(PN : s, PB : p.Budg, DN : d.DName)} \\
\text{from depts d, d.DProjs s, Proj p} \\
\text{where s = p.PName and p.CustName = "CitiBank"}
\]

We deal only with set semantics in this dissertation, thus we omit writing the keyword distinct from now on. The translation of the above OQL query is then given below. Thus, if \( d \) is an oid in depts the implicit dereferencing in \( d.DName \) corresponds to the dictionary lookup in \( \text{Dept}[d].DName \). Also the extent depts is translated as \( \text{domDept} \).

\[
(Q) \text{select struct(PN : s, PB : p.Budg, DN : Dept[d].DName)} \\
\text{from domDept d, Dept[d].DProjs s, Proj p} \\
\text{where s = p.PName and p.CustName = "CitiBank"}
\]

Physical schema. For this example, we assume that the relation \( \text{Proj} \), stored as a table (a set of records), and the dictionary \( \text{Dept} \), stored in an index-like way, are also part of the physical schema, who therefore is not disjoint from the logical schema; this is a common situation. In addition, we assume that the following indexes are maintained: a primary index \( I \) on the key \( PName \) of relation \( \text{Proj} \) and a secondary index \( SI \) on \( \text{CustName} \) of relation \( \text{Proj} \)(we could have also added an index between the key \( DName \) and the extent of \( \text{Dept} \) but we don’t need it for the example). Both indexes are represented by dictionaries (see figure 1.6). For example, \( I[s] \) returns the record \( r \) in \( \text{Proj} \) such that \( r.PName = s \). Similarly, \( SI[c] \) gives back the set of records\(^3\) \( r \) in \( \text{Proj} \) such that \( r.CustName = c \). Finally, the physical schema materializes the physical access structure defined by:

\[
(JI) \text{select struct(DOID : d, PN : p.PName)}
\]

\(^3\)In an implementation this may be a set of record ids rather than a set of records (if \( SI \) is not a clustered index), and similarly for the case of the primary index. This would introduce an additional level of indirection that we chose not show here for simplicity of presentation.
Dept : Dict(Doid, Struct{string DName:
  Set{string DProjs; string MgrName})
Proj : Set{Struct{string PName; string CustName;
  string PDept; string Budg}}
I : Dict{string, Struct{string PName; string CustName;
  string PDept; string Budg}}
SI : Dict{string, Set{Struct{string PName; string CustName;
  string PDept; string Budg}}}  
JI : Set{Struct(Doid DOID; string PN)}

Figure 1.6: The physical schema

from depts d, d.DProjs s, Proj p
where s = p.PName

Note that JI is both a generalized access support relation [KM90a] and a generalized join index [Val87] since it involves a relation and a class.

Alternative query plans With this physical schema, under the constraints specified in the logical schema, we give three examples of alternative query plans for the query Q we saw earlier (Q itself may be a reasonable plan, even though the three plans below are potentially significantly better).

(P1) `select, struct(PN : p.PName, PB : p.Budg, DN : p.PDept)
  from Proj p
  where p.CustName = "CitiBank"

(P2) `select, struct(PN : p.PName, PB : p.Budg, DN : p.PDept)
  from SI ["CitiBank"] p

(P3) `select, struct(PN : j.PN, PB : I[j.PN].Budg, DN : Dept[j.DOID].DName)
  from JI j
  where I[j.PN].CustName = "CitiBank"

Obtaining P1 from Q is what is usually understood by semantic optimization. Notice that both Q and P1 do not involve any schema elements that are not in the logical schema, thus they can be thought of queries at the logical level. Their equivalence is governed by the semantic constraints describing the logical schema. On the other hand, the other two plans involve the additional physical schema elements. Depending on the cost model (especially in a distributed heterogeneous system), either one of Q, P1, P2, and P3 may be cheaper than the other. As we shall see, although they are quite different in nature, our optimization algorithm is able to generate systematically all these plans.

Example 1.2.2 (Index use enabled by semantic constraints) This is a very simple and common relational scenario adapted from [Bak99], showing the benefits of exploiting referential integrity constraints.

Consider a relation R(A, B, C, E) and a query that asks for all tuples in R with given values for the attributes B and C:
\[(Q)\quad \text{select struct } (A = r.A, E = r.E)\]
\[\text{from } R r\]
\[\text{where } r.B = b \text{ and } r.C = c\]

The relation is very large, but the number of tuples that meet the where clause criteria is very small. However, the SQL engine is taking a long time in returning an answer. Why isn’t the system using an index on \(R\)? Simply because there is no index on the attributes \(B\) and \(C\). The only index on \(R\) that includes \(B\) and \(C\) is an index, call it \(I\), on \(ABC\). There is no index with \(B\) and/or \(C\) in the high-order position(s), and the SQL optimizer chooses to do a table scan over \(R\) to answer the query (it might have been better to choose an index scan over \(I\) instead of a scan over the whole relation \(R\)).

There are several solutions to force the SQL optimizer to use the index on \(ABC\): for example, if all possible values of \(A\) are known to be in the set \(\{01', 02', 03', 04'\}\), one can hard-code in the where clause the condition \(A \in \{01', 02', 03', 04'\}\) and the problem is solved. Of course, this is not a real solution because tomorrow the values for \(A\) might change! The reader can find several other solutions in [Bak99] but none are satisfactory except one: rewrite \(Q\) into an equivalent query that does a join of \(R\) with a small table \(S\) on attribute \(A\) knowing that there is a foreign key constraint from \(R\) into \(S\) on \(A\):

\[(Q')\quad \text{select struct } (A = r.A, E = r.E)\]
\[\text{from } R r, S s\]
\[\text{where } r.B = b \text{ and } r.C = c \text{ and } r.A = s.A\]

Although we have not selected any attributes from \(S\), the join with \(S\) is of a great benefit. The SQL optimizer chooses (only now!) to use \(S\) as the outer table in the join and while scanning \(S\), as each value \(a\) for \(A\) is retrieved, the index \(I\) is used to lookup the tuples corresponding to \(a\), \(b\), \(c\). As we shall later see, our optimizer has the capability of finding, automatically, such a plan.

**Example 1.2.3 (Use of views enabled by key constraints)** Here we show that integrity constraints also create opportunities for rewriting queries using materialized views. Consider the query \(Q\) given below, which joins relations \(R_1(K, A_1, A_2, F, \ldots), R_2(K, A_1, A_2, \ldots)\) with \(S_{ij}(A_i, B_i, \ldots)\) \((1 \leq i \leq 2, 1 \leq j \leq 2)\). Figure 1.7 depicts \(Q\)’s join graph, in which the nodes represent the bindings of the query variables and the edges represent equijoins between them. The join conditions are shown on the edge labels.

\[(Q)\quad \text{select struct } (B_{11} : s_{11}.B, B_{12} : s_{12}.B, B_{21} : s_{21}.B, B_{22} : s_{22}.B)\]
\[\text{from } R_1 r_1, S_{11} s_{11}, S_{12} s_{12}, R_2 r_2, S_{21} s_{21}, S_{22} s_{22}\]
\[\text{where } r_1.F = r_2.K \text{ and } r_1.A_1 = s_{11}.A_1 \text{ and } r_1.A_2 = s_{12}.A_2\]
\[\text{and } r_2.A_1 = s_{21}.A_1 \text{ and } r_2.A_2 = s_{22}.A_2\]

One can think of \(R_1, S_{11}\) and \(S_{12}\) as storing together one large conceptual relation \(U_1\) that has been normalized for storage efficiency. Thus, the attributes \(A_1\) and \(A_2\) of \(R_1\) are foreign keys into \(S_{11}\) and, respectively, \(S_{12}\). The attribute \(K\) of \(R_1\) is the key of \(U_1\) and therefore of \(R_1\). Similarly, \(R_2, S_{21}\) and \(S_{22}\) are the result of normalizing another large conceptual relation \(U_2\). For simplicity, we used the same name for attributes \(A_1, A_2\) and \(K\) of \(U_1\) and \(U_2\) but they can store different kind of information. In addition, the conceptual relation \(U_1\) has a foreign key attribute \(F\) into \(U_2\) and this attribute is stored in \(R_1\). We want to perform the foreign key join of \(U_1\) and \(U_2\), which translates to a complex join across the entire database. The query returns the values of the attribute \(B\) from each of the ”corner” relations \(S_{11}, S_{12}, S_{21}, S_{22}\). (Again for simplicity we use the same name \(B\) here, but each relation may store different kind of information).

Suppose now that the attributes \(B\) of the ”corner” relations have few distinct values, therefore the size of the result is relatively small compared to the size of the database. However, in the absence of any indexes on the attributes \(B\) of the ”corner” relations the execution time of the query is very long. Instead of indexes, we assume the existence of materialized views \(V_i(K, B_i, B_j)\) \((1 \leq i \leq 2)\), where each \(V_i\) joins \(R_i\) with \(S_{1i}\) and \(S_{2i}\) and retrieves the \(B\) attributes from \(S_{1i}\) and \(S_{2i}\) together with the key \(K\) of \(R_i\):
(V_i)

\[
\begin{align*}
\text{select} & \quad \text{struct}(K : r.K, B_1 : s_1.B, B_2 : s_2.B) \\
\text{from} & \quad R \ r, S_{11} s_1, S_{22} s_2 \\
\text{where} & \quad r.A_1 = s_1.A_1 \text{ and } r.A_2 = s_2.A_2
\end{align*}
\]

It is easy to see that the join of R_2, S_{21}, and S_{22} can be replaced by a scan over V_{21}:

\[
\text{(Q')} \quad \begin{align*}
\text{select} & \quad \text{struct}(B_{11} : s_{11}.B, B_{12} : s_{12}.B, B_{21} : v_2.B_1, B_{22} : v_2.B_2) \\
\text{from} & \quad R_1 r_1, S_{11} s_{11}, S_{12} s_{12}, V_2 v_2 \\
\text{where} & \quad r_1.F = v_2.K \quad \text{and} \\
& \quad r_1.A_1 = s_{11}.A_1 \text{ and } r_1.A_2 = s_{12}.A_2
\end{align*}
\]

Less intuitively though, the join of R_1, S_{11}, and S_{12} cannot be replaced by a scan over V_1. Q'', the obvious candidate for a rewriting of Q using both V_1 and V_2 is not equivalent to Q in the absence of additional semantic information.

\[
\text{(Q'')} \quad \begin{align*}
\text{select} & \quad \text{struct}(B_{11} : v_1.B_1, B_{12} : v_1.B_2, B_{21} : v_2.B_1, B_{22} : v_2.B_2) \\
\text{from} & \quad R_1 r_1, V_1 v_1, V_2 v_2 \\
\text{where} & \quad r_1.K = v_1.K \text{ and } r_1.F = v_2.K
\end{align*}
\]

The reason is that V_1 does not contain the F attribute of R_1, and there is no guarantee that joining the latter with the V_1 will recover the correct value of F. If, on the other hand, K were a key in R_1, Q'' would be equivalent to Q, being therefore an additional (and likely better) plan. We will see, how our optimization strategy is able to consider key constraints (other constraints as well) in order to find such rewritings with views.

**Example 1.2.4 (Interaction between views and indexes)** Assume a logical schema with relations \( R(A, B) \) and \( S(B, C) \), and a physical schema that has \( R \) and \( S \) too (direct mapping!), as well as a materialized view \( V = \Pi_A (R \bowtie S) \) and secondary indexes \( I_R \) and \( I_S \) on attributes \( A \) and \( B \) of \( R \) and \( S \), respectively. We want to optimize the logical query \( Q = R \bowtie S \) (expressible in our language in the obvious way).

\( Q \) itself is a valid query plan. However, we want to take advantage of \( V \) and of the two indexes and find possible better plans. By considering only the view \( V \), the following rewriting is equivalent to \( Q \):

\[
\text{(P)} \quad \begin{align*}
\text{select} & \quad \text{struct}(A : r.A, B : s.B, C : s.C) \\
\text{from} & \quad V v, R r, S s \\
\text{where} & \quad v.A = r.A \text{ and } r.B = s.B
\end{align*}
\]

However, the above rewriting \( P \) is not minimal: the scan over \( V \) is, obviously, redundant, and can be eliminated from \( P \) to produce a smaller query (\( Q \), in this case). In fact, there is no rewriting that uses \( V \) and has a minimal number of scans in the from clause. The classical algorithms for rewriting queries using views are based on the
results of [LMSS95] for conjunctive queries, and they only explore minimal rewritings\(^4\). Thus, they fail to find any rewritings for this example.

However, if \( V \) is a small relation, the above query \( P \) can have a better execution plan than the original \( Q \). This plan, based on the existence of indexes performs a scan of \( V \) first, then uses for each tuple in \( V \) the value of the \( A \) attribute to lookup in the index \( I_R \) for \( R \), then performs lookups in the index for \( I_S \) for \( S \). Since our language can have indexes (through the use of dictionaries), the above execution plan can then be expressed as the following query plan:

\[
(P') \quad \text{select struct}(A : r.A, B : s.B, C : s.C) \\
\text{from} \quad V v, I_R[v.A] r, I_S[r.B] s
\]

from which no scan in the \text{from} clause can be eliminated, and is therefore minimal. We will see in section 6.4 that our optimizer, by the simple fact of incorporating indexes explicitly at the language level in the physical schema, is able to explore and find such plans (inexpressible as conjunctive queries).

### 1.3 Approach and Contributions

In this section we summarize our main contributions as well as we give an account of the different phases, algorithms and concepts involved in the chase and backchase optimization. In addition, in subsequent chapters, we will point out other specific contributions. The following is a list of the main techniques and concepts that we introduce:

- **A new language able to deal with nested sets and with dictionaries.** Dictionaries (as already introduced in section 1.2) are finite functions that allow a natural description of both storage and fast access capabilities of index-like structures as well as OO classes. Our language allows for defining, as dictionary views, physical schema elements that were not fully expressible in the literature on relational and OO algebras.

  A dictionary is characterized by a finite domain of keys and for each key there is an entry associated with it. The operation that given a key in the domain returns the corresponding entry in the dictionary is the dictionary lookup. For example, a primary index on a primary key \( A \) of a relation \( R \) can be defined as a dictionary having as keys the set of all values for \( A \) in \( R \), while the entries are the tuples corresponding to each key. A secondary index on some non-key attribute is defined similarly with the difference that entries are sets of tuples rather than one unique tuple. Other physical access structures such as join indexes, access support relations, sources with limited capabilities (binding patterns) can be defined naturally with dictionaries. In addition, logical schema elements such as OO classes with extents are expressible via dictionaries. A class is modeled as a dictionary having as domain the set of all oids of that class, i.e. the extent of the class. Moreover, the entry in the dictionary corresponding to a particular oid is the record value associated in the class to the oid. The operation of oid dereferencing becomes then dictionary lookup.

- **Fundamental use of constraints that make the different optimization techniques cooperate easily.** We develop a constraint language that can express both semantic constraints, relational and OO (inverse relationship constraints, for example), and physical constraints: constraints that equivalently characterize (and can therefore replace) physical schema definitions of materialized views and indexes. Semantic and physical constraints have the same syntactic form and thus we will be able to use them in the same way during rewriting. The main idea behind physical constraints is as follows: typically a physical access structure \( V \) (index, materialized view, join index, GMAP, etc.) has a definition expressed as a query \( V = Q(R) \) in terms of the logical schema elements, denoted here collectively by \( R \). Instead of this definition

\(^4\)Otherwise, the space of non-minimal rewritings is infinite.
what we use in the optimization is two "complementary" constraints corresponding to the two inclusions: $V \subseteq Q(R)$ and $Q(R) \subseteq V$. Notice that $Q$ can either be a set-valued query or a dictionary-valued query.

- **Chase for checking equivalence of queries under constraints.** A prerequisite in optimization is deciding equivalence of queries. Before choosing a (optimal) plan $P$ for a query $Q$ we need to make sure that $P$ is equivalent to $Q$. In our framework, checking equivalence of queries in a complex schema encompassing both logical and physical elements becomes checking equivalence of queries under constraints (or dependencies; we will use the two terms interchangeable).

We develop the *equational chase* method for checking equivalence of queries under constraints. The chase is a rewriting procedure that transforms queries into equivalent queries based on the existing constraints in the schema. Each chase step is a rewrite $Q \overset{d}{\longrightarrow} Q'$ where $d$ is a constraint in the schema. In turn, a chase step amounts itself to two steps: first, finding whether the constraint $d$ is applicable. This amounts to finding whether the conditions required by $d$ are implied $Q$. Checking whether this implication holds is done by looking for a certain substitution (homomorphism) from the variables of the constraint to the variables of the query. If such homomorphism exists, then in the second step, the conditions that the constraint guarantees to satisfy are "added" to the query. These can be: new variables ranging over logical/physical sources, added to the *from* clause, and/or new predicates, added to the *where* clause. Chasing a query with a set of constraints means chasing the query, in any order, with all the constraints in the set. The chase is an equivalence-preserving transformation\(^5\) and one can check whether two queries are equivalent by chasing them to normal form.

Equivalence is related, inter-reducible in fact, to the problem of constraint implication. We show that our equational chase generalizes the classical relational chase of [ABU79, MMS79, BV84b]. We also show that the main results that allow one to use the chase as a *complete* proof procedure for containment/equivalence of queries and implication of dependencies still hold when we move from relational conjunctive queries and dependencies to *path-conjunctive queries and dependencies.*

The PC simplification. Path-conjunctive (PC) queries and embedded path-conjunctive dependencies (EPCDs) are the "conjunctive" fragment of our dictionary based language. This is the language for which the chase is defined in its more general form, and also the language used in all our theoretical results as well as the the current implementation of our prototype. While not allowing union/disjunction nor grouping/aggregates nor negation, this language is still powerful enough to express the (conjunctive) core of OQL (in particular SQL). This is still a very expressive language. For example, all the queries and constraints in section 1.2 are path-conjunctive (with the exception of the plans using the non-failing lookup operation; we show how to deal with these in section 6.4). When no nested sets or dictionaries are present, the PC language expresses exactly the conjunctive relational queries.

In addition to the above mentioned completeness of the chase (for path-conjunctive queries and dependencies), we also show that other classical results from the relational theory of conjunctive/tableaux queries generalize as well to path-conjunctive queries: NP-completeness of query containment (under all instances), decidability of query containment (under dependencies) and dependency implication, for a class of EPCDs that we call *full* and generalize the tgd's of [BV84b]. Thus the class of path-conjunctive queries and constraints is the natural class to consider as the foundation for our (both OO and relational) optimization framework. Extensions that include union/disjunction, grouping/aggregates are possible and discussed in Chapter 7.

- **Chase for discovery of relevant physical and logical data sources.** While the chase is useful for checking equivalence of two given queries (a logical query and a rewriting over the physical schema), it does not (apparently!) say anything about *where/how* to look for a (optimal) rewriting. Optimization is thus a harder problem, and it requires discovery of alternative equivalent queries/plans.

The solution that we adopt uses, again, the chase, in a novel manner. Given a logical query $Q$, a logical schema with semantic constraints $D$, and a physical schema described with physical constraints $D'$, we

\(^5\) Only under set semantics (see later Section 2.1). We adopt set semantics all throughout this dissertation.
chase $Q$ with all the constraints in $D$ and $D'$ to produce a larger query that contains within it (the join of) all relevant physical and logical sources that can answer the query. We call this larger query the universal plan $U$.

- **Subqueries of the universal plan define the search space.** The universal plan (call it $U$) is equivalent to the input query $Q$, but it is not a very efficient query because it is highly redundant. However, we observed that by backchasing, i.e., applying chase steps in reverse that go from larger queries to smaller queries, we can find several equivalent rewritings of the universal plan that are smaller in size. For example, the original query $Q$ can be obtained by backchasing with a sequence of constraints that is the reverse of the one used for chasing $Q$ into the universal plan $U$. During this particular backchase sequence exactly those elements that were added to $Q$ during the chase are now removed. But by using a different sequence of constraints during the backchase, other elements of the universal plan, such as the ones that were in $Q$, can be removed while leaving the new schema elements in place. The result is then a query that can look very different from the original query. Hence, by removing in various ways the redundancy that exists within the universal plan, one can find in a systematic way many different queries, equivalent to $Q$, and potentially more efficient. Figure 1.8 illustrates the general situation. All of the resulting queries have the property that they are subqueries of the universal plan.

![Figure 1.8: Chase and back $d_1, \ldots, d_n$ is the sequence of constraints used during chasing $Q$ into $U$.](image)

Thus, we define the search space for rewritings of $Q$ as being the set of all equivalent subqueries of the universal plan that mention physical schema elements. Figure 1.9 graphically depicts this definition. We denote there collectively by $\hat{V}, \hat{V}_1, \hat{V}_2$ elements of the physical schema, and by $R_1, R_2$ elements of the logical schema. The original query mentions only $R_1$, while the universal plan has additional logical sources $R_2$ and physical sources $\hat{V}$. The subqueries that are executable are those that use subsets $\hat{V}_1, \ldots, \hat{V}_k$ of $\hat{V}$. The rewritings that we are particularly interested in are the equivalent subqueries of the universal plan having minimal number of joins. We show that when limiting the physical schema to path-conjunctive materialized views and in the absence of logical constraints, every minimal rewriting of the input query is a subquery of the universal plan, thus the universal plan is a complete search space for minimal rewritings.

![Figure 1.9: Search space for minimal rewritings.](image)

- **Backchase minimization as search strategy.** Exploring the search space to find minimal subqueries of the universal plan is a minimization problem. It subsumes relational tableau minimization [AHV95]
but it is more general due to the presence of nesting and dictionaries in the query language and because equivalence is considered under constraints as opposed to equivalence under all instances.

Backchase minimization is our search procedure for enumerating minimal equivalent subqueries of the universal plan. For each subquery explored we check the equivalence (with the universal plan) via chase. There are two basic ways, and conceptually equivalent, of implementing the backchase minimization:

- **Top-down.** The first one is a top-down, decremental, procedure that goes from the universal plan down to its subqueries by eliminating, exhaustively, one scan at a time from the *from* clause. The algorithm stops descending on a branch whenever a *non-equivalent* subquery is found. The last equivalent query on that branch is a minimal equivalent subquery of the universal plan. We prove that all equivalent subqueries can be found this way, i.e., completeness for the path-conjunctive case.

- **Bottom-up.** Symmetrically, the second way of implementing the backchase minimization is a bottom-up, incremental, procedure, that assembles subqueries of the universal plan starting from the smaller ones. The algorithm stops ascending on a certain branch when an *equivalent* subquery is found (in contrast to the top-down algorithm). Every such equivalent subquery is a minimal equivalent subquery of the universal plan. *The crucial advantage of the bottom-up backchase is that it can be mixed with cost-based pruning.*

For presentation purposes, we will prefer many times to use the top-down variant of backchase. However, in Chapter 6, we will see the importance in practice of the bottom-up approach (when mixed with cost-based pruning).

It is worthwhile mentioning that even in the absence of semantic or physical constraints, the backchase minimization can provide useful optimization. This amounts to eliminating redundant joins in a query (see also the cost monotonicity discussion below) and we believe that this in itself is a requirement in the context of complex systems that compose (automatically) queries with views. Such examples include mediator systems that integrate semi-structured (or XML) data sources with relational sources, in which the relational sources are exported as semi-structured/XML views. Complex queries posed in a language such as UnQL [BDHS96] or XML-QL [DFF+99] must then be decomposed into relational queries [FTS99, FPS97]. Typically the resulting relational queries have a lot of redundant joins and being able to eliminate them in a systematic (non-heuristical) way is crucial.

**Cost monotonicity assumption.** An important assumption is used in the chase/backchase approach: *candidate plans that have more joins are more expensive than candidate plans with less joins, and thus not considered for cost-evaluation.* This assumption, also used implicitly in join elimination of [CGK+99], rewriting with materialized views [LMS95], tableau elimination, etc., allows us to give the efficient bottom-up implementation of the backchase in which the search space is pruned. We argue in section 6.2.4 that cost monotonicity is essentially true under one important condition: joins are implemented against by methods other than index-based. Then a minimal candidate plan can always be favored against a candidate plan that has additional redundant joins. The situation in which the monotonicity assumption becomes false (by a significant margin of error) is one in which indexes are part of physical plans. We will see in Chapter 6 that the bottom-up backchase minimization is extended to search (in a controlled way, with not much additional overhead) for such plans that are not minimal but have a potentially good cost.

- **C&B enumerator.** Thus, our main strategy for enumeration of candidate plans consists of two phases: chase to obtain the universal plan, followed by backchase minimization to find minimal reworkings. The minimal reworkings can use various physical sources, different in general from the ones that explicitly occur in the original query. This variety is enabled by the use of both semantic constraints and existing physical schema elements, and here is where the strength of our approach lies. We call the two phases combined the C&B (chase and backchase) enumeration. The main architecture of the C&B enumerator is summarized in figure 1.10.

---

6With a not too large margin of error.
**C&B with cost-based pruning.** The minimal candidate plans that result after the C&B enumeration are not yet plans ready for evaluation. These candidate plans specify what physical access paths to use, but they do not tell yet how to use them. An additional module, see figure 1.4, in which techniques such as join reordering, pushing selections down to sources, choosing various join algorithms (sorting, hashing, etc.) are used to translate a candidate plan into an actual physical plan, is needed. This step must be cost-based and the result, for each candidate plan, is the cheapest physical plan that implements the candidate plan. The final result of the optimization is then the cheapest among all physical plans implementing the candidate plans enumerated by the C&B phase.

While conceptually the C&B enumeration is separated from the cost-based phase, in reality it is highly desirable to mix the two. In Chapter 6 we show that, by using cost-based pruning, we can short-circuit the exhaustive enumeration of minimal rewritings, and find directly the best physical plan. Due to the nature of the search space involved (subqueries of the universal plan), and based on a monotonicity of cost assumption, cost-based pruning in our context is very effective: when a subquery is found to have a higher cost then the best cost found so far, the subquery is pruned together with all of its superqueries. This pruning strategy works in conjunction with a bottom-up variant of the backchase. The improvement in performance (optimization time) is then substantial, sometimes over an order of magnitude (for large enough queries).

Cost evaluation itself is more complex for the queries and data that we consider than for the relational case. In Chapter 6 we design a language for describing physical plans for OO (with nested collections) and relational queries. This language is centered on two basic access primitives: scan of a set, and lookup into a dictionary. We give then a cost-model for evaluating such plans, based on database statistics. Then, we show how we can explore the space of physical plans, by mainly reducing the problem to a join enumeration problem. The reduction, based on a new technique that we call query fragmentation, allows us to immediately generalize the classical dynamic programming algorithm for join enumeration. In addition to handling dependent joins (typical for OO queries) and scans over nested collections, classical techniques such as pushing down selections and projections are also performed during the join enumeration.

**Experiments.** The natural question that we raise next is whether the C&B technique is practical. This means two sets of issues:

1. Are there feasible implementations of the technique? In particular:
(a) Is the chase phase feasible, given that even determining if a constraint is applicable requires searching among exponentially many variable mappings (homomorphisms)?

(b) Is the backchase feasible, given that even if each chase or backchase step is feasible, the backchase phase may visit exponentially many subqueries?

(c) What is the effect of using cost information on the performance of backchase?

2. Is the technique worthwhile? That is, when you add the significant cost of C&B optimization, is the cost of an alternative plan that only the C&B technique would find still better than the cost of the plan you had without C&B?

In order to answer the above questions, we have built a prototype implementation of the C&B technique for path-conjunctive queries and constraints. We have also built a cost-based optimizer for path-conjunctive queries that is used either in a separate phase that chooses the best physical plan among all candidate plans produced by the C&B enumerator, or mixed with a bottom-up backchase implementation of the C&B technique. With this implementation, we have used experimental configurations to answer the questions summarized above. The experimental configurations cover both relational and OO optimization, and the scenarios considered exhibit the systematic interaction between semantic and physical optimization that allows us to find high quality plans. Our experiments cover and go beyond the experiments of [CGK+99, TSI94, YL87, S089]. We reconstructed those experiments and found that our optimizer can also find the desired plans for a set of chosen queries. However, we went further by repeating the experiments on families of queries and schemas of similar structure but of increasing complexity. This allows us to find out how far the technique can take us and to show that the applicability range of the implementation likely includes the range of practical queries. And, for one of the configurations where we can use a conventional execution engine we have also measured the global benefit of the C&B technique by measuring the reduction in total processing (optimization + execution) time, as a function of the complexity of the queries and the schema. The experiments were done for both pure C&B-enumeration (Chapter 5) and C&B with cost-based pruning (Chapter 6).

Feasibility of the C&B approach. In Chapters 5 and 6 we show the following:

1. The technique is definitely feasible, for practical schemas and queries, as follows:

   (a) By using congruence closure and a homomorphism pruning technique, we can implement the chase very efficiently in practice. This is very important, since the backchase phase uses the chase very frequently. 

   (b) The backchase enumeration, in the absence of cost information, quickly becomes impractical if we increase both query complexity and the size of the constraint set. As alternative search strategies, we have designed several stratification techniques [PdT00] that are variations of the basic backchase algorithm and split the search space, whenever possible, into several search spaces of smaller sizes. We show that these strategies are efficient and worthwhile even for quite challenging queries, thus making the whole approach scalable. Moreover, one of these strategies is complete for the important case of path-conjunctive materialized views [DPT99, Lev] just like the general technique.

   (c) Finally, we show that by taking advantage of cost information and mixing the backchase exploration phase with cost evaluation, the overall performance improves significantly. The C&B with cost-based pruning performs well in many common situations even when no stratification is applicable. The whole approach becomes then even more practical and worthwhile. Further mixing of stratification and cost-based pruning yields additional improvement for the case of path-conjunctive materialized views. For that case, such mixing offers a very good scalability with the query size and the number of views.

7No doubt such breaking points also exist for the implementations in the cited papers, but no information about them has been published.
2. We find the technique very valuable when only the presence of semantic integrity constraints enables the use of physical access structures or materialized views. The total processing time when C&B optimization is employed can become significantly smaller in such situations (in spite of the fact that the amount of time spent on optimization, relative to total processing time, is more significant than when traditional optimization is used). This clearly justifies the original intuition for this research direction [DPT99, PT99].

1.4 Overview of the Dissertation

The rest of the dissertation consists of the following:

   (a) Section 2.1: the chase as a procedure for checking equivalence (its complete version is the PC chase defined in Chapter 3).
   (b) Section 2.2: the top-down backchase minimization algorithm.
   (c) Section 2.3: the method by which we describe physical access sources through constraints.
   (d) Section 2.4: the chase as a procedure for discovering relevant physical and logical sources.
   (e) Section 2.5: several detailed examples that show how the chase/backchase method can be used to produce alternative plans.

2. Chapter 3: completeness results regarding the equational chase as a method for checking equivalence.
   (a) Section 3.1: introduces the path-conjunctive (PC) language for queries and constraints (dependencies).
   (b) Section 3.2: gives a canonical instance construction, essential for proving the subsequent results.
   (c) Section 3.3: we prove two theorems that: 1) characterize with homomorphisms, and 2) show NP-completeness of containment/equivalence of path-conjunctive queries under all instances and validity of path-conjunctive constraints.
   (d) Section 3.4: presents the PC chase, explains in which ways it differs from the relational chase, and gives the main theorems regarding its completeness as a procedure for checking containment/equivalence under constraints of PC queries, and implication of constraints. We also discuss issues such as termination, confluence and introduce a class of full dependencies, for which we show that such properties hold.
   (e) Subsequent sections of chapter 3: the proofs of the theorems presented in section 3.4.

3. Chapter 4: completeness results regarding the chase as used for discovery of relevant logical/physical sources, and the backchase minimization as enumeration procedure.
   (a) Section 4.2: we prove that for the case of optimizing PC queries with PC materialized views any "minimal" rewriting of the input query that is allowed to use views must be a subquery of the result of chasing the input query.
   (b) Section 4.3: proves that the top-down, decremental, backchase minimization enumerates all equivalent minimal PC subqueries of the universal plan.

4. Chapter 5: we describe preliminary experimental results regarding the C&B enumeration of minimal plans. No cost information is used in these experiments. (We call this method the pure C&B enumeration).
   (a) Sections 5.1 and 5.4.2: shows that the chase itself is fast and scales quite well.
(b) Section 5.2: we develop stratification techniques in order to speed-up the search for minimal plans. These techniques partition either the set of input constraints (off-line constraint stratification, OCS, in section 5.2.2) or partition the set of input constraints and the input query into query fragments (on-line query fragmentation OCS in section 5.2.1). We show that the second one is complete in the sense that it is guaranteed not to lose minimal plans, in a restricted but common situation in which materialized views and indexes are allowed.

(c) Section 5.4.3: compare experimentally the non-stratified full backchase (FB) with OQF and OCS in, showing that while FB is the bottleneck when the query size and the number of input constraints become moderately large, OCS and OQF can be efficiently used up to significantly larger numbers.

5. Chapter 6: we describe how to efficiently combine the pure C&B enumeration studied in the previous chapter with cost-based optimization. The goal is to avoid the exhaustive enumeration of minimal plans and thus produce an efficient optimizer based on chase and backchase.

(a) Section 6.1: the space of physical plans onto which PC queries are mapped. The physical plans resemble operator trees from the relational optimization. However, they are presented in a programming language style, and they are more complicated than the relational counterparts, because they consider access methods specific to dictionaries and nested sets. The two basic access primitives are scan and lookup.

(b) Section 6.2: a language for describing cost information when the data model has nested sets and dictionaries. In addition, we specify how to compute the cost of a physical plan. The cost model is a generalization of a simple relational cost model.

(c) Section 6.3: a generalized dynamic programming algorithm in the spirit of System R, used to enumerate the space of physical plans for a PC query and to select its best physical plan.

(d) Section 6.4: gives an algorithm BottomUpFB+Prune that efficiently combines bottom-up backchase with cost-based pruning, in order to find the best physical plan.

(e) Section 6.5: shows that BottomUpFB+Prune outperforms (by an order of magnitude, for large enough queries) the pure C&B enumeration. The use of cost information makes the whole approach very effective even when universal plans become large. We also give comparisons with the stratified techniques.

(f) Section 6.6: summary of the chapter

6. Chapter 7:

(a) Section 7.1: related work.

(b) Section 7.2: summary of our contributions.

(c) Section 7.3: we discuss the limitations of our approach.

(d) Section 7.4: future research items.
Chapter 2

Optimization with Chase and Backchase: The C&B Approach

This chapter introduces the basic principles that we use in a new optimization framework that smoothly integrates fundamental optimization techniques previously believed of different nature: semantic optimization, physical data independence, use of materialized views, tableau-like minimization.

Useful terminology. We distinguish between a candidate plan and an actual plan. A candidate plan is an equivalent rewriting of the input logical query that uses only physical schema elements. We will also use the term physical query for a candidate plan. A candidate plan specifies what schema elements are used to answer the query, i.e. it chooses the physical access paths. For example, a candidate plan specifies whether to use an index, a materialized view or the file when accessing a relation. A candidate plan is obtained via rewriting and may be totally different from the input query.

A candidate plan (or physical query) does not yet specify how the query is to be answered, i.e. join order, join algorithms, etc. In contrast, a physical plan that implements a candidate plan completely specifies the physical algorithms used to answer the candidate plan, i.e. it chooses the physical access methods. For example if, in the physical query, an index was already chosen for accessing a relation, then the physical plan is committed to use the index. (Physical plans corresponding to access paths other than the index will be discovered for other physical queries, and the rewriting phase must be able to produce all such physical queries.) Then, while producing the physical plan we will have a choice between, say, a nested loops join in which an index scan is involved, or an index-based join, in which an index lookup is used. Choosing the best physical plan implementing a candidate plan must be done in a cost-based fashion and the techniques that we use are explained in Chapter 6.

Rewriting with chase and backchase. This chapter focuses, mainly through examples, on the first conceptual phase of the optimizer: generating "good" candidate plans from a certain search space of candidate plans. It is quite important how we define this search space: searching through all possible candidate plans is not only impractical but theoretically impossible (there may be infinitely many queries equivalent to the input query). Our approach defines the search space for candidate plans using the chase method: relevant physical data sources are discovered by chasing the input query towards the universal plan. Then, a second stage, called the backchase minimization, explores subqueries of the universal plan trying to find better, non-redundant candidate plans.

The result of the backchase minimization (aka tableau minimization or join elimination) is a set of candidate plans.

1 In practice the cost-based phase must be performed at the same time, in order to prune the search space. This is addressed in Chapter 6.
Given a constraint $d$ of the form

$$\forall (r_1 \in R_1) \cdots \forall (r_m \in R_m) \left[ B_1 \Rightarrow \exists (s_1 \in S_1) \cdots \exists (s_n \in S_n) \ B_2 \right]$$

the corresponding chase step (in a simplified form) is the rewrite

$$(Q) \quad \text{select } O(\bar{r}) \quad \text{from } \ldots , R_1 r_1 \ldots , R_m r_m \ldots \quad \stackrel{d}{\rightarrow} \quad (Q') \quad \text{select } O(\bar{r}) \quad \text{from } \ldots , R_1 r_1 \ldots , R_m r_m , S_1 s_1 \ldots , S_n s_n \ldots \quad \text{where } \ldots \ \text{and } B_1 \ \text{and } \ldots$$

$^a$The general form requires the PC restriction. See later in Chapter 3 the PC chase definition, for which we prove completeness. In the simplified form shown here, the chase step can be applicable (as sound) to a larger class of queries than PC.

Figure 2.1: A chase step.

plans, none of which can be minimized any further (i.e. minimal candidate plans). Later in Chapter 4 we will show that, for a particular but important case (PC queries and PC materialized views), the universal plan contains all minimal equivalent rewritings of the input query. This result gives us a theoretical justification for using the universal plan to define our search space. For the more general case where we don’t have such a completeness result (and this is an open problem), we justify the use of the universal plan as the search space for finding enough good plans, through examples.

We start by explaining the chase as a procedure for checking equivalence of queries under constraints.

### 2.1 Chase: Checking Query Equivalence

Being able to check equivalence during optimization is a requirement. When we want to mix semantic optimization with use of physical definitions, all in a context that includes object-oriented and complex value query capabilities, the problem of deciding equivalence is actually a difficult one. One of the important contributions of this dissertation consists of several extensions of the theory of relational conjunctive query containment/equivalence and dependency implication. We show in Chapter 3 that checking containment/equivalence under all instances of two path-conjunctive queries is decidable and it can be characterized, as in the relational case [CM77], with homomorphisms (appropriately generalized). Moreover, we extend the relational chase to the path-conjunctive chase and we show that chasing is a complete proof procedure for checking equivalence of two path-conjunctive queries under constraints. Without entering in formal details here, we informally present a simplified version of the path-conjunctive chase and we give an example of how we use it to test equivalence. We point out that the restrictions imposed in Chapter 3 on path-conjunctive queries regarding set/dictionary equality are not needed for the soundness of the method.

The chase step, shown in figure 2.1 is sound, i.e. it rewrites $Q$ into a query $Q'$ such that $Q$ and $Q'$ are equivalent under all instances that satisfy the constraint $d$. (We are using the same language as introduced in Chapter 1.) An important observation is that the two queries are equivalent only under set semantics! Under bag semantics, for example, $Q'$ can have more duplicates than $Q$. In fact, the entire C&B approach (see Sections 2.2 and 2.4) works for queries with set semantics.

Chasing a query $Q$ with a set of constraints $D$ consists of applying repeatedly chase steps w.r.t any applicable constraint from $D$. "Applicable" must be defined carefully to avoid trivial loops and to allow for chasing even when the query and the constraint do not match syntactically as easily as we have seen in the simplified form.
above. We can stop this rewriting anytime and it will still be sound (under the constraints) for a large class of queries, views, indexes and constraints. We show in Chapter 3 that while the chase does not always terminate, it does so for certain classes of constraints and queries, yielding an essentially unique result whose size is polynomial \(^2\) in that of \(Q\).

**Example 2.1.1** Recall the queries \(Q\) and \(P_1\) from example 1.2.1. Here we show how one can use the chase to verify that the two queries are equivalent under the constraints given in the logical schema. We chase first \(Q\), shown again below:

\[
(Q) \quad \text{select struct}(PN : s, PB : p.Budg, DN : Dept[d].DName) \\
\quad \text{from dom Dept d, Dept[d].DProjs s, Proj p} \\
\quad \text{where } s = p.PName \text{ and } p.CustName = "CitiBank"
\]

(INV1) is applicable and \(Q\) rewrites to:

\[
(Q') \quad \text{select struct}(PN : s, PB : p.Budg, DN : Dept[d].DName) \\
\quad \text{from dom Dept d, Dept[d].DProjs s, Proj p} \\
\quad \text{where } s = p.PName \text{ and } p.CustName = "CitiBank" \text{ and } p.PDept = Dept[d].DName
\]

No other constraint is then applicable to \(Q'\) (rewriting \(Q'\) with any of the constraints would introduce redundancy and the chase avoids this). Now, we chase \(P_1\). (RIC2) is applicable and \(P_1\) rewrites to:

\[
(P_1') \quad \text{select struct}(PN : p.PName, PB : p.Budg, DN : p.PDept) \\
\quad \text{from Proj p, dom Dept d} \\
\quad \text{where } p.CustName = "CitiBank" \text{ and } p.PDept = Dept[d].DName
\]

(INV2) becomes now applicable and \(P_1'\) rewrites to (and this is where the chase stops):

\[
(P_1'') \quad \text{select struct}(PN : p.PName, PB : p.Budg, DN : p.PDept) \\
\quad \text{from Proj p, dom Dept d, Dept[d].DProjs s} \\
\quad \text{where } p.CustName = "CitiBank" \text{ and } p.PDept = Dept[d].DName \text{ and } s = p.PName
\]

It is easy to see now that \(Q'\) and \(P_1''\) are equivalent (under all instances because one can find two homomorphisms from one into the other, see also Theorem 3.3.9 in Chapter 3. The two queries are in fact isomorphic.). But, since the rewritings preserve equivalence under constraints, it follows that \(Q\) and \(P_1\) are equivalent under the constraints.

We point out however that in our optimization algorithm we do not check for equivalence of two queries by chasing the queries themselves. Since we always verify whether a query is equivalent to one of its subqueries\(^3\) is enough to check whether a certain constraint \(\delta\) guaranteeing this equivalence is implied by the set of constraints \(D\). To test this implication we chase then \(\delta\) with \(D\) (see also Chapter 3 for details on chasing constraints and dependency implication).

### 2.2 Backchase Minimization

There are two main kinds of minimization that have been considered so far in the literature, and we illustrate them below via some simple relational examples.

\(^2\)This bound could be used as a heuristic for stopping the chase when termination is not guaranteed.

\(^3\)See the backchase step in the next section.
• **trivial minimization** (also known as tableau minimization [AHV95])

\[
(Q) \quad \text{select} \quad \text{struct}(A = r_1.A) \\
\quad \text{from} \quad R_{r_1}, R_{r_2} \\
\text{where} \quad r_1.A = r_2.A
\quad \rightarrow \quad (Q') \quad \text{select} \quad \text{struct}(A = r_1.A) \\
\quad \text{from} \quad R_{r_1}
\]

It is not hard to see that \(Q'\) is equivalent to \(Q\) because there exists a containment mapping (homomorphism) \(h\) from \(Q\) into \(Q'\) (and an identity one from \(Q'\) into \(Q\)). The same existence of the homomorphism \(h\) is in fact equivalent to saying that the following constraint is true in all instances (it is trivial, see [BV84b] for definition of trivial tuple-generating dependencies TGDs and also Chapter 3 for definition of trivial embedded path-conjunctive dependencies EPCDs and their characterization via homomorphisms):

\[
\forall (r_1 \in R) \exists (r_2 \in R) r_1.A = r_2.A
\]

Evaluation of \(Q'\) is cheaper than evaluation of \(Q\) under any cost model.

• **join elimination** [CGK+99, SO89]

\[
(Q) \quad \text{select} \quad \text{struct}(A = r.A) \\
\quad \text{from} \quad R, S, s \\
\text{where} \quad r.A = s.A
\quad \rightarrow \quad (Q') \quad \text{select} \quad \text{struct}(A = r.A) \\
\quad \text{from} \quad R
\]

\(Q'\) is equivalent to \(Q\) provided that the following constraint is true:

\[
\forall (r \in R) \exists (s \in S) r.A = s.A
\]

As opposed to trivial minimization, the relations in the from clause of \(Q\) can be different. The cost of evaluating \(Q'\) is in many cases cheaper than the cost of evaluating \(Q\). However, this is not always true, and we postpone a full discussion for Chapter 6. Typically the constraint that guarantees the equivalence is a referential integrity constraint stated explicitly in the schema. However, the transformation can be done in a more general way, by looking also at constraints that are implied by the constraints in the schema.

Both transformations above rewrite a query into a query with fewer joins (a subquery, as we shall see in a moment), and since joins are the most expensive operation, the savings in execution time can be significant. The backchase minimization algorithm that we introduce in this section unifies and generalizes the two techniques in two dimensions: it applies to our more general language with dictionaries and nested sets, and it uses implied constraints (in addition to trivial constraints or constraints in the schema).

**Subquery.** The notion of subquery is central to the minimization component of our optimizer and needs to be defined carefully. We define, informally (see Chapter 4 for full definition), a subquery \(Q'\) of a query \(Q\) as follows:

• the from clause of \(Q'\) is a subset of the from clause of \(Q\) (that is, we eliminate one or more scans from \(Q\)),

• the select clause of \(Q'\) is an equivalent rewriting of the select clause of \(Q\) (that doesn’t use variables that have been eliminated), and

• the where clause of \(Q'\) is implied by the conditions in the where clause of \(Q\)

In Chapter 3 we show how we can decide in polynomial time, when the conditions in the where clause are only equalities, whether there exists such an equivalent rewriting required by the second condition, or whether the implication required by the third condition holds. Here is a simple example. The following query \(Q'\):

\[
\text{select} \quad \text{struct}(A = x_2.A) \\
\text{from} \quad R_0, x_2, R_0, x_3 \\
\text{where} \quad x_2.A = x_3.A
\]

is a subquery of
Let $D$ be a set of constraints and let $Q$ be a query of the form:

```plaintext
select $O(x,y)$
from $R_1, x_1, \ldots, R_y, \ldots, R_m x_m$
where $C(x,y)$
```

and let $Q'$ be a strict subquery obtained from $Q$ by eliminating the scan $R_y$ (if there exists such $Q'$):

```plaintext
select $O'(\bar{x})$
from $R_1, x_1, \ldots, R_m x_m$
where $C'(\bar{x})$
```

Then the rewrite $Q \rightarrow Q'$ is a backchase step provided that the following constraint\(^*\) is implied by $D$:

$$(\delta) \forall (x_1 \in R_1) \ldots \forall (x_m \in R_m) \left[ C'(\bar{x}) \Rightarrow \exists (y \in R) C(\bar{x}, y) \right]$$

\(^*\)The complete version of the backchase step uses a stronger constraint than $\delta$, see further Chapter 4. Also, it requires the PC restriction, under which we prove that the backchase can provide a complete enumeration of equivalent subqueries.

Figure 2.2: A backchase step.

```plaintext
select struct($A = x_1.A$)
from $R_1, x_1, R_2, x_2, R_3, x_3$
where $x_1.A = x_2.A$ and $x_1.A = x_3.A$
```

obtained by eliminating the scan $R_1 x_1$. Here $x_2.A$ is an equivalent rewriting for $x_1.A$ in the select clause of $Q'$ (the equality $x_1.A = x_2.A$ is in fact part of the where clause of $Q$), while the equality $x_2.A = x_3.A$ is implied (via transitivity) by the equalities in the where clause of $Q$. In general, because of dictionary operations, transitivity is not enough for this kind of reasoning, and in Chapter 3 we give a congruence-closure internal representation of a query (the canonical instance) on which complete checking of such conditions is possible.

We call a subquery $Q'$ of $Q$ a strict subquery of $Q$ if the from clause of $Q'$ is a strict subset of the from clause of $Q$. We call a subquery $Q'$ of $Q$ a maximal subquery of $Q$ if the where clause of $Q'$ contains all the conditions involving only variables bound in the from clause of $Q'$ that can be implied from the conditions in the where clause of $Q$. When we have only equalities maximality is a decidable condition (in PTIME).

A final remark is that, from the way we defined a subquery $Q'$ of $Q$, it is always the case that $Q$ is contained in $Q'$. Whenever the from clause of $Q$ is instantiated to some actual values such that the where clause is satisfied and a tuple $t$ is emitted in the answer of $Q$, the same instantiation satisfies the where clause of $Q'$ and, moreover, the same tuple $t$ is emitted.

**Backchase Minimization Algorithm.** We introduce first (in a simplified form) the backchase step and then we show how we can use this in an algorithm for enumerating minimal subqueries of a given query. In Chapter 4 we will give the more general definition of a backchase step that guarantees completeness of the above enumeration.

A backchase step, shown in figure 2.2, tries to eliminate some scan from $Q$ while preserving equivalence under $D$. The above constraint implies that the two queries are equivalent; in fact it is easy to see that there is a chase step\(^4\) with $\delta$ from $Q'$ to a query equivalent to $Q$ (under all instances).

\(^4\)Thus, the name backchase.
In general, we can define a backchase step as a rewrite that eliminates more than one scan at a time (and this will be actually used in the bottom-up backchase algorithm of Chapter 6). In chapter 4 we show that a backchase step that eliminates \( k \) scans can always be reduced to a sequence of \( k \) backchase steps removing only one scan each. Trying to see whether constraint \( (\delta) \) is implied by \( D \) is done using with the chase. The backchase minimization algorithm can now be described as follows:

**Algorithm 2.2.1 Minimize\((Q, D)\):**

input query \( Q \), constraints \( D \)
for every scan \( R_i \ x_i \) in the from clause of \( Q \) do
  compute the maximal subquery \( Q' \) of \( Q \) that eliminates scan \( R_i \ x_i \);
  if there exists such \( Q' \) then
    if \( Q' \) is equivalent to \( Q \) under \( D \) then
      Minimize \((Q', D)\);
    if no equivalent subquery \( Q' \) was found then output \( Q \);

Thus the algorithm essentially tries to apply backchase steps removing one scan each in all possible ways until no such step is possible anymore. The algorithm is essentially an exponential, **top-down**, enumeration of all equivalent subqueries of \( Q \) that stops when equivalence is not preserved. We point out that, as opposed to the relational tableau minimization, where the minimal form is unique up to isomorphism [AHV95], the backchase minimization can end up with several minimal forms.

**Bottom-up Backchase Minimization.** The backchase search for minimal subqueries can also be done in a **bottom-up** fashion by exploring subqueries with one scan first, then subqueries with two scans and so on, until equivalent subqueries are found. This strategy, while still exponential in the worst case, explores non-equivalent subqueries and it can outperform the top-down approach when the minimal forms are small compared to the input query, i.e. the original query has high redundancy. In the case when the original query is not redundant at all then the top-down approach outperforms the bottom-up because it stops after exploring subqueries with one less scan, while the bottom-up approach would have to explore most of the search tree. However, the bottom-up approach has the additional, important, advantage of being able to be mixed with **cost-based pruning**. This will be the central idea of Chapter 6 where we show, experimentally, the benefits of bottom-up backchase with cost-based pruning.

Backchase minimization alone only looks at subqueries of \( Q \) and it doesn’t take into account: 1) additional logical schema info that might be useful in reducing further the number of joins in the query, and 2) relevant physical access paths (not appearing explicitly in the query) that can be used in answering the query. In section 2.4 we show how we can use the chase to discover such logical and physical elements before performing backchase minimization, and we show the benefits of this strategy.

### 2.3 Physical Structures as Constraints

We show here how typical physical access structures are captured by constraints.

**Indexes and classes** The dictionary construction operation allows us to define explicitly **primary** and **secondary indexes** such as \( I \) and \( SI \) from example 1.2.1: 

\[
I \overset{\text{def}}{=} \begin{array}{c}
\text{dict } k \text{ in } \Pi_{\text{PName}}(\text{Proj}) \\
\Rightarrow \\
\text{element}(\text{select } p \text{ from } \text{Proj } p \text{ where } p.\text{PName} = k)
\end{array}
\]

\[
SI \overset{\text{def}}{=} \begin{array}{c}
\text{dict } k \text{ in } \Pi_{\text{CustName}}(\text{Proj}) \\
\Rightarrow \\
(\text{select } p \text{ from } \text{Proj } p \text{ where } p.\text{CustName} = k)
\end{array}
\]

28
Here $\Pi_A(R)$ is a shorthand for the query that projects relation $R$ on $A$ and $\text{element}(C)$ is the OQL operation that extracts the unique element of the singleton collection $C$ and fails if $C$ is not a singleton. Luckily, the use of constraints allows us to avoid using this messy operation. Both primary and secondary indexes are completely characterized by constraints, e.g., for $I$ we use $(P1, P2)$ and for $SI$ we use $(S1, S2, S3)$ where

\[
\begin{align*}
(P11) & \forall (p \in \text{Proj}) \exists (i \in \text{dom}I) \ [i = p.PName \ and \ I[i] = p] \\
(P12) & \forall (i \in \text{dom}I) \exists (p \in \text{Proj}) \ [i = p.PName \ and \ I[i] = p] \\
(S11) & \forall (p \in \text{Proj}) \exists (k \in \text{dom}SI) \exists (t \in SI[k]) \ [k = p.CustName \ and \ p = t] \\
(S12) & \forall (k \in \text{dom}SI) \forall (t \in SI[k]) \exists (p \in \text{Proj}) \ [k = p.CustName \ and \ p = t] \\
(S13) & \forall (k \in \text{dom}SI) \exists (t \in SI[k]) \text{true}
\end{align*}
\]

Notice that each of $(P11, P12, S11, S12)$ is an inclusion constraint while $(S13)$ is a non-emptiness constraint. In fact, taken together, the pairs of inclusion constraints also state an inverse relationships between the dictionaries and Proj. Similarly, we can represent the relationship between the class $\text{Dept}$ and the dictionary implementing it, $\text{Dept}$, with two constraints. We show one of them (the other is "inverse"):

\[
(\delta_{\text{Dept}}) \ \forall (d \in \text{depts}) \forall (s \in d.DProjs) \exists (d' \in \text{domDept}) \exists (s' \in \text{Dept}[d'.DProjs]) \ [d = d' \ and \ s = s']
\]

**Hash tables** An interesting extension to this idea are hash tables. A hash table for a relation can be viewed as a dictionary in which keys are the results of applying the hash function to tuples in the relation, while the entries are the buckets (sets of tuples). Thus, a hash table can be represented similarly to secondary indexes. A hash table differs from an index because it is not usually materialized, however a hash-join algorithm would have to compute it on the fly. In our framework, we can rewrite join queries into queries that correspond to hash-join plans, provided that the hash-table exists, in the same way we rewrite queries into plans that use indexes.

**Materialized views/Source capabilities** Materialized conjunctive or PSJ (project-select-join) views, or cached results of conjunctive/PSJ queries over a relational schema $R$ have been used in answering other conjunctive/PSJ queries over $R$ [YL87, CR94, CKPS95, LMSS95, Qia96]. We consider the more general form

\[
v \overset{\text{def}}{=} \text{select } O(\bar{x}) \text{ from } \bar{P} \bar{x} \text{ where } B(\bar{x})
\]

Here we denote by $\bar{P} \bar{x}$ an arbitrary sequence of bindings $P_1 x_1, \ldots, P_n x_n$, by $O(\bar{x})$ we denote the fact that variables $x_1, \ldots, x_n$ can appear in the output record $O$ (and similar for $B(\bar{x})$). Like indexes, such structures can be characterized by constraints, namely:

\[
\begin{align*}
\delta_v & \overset{\text{def}}{=} \forall (\bar{x} \in \bar{P}) \ [B(\bar{x}) \Rightarrow \exists (v \in V) \ O(\bar{x}) = v] \\
\delta'_v & \overset{\text{def}}{=} \forall (v \in V) \ \exists (\bar{x} \in \bar{P}) \ [B(\bar{x}) \ and \ O(\bar{x}) = v]
\end{align*}
\]

Note that $\delta_v$ corresponds to the inclusion $\text{select } O(\bar{x}) \text{ from } \bar{P} \bar{x} \text{ where } B(\bar{x}) \subseteq V$ while $\delta'_v$ corresponds to the inverse inclusion. The two are, in general, constraints between the physical and the logical schema.

In example 1.2.1, JI is expressed as such a view and $\delta_{JI}$ is (we don’t show here $\delta'_{JI}$):

\[
(\delta_{JI}) \ \forall (d \in \text{domDept}) \forall (s \in \text{Dept}[d].DProjs) \forall (p \in \text{Proj}) \\
[ s = p.PName \Rightarrow \exists (j \in JI) \ j.DID = d \ and \ j.PN = p.PName ]
\]

Source capabilities often used in information integration systems can be described by either such materialized views or by dictionaries modeling the binding patterns of [RSU95].
Join indexes [Val87] were introduced as a technique for join navigation and shown to outperform even hybrid-hash join in most cases with high join selectivity. The technique assumes that tuples have unique, system-generated identifiers called surrogates (if the relations have keys, these can be used instead), and that the relations are indexed on surrogates. A join index for the join of relations $R$ and $S$, denoted $J_{RS}$, is a precomputed binary relation associating the surrogates of $R$-tuples to surrogates of $S$-tuples whenever these tuples agree on the join condition. The join is computed by scanning $J_{RS}$ and using the surrogates to index into the relations.

Since indexes are not first class citizens of relational algebra / OQL, [Val87]'s join evaluation algorithm cannot be expressed in these query languages. In contrast, the query $P_3$ from example 1.2.1 is a candidate plan describing precisely the above algorithm: it iterates over materialized view $V_I$ and explicitly indexes by name into $I$ (the primary index of $\text{Proj}$ on key attribute $\text{Name}$), and by oid into the dictionary $\text{Dept}$ representing the class of departments. The success in expressing this query plan stems from the ability of expressing index lookup in our language. We can therefore fully describe a join index by a triple consisting of a materialized binary relation view and two indexes. In our example, the join index for joining $\text{Dept}$ with $\text{Proj}$ is $(\text{Dept}, I, J_I)$.

Access support relations [KM90a, KM90b] generalize path indexes [MS86, Ber94, BK89] and translate the join index idea from the relational to the object model, generalizing it from binary to n-ary relations. An access support relation (ASR) for a given path is a separate precomputed relation that explicitly stores the oids of objects related to each other via the attributes of the path. As with join indexes, ASRs are used to rewrite navigation style path queries to queries which scan the access support relation, project out the oids of the source and target objects for the path and dereference these oids to access the objects. The oid dereferencing operation is performed implicitly in OQL, which therefore can express this algorithm, but fails to express its join index based relational counterpart because of the lack of explicit dictionary lookup operations. In our approach, access support relations and join indexes are unified using dictionaries both for representing classes with extents and indexes. Analogous to join indexes, we model access support relations for a given path as the materialized relation storing the oids along the path, together with the dictionaries modeling the classes of the source and target objects of the path.

Gmaps [TSI96] specify physical access structures as materialized PSJ views over logical schema. [TSI96] gives a sound (not complete) algorithm for rewriting PSJ queries against the logical schema in terms of materialized gmaps. Our framework subsumes gmaps: PSJ queries alone (in the absence of dictionaries) only approximate index structures with their graph relations (binary relations associating keys to values, which are called input respectively output nodes in gmap terminology). In contrast, we capture the intended meaning of a general gmap definition using dictionaries:

$$\text{dict } \exists \bar{x} \in (\text{select } O_1(\bar{x}) \text{ from } \tilde{P} \bar{x} \text{ where } B(\bar{x})) \Rightarrow \text{select } O_2(\bar{x}, \bar{z}) \text{ from } \tilde{P} \bar{x} \text{ where } B(\bar{x})$$

Here $O_1, O_2$ have flat record type (as outputs of PSJ queries in the original definition). Notice the correlation between the domain and range of the dictionary: they are given by queries which differ only in the projection of the select clause, a limitation resulting from the gmap definition language. We can generalize gmaps by overcoming this limitation and supplying different queries for the domain and range of our dictionaries. Similarly to the case of secondary indexes, we can model this generalized form of gmaps with dependencies.

In the PSJ modeling of gmaps, queries rewritten in terms of gmaps perform relational joins and don’t explicitly express index lookups. Just by looking at the rewritten query, the optimizer cannot decide whether a join should be implemented as such or in an index-based fashion. In other words, PSJ queries used in the gmap approach are not as close to query plans as queries in our language.
2.4 Chase: Discovery of Relevant Physical Sources

As we saw in section 2.1 applying a chase step with constraint \(d\) to a query \(Q\) has the result of adding to the from and where clauses of \(Q\) the existentially quantified part of \(d\). The chase step is applicable only if there is a match between the universally quantified part of \(d\) and \(Q\). Thus, the chase step rewrites \(Q\) into an equivalent \(Q'\) that brings in new elements of the schema that are relevant to the query. For example, a query that has a scan over \(\text{Proj}\) can be rewritten into one that adds a scan over the primary index \(I\) for \(R\) by chasing with \(PI1\) of section 2.3. On the other hand, for a query that doesn’t mention \(\text{Proj}\) the chase step with \(PI1\) will not be applicable (\(I\) is not relevant to the query in that case). The reader can imagine more complicated examples in which by chasing with constraints such as \(\delta\) we add views that are relevant to the query (can be used in answering the query).

Thus, the first phase of the C&B optimization strategy is the chase phase. The role of it is to bring, in a systematic way, all the relevant physical structures into the logical query. The result of the chase applied to a logical query \(Q\) is a larger query that holds in one place essentially all possible physical plans for \(Q\) expressible in our language. We call this larger query the universal plan and we usually denote it by \(U\).

**Example 2.4.1** On our \(\text{Proj-Dept}\) schema of example 1.2.1 and with the constraints describing the indexes \(I, SI\) and \(JI\) from section 2.3, we illustrate how the chase phase produces the universal plan for \(Q\). We have seen already in example 2.1.1 that \(Q\) rewrites in one chase step with \(\text{(INV1)}\) to

\[
\text{(Q')} \quad \text{select } \text{struct(PN : } s, \text{ PB : } p.\text{Budg, DN : Dept[d].DName)} \\
\text{from } \text{dom Dept d, Dept[d].DProjs s, Proj p} \\
\text{where } s = p.\text{PName and p.CustName = "CitiBank" and p.PDept = Dept[d].DName}
\]

With the additional constraints describing the three indexes, \(I, SI,\) and \(JI,\) the chase doesn’t stop here. By chasing with \(\delta\), then with \(SI\) and \(PI1\), the universal plan \(U\) is obtained as follows:

\[
\text{select } \text{struct(PN : } s, \text{ PB : } p.\text{Budg, DN : Dept[d].DName)} \\
\text{from } \text{dom Dept d, Dept[d].DProjs s, Proj p, JI j, dom SI k, SI[k] t, dom I i} \\
\text{where } s = p.\text{PName and p.CustName = "CitiBank" and p.PDept = Dept[d].DName} \\
\text{and } j.DDID = d \text{ and } j.PN = p.\text{PName} \\
\text{and } k = p.\text{CustName and } p = t \text{ and } i = p.\text{PName and } I[i] = p
\]

The universal plan still references elements of the logical schema, and it is not an actual plan (to be evaluated). However, in the second phase we perform backchase minimization and retain only the minimal subqueries of the universal plan that refer to physical schema elements only. Putting the two phases together, we obtain what we call the C&B (chase and backchase) enumeration of candidate plans.

2.5 Examples of C&B Enumeration

**Example 2.5.1 (Semantic optimization)** Consider the universal plan \(U\) obtained in example 2.4.1. One minimization path during backchasing \(U\) consists of

1. eliminate the scan \(\text{dom I i}\) (and the two conditions \(i = p.\text{PName and } I[i] = p\)) with a backchase step using a constraint implied by \(PI1\):

\[
\forall(d \in \text{dom Dept}) \forall(s \in \text{Dept[d].DProjs}) \forall(p \in \text{Proj}) \forall(j \in \text{JI}) \forall(k \in \text{dom SI}) \forall(t \in \text{SI[k]}) \\
[ s = p.\text{PName and p.CustName = "CitiBank" and p.PDept = Dept[d].DName} \\
\]
and \( j.\text{DOID} = d \) and \( j.\text{PN} = p.\text{Name} \) 
and \( k = p.\text{CustName} \) and \( p = t \) 
\[ \Rightarrow \exists (i \in \text{dom} I) i = p.\text{Name} \text{ and } I[i] = p \]

To see that PI1 implies the above constraint just notice that the latter contains a stronger universally quantified part and more conditions in the left hand side of \( \Rightarrow \).

2. eliminate the scan \( \text{SI}[k] t \) (and the condition \( p = t \)) with a backchase step using a constraint \( \delta \) implied by SI1:

\[
\forall (d \in \text{dom} \text{Dept}) \forall (s \in \text{Dept}[d].\text{DProjs}) \forall (p \in \text{Proj}) \forall (j \in \text{JI}) \forall (k \in \text{dom} \text{SI}) \left[ \begin{array}{c} s = p.\text{Name} \text{ and } p.\text{CustName} = "\text{CitiBank}" \text{ and } p.\text{PDept} = \text{Dept}[d].\text{DName} \\
\text{and } j.\text{DOID} = d \text{ and } j.\text{PN} = p.\text{Name} \\
\text{and } k = p.\text{CustName} \end{array} \right]
\Rightarrow 
\exists (t \in \text{SI}[k]) p = t
\]

A simple way to see why \( \delta \) is implied by SI1 is to consider the following constraint, \( \delta' \):

\[
\forall (p \in \text{Proj}) \forall (k \in \text{dom} \text{SI}) \left[ k = p.\text{CustName} \Rightarrow \exists (t \in \text{SI}[k]) p = t \right]
\]

which is implied by SI1. Since the \( \delta \) can be obtained from \( \delta' \) by adding more bindings in the universal part and more conditions in the left hand side of the implication, \( \delta \) is implied by \( \delta' \) and, therefore, is implied by SI1. In chapter 3 we show how we can use the chase to implement such reasoning about constraints.

In the same fashion as above:

3. eliminate the scan \( \text{dom} \text{SI} k \) with a backchase step using also a constraint implied by SI1. We remark here that steps 2 and 3 above could have been combined into one backchase step eliminating two scans at once.

4. eliminate the scan \( \text{JI} j \) with a backchase step using a constraint implied by \( \delta_{JI} \). The query \( Q' \) obtained at this point is one resembling the original query \( Q \):

\[
(Q') \textbf{select } \textbf{struct}(\text{PN} : s, \text{PB} : p.\text{Budg}, \text{DN} : \text{Dept}[d].\text{DName}) \\
\text{from } \text{dom} \text{Dept} d, \text{Dept}[d].\text{DProjs} s, \text{Proj} p \\
\textbf{where } s = p.\text{Name} \text{ and } p.\text{CustName} = "\text{CitiBank}" \text{ and } p.\text{PDept} = \text{Dept}[d].\text{DName}
\]

Indeed, \( Q \) can be obtained from \( Q' \) by removing the equality \( p.\text{PDept} = \text{Dept}[d].\text{DName} \) from the \( \textbf{where} \) clause of \( Q' \). The elimination of this equality is justified by the fact that the equality is implied: see INV1 on the logical schema Proj-Dept. However, the backchase minimization algorithm focuses on eliminating joins rather than conditions in the \( \textbf{where} \) clause. Handling of conditions can then be done on (join-) minimal forms in a stage that follows the backchase minimization. This treatment must be done based on cost and includes: elimination of conditions (e.g. expensive predicates), pushing selections towards sources, etc. Of course, we use here the assumption that as long as a query is not join-minimal, removing a join has a higher benefit than removing a condition from the \( \textbf{where} \) clause. This may not be always true, but we use it as an important heuristic to limit the search space for plans. The alternative would be to generalize the backchase step to consider subqueries that are not strict (with significant increase in complexity).

Coming back to our example, since \( Q' \) has still redundant joins in the \( \textbf{select} \) clause, the backchase minimization can continue as follows:

5. eliminate the scan \( \text{Dept}[d].\text{DProjs} s \) with a backchase step with a constraint implied by (INV2). Note that the equality \( s = p.\text{Name} \) enables the replacement of \( s \) in the \( \textbf{select} \) clause with the equivalent \( p.\text{Name} \).

6. finally, eliminate the scan \( \text{dom Dept} d \) with a backchase step with a constraint implied by (RIC2). Here the equality \( p.\text{PDept} = \text{Dept}[d].\text{DName} \) allows the replacement of \( \text{Dept}[d].\text{DName} \) in the \( \textbf{select} \) clause with \( p.\text{PDept} \). The result of this minimization path is:
from Proj p
where p.CustName = "CitiBank"

which is the plan $P_1$ discussed in example 1.2.1.

Thus, by choosing a certain path during the minimization phase our algorithm is able to perform semantic optimization, even though additional physical elements were considered along the way. The next example shows how choosing a different path the algorithm is able to produce plans that use such physical elements.

Example 2.5.2 (Mapping to indexes) Consider the following variation of the minimization path of the previous example: eliminate scans $\text{dom} \ I \ i, \ JI \ j, \ \text{Dept}[d] \ \text{Depro}s \ s, \ \text{domDept} \ d$ while keeping the scans involving $SI$. The result of these elimination steps is the following query $Q_1$:

$$\begin{align*}
\text{select struct}(\text{PN} : p.\text{PName}, \text{PB} : p.\text{Budg}, \text{DN} : p.\text{PDept}) \\
\text{from} \ Proj \ p \\
\text{where} \ p.\text{CustName} = "\text{CitiBank}" \ \text{and} \ k = p.\text{CustName} \ \text{and} \ p = t
\end{align*}$$

The next backtack step, in which we choose to eliminate $\text{Proj} p$, needs to be explained carefully. While in general there is more than one subquery of $Q_1$ that can be obtained by eliminating the scan $\text{Proj} p$, the backtack minimization algorithm considers only (and for good reason) the maximal subquery, call it $Q_2$, of $Q_1$. In other words, the $\text{where}$ clause of $Q_2$ must contain all the equalities that can be inferred from the $\text{where}$ clause of $Q_1$ and, of course, use only variables $k$ and $t$. Here, the equality $k = "\text{CitiBank}"$ is implied from the equalities $p.\text{CustName} = "\text{CitiBank}"$ and $k = p.\text{CustName}$, thus we put it in the $\text{where}$ clause of $Q_2$. There is one more equality involving $k$ and $t$ that we can infer: $t.\text{CustName} = k$.

We also need to find "equal" replacements for $p.\text{PName}$, $p.\text{Budg}$ and $p.\text{PDept}$ in the $\text{select}$ clause. From the condition $p = t$ in the $\text{where}$ clause of $Q_1$ we derive that $p.\text{PName} = t.\text{PName}$, $p.\text{Budg} = t.\text{Budg}$ and $p.\text{PDept} = t.\text{PDept}$, and $Q_2$ is thus obtained as follows:

$$\begin{align*}
\text{select struct}(\text{PN} : t.\text{PName}, \text{PB} : t.\text{Budg}, \text{DN} : t.\text{PDept}) \\
\text{from} \ \text{domSI} \ k, \ \text{SI}[k] \ t \\
\text{where} \ k = "\text{CitiBank}" \ \text{and} \ t.\text{CustName} = k
\end{align*}$$

We can check now that $Q_2$ is indeed equivalent to $Q_1$. The constraint that guarantees this equivalence is the following:

$$\forall(k \in \text{domSI}) \ \forall(t \in \text{SI}[k]) \\
\left[ k = "\text{CitiBank}" \ \text{and} \ t.\text{CustName} = k \\
\Rightarrow \\
\exists(p \in \text{Proj}) \ p.\text{CustName} = "\text{CitiBank}" \ \text{and} \ k = p.\text{CustName} \ \text{and} \ p = t \right]$$

The constraint is then implied by $SI2$. Failing to include the equality $k = "\text{CitiBank}"$ in the $\text{where}$ clause of $Q_2$ would have resulted in a subquery not equivalent to $Q_1$: the same constraint as above from which we remove the mentioned equality from the left-hand side of $\Rightarrow$ is no longer implied by $SI2$. Thus the backtack minimization would fail to eliminate $\text{Proj} p$.

The last query is a scan-minimal query (no scan can be eliminated anymore) and it is one of the outputs of the backtack minimization phase. However, there are some additional things that we can do. First, as we mentioned in the previous example, we can focus now on eliminating equalities. This can only be done in cost-based fashion. In some cases keeping an equality rather than eliminating it may be useful to filter out some

---

Footnote: This is a congruence rule for records. See Chapter 3 for a detailed explanation of how we can use a congruence-closure method to facilitate reasoning about equality.
intermediate result thus decreasing the overall cost. In some other cases, an expensive condition that can be
eliminated can significantly reduce the cost. In our example, the equality $t.\text{CustName} = k$ can be eliminated
and it is likely that we obtain a better plan if we do so. The elimination is justified by the following constraint,
implied by $SI2$:

$$\forall (k \in \text{dom}SI) \forall (t \in SI[k]) \ k = t.\text{CustName}$$

In general, we can perform elimination of equalities in a systematic way, by using a variation of the backchase
step in which the constraint $\delta$ has no existential quantification (i.e. $\delta$ is an EGD).

Finally, we can rewrite the resulting query into the following, more operational (and likely to have a faster
evaluation), version that uses the non-failing lookup operator:

$$(P_3) \quad \text{select, struct}(PN: p.PName, PB: p.Budget, DN: p.DName)
\text{from} \ SI["Citibank"] \ p$$

which is precisely one of the plans shown in example 1.2.1. In a similar way we can obtain the plan $P_3$ and
others such as a scan over $\text{dom}I$.

In the spirit of the previous examples, the next two examples demonstrate the advantages of chasing with
existing logical constraints (key constraints, referential integrity constraints) before performing the backchase
minimization. The first example shows that even a simple chase with key constraints can have great benefits.
The example illustrates also the kind of redundancy that can appear while composing queries with views.

**Example 2.5.3 (Minimization under key constraints)** Consider a logical schema consisting of a flat relation
$\text{Students}$ and a nested relation $\text{Books}$ shown in figure 2.3. The types $\text{BookID}$ and $\text{CopyID}$ are some fresh
types used for book ids and book copy ids. Each book has a set of copies, and the information associated to each
copy consists of a copy id, a boolean flag saying whether the copy is currently borrowed or not, and the name of
the borrower if the copy is borrowed. We assume that students are identified by their names, while books by
their $\text{bookId}$ attribute which is a primary key.

```plaintext
Students : Set < Struct{
    string name;
    string major;
    ... } > ;
Books : Set < Struct{
    BookID bookId;
    string title;
    Set < Struct{
        CopyID copyId;
        boolean borrowed;
        string borrower;
    } > copies;
    string publisher;
    ... } >
primary key bookId;
```

Figure 2.3: A complex-value $\text{Books-Students}$ logical schema

Suppose now that we have the following non-materialized view defining the set of all book ids that are currently
available:

$$(V) \quad \text{select, struct}(bid: b.bookId)
\text{from} \ Books b, b.copies c
\text{where} \ c.borrowed = \text{false}$$

The user asks the following query that retrieves the titles of all books currently available and it does so by using
the previously defined view $V$:

$$(Q) \quad \textbf{select} \quad \textbf{struct}(\text{title} : b.\text{title})$$
$$\quad \textbf{from} \quad \text{Books} \ b, \ V \ v$$
$$\quad \textbf{where} \quad v.\text{bid} = b.\text{bookId}$$

Since the view is non-materialized the optimizer is likely to perform first the composition of $Q$ with $V$ rather than materializing the view and then optimizing $Q$ as it is. The result of the composition is the following query:

$$(Q') \quad \textbf{select} \quad \textbf{struct}(\text{title} : b.\text{title})$$
$$\quad \textbf{from} \quad \text{Books} \ b, \ \text{Books} \ b', \ b'.\text{copies} \ c$$
$$\quad \textbf{where} \quad b'.\text{bookId} = b.\text{bookId} \ \text{and} \ c.\text{borrowed} = \text{false}$$

It is quite intuitive that $Q'$ contains a redundant scan over $\text{Books}$. However, if we try to apply the minimization algorithm on $Q'$, we don’t eliminate anything. Here is why. Suppose we try to eliminate the scan $\text{Books} \ b$, thus we are looking for an equivalent subquery of $Q'$ that scan. We need to find an equal replacement for $b.\text{title}$ that appears in the $\textbf{select}$ clause. However, the obvious candidate for the replacement, $b'.\text{title}$, cannot be found equivalent to $b.\text{title}$ just by looking at the $\textbf{where}$ clause of the query. The reason is that we cannot infer from $b.\text{bookId} = b'.\text{bookId}$ that $b.\text{title} = b'.\text{title}$. For the same reason, relational tableaux minimization (which is a particular case of back-chase minimization) fails to do any reduction in such a situation. What we need is the key constraint information, and this information is not part of the query. The solution is to chase first $Q'$ with the key constraint for $\text{bookId}$, and then to apply the minimization algorithm. The result of chasing $Q'$ with the key constraint

$$\forall(b \in \text{Books}) \forall(b' \in \text{Books}) \quad b.\text{bookId} = b'.\text{bookId} \ \Rightarrow \ b = b'$$

is the query:

$$(U) \quad \textbf{select} \quad \textbf{struct}(\text{title} : b.\text{title})$$
$$\quad \textbf{from} \quad \text{Books} \ b, \ \text{Books} \ b', \ b'.\text{copies} \ c$$
$$\quad \textbf{where} \quad b'.\text{bookId} = b.\text{bookId} \ \text{and} \ c.\text{borrowed} = \text{false} \ \text{and} \ b = b'$$

Now backchasing $U$ with a constraint implied by the following trivial constraint:

$$\forall(b' \in \text{Books}) \exists(b \in \text{Books}) \quad b = b'$$

results in:

$$(Q'_m) \quad \textbf{select} \quad \textbf{struct}(\text{title} : b'.\text{title})$$
$$\quad \textbf{from} \quad \text{Books} \ b', \ b'.\text{copies} \ c$$
$$\quad \textbf{where} \quad c.\text{borrowed} = \text{false}$$

which is a better query than $Q'$. The idea of chasing with functional dependencies to create opportunities for successful minimization appears in [AHV95] in the context of relational tableaux.

**Example 2.5.4** We use the same schema of figure 2.3 for this example. Consider the following query asking for titles and copy ids of all books that are currently borrowed

$$(Q) \quad \textbf{select} \quad \textbf{struct}(\text{title} : b.\text{title}, \text{copyId} : c.\text{copyId})$$
$$\quad \textbf{from} \quad \text{Books} \ b, \ b.\text{copies} \ c$$
$$\quad \textbf{where} \quad c.\text{borrowed} = \text{true}$$

and assume the following scenario. The $\text{Books}$ is a very large table and it takes hours before $Q$ returns an answer which happens to be small in size (compared to the size of $\text{Books}$, there are not so many people borrowing books!). An inverse path index, storing for each student the title and copy ids of the books borrowed by the student, is

---

35
maintained as part of the physical implementation. The definition of this index is a through a dictionary query:

\[
\text{I} \triangleq \text{dict } k \text{ in } \Pi_{\text{Name}}(\text{Students}) \\
\Rightarrow \text{select } \text{struct}(\text{title}: b.\text{title}, \text{copyId}: c.\text{copyId}) \\
\text{from } \text{Books } b, \text{b.copies } c \\
\text{where } c.\text{borrowed} = \text{true and } c.\text{borrower} = k
\]

The index is smaller than Books and we would like, if possible, to use the index rather than Books. In the C&B optimizer, the index is represented internally through two constraints. Here is one of them, essentially stating that for every student and his borrowed books there must exist an entry in the index that associates with the name of the student the titles and copy ids of the borrowed books.

\[
(\delta_I) \quad \forall(s \in \text{Students}) \forall(b \in \text{Books}) \forall(c \in b.\text{copies}) \\
[c.\text{borrowed} = \text{true and } c.\text{borrower} = s.\text{name}] \\
\Rightarrow \exists(n \in \text{dom } I) \exists(t \in I[n]) \\
\quad t.\text{title} = b.\text{title and } c.\text{copyId} = t.\text{copyId and } n = s.\text{name}
\]

To rewrite \(Q\) into a query that uses \(I\) we need to chase \(Q\) with \(\delta_I\). Unfortunately, the constraint is not applicable to \(Q\) because we cannot match the scan over \text{Students} that occurs in the universal part of \(\delta_I\) with any of the scans in \(Q\). On the other hand suppose that we know that all the borrowed books were borrowed by students, that is we assume that the following semantic constraint holds:

\[
(d) \quad \forall(b \in \text{Books}) \forall(c \in b.\text{copies}) \\
[c.\text{borrowed} = \text{true}] \\
\Rightarrow \exists(s \in \text{Students}) s.\text{name} = c.\text{borrower}
\]

Now we can chase \(Q\) with \(d\) to produce:

\[
\text{select } \text{struct}(\text{title}: b.\text{title}, \text{copyId}: c.\text{copyId}) \\
\text{from } \text{Books } b, \text{b.copies } c, \text{Students } s \\
\text{where } c.\text{borrowed} = \text{true and } s.\text{name} = c.\text{borrower}
\]

On this new query the constraint \(\delta_I\) becomes applicable and in one chase step we obtain the universal plan:

\[
\text{select } \text{struct}(\text{title}: b.\text{title}, \text{copyId}: c.\text{copyId}) \\
\text{from } \text{Books } b, \text{b.copies } c, \text{Students } s, \text{dom } I n, I[n] t \\
\text{where } c.\text{borrowed} = \text{true and } s.\text{name} = c.\text{borrower} \\
\quad \text{and } t.\text{title} = b.\text{title and } c.\text{copyId} = t.\text{copyId and } n = s.\text{name}
\]

Without entering in further details, we only say that the backchase minimization produces from the above query two scan-minimal plans. The first one is the same as the original query \(Q\) while the second one is a query using only \(I\):

\[
\text{select } \text{struct}(\text{title}: t.\text{title}, \text{copyId}: t.\text{copyId}) \\
\text{from } \text{dom } I n, I[n] t
\]

Full specification of chase and backchase, and of theorems on which the C&B enumeration is based are left for Chapters 3 and 4.

We have essentially argued in this section that the more constraints one uses during the optimization the better.
are the plans produced. However the increase in the number of constraints being considered increases also the size of the search space (universal plan) on which the backchase minimization works. We will show, in Chapters 5 and 6, that the C&B technique can be efficiently implemented so that it is practical for reasonably large universal plans.
Chapter 3

The Theory of Path-Conjunctive Chase

In this chapter we generalize the relational chase to the path-conjunctive (PC) language with nested sets and dictionaries described in section 3.1, and we give the main results regarding PC query containment, embedded path-conjunctive dependency (EPCD) implication and the path-conjunctive chase. The key new ideas of this chapter are in the canonical instance construction of section 3.2. Section 3.3.3 proves the NP-completeness of PC containment/EPCD validity, extending the result of [CM77]. Section 3.4 is an overview of the path-conjunctive chase and states the main theorems without the proofs. The proofs are relegated to the subsequent sections, and their key ideas are in using the canonical instance to adapt and extend the proof techniques of [BV84b] from the relational chase. We summarize below the main points of interest of this chapter:

- **NP-completeness of PC containment and EPCD validity** and their characterization with homomorphisms: Theorems 3.3.9 and 3.3.11 in section 3.3.3.

- **path-conjunctive chase**: Definition 3.4.2 in section 3.4.

- **terminating chase as a decision procedure** for EPCD implication / PC query containment under EPCDs: Theorem 3.4.4 in section 3.4 with proof in section 3.5.

- **termination of chase for full EPCDs**: Theorem 3.4.6 in section 3.4 with proof in section 3.6.1.

- **complexity analysis for chase with full EPCDs**: Proposition 3.6.9 in section 3.6.2.

- **Church-Rosser and semantic invariance for chase with full EPCDs**: Theorems 3.4.8 and 3.4.9 in section 3.4 with proofs in section 3.6.3.

- **(infinite) chase as a complete proof procedure** for EPCD implication / PC query containment under EPCDs: Theorem 3.4.10 in section 3.4 with proof in section 3.7.

It is worth mentioning that in [PT99] and [PT98] we give an axiomatization that is sound and complete for checking equivalence of PC queries under EPCDs. This axiomatization is based on the monad algebra laws of [LT97] and uses an additional fundamental equivalence law for sets called \textit{idemloop}. Then the soundness of the equational chase that we present in this chapter can be justified as a sequence of finer granularity rewrites that use the above mentioned axiomatization. The completeness of the axioms system is proved via reduction to the PC chase.
3.1 The Path-Conjunctive Language

In this chapter we focus our attention to a restriction of the language introduced in section 1.2. One important restriction is that we do not allow navigation-based queries. We will see, at the end of this section, that typical OO navigation queries are translated by breaking each navigation path into explicit joins.

We call this fragment path-conjunctive queries and dependencies. This is the language for which we prove the completeness results of this chapter and of the next one. Chapter 6 then considers a class of physical plans for PC queries, in which navigation-based plans for PC queries, in addition to explicit join plans, are rediscovered (and used in the optimizer). PC queries and constraints have the same form as in section 1.2, however the paths have some important restrictions.

The following are expressions that can occur within a query. We call them paths.

$$P ::= x \mid R \mid P.A \mid \text{dom } P \mid P[x] \mid \text{true} \mid \text{false}$$

Path-conjunctions (which can occur in the where clause of a query) are conjunctions of equalities between paths:

$$C ::= P_1 = P'_1 \land \cdots \land P_n = P'_n$$

A path-conjunctive (PC) query has the form

$$\begin{array}{c}
\text{select} & \text{struct}(A_1 : P_1, \ldots, A_n : P_n) \\
\text{from} & P_1 x_1, \ldots, P_m x_m \\
\text{where} & C
\end{array}$$

where $$P_i, P'_i$$ are paths, and $$C$$ is a path-conjunction. We will use, to abbreviate, the notation $$\bar{x}$$ to refer to a set of variables $$\{x_1, \ldots, x_n\}$$. A pair $$P_i x_i$$ occurring in the from clause will be called either a scan or a binding. Each path $$P_i$$, of set type, will be called a generator.

The scans of a PC query may depend on each other. In other words, a generator $$P_i$$ may depend on a variable $$x_j$$ with $$j < i$$. This is a usual situation for OO queries: in order to query nested structures, one may need several nested scans that are dependent on each other. This situation is also encountered when secondary indexes are used. Many of the queries in the examples of Chapters 1 and 2 have such dependent scans (and in fact, they are PC queries). In the absence of nested sets and dictionaries, a path-conjunctive query becomes just a conjunctive query [CM77].

An embedded path-conjunctive dependency (EPCD) is a logical formula with the following form:

$$\forall(x_1 \in P_1) \ldots \forall(x_n \in P_n) [ B_1(\bar{x}) \Rightarrow \exists(y_1 \in P'_1) \ldots \exists(y_m \in P'_m) B_2(\bar{x}, y)]$$

where $$P_i, P'_i$$ are paths while $$B_1$$ and $$B_2$$ are path-conjunctions.

An equality-generating dependency (EGD) is an EPCD of the form:

$$\forall(x_1 \in P_1) \ldots \forall(x_n \in P_n) [ B_1(\bar{x}) \Rightarrow P = P']$$

We will call a pair (variable, set) of the form $$x \in S$$, occurring in either the universal or the existential part of an EPCD, a binding. As before we can have dependent bindings. Thus EPCDs (and EGDs) are more general than first-order logic formulas, and this reflects the nested characteristic of our data model. In the absence of nested sets and dictionaries, EPCDs and EGDs become relational tgd's and egd's [BV84a, BV84b]. All the constraints shown in the examples of Chapters 1 and 2 are EPCDs and EGDs (but few of them are tgd's or egd's).

A PC tableau consists of a context and a path-conjunction of the form $$T ::= \{\bar{x} \in P_1 ; C_1(\bar{x})\}$$
A PC tableau is a generalization of a relational tableau [BV84a, BV84b]. If no nested sets or dictionaries are present (i.e., only relations are present), then a PC tableau corresponds exactly to a relational tableau. The main difference is only one of notation: in a PC tableau, variables range over tuples rather than individuals, and equalities are explicit rather than implicit.

For an EPCD as above we will also use the notation $\text{dep}(T, T')$, where $T$ is as above and $T' = \{ \bar{x} \in \bar{\bar{F}}_1, \bar{y} \in \bar{\bar{F}}_2(\bar{x}) ; C_1(\bar{x}) \text{ and } C_2(\bar{x}, \bar{y}) \}$. This is in the spirit with the notation for tuple generating dependencies using tableaux in [BV84a] and [BV84b]. Note however that our formalism doesn’t necessarily distinguish between EPCDs and EGDs: any EGD can be written as $\text{dep}(T, T')$, where $T' = \{ \bar{x} \in \bar{\bar{F}}_1 ; C_1(\bar{x}) \text{ and } \bar{F}_2(\bar{x}) = F_3(\bar{x}) \}$. For a PC query $Q$ as above $T = \{ x_1 \in F_1, \ldots, x_m \in F_m ; C \}$ is the tableau corresponding to $Q$. Note that the tableau $T$ of a PC query is uniquely determined (consisting of all elements in the from and where clauses). Similarly, the tableaux $T$ and $T'$ associated to an EPCD are uniquely determined. Thus, any path occurring in a query or constraint has a unique tableau surrounding it. We will use this tableau as a context with respect to which we are able to assert properties about paths.

As opposed to section 1.2, any lookup operation $P_2[P_1]$ must be such that $P_1$ is a variable. This restriction together with the well-definedness restriction below ensure well-definedness (i.e., non-failure) of queries (see the next section for detailed treatment). We could easily add arbitrary constants at base types to the language above and our results will still hold, however, for simplicity, we choose to ignore them in this chapter.

All PC queries, tableaux and EPCDs are additionally restricted as follows:

- A finite set type is a type of the form $\text{Set}(\tau)$ where the only base type occurring in $\tau$ is bool or $\text{Struct}()$ (the empty record type). We do not allow in PC queries or EPCDs bindings of the form $x \in P$ such that $P$ is of finite set type $^1$.

- Any expression $P_2[x]$ defined must be such that it either occurs in the scope of a binding $x \in \text{dom} P$ of the surrounding tableau $T$ or, more general, there exists a binding $x' \in S$ in $T$ such that

\[
\forall (\bar{x} \in \bar{\bar{F}}_1) \left[ C_1(\bar{x}) \Rightarrow x=x' \text{ and } S=\text{dom } P \right]
\]

is a valid EPCD (true in all instances) $^2$.

- A simple type is defined (inductively) as either a base type or a record type in which the types of the components are simple types (in other words, it doesn’t involve set or dictionary types). Dictionary types $\text{Dict}(\sigma, \tau)$ are restricted such that $\sigma$ is a simple type.

For the completeness part of some of our theorems, we will require an additional restriction on set/dictionary equality in path-conjunctions and on the paths occurring in the select clause of a query:

- The paths $P_1, P'_1$ appearing in path conjunctions must be of simple type. Also the paths occurring in the select clause will have to be restricted so that they are of simple type as well.

In some of our results, we will be able to drop the simple-type restriction in equalities or in the path expression in the select clause of a query. Therefore, we will mention explicitly whenever the simple-type restrictions are needed. However, keys in dictionaries will always be of simple type. We give next some useful definitions and notations.

$^1$Finite set types cause some difficulties in our current proof method. However there are more serious reasons to worry about them; it is shown in [BNTW95] that they can be used to encode set difference, although the given encoding uses a language slightly richer than that of PC queries.

$^2$This restriction could be removed at the price of tedious reasoning about partiality, but we have seen no need to do it for the results and examples discussed here.
Definition 3.1.1 A valuation of a tableau $T = \{ \bar{x} \in \bar{P} : C(\bar{x}) \}$ into an instance $I$ is a type-preserving mapping $v : \bar{x} \rightarrow I$ that can be extended to path expressions and path conjunctions over $\bar{x}$ (i.e. $v(R) = R'$, for any name $R$, $v(P.A) = v(P).A$, etc.) such that the following two conditions hold:

1. If $x \in P$ occurs in $T$ then $v(x)$ is an element of $v(P)$ in $I$ (context-preserving property)
2. $C(v(\bar{x})) = \text{true}$

We will show in the next section that the restriction to well-defined tableaux guarantees that no valuation ever fails. To be more precise, for a well-defined expression of the form $P[\bar{x}]$ over some well-defined tableau $T$ and a valuation $v$ from $T$ into some arbitrary instance, $v(P)[v(\bar{x})]$ doesn’t fail because it is always the case that $v(x) \in \text{dom } v(P)$.

The notion of valuation is useful in giving meaning of expressions with free variables, i.e. path expressions $P(\bar{x})$ over a tableau $T$. Thus, we are able to express in terms of valuations the notion of satisfiability of an EPCD by an instance.

Definition 3.1.2 Let $I$ be an instance and $d = \text{dep}(T,T')$ an EPCD. Then $I$ satisfies $d$, written as $I \models \text{dep}(T,T')$, if for any valuation $v : T \rightarrow I$ there exists a valuation $v' : T' \rightarrow I$ such that $v'$ coincides with $v$ on $T$ (i.e. on variables $\bar{x}$ of $T$).

The following notations will be used:

$Q_1 \subseteq^{\text{unr}} Q_2$ means containment under all unrestricted (i.e. finite and infinite) instances. We will also write it as $Q_1 \subseteq Q_2$.

$Q_1 \subseteq^{\text{fin}} Q_2$ means containment under all finite instances.

$Q_1 \subseteq^{\text{D}} Q_2$ means containment under all (unrestricted) instances that satisfy the set of EPCDs $D$ (sometimes we make the superscript unr explicit). Similarly, we may have $Q_1 \subseteq^{\text{fin}}_D Q_2$ with the obvious meaning.

$D \models d$ means that EPCD $d$ is a logical consequence of the set of EPCDs $D$ (under all unrestricted instances). Similarly, $D \models^{\text{fin}} d$ means finite implication.

**OO navigation-join queries.** The above syntax for path expressions does not allow for expressing typical navigation-style joins (we refer to them as pointer-based joins) used in OO query languages. For example, a path of the form $P_3[\;[P_1[x_1]]\;]$ is not syntactically valid. While apparently this is an important limitation, in all common cases it is possible to break such a path into several simple paths having only one lookup operation, by introducing additional scans and explicit join conditions. To illustrate, the following query $Q_1$ (not a PC query):

\[
\text{select } P_2[\;[P_1[x_1]]\;] \text{ from dom } P_1 \; x_1
\]

can be rewritten as $Q_2$ (a PC query):

\[
\text{select } P_2[\;z\;] \text{ from dom } P_1 \; x_1. \; \text{dom } P_2 \; z \text{ where } z=P_1[\;x]\]

However, the two queries are not always equivalent: $Q_1$ can fail while $Q_2$ never fails. This is because the implicit constraint that one assumes when writing $Q_1$, namely (RIC) $\forall x \in \text{dom } P_1 \; \exists z \in \text{dom } P_2 \; z=P_1[\;x\;]$, may not hold. Clearly, $Q_1$ and $Q_2$ are equivalent under all instances satisfying (RIC): the first query is a pointer-based join, while the second one is a value-based join [SC90, BK93]. While logically, we can always live with only the value-based join form at the language level (because it is actually the one having the intended meaning, anyway), this is no longer the case when we want to evaluate queries. It is shown in [SC90] that in some situations a
pointer-based join is cheaper to evaluate than its equivalent value-based join, while in other situations it is the other way around. Thus, an optimizer must consider both forms when searching for the optimal plan. The full details will be addressed in Chapter 6 where we show how we can map PC queries such as $Q_2$ into physical plans that correspond to either value-based or pointer-based joins.

Having explained this issue, all queries that we consider throughout this chapter and the next one will be PC (i.e. no pointer-joins allowed).

3.2 A Canonical Instance Construction

We associate to each tableau $T = \{ \bar{x} \in \bar{\bar{P}} : C(\bar{x}) \}$ a special instance, $\text{Inst}(T)$, crucial for proving our decidability and completeness results. Intuitively, $\text{Inst}(T)$ is the minimal instance that contains the "structure" of $T$, and it allows us to express syntactical conditions on $T$ as necessary and sufficient conditions on $\text{Inst}(T)$. For example, we will use it to decide whether an equality of paths follows from the equalities in the where clause of a query. This check is one of the prerequisites for finding homomorphisms, a very frequent operation during the chase.

The concept of canonical instance is not new. On the contrary, it was widely used in the theory of relational conjunctive queries and dependencies. For example to show that the (relational) chase is a complete procedure for proving dependency implication $D \models d$, one constructs (through chase) a (canonical!) instance $I$ that satisfies $D$, thus satisfying $d$, from which it is inferred that a certain "valuation"/homomorphism must exist [BV84b]. In the relational case the canonical instance is isomorphic to the tableau itself. However, in our case, the construction is significantly more complicated due to several reasons:

- **nested sets.** While there are some current extensions of the relational tableaux to nested relational tableaux [HD99], these are not enough when we add dictionaries.

- **dictionaries.** We choose to give up the nice graphical tableau representation in favor of a graph-based construction that is able to represent all reachable components of an object (through record projections or, more complicated, through lookup operations). Moreover the construction represents in a concise way all equalities between these components that can be inferred from $C(\bar{x})$. Computing such equalities is done by propagating the equalities in $C(\bar{x})$ via a congruence closure construction. This construction uses a significant extension of the method in [NOS80].

- **well-defined lookups.** We want to ensure that every lookup operation $P[x]$ that we consider is well defined (it doesn't fail) in the canonical instance. This ensures that all queries that we consider are "safe" (never fail), thus reasoning about containment/equivalence of queries doesn't need to take into account undefined values, hence being an easier task.

- **extensionality of sets.** If in the canonical instance two objects of set type have the same elements then they must be identified, even though their equality may not be a consequence of $C(\bar{x})$. As we'll see this causes significant difficulties when reasoning about queries/dependencies with set/dictionary equality. The only way we could prove the main theorems in this chapter was by (partially) giving up set/dictionary equality.

For this section there are no restrictions in the type of equalities that can occur in $C(\bar{x})$. In particular, set/dictionary equality is allowed. The construction, which has two stages, is sketched next. The nice inductive structure exhibited by the construction will make our life a lot easier when proving properties about queries/dependencies.
**A non-extensional instance:** $\text{Clns}(T)$. In the first stage we build, in parallel, a set $G$ of path expressions over the variables $\bar{x}$, and a binary relation $\simeq \subseteq G \times G$ such that $G$ will represent the set of all possible well-defined paths over $T$ while $\simeq$ will be the congruence closure of the relation \{ $(Q_1, Q_2) \mid Q_1 = Q_2$ occurs in $T$ \}. We will use as a notation $Q : G$ to assert that the path expression $Q$ is an element of $G$. Then, $G$ and $\simeq$ are defined to be the least set and binary relation that are closed under the below rules. The first group of rules, the *add rules*, specify when a path expression $Q$ belongs to $G$:

\[
\begin{align*}
\text{(prj-add)} & \quad Q : G \\
& \implies Q.A : G \\
\text{(dom-add)} & \quad Q : G \\
& \implies \text{dom } Q : G \\
\text{(root-add)} & \quad \text{R in the schema} \\
& \implies R : G
\end{align*}
\]

\[
\begin{align*}
\text{(var-add)} & \quad x \in S \text{ in } T, \quad S : G \\
& \implies x : G \\
\text{(true-add)} & \quad \text{true} : G \\
\text{(false-add)} & \quad \text{false} : G
\end{align*}
\]

\[
\begin{align*}
\text{(lookup-add)} & \quad x \simeq x', \quad x' \in S \text{ in } T, \quad S \simeq \text{dom } Q \\
& \implies Q[ x ] : G
\end{align*}
\]

We remark here that the (var-add) rule is not an axiom, in other words to infer that $x : G$ we need to make sure first that the set $S$ to which $x$ is bound is well-defined. This reflects the fact that in $\bar{x} \in \bar{P}$ we may have dependent bindings (for example: $y \in R, x \in y$). Also note that the last rule requires in the hypothesis two conditions involving $\simeq$. The second group of rules, the *equality rules*, specify when two paths $Q_1$ and $Q_2$ are to be considered equal as a consequence of $C(\bar{x})$:

\[
\begin{align*}
\text{(eq)} & \quad Q_1 = Q_2 \text{ occurs in } C(\bar{x}), \quad Q_1 : G, \quad Q_2 : G \\
& \implies Q_1 \simeq Q_2
\end{align*}
\]

\[
\begin{align*}
\text{(refl)} & \quad Q : G \\
& \implies Q \simeq Q \\
\text{(sym)} & \quad Q_1 \simeq Q_2 \\
& \implies Q_2 \simeq Q_1 \\
\text{(trans)} & \quad Q_1 \simeq Q_2, \quad Q_2 \simeq Q_3 \\
& \implies Q_1 \simeq Q_3
\end{align*}
\]

\[
\begin{align*}
\text{(prj-cong)} & \quad Q \simeq Q' \\
& \implies Q.A \simeq Q'.A \\
\text{(dom-cong)} & \quad Q \simeq Q' \\
& \implies \text{dom } Q \simeq \text{dom } Q' \\
\text{(red-ext)} & \quad Q.A_1 \simeq Q'.A_1 \ldots Q.A_n \simeq Q'.A_n \\
& \implies Q \simeq Q'
\end{align*}
\]

\[
\begin{align*}
\text{(lookup-cong)} & \quad x \simeq y, \quad Q_1 \simeq Q_2 \\
& \implies Q_1[ x ] : G \quad Q_2[ y ] : G \\
& \implies Q_1[ x ] \simeq Q_2[ y ]
\end{align*}
\]

Observe that $Q_1 \simeq Q_2$ implies $Q_1 : G$ and $Q_2 : G$ (a simple induction on the derivation of $Q_1 \simeq Q_2$). In other words we only equate paths that we already know to be well-defined. The add rules and the equality rules are cyclic. As a consequence of equating paths via $\simeq$, more paths may become members of $G$ (and therefore well-defined) and vice-versa.
The important observation now is that \( G \) and \( \simeq \) are finite, and moreover, constructible in polynomial time in the size of the tableau \( T^3 \). This is an immediate consequence of the fact that the set of all syntactically valid paths with variables in \( \vec{x} \) is finite and polynomial in the size of \( T \). To see why this is true it suffices to observe that all our operators (the lookup operation being the only exception) are deconstructors that reduce the size of the input type, and that in the right-hand side of the lookup operator only variables are allowed. An algorithm that computes \( G \) and \( \simeq \) will always make sure that at each step produces at least one new path \( Q : G \) or a new pair \( Q_i \simeq Q_j \). Since \( G \) and \( \simeq \) are polynomial in the size of \( T \) the algorithm must terminate within polynomially many steps. It is easy to see that each step of the algorithm (mainly deciding which rule to be applied) can be implemented in polynomial time as well. We will refer to \( G \) and \( \simeq \) as the set of well-defined paths over \( T \) and, respectively, the congruence closure of \( T \).

**Definition 3.2.1** (1) A tableau \( T \) is a well-defined tableau if, for every \( Q \) that occurs in \( T \), \( Q : G \). A path \( Q \) with variables in \( \vec{x} \) is a well-defined path over \( T \) if \( Q : G \).

(2) An EGD of the form \( \forall (\vec{x} \in \vec{P}) [ C(\vec{x}) \Rightarrow Q = Q' ] \) is well-defined if the tableau \( T = \{ \vec{x} \in \vec{P} ; C(\vec{x}) \} \) is well-defined and \( Q \) and \( Q' \) are well-defined over \( T \).

(3) An EPCD \( \text{dep}(T, T') \) is well-defined if both tableaux \( T \) and \( T' \) are well-defined.

Note that it is possible that a tableau \( T \) is well-defined and there are paths that are not well-defined over \( T \) (however, they do not occur in \( T \)). It is also possible to have well-defined paths over tableaux that are not well-defined. We will always work with well-defined tableaux and with well-defined paths over well-defined tableaux. The next lemma shows that a well-defined EGD is a special case of a well-defined EPCD.

**Lemma 3.2.2** Let \( T = \{ \vec{x} \in \vec{P} ; C(\vec{x}) \} \) be a well-defined tableau. Then \( Q(\vec{x}) \) and \( Q'(\vec{x}) \) are well-defined paths over \( T \) if and only if the tableau \( T' = \{ \vec{x} \in \vec{P} ; C(\vec{x}) \text{ and } (Q = Q') \} \) is well-defined. Thus the EGD \( \forall (\vec{x} \in \vec{P}) [ C(\vec{x}) \Rightarrow Q = Q' ] \) is the same as the EPCD \( \text{dep}(T, T') \).

**Proof.** Let \( \simeq \) and \( \simeq' \) be, respectively, the corresponding relations for \( T \) and \( T' \). Similarly, we have \( G \) and \( G' \). For the first direction, assume that \( Q \) and \( Q' \) are well-defined over \( T \), i.e. \( Q : G \) and \( Q' : G' \). An easy induction (by the derivation of \( P : G \) or \( P_1 \simeq P_2 \)) shows that \( P : G \) implies \( P : G' \) and \( P_1 \simeq P_2 \) implies \( P_1 \simeq P_2 \). Thus \( Q \) and \( Q' \) are well-defined over \( T' \), and hence \( T' \) is well-defined.

Conversely, we prove by induction that for any \( P : G' \), \( P_1 \not\simeq P_2 \) with derivations that do not use \( Q = Q' \), we have \( P : G \) and \( P_1 \simeq P_2 \). The interesting case is when the last rule applied in the derivation of \( P_1 \not\simeq P_2 \) is (eq.). It must be the case then that \( P_1 = P_2 \) occurs in \( C(\vec{x}) \) since the derivation doesn’t use \( Q = Q' \). Since \( P_1 : G' \) and \( P_2 : G' \), by the inductive hypothesis, we have \( P_1 : G \) and \( P_2 : G \). Thus, by (eq.), \( P_1 \simeq P_2 \). The other cases are straightforward. We observe now that to derive \( Q : G' \) and \( Q' : G' \) we don’t need \( Q = Q' \) (\( Q : G' \) and \( Q' : G' \) must be already derived before we can apply the (eq) rule with \( Q = Q' \)). Thus, \( Q : G \) and \( Q' : G \). **End of Proof.**

Next, we build a directed acyclic labeled graph with \( G \) as the set of nodes. The edges are as follows: for any \( Q : G \) and \( Q \wedge : G \), add an edge, labeled \( A \) from \( Q \) into \( Q.A \). For any \( Q : G \) and \( \text{dom} Q : G \), add an edge labeled \( \text{dom} \) from \( Q \) into \( \text{dom} Q \). For any \( x : G \), \( Q : G \), \( Q[x] : G \), add two edges labeled \( [ ] \), one from \( x \) into \( Q[x] \) and one from \( Q \) into \( Q[x] \). Finally, we populate nodes corresponding to set values: for each binding \( x \in S \) occurring in \( T \), add an edge labeled with \( \varepsilon \) from \( S \) into \( x \).

Each \( \approx \)-equivalence class becomes a node in a new graph, \( \text{CInst}(T) \). Add an edge from a node \([P_1, \ldots, P_n]\) into a node \([Q_1, \ldots, Q_k]\), if there is at least one edge with the same label from some \( P_i \) into some \( Q_j \) in \( G(T) \). We

---

However, it may be exponential in the height of the schema. The height of the schema is the maximum number of levels of nesting of record or dictionary types. In this paper we always consider the schema fixed.
will use the following notations: if $P$ and $Q$ are two nodes in $CInst(T)$ such that there is an edge labeled $\texttt{A}$ from $P$ into $Q$ then we denote the node $Q$ by $P.A$. For a given node $P$ (of record type) there is a unique node $P.A$ (since $\simeq$ is closed under (prj-cong). Similar, for edges labeled $[]$ and $\texttt{dom}$. Thus, $\texttt{A}$, $[]$ and $\texttt{dom}$ can be viewed as operations\textsuperscript{4} on nodes of $CInst(T)$. Also, observe that these operations are always defined. For nodes $P$ and $Q$ in $CInst(T)$ such that there is an edge labeled $\in$ from $Q$ into $P$ we use the notation $P \in Q$.

\textbf{Example.} Consider the tableau

$$ T = \{ x \in S, z \in \mathcal{M}[x].A : (\mathcal{M}[x].B=z) \ \textbf{and} \ (S=\texttt{dom}\mathcal{M}) \ \textbf{and} \ (\mathcal{M}[x].B=x) \} $$

where $\mathcal{M} : \text{Dict}(b, \text{Struct}(A : \text{Set}(b).B : b)), S : \text{Set}(b)$, and $b$ is some base type. Then $G$ is shown in figure 3.1a, while $CInst(T)$ is shown in figure 3.1b. The relation $\simeq$ can be read from the nodes of $CInst(T)$: each node is a $\simeq$-equivalence class. Note that $\mathcal{M}[x]$ is well-defined over $T$ since $x \in S$ and $S \simeq \texttt{dom}\mathcal{M}$. $\mathcal{M}[z]$ is well-defined over $T$ as well, since $z \simeq x$. However, $\mathcal{M}[x].B$ is not well-defined, even though $\mathcal{M}[x].B \simeq x$, because it is not syntactically valid. Observe that all the well-defined paths over $T$ appear in $G$, and $T$ is a well-defined tableau.

\textbf{Proposition 3.2.3} Let $T$ be a well-defined tableau and let $Q$ be a well-defined path over $T$. Then, for any instance $I$ and for any valuation $v : T \rightarrow I$, $v(Q)$ doesn't fail.

\textbf{Proof.} We prove by simultaneous induction on the derivation of $Q : G$ or $Q_1 \simeq Q_2$ that $Q : G$ implies $v(Q)$ doesn't fail and $Q_1 \simeq Q_2$ implies that neither $v(Q_1)$ nor $v(Q_2)$ fails and $v(Q_1) = v(Q_2)$. The base cases, (root-add), (true-add), (false-add) are obvious. For the inductive cases, (prj-add), (dom-add), (sym), (trans), (prj-cong), (dom-cong), and (red-ext) require a simple application of the inductive hypothesis and of the definition of a valuation. (var-add) is trivial. The first interesting case is (lookup-add). Then $v(Q[x]) = v(Q)[v(x)]$ and $v(x' \in v(S)$. But since $S \simeq \texttt{dom}Q$ it follows by the inductive hypothesis that $v(S) = v(\texttt{dom}Q) = \texttt{dom}v(Q)$. Similarly, $x \simeq x'$ implies $v(x) = v(x')$. Thus $v(x) \in \texttt{dom}v(Q)$ and therefore $v(Q)[v(x)]$ doesn't fail. For the (eq) rule, by the inductive hypothesis, we have that $v(Q_1)$ and $v(Q_2)$ don't fail. Since $v$ is a valuation it must be the case that $v(Q_1) = v(Q_2)$. In the case of (lookup-cong) we make use of the fact that $v(Q_1[x])$ and $v(Q_2[y])$ must be defined. The (refl) case is similar. \textbf{End of Proof.}

\textsuperscript{4}In which, even though we don’t say it explicitly, the order of arguments of $[]$ matters, of course.
We conclude the construction of \( C\text{Inst}(T) \) and the discussion on well-definedness with an interesting remark. Suppose \( T \) is a tableau as before and \( Q_1 = Q'_1, \ldots, Q_n = Q'_n \) are, each of them, well-defined equalities over \( T \). Then the EPCD
\[
d = \forall (\bar{x} \in \bar{P}) \left[ C(\bar{x}) \Rightarrow (Q_1 = Q'_1) \text{ and } \ldots \text{ and } (Q_n = Q'_n) \right]
\]
is obviously well-defined. Moreover it is equivalent to the set of EGDs:
\[
D = \{ \forall (\bar{x} \in \bar{P}) \left[ C(\bar{x}) \Rightarrow Q_i = Q'_i \right] \mid i = 1, n \}
\]
However, there are cases in which an well-defined EPCD of the same form as \( d \) is not equivalent with \( D \) simply because the EGDs in \( D \) may not be well-defined when taken individually. A simple example is to take \( d \) to be:
\[
\forall (x \in S) \left[ \text{true} \Rightarrow (\text{true} = \text{dom } M) \text{ and } (M[x] = x.A) \right]
\]
While \( d \) is well-defined, obviously the EGD \( \forall (x \in S) \left[ \text{true} \Rightarrow M[x] = x.A \right] \) is not well-defined. However, we will use the above equivalence without explicitly mentioning, whenever it is clear from the context that the well-definedness conditions are satisfied.

The canonical instance: \( \text{Inst}(T) \). \( C\text{Inst}(T) \) has all the properties to be a valid instance with one exception: there may be two distinct nodes of set type \( S_1, S_2 \) that have the same set of \( \varepsilon \)-successors, \( \{e_1, \ldots, e_m\} \). Thus, \( C\text{Inst}(T) \) does not satisfy the extensionality property of sets. In this case, we will identify \( S_1 \) and \( S_2 \). Similar, if two dictionaries \( M \) and \( N \) have the same domain, i.e. \( \text{dom } M \) and \( \text{dom } N \) are the same node, and for all \( \varepsilon \)-successors \( x \) of \( \text{dom } M \), \( M[x] \) and \( N[x] \) are identical, then we identify the nodes \( M \) and \( N \). Of course, we may need to propagate these identifications in the graph. More formally, we define an equivalence relation on nodes of \( C\text{Inst}(T) \) (which in turn consist of congruence classes with respect to \( C(\bar{x}) \)), denote it by \( \sim \), as the smallest relation closed under the following extensionality rules:

\[
\begin{align*}
\text{(refl)} & \quad Q \sim Q \\
\text{(sym)} & \quad Q_1 \sim Q_2 \qquad Q_2 \sim Q_1 \\
\text{(trans)} & \quad Q_1 \sim Q_2 \quad Q_2 \sim Q_3 \Rightarrow Q_1 \sim Q_3 \\
\text{(set-ext)} & \quad \forall e \in S_1. \exists e' \in S_2. e \sim e' \quad \forall e' \in S_2. \exists e \in S_1. e' \sim e \\
\text{(dict-ext)} & \quad M \leq N \quad N \leq M \Rightarrow M \sim N
\end{align*}
\]

where \( M \leq N \) means: for all \( e \in \text{dom } M \) there exists \( e' \in \text{dom } N \) such that \( e \sim e' \land M[e] \sim N[e'] \). Note that \( M \leq N \) and \( N \leq M \) implies \( \text{dom } M \sim \text{dom } N \).

It is obvious that \( \sim \) exists and is unique. The graph consisting of \( \sim \)-equivalence classes is \( \text{Inst}(T) \) (we add edges between \( \sim \)-equivalence classes in the same way we did for \( C\text{Inst}(T) \)). \( \text{Inst}(T) \) is an extensional instance, and one can see that its construction is in \( \text{PTIME} \). We consider two canonical mappings, \( \text{eval} : T \rightarrow C\text{Inst}(T) \), and \( \text{eval} : T \rightarrow \text{Inst}(T) \), associating to each path expression \( P \) occurring in \( T \) nodes in \( C\text{Inst}(T) \) and, respectively, \( \text{Inst}(T) \). We denote by \( \text{collapse} \) the function mapping nodes in \( C\text{Inst}(T) \) to their corresponding \( \sim \)-equivalence classes in \( \text{Inst}(T) \). We have then \( \text{eval} = \text{collapse} \circ \text{eval} \).

Example. Consider again the tableau \( T \) from the previous example. Since in \( C\text{Inst}(T) \) the nodes \( [S, \text{dom } M] \) and \( [M[x].A] \) have the same \( \varepsilon \)-successor, it follows that \( [S, \text{dom } M] \sim [M[x].A] \). Thus they are collapsed in \( \text{Inst}(T) \)
(see figure 3.2). For simplicity, we represented the \( \sim \)-equivalence class consisting of the two nodes as the union of the elements in each of the two \( \approx \)-equivalence classes. We note that, in \( \mathit{Inst}(T) \), \( \mathcal{M}[x]A \) and \( \mathit{dom}\mathcal{M} \) are equal (\( \mathit{eval}(\mathcal{M}[x]A) = \mathit{eval}(\mathit{dom}\mathcal{M}) \)) even though their equality doesn’t follow (by congruence) from the original equalities of \( T \). Also there exist instances into which we can "map" \( T \) via valuations but still these instances don’t satisfy the equality of \( \mathcal{M}[x]A \) and \( \mathit{dom}\mathcal{M} \). Since our goal is to characterize dependencies that hold in all instances by checking their satisfiability in the canonical instance, this is apparently a problem. We will see in a little while how we can overcome this difficulty.

![Diagram](image)

Figure 3.2: \( \mathit{Inst}(T) \)

We prove first some technical lemmas that ensure that our construction is well-defined, and are essential for the next results. The first one says that set nodes in \( C\mathit{Inst}(T) \) are identified by \( \mathit{Inst}(T) \) only as a consequence of the extensionality rule. The same property holds for dictionaries. We will see how this fails if we allow keys of non-simple type to appear in dictionaries.

**Lemma 3.2.4** 1. If \( S_1 \) and \( S_2 \) are set nodes in \( C\mathit{Inst}(T) \) then the following is a derived rule:

\[
\begin{array}{c}
\text{(set-inv-ext)} \\
\forall e \in S_1 \cdot \exists e' \in S_2 . e \sim e' \quad \forall e' \in S_2 . \exists e \in S_1 . e' \sim e
\end{array}
\]

2. If \( M \) and \( N \) are dictionary nodes in \( C\mathit{Inst}(T) \) then the following is a derived rule:

\[
\begin{array}{c}
\text{(dict-inv-ext)} \\
M \sim N \quad M \leq N \quad N \leq M
\end{array}
\]

**Proof.** The only way to infer \( S_1 \sim S_2 \) other than by (refl), (sym) or (trans) is by (set-ext). A simple induction concludes the proof. Similar for the dictionary case. **End of Proof.**

**Lemma 3.2.5** If \( Q_1 \) and \( Q_2 \) are of simple type such that \( Q_1 \sim Q_2 \) then \( Q_1 = Q_2 \).

**Lemma 3.2.6** \( \sim \) is a congruence relation, i.e. it is closed under rules (prj-cong), (!-cong) and dom-cong.

**Proof.** For (prj-cong) we use a similar observation as in the proof of Lemma 3.2.4. (dom-cong) is an immediate consequence of Lemma 3.2.4. The interesting case is (lookup-cong). Suppose \( Q_1 \sim Q_2 \) of simple type, and \( M_1 \sim M_2 \). We observe first that \( Q_1 = Q_2 \) (by the previous lemma), and second that \( Q_1 \in \mathit{dom} M_1 \) if and only if \( Q_1 \in \mathit{dom} M_2 \) (using the fact that \( \mathit{dom} M_1 \sim \mathit{dom} M_2 \) and Lemma 3.2.4 and, again, previous lemma). Suppose that \( Q_1 \in \mathit{dom} M_1 \) Since \( M_1 \leq M_2 \), there exists some \( Q' \in \mathit{dom} M_2 \) such that \( Q_1 \sim Q' \) (and therefore \( Q_1 = Q' \)) and \( M_1[Q_1] \sim M_2[Q'] \). Thus, we conclude \( M_1[Q_1] \sim M_2[Q_2] \). **End of Proof.**

47
The last lemma allows us, as in the case of $CInst(T)$, to define in a correct way, the operations $\cdot$, $\sqcup$ and $\text{dom}$ on nodes of $Inst(T)$. We also observe that without the restriction on the type of keys of dictionaries, Lemma 3.2.6 fails. A simple counterexample to it is $T = \{x \in \text{dom} t, y \in \text{dom} t, z \in \mathbb{N}[y] : \text{true} \}$, where $x$ and $y$ are of the same set type. Here, we infer $\text{eval} - x \sim \text{eval} - y$, by (set-ext), since they are both empty, and $\text{eval} - (\text{dom} t) \sim \text{eval} - (\text{dom} t)$, by (set-ext) again, but there is no way to infer $\text{eval} - (\mathbb{N}[x]) \sim \text{eval} - (\mathbb{N}[y])$ with only the rules listed in the definition of $\sim$. In this case we would have to postulate explicitly the congruence rules as part of the definition of $\sim$. However, one can easily see that Lemma 3.2.4 fails in that case.

**Lemma 3.2.7** The mappings $\text{eval} -$, $\text{collapse}$ and $\text{eval}$ are algebraic homomorphisms\(^5\) with respect to operations $\cdot$, $\sqcup$ and $\text{dom}$ on path expressions over $T$, nodes in $CInst(T)$ and nodes in $Inst(T)$.

And this concludes our construction!

### 3.3 Trivial dependencies and query containment

#### 3.3.1 Trivial EGDs

We show first that an EGD (with set/dictionary equality) is true in all (finite and unrestricted) instances (trivial EGD) if and only if it is satisfied in $CInst(T)$ under the canonical mapping $\text{eval} -$. Since the construction of $CInst(T)$ can be carried out in PTIME, it follows that deciding triviality of EGDs is in PTIME as well.

**Lemma 3.3.1 (EGD Lemma)** Let $d = \forall(\bar{x} \in \bar{P}) \ [ C(\bar{x}) \Rightarrow Q(\bar{x}) = Q'(\bar{x}) ]$ be an EGD over $T = \{\bar{x} \in \bar{P}; C(\bar{x})\}$. $T$ may have set/dictionary equality.

1. **If $Q$ and $Q'$ are of simple type then the following are equivalent:**
   
   a. $d$ is trivial (fin and/or unr)
   
   b. $\text{eval} - Q = \text{eval} - Q'$
   
   c. $\text{eval} Q = \text{eval} Q'$

2. **If $Q$ and $Q'$ are allowed to have set/dictionary equality then the following are equivalent:**

   a. $d$ is trivial (fin and/or unr)
   
   b. $\text{eval} - Q = \text{eval} - Q'$

**Proof. Part 1:** One direction, (b) implies (a), is true even for $Q$ and $Q'$ of non-simple type, and we already proved it in Proposition 3.2.3. Indeed, $\text{eval} - Q = \text{eval} - Q'$ means that $Q \simeq Q'$, and thus, for any instance $I$ and for any valuation $v : T \rightarrow I$, $v(Q) = v(Q')$. Since we didn’t make any assumption about $I$, it follows that $d$ is trivial both in the finite and unrestricted case. The direction (a) implies (c) is obvious. For the last direction in part (1), (c) implies (b), assume $\text{eval} Q = \text{eval} Q'$. This means that $\text{collapse}(\text{eval} - Q) = \text{collapse}(\text{eval} - Q')$, or $\text{eval} - Q \sim \text{eval} - Q'$. If $Q$ and $Q'$ are of simple type then, by Lemma 3.2.5, $\text{eval} - Q = \text{eval} - Q'$.

**Part 2:** We need to show (a) implies (b) even when set/dictionary equality is allowed. We already proved (b) $\Rightarrow$ (a) in this case. Suppose $d$ is trivial, $Q$ and $Q'$ are of set or dictionary type, and $\text{eval} - Q \neq \text{eval} - Q'$.

\(^{5}\text{We use the term algebraic homomorphism for a mapping that commutes with operations } \cdot, \sqcup \text{ and dom, not to be confused with a homomorphism, defined in section 3.3, which, besides being an algebraic homomorphism, must satisfy some additional conditions.} \)
We construct another trivial EGD $d' : \forall(x \in \bar{P}) \forall(y \in \bar{R}) \ [ C(x) \Rightarrow Q(x) = Q'(x) ]$ for which we show that $\text{eval}^t Q \neq \text{eval}^t Q'$, where $\text{eval}^t$ is the canonical valuation associated to $T' = \{ x \in \bar{P}, y \in \bar{R} \mid C(x) \}$. Thus, $\text{eval}^t$ and $\text{Inst}(T')$ provide a counterexample to the fact that $d'$ is trivial.

First, it is easy to see that any $d'$ of the above form is a logical consequence of $d$: for any instance $I \models d$, if $v$ is a valuation from $T'$ into $I$ then $v$ restricted to $x$ is a valuation from $T$ into $I$, thus $v(Q) = v(Q')$, and therefore $I \models d$. Hence $d$ trivial implies $d'$ trivial.

We explain the construction of the new bindings $y \in \bar{R}$ that are added to $d$. If $Q$ is of set type, then we generate the first binding: $y \in Q$, where $y$ is a new variable. Now $y$ may generate new bindings as well. Here are all possible cases:

- $y$ is of set type. Add $z \in y$ and continue recursively with $z$.
- $y$ is of simple type. Then $y$ doesn’t generate any bindings any more.
- $y$ is of dictionary type. Then add $z \in \text{dom} y$ and continue recursively with $y[z]$. $z$ itself must be of simple type, therefore it cannot generate any more bindings.
- $y$ is of record type. Then for each attribute $A$ such that $y.A$ is of set or dictionary type, continue recursively with $y.A$.

Similarly, if $Q$ is of dictionary type, the first binding is $y \in \text{dom} Q$, and then we continue in the same way as above with $y$. Notice that this process terminates, because the type of $Q$ is finite. It is easy to see that this construction reflects in the fact that $\text{Inst}(T')$ differs from $\text{Inst}(T)$ by having an additional tree rooted at $y$. This tree is entirely disjoint from $\text{Inst}(T)$, since $C(x)$ doesn’t involve any of the new variables. Also, this tree doesn’t have any set nodes with no $\varepsilon$-successors (our construction rules out explicitly empty sets). This ensures that for any node $Q''$ in this tree, $Q'' \neq P$ for any node $P$ outside of the tree. In particular, $\text{eval}^t_y P$ for any $P$ outside of the tree (here $y$ is the first variable in $y$, the one added in the first step to $Q$ or $\text{dom} Q$). This, together with the fact that $\text{eval}^t_y Q \neq \text{eval}^t_y Q'$ (and therefore $\text{eval}^t_y Q \neq \text{eval}^t_y Q'$), implies that $\text{eval}^t_y Q \neq \text{eval}^t_y Q'$ (we make use here of Lemma 3.2.4). Thus, $\text{eval}^t Q \neq \text{eval}^t Q'$. \textbf{End of Proof.}

Note that the the direction (c) $\Rightarrow$ (b) in part (1) fails when $Q$ and $Q'$ are of non-simple type. For example, $d = \forall(a \in A) \forall(b \in B) [ a = b \Rightarrow A = B ]$ is satisfied in $\text{Inst}(T)$ under the canonical valuation $\text{eval}$. This is because the nodes corresponding to $A$ and $B$ are identified by the (set-ext) rule. Nonetheless $d$ is not trivial (take an instance with $A = \{ a, b \}, B = \{ a, b' \}$). A similar counterexample using dictionaries: $\forall(a \in \text{dom} \bar{M}) [ \text{dom} \bar{M} = \text{dom} \bar{N} \text{ and } \bar{M}[a] = \bar{N}[a] \Rightarrow \bar{M} = \bar{N} ]$. If we look at our example in the previous section, $\text{eval}(\bar{M}[x].A) = \text{eval}(\text{dom} \bar{M})$ but $\text{eval}^t(\bar{M}[x].A) \neq \text{eval}^t(\text{dom} \bar{M})$ and thus $\forall(x \in S) \forall(z \in \bar{M}[x].A) [ \bar{M}[x].B = z \text{ and } S = \text{dom} \bar{M} \text{ and } \bar{M}[x].B = x \Rightarrow z = \bar{M}[x].A = \text{dom} \bar{M}$ is not trivial, as expected.

\textbf{Theorem 3.3.2} An EGD (with equality at set/dictionary type) holds in all unrestricted instances iff it holds in all finite instances. Triviality of EGDs is decidable in PTIME.

\textbf{3.3.2 Homomorphisms of tableaux}

In the next subsection we will prove our first important results, Theorem 3.3.9 and 3.3.11, that relate trivial dependencies and query containment with existence of special mappings between tableaux that generalize the notion of homomorphism from the relational case. As we’ll see, the completeness part of the theorem (the difficult direction) makes use of the fact that, given two tableaux $T_1$ and $T_2$, a valuation $v$ from $T_2$ into $\text{Inst}(T_1)$ induces a homomorphism $h$ from $T_2$ into $T_1$. 

49
Definition 3.3.3 (Homomorphism) Let \( T = \{ \bar{x} \in \bar{P} : C(\bar{x}) \} \) and \( T' = \{ \bar{y} \in \bar{R} : D(\bar{y}) \} \) be two tableaux. A homomorphism \( h : T' \to T \) is a type-preserving mapping from variables \( \bar{y} \) into variables \( \bar{x} \) such that, when extended to path expressions over \( T' \) in the usual way (algebraic homomorphism), \( h \) satisfies:

1. \( h \) maps well-defined paths over \( T' \) to well-defined paths over \( T \),
2. for any \( y_i \in R_i \) in \( T' \) and \( x_j \in P_j \) in \( T \), if \( h(y_i) = x_j \) then the following EGD is trivial
   \[ \forall (\bar{x} \in \bar{P}) \, [ \, C(\bar{x}) \Rightarrow P_j = h(R_i) \, ] \]
3. for each \( Q(\bar{y}) = Q'(\bar{y}) \) that occurs in \( D(\bar{y}) \) the following EGD is trivial
   \[ \forall (\bar{x} \in \bar{P}) \, [ \, C(\bar{x}) \Rightarrow Q(h(\bar{y})) = Q'(h(\bar{y})) \, ] \]

While condition (1) is a simple well-definedness property, conditions (2) and (3) are the interesting ones. Condition (3) says that the image through \( h \) of the equalities in \( T' \) must follow from the equalities of \( T \). The same property must be satisfied by homomorphisms (containment mappings) in the case of relational conjunctive queries. Finally, condition (2) is a generalization of the relational requirement that goals must be mapped into goals with the same relation name\(^6\).

It is easy to see that checking whether a mapping \( h \) is a homomorphism is in \( \text{PTIME} \). Also, we observe that composing a valuation with a homomorphism yields a valuation. Using this, and the EGD lemma, one can easily verify the following proposition, which gives an equivalent and often convenient characterization of homomorphisms in terms of \( G \) and \( \simeq \) of section 3.2. As an immediate application of this proposition, composition of two homomorphisms is a homomorphism.

Proposition 3.3.4 Let \( T = \{ \bar{x} \in \bar{P} : C(\bar{x}) \} \) and \( T' = \{ \bar{y} \in \bar{R} : D(\bar{y}) \} \) be two tableaux, and let \( h \) be a type-preserving mapping from variables \( \bar{y} \) into variables \( \bar{x} \). Extend \( h \) to path expressions such that \( h \) is an algebraic homomorphism. Let \( G, \simeq, \text{ and } G', \simeq' \) be the set of well-defined paths and the congruence closure for \( T \), respectively \( T' \). Then \( h \) is a homomorphism if and only if the following conditions hold:

1. for any \( Q : G' \), \( h(Q) : G \)
2. for any \( y_i \in R_i \) in \( T' \) and \( x_j \in P_j \) in \( T \), if \( h(y_i) = x_j \) then \( P_j \simeq h(R_i) \)
3. for any \( Q_1 \simeq' Q_2 \), \( h(Q_1) \simeq h(Q_2) \)

The next lemma shows that given a valuation from a tableau \( T_2 \) into \( \text{Inst}(T_1) \) we can "shift" this to an "equivalent" valuation from \( T_2 \) into \( \text{ClInst}(T_1) \). However, this equivalence holds only with respect to simple type equalities.

Lemma 3.3.5 Let \( T_1 = \{ \bar{x} \in \bar{P}_1 : C_1(\bar{x}) \} \) and \( T_2 = \{ \bar{y} \in \bar{P}_2 : C_2(\bar{y}) \} \) in which \( C_1 \) may have set/dictionary equality but \( C_2 \) is simple-type restricted. Then, for any valuation \( v : T_2 \to \text{Inst}(T_1) \) there exists a context-preserving, algebraic homomorphism, \( v_c : T_2 \to \text{ClInst}(T_1) \) such that \( v = \text{collapse} \circ v_c \). Moreover, \( v \) and \( v_c \) satisfy the same set of formulas \( Q = Q' \) over \( T_2 \) with \( Q \) and \( Q' \) of simple type.

Proof Sketch\(^7\). We define \( v_c \) such that \( v = \text{collapse} \circ v_c \) by inducting over \( \bar{y} \in \bar{P}_2 \). Base case: \( y \in R \) where \( R \) must be a root name of set type. It must be the case that \( v(y) = v(R) \) in \( \text{Inst}(T_1) \). Since \( v(R) = R^{\text{Inst}(T_1)} \), \( \text{val}_{\text{cl}}(R) = v(R) \). We know that, by the construction

\(^6\)As opposed to the relational case, in our case the expressions that appear as bounding sets for variables are not only relation names but arbitrary path expressions.

\(^7\)A more formal proof, along the same lines, uses induction on the derivation of \( Q : G_2 \) or \( Q \simeq Q' \). See the proof of Lemma 3.7.2
of $\text{Inst}(T_1)$, there must be some node $S$ in $\text{ClInst}(T_1)$ such that $\text{collapse}(S) = v(R)$ and a node $e$ in $\text{ClInst}(T_1)$ such that $\text{collapse}(e) = v(y)$, and $e \in S$. But $S \sim \text{val}_c R$, thus by Lemma 3.2.4, there is some $e', e' \sim e$ such that $e' \in \text{val}_c R$. Define $v_c(y) = e'$, thus $\text{collapse}(v_c(y)) = v(y)$. We then extend $v_c$ on paths over the variable $y$, so that $v_c$ is an algebraic homomorphism. It will be the case that $v_c(R) = \text{val}_c R$. We can verify that $v_c(y) \in v_c(R)$ in $\text{ClInst}(T_1)$ and that $v = \text{collapse}_c \circ v_c$ on paths over $y$. The case when $y \in \text{dom} M$ with $M$ a root name dictionary type is handled similarly, making use again of Lemma 3.2.4 (the dictionary part).

**Induction case:** $y_n \in P_2(y_1, \ldots, y_{n-1})$. If $P_2$ involves only a root name, we define $v_c$ on $y_n$ as before. Suppose $P_2$ does depend on $y_1, \ldots, y_{n-1}$. Then, by the inductive hypothesis, $v_c(P_2)$ is defined and $v(P_2) = \text{collapse}(v_c(P_2))$. We know that $v(y) \in v(P_2)$, thus there must exist an element $e$ (in $\text{ClInst}(T_1)$) such that $\text{collapse}(e) = v(y)$ and an element $S$ such that $\text{collapse}(S) = v(P_2)$ and $e \in S$. Applying Lemma 3.2.4, we infer the existence of some $e', e' \sim e$, such that $e' \in v_c(P_2)$. Define $v_c(y) = e'$. We can verify that $\text{collapse}(v_c(y)) = v(y)$. We then extend $v_c$ on paths depending on $y_n$.

Finally, we need to check that $v(Q) = v(Q')$ if $v_c(Q) = v_c(Q')$ whenever $Q$ and $Q'$ are of simple type. One direction is true even when $Q$ and $Q'$ are arbitrary: $v_c(Q) = v_c(Q')$ implies $v(Q) = v(Q')$. Conversely, $v(Q) = v(Q')$ means $\text{collapse}(v_c(Q)) = \text{collapse}(v_c(Q'))$. But, by Lemma 3.2.5, $\text{collapse}$ is injective on simple type nodes. Thus $v_c(Q) = v_c(Q')$, if $Q$ and $Q'$ are of simple type. **End of Proof.**

We stop here to remark that for any mapping $v_c$ as above, for any well-defined path $P$ over $T_2$, $v_c(Q)$ is defined in $\text{ClInst}(T_1)$. The proof is essentially the one given in Proposition 3.2.3.

**Lemma 3.3.6** Let $T_1 = \{ \bar{x} \in \bar{P}_1 : C_1(\bar{x}) \}$ and $T_2 = \{ \bar{y} \in \bar{P}_2 : C_2(\bar{y}) \}$ in which $C_1$ and $C_2$ are allowed to have equality at set/dictionary type. Then, for any context-preserving, algebraic homomorphism $v_c : T_2 \rightarrow \text{ClInst}(T_1)$ such that $v_c(C_2(\bar{y})) = \text{true}$ there exists a homomorphism $h : T_2 \rightarrow T_1$ such that $\text{val}_c \circ h = v_c$.

**Proof Sketch.** We define mapping $h$ by induction on the bindings $\bar{y} \in \bar{P}_2$ such that $h$ maps well-defined paths to well-defined paths is context preserving and $\text{val}_c \circ h = v_c$. Base case: $y \in R$ in $T_2$, where $R$ is a root name of set type. We know that $v_c(y) \in v_c(R)$ in $\text{ClInst}(T_1)$. Also, we must have $v_c(R) = \text{val}_c R$ (since $v_c$ is an algebraic homomorphism). By the construction of $\text{ClInst}(T_1)$, there exists at least one $S$ in the congruence class $\text{val}_c R$ and one $x$ in the congruence class $v_c(y)$ such that $x \in S$ is a binding in $T_1$. Define $h(y) = x$, and extend $h$ to well-defined paths with at most one variable, $y$. This includes as a limit case $h(R) = R$. Since $\text{val}_c S = \text{val}_c R$, it follows that $\forall (\bar{x} \in \bar{P}_1) [ C_1(\bar{x}) \Rightarrow S = h(R) ]$ is trivial, by Lemma 3.3.1.

We prove now by induction on the size of $Q$ that, for all well-defined path $Q$ over $T_2$, $\text{h}(Q(y))$ is well-defined over $T_1$ and $\text{val}_c \circ h(Q(y)) = v_c(Q(y))$. **Base case.** $y : h(y) = x$ is well-defined over $T_1$, by (var-add), and $\text{val}_c h(y) = v_c(y)$. $Q = R$, where $R$ is any root name : $h(R) = R$ is well-defined, by (root-var), and $\text{val}_c h(R) = v_c(R)$. The cases $Q = \text{true} \text{ false}$ are obvious. **Induction case.** $Q = Q' : Q'$ must be well-defined over $T_2$ (the rule applicable is (proj-add)) and $h(Q) = h(Q')$. By the inductive hypothesis, $h(Q')$ is well-defined over $T_1$, and $\text{val}_c h(Q') = v_c(Q')$. Then $h(Q)$ is well-defined by (proj-add) (this time over $T_1$) and it is easily verified that $\text{val}_c h(Q) = v_c(Q)$. $Q = \text{dom} Q'$: similar with the previous case. Finally, the interesting case is $Q = Q'[y]$. The general picture, including the way $h(y)$ was defined, is described in figure 3.3.

Since $Q'[y]$ is well-defined, it must be the case that $Q'$ is well-defined as well and, by the inductive hypothesis, $h(Q')$ is well-defined and $\text{val}_c h(Q') = v_c(Q')$. By (dom-add), $\text{dom} h(Q')$ is well-defined and $\text{val}_c \text{dom} h(Q') = v_c(\text{dom} Q')$. We also know that $v_c(Q'[y])$ must be defined in $\text{ClInst}(T_1)$, by the previous remark. Thus, it must be the case that $v_c(y) \in \text{dom} v_c(Q'[y]) = v_c(\text{dom} Q')$ in $\text{ClInst}(T_1)$. Hence, there must exist some binding $z \in P$ in $T_1$ such that $\text{val}_c z = v_c(y)$ (and therefore $z \sim x$) and $\text{val}_c P = v_c(\text{dom} Q')$ (and therefore $P \sim \text{dom} h(Q')$). Thus, by (lookup-add), $h(Q'[x])$ is well-defined over $T_1$. Moreover, by (lookup-cong), $\text{val}_c h(Q'[x]) = v_c(Q'[y])$. But since $h(Q'[y]) = h(Q')[x]$, we proved what we wanted.
Figure 3.3: Proof of Lemma 3.3.6: well-definedness of $h$.

For the case when $y \in \text{dom } M$ where $M$ is a root name of dictionary type, $h$ is defined in a similar way. The above proof regarding $h$ still applies.

**Induction case:** $y_n \in P_2(y_1, \ldots, y_{n-1})$. If $P_2$ involves only a root name, we define $h$ on $y_n$ as before. Suppose $P_2$ does depend on $y_1, \ldots, y_{n-1}$. Then, by the inductive hypothesis, $h(P_2)$ is defined and $v_c(P_2) = \text{val}_c(h(P_2))$. As with the base case there must be at least one $P_1$ over $T_1$ such that $\text{val}_c.P_1 = v_c(P_2)$ and at least one variable $x_n$ in $T_1$ such that $\text{val}_c.x_n = v_c(y_n)$. Define $h(y_n) = x_n$ and extend it to paths over variables $y_1, \ldots, y_n$. We have that $v_c(y_n) = \text{val}_c(h(y_n))$. Moreover, $\text{val}_c.P_1 = v_c(P_2) = \text{val}_c(h(P_2))$, by the inductive hypothesis. Thus, $\forall(\bar{x} \in \bar{P}_1) \left[ C_1(\bar{x}) \Rightarrow P_1 = h(P_2) \right]$ is trivial. The previous proof regarding well-definedness of $h$ for paths depending of $y_1, \ldots, y_n$, and the comutativity of $\text{val}_c \circ h$ with $v_c$ still applies.

Finally, we have to verify that $\forall(\bar{x} \in \bar{P}_1) \left[ C_1(\bar{x}) \Rightarrow Q(h(\bar{y})) = Q'(h(\bar{y})) \right]$ is trivial for any $Q(\bar{y}) = Q'(\bar{y})$ that occurs in $C_2(\bar{y})$. By Lemma 3.3.1 it suffices to verify that $\text{val}_c(Q(h(\bar{y}))) = \text{val}_c(Q'(h(\bar{y})))$. But, since $\text{val}_c \circ h = v_c$, this is equivalent to $v_c(Q) = v_c(Q')$ which we know is true. **End of Proof.**

Note that we used only one direction (the easy one) of the Lemma 3.3.1: $\text{val}_c.Q = \text{val}_c.Q'$ implies triviality. Putting together the last two lemmas we obtain the following:

**Corollary 3.3.7** Let $T_1 = \{ \bar{x} \in \bar{P}_1; C_1(\bar{x}) \}$ and $T_2 = \{ \bar{y} \in \bar{P}_2; C_2(\bar{y}) \}$ in which $C_1$ may have set/dictionary equality while $C_2$ is restricted to simple type equality only. Then, for any valuation $v : T_2 \rightarrow \text{Inst}(T_1)$, there exists a homomorphism $h : T_2 \rightarrow T_1$ such that $\text{val}_c \circ h = v$.

### 3.3.3 Trivial EPCDs and query containment

The main idea used in proving Theorems 3.3.9 and 3.3.11 of this section, theorems that characterize PC query containment/triviality of EPCDs in terms of homomorphisms, is as follows. If $Q_1 \subseteq Q_2$ (containment under all instances) then the containment must be satisfied under the canonical instance associated to $Q_1$, call it $\text{Inst}(T_1)$. We also know that the output path of $Q_1$ is always in the result of $Q_1$ on database $\text{Inst}(T_1)$ and thus, by the previous observation, in the result of $Q_2$ on $\text{Inst}(T_1)$. Therefore there must exist a valuation from $Q_2$ (in fact, from the tableau $T_2$ of $Q_2$) into $\text{Inst}(T_1)$. Then we use Corollary 3.3.7 of the previous subsection to infer the existence of a homomorphism from $T_2$ into $T_1$.

We show first that PC query containment is reducible to triviality of EPCDs and vice-versa. Let $Q_1$ and $Q_2$ be two PC queries and define $Q_1 \cap Q_2$ as follows:

\[ Q_1 \overset{\text{def}}{=} \text{select } O_1(\bar{x}) \text{ from } \bar{P}_1 \bar{x} \text{ where } C_1(\bar{x}) \quad Q_2 \overset{\text{def}}{=} \text{select } O_2(\bar{y}) \text{ from } \bar{P}_2 \bar{y} \text{ where } C_2(\bar{y}) \]

\[ ^8 \text{Going back to [CM77].} \]

52
\[ Q_1 \cap Q_2 \overset{\text{def}}{=} \text{select } O_1(\bar{x}) \text{ from } \tilde{P}_1 \bar{x}, \tilde{P}_2 \bar{y} \text{ where } C_1(\bar{x}) \text{ and } C_2(\bar{y}) \text{ and } O_1(\bar{x}) = O_2(\bar{y}) \]

As the notation suggests, it is clear that the meaning of \( Q_1 \cap Q_2 \) is the intersection of the results of \( Q_1 \) and \( Q_2 \).

Consider now the following EPCD:

\[
\text{cont}(Q_1, Q_2) \overset{\text{def}}{=} \forall (\bar{x} \in \tilde{P}_1) \ [ \ C_1(\bar{x}) \Rightarrow \exists (\bar{y} \in \tilde{P}_2) \ C_2(\bar{y}) \text{ and } O_1(\bar{x}) = O_2(\bar{y}) ]
\]

Clearly, if \( \text{cont}(Q_1, Q_2) \) holds then \( Q_1 \) and \( Q_1 \cap Q_2 \) are equivalent and therefore \( Q_1 \) is contained in \( Q_2 \). In fact the rewrite step from \( Q_1 \) to \( Q_1 \cap Q_2 \) is nothing but a form of chase of \( Q_1 \) with \( \text{cont}(Q_1, Q_2) \). The full definition of chase in section 3.4 requires an extra condition that the result of the rewrite step is not trivially equivalent with the original query, thus guaranteeing termination. Nonetheless, the rewrite step that we use here to rewrite \( Q_1 \) into \( Q_1 \cap Q_2 \) is sound even though it may not satisfy that extra condition. In particular when \( \text{cont}(Q_1, Q_2) \) is trivial, \( Q_1 \cap Q_2 \) is trivially equivalent to \( Q_1 \), thus \( Q_1 \subseteq Q_2 \).

The next lemma, using just semantic arguments involving valuations and instances, shows that the other direction holds as well, i.e. if \( Q_1 \subseteq Q_2 \) then \( \text{cont}(Q_1, Q_2) \) is trivial. Thus, PC query containment is reducible to triviality of EPCDs. Similarly, triviality of EPCDs is reducible to query containment. More precisely, let \( d \) be an EPCD:

\[
\forall (\bar{x} \in \tilde{P}_1) \ [ \ C_1(\bar{x}) \Rightarrow \exists (\bar{y} \in \tilde{P}_2(\bar{x})) \ C_2(\bar{x}, y) ]
\]

Define:

\[
\text{front}(d) \overset{\text{def}}{=} \text{select } \text{Struct}(\bar{\alpha} : \bar{x}) \text{ from } \tilde{P}_1 \bar{x} \text{ where } C_1(\bar{x}) \quad (\bar{\alpha} \text{ are fresh labels})
\]

\[
\text{back}(d) \overset{\text{def}}{=} \text{select } \text{Struct}(\bar{\alpha} : \bar{x}) \text{ from } \tilde{P}_1 \bar{x}, \tilde{P}_2 \bar{y} \text{ where } C_1(\bar{x}) \text{ and } C_2(\bar{x}, \bar{y})
\]

**Lemma 3.3.8 (Reducibility)** Let \( Q_1 \) and \( Q_2 \) be two set-valued PC queries and let \( d \) be an EPCD, with no restriction to simple types. Then:

1. \( Q_1 \subseteq Q_2 \) if and only if \( \text{cont}(Q_1, Q_2) \) is trivial
2. \( d \) is trivial if and only if \( \text{front}(d) \subseteq \text{back}(d) \)

**Theorem 3.3.9 (Containment)** Let

\[
Q_1 = \text{select } O_1(\bar{x}) \text{ from } \tilde{P}_1 \bar{x} \text{ where } C_1(\bar{x}) \quad Q_2 = \text{select } O_2(\bar{y}) \text{ from } \tilde{P}_2 \bar{y} \text{ where } C_2(\bar{y})
\]

be two PC queries (and \( T_1 \), respectively \( T_2 \), their tableaux). \( C_2, O_1 \) and \( O_2 \) are simple-type restricted, while \( C_1 \) may have set/dictionary equality. Then the following are equivalent:

(a1) \( Q_1 \subseteq^\text{unr} Q_2 \)

(a2) \( Q_1 \subseteq^\text{fin} Q_2 \)

(b) there exists a containment mapping from \( Q_2 \) into \( Q_1 \), i.e.

a homomorphism \( T_1 \overset{h}{\rightarrow} T_2 \) s. t. \( \forall (\bar{x} \in \tilde{P}_1) \ [ \ C_1(\bar{x}) \Rightarrow O_1(\bar{x}) = O_2(h(\bar{y})) ] \)

**Proof:** Suppose (a) is true. Then \( \text{eval}(O_1(\bar{x})) \in Q_1(\text{Inst}(T_1)) \subseteq Q_2(\text{Inst}(T_1)) \). Thus, there must exist a valuation \( v : T_2 \rightarrow \text{Inst}(T_1) \) such that \( \text{eval}(O_1(\bar{x})) = v(O_2(\bar{y})) \). By Corollary 3.3.7 there exists a homomorphism \( h : T_2 \rightarrow T_1 \) such that \( v(O_2(\bar{y})) = \text{eval} \circ h(O_2(\bar{y})) \). Hence, \( \text{eval}(O_1(\bar{x})) = \text{eval} \circ h(O_2(\bar{y})) \). By Lemma 3.3.1, it follows that

\(^{10}\text{Notice that this is a reduction from query containment to query equivalence.}

\(^{11}\text{A more fundamental explanation of the above soundness is given in } [\text{PT98, PT99}] \text{ in the context of a complete axiomatic system centered around the idemloop law.}

\(^{12}\text{The two reductions are basic properties that hold, in principle, for a larger class of queries and dependencies, not only path-conjunctive.}

\(^{13}\text{Here } O_1 \text{ and } O_2 \text{ are records, thus their equality is represented by a conjunction of equalities. However, since everything is well-defined, by our observation in section 3.2, this is equivalent to a set of EGDs, and thus its triviality is a PTIME condition.}
\( \bar{x} \in \bar{F}_1 \vdash C_1(\bar{x}) \Rightarrow O_1(\bar{x})=O_2(h(\bar{y})) \) is trivial. Since \( \text{Inst}(T_1) \) is finite, the same proof works for both \((a1) \Rightarrow (b)\) and \((a2) \Rightarrow (b)\).

Conversely, suppose \((b)\) holds, and let \( I \) be an instance and \( v : T_1 \rightarrow I \) a valuation. The result of \( Q_1 \) on \( I \) for valuation \( v \) is \( v(O_1) \). Observe that \( v \circ h \) is a valuation from \( T_2 \) into \( I \). Moreover, \( v \circ h(O_2) = v(O_1) \) since \( \forall (\bar{x} \in \bar{F}_1) \ [ C_1(\bar{x}) \Rightarrow O_1(\bar{x})=O_2(h(\bar{y})) ] \) is trivial. Thus the result of \( Q_2 \) on \( I \) contains \( v(O_1) \). Since \( I \) and \( v \) were chosen arbitrary, and \( I \) can be either finite or infinite, it follows that \( Q_1 \subseteq Q_2 \) in the finite and unrestricted.

**End of Proof.**

Before we can allow non-simple type variables in \( \bar{x} \) in theorem 3.3.11 we need to strengthen Lemma 3.3.5. It is the case then that Lemma 3.3.5 becomes a particular case of the following lemma when \( \bar{x} \) is taken to be empty.

**Lemma 3.3.10** Let \( T_1 = \{ \bar{x} \in \bar{F}_1, \bar{z} \in \bar{F}_1(\bar{x}) : C_1(\bar{x}) \text{ and } C_2(\bar{x}, \bar{z}) \} \) and \( T_2 = \{ \bar{x} \in \bar{F}_1, \bar{y} \in \bar{F}_2(\bar{x}) : C_1(\bar{x}) \text{ and } C_3(\bar{x}, \bar{y}) \} \) in which \( C_1 \) and \( C_2 \) may have set/dictionary equality but \( C_3 \) is simple type restricted. Then, for any valuation \( v : T_2 \rightarrow \text{Inst}(T_1) \) such that \( v(\bar{x}) = \text{val}(\bar{x}) \)\(^{13}\) there exists a context-preserving, algebraic homomorphism, \( v_c : T_2 \rightarrow \text{ClInst}(T_1) \) such that \( v = \text{collapse} \circ v_c \) and \( v_c(\bar{x}) = \text{val}_c(\bar{x}) \). Moreover, \( v \) and \( v_c \) satisfy the same set of formulas \( Q=Q' \) over \( T_2 \) with \( Q \) and \( Q' \) of simple type.

**Proof Sketch.** The proof is similar with that of Lemma 3.3.5. However, \( v_c(x) \) must be defined as \( \text{val}_c(x) \) in this case. In contrast, for \( v_c(y) \) we had the liberty to choose an element \( e \) with \( \text{collapse}(e) = v(y) \) such that the requirements for \( v_c \) were satisfied. Thus, we have to verify that indeed \( v_c(x) \in v_c(P_1) \) for any binding \( x \in P_1 \) (it is obvious that \( \text{collapse}(v_c(x)) = v(x) \)). We prove this by induction on \( x \in \bar{F}_1 \).

**Base case:** \( x \in R \) where \( R \) is a root name of set type. Then since \( x \in R \) is also in \( T_1 \) it must be the case that \( \text{val}_c \cdot x \in \text{val}_c(R) \) in \( \text{ClInst}(T_1) \). Also, we must have \( v_c(R) = \text{val}_c(R) \), thus \( v_c(x) \in v_c(R) \). The dictionary case is similar.

**Inductive case:** The root name case is as before. Suppose that \( x_n \in P_1(x_1, \ldots, x_{n-1}) \). \( v(x_n) \in v(P_1(x_1, \ldots, x_{n-1})) \).

and by the inductive hypothesis \( v_c(P_1) \) is defined and \( v_c(P_1) = \text{val}_c(P_1) \). Again, we use the fact that \( x_n \in \bar{F}_1 \)

is also in \( T_1 \) and therefore \( \text{val}_c(x_n) \in \text{val}_c(P_1) \), thus \( v_c(x_n) \in v_c(P_1) \).

On \( \bar{y} \), \( v_c \) is defined in a similar manner as in the proof of Lemma 3.3.5. Finally, as in that proof, \( v_c \) and \( v \) satisfy the same set of simple-type formulas over \( T_2 \). **End of Proof.**

**Theorem 3.3.11 (Trivial dependencies)** Let \( d \) be an EPCD: \( \forall (\bar{x} \in \bar{F}_1) [ C_1(\bar{x}) \Rightarrow \exists (\bar{y} \in \bar{F}_2(\bar{x})) C_2(\bar{x}, \bar{y}) ] \) and \( T_1 \) and \( T_2 \) such that \( d = \text{dep}(T_1, T_2) \). \( C_2 \) is simple-type restricted while \( C_1 \) may have set/dictionary equality. Then the following are equivalent:

- \((a1) \) \( d \) is trivial (unrestricted)
- \((a2) \) \( d \) is trivial (finite)
- \((b) \) there exists homomorphism \( T_1 \xrightarrow{h} T_2 \) such that \( \forall (\bar{x} \in \bar{F}_1) [ C_1(\bar{x}) \Rightarrow \bar{x}=h(\bar{y}) ] \)\(^{14}\) is trivial

**Proof:** Suppose \((a) \) holds. Then \( \text{front}(d) \subseteq \text{back}(d) \). We cannot apply Theorem 3.3.9 because \( \bar{x} \) may be of non-simple type \(^{15}\), but we extend the proof of it. Reasoning like there we conclude that there exist a valuation \( v : T_2 \rightarrow \text{Inst}(T_1) \) such that \( v(\bar{x}) = \text{val}(\bar{x}) \). By Lemma 3.3.10 (in which \( \bar{z} \) and \( C_3 \) are empty) there exists \( v_c : T_2 \rightarrow \text{ClInst}(T_1) \) such that \( v_c(\bar{x}) = \text{val}_c(\bar{x}) \) and \( v_c(C_2(\bar{x}, \bar{y})) = \text{true} \) (provided that \( C_2 \) involves only simple-type equality). Then, by Lemma 3.3.6 there exists homomorphism \( h : T_2 \rightarrow T_1 \) such that \( \text{val}_c \circ h = v_c \).

\(^{13}\)Here \( \text{val} \) and \( \text{val}_c \) are the canonical mappings from \( T_1 \) into \( \text{Inst}(T_1) \), respectively \( \text{ClInst}(T_1) \).

\(^{14}\)Same observation as in the previous theorem.

\(^{15}\)Nonetheless, the fact that we are being able to extend that proof to this case encourages the hope that the simple type restriction on \( O_1 \) and \( O_2 \) can be removed for the containment theorem.
Therefore $\text{cval}_c(h(\bar{x})) = \text{cval}_c(\bar{x})$. Then we apply the EGD lemma (the non-simple type part) and conclude that $\forall(\bar{x} \in \bar{P}) \ [ C_1(\bar{x}) \Rightarrow \bar{x} = h(\bar{x}) ]$ is trivial. For (b) $\Rightarrow$ (a) the proof is similar with the proof of (b) $\Rightarrow$ (a) in Theorem 3.3.9. End of Proof.

We remark here that Theorem 3.3.11 is a stronger statement than Theorem 3.3.9. Indeed, there is a direct proof of the containment theorem from the triviality theorem that uses the reducibility lemma.

**Corollary 3.3.12** Existence of a homomorphism of tableaux, and therefore containment/equivalence of PC queries and EPCD triviality are decidable and in NP (and hence NP-complete by [CM77]).

### 3.4 The PC Chase

This section introduces the path-conjunctive chase and discusses the important issues related with it: termination, completeness, confluence, etc. The main theorems are also stated here, while the proofs of the theorems are relegated for the next sections. Before defining the chase we observe that the reducibility lemma holds in the presence of dependencies as well.

**Lemma 3.4.1 (Reducibility under dependencies)** Let $Q_1$ and $Q_2$ be two set-valued PC queries, let $d$ be an EPCD, and let $D$ be a set of EPCDs, all with no restriction to simple types. Then:

1. $Q_1 \subseteq_D Q_2$ if and only if $D \models \text{cont}(Q_1, Q_2)$
2. $D \models d$ if and only if $\text{front}(d) \subseteq_D \text{back}(d)$

**Definition 3.4.2 (Chase step)** Given an EPCD $d$ of the form $\forall(\bar{r} \in \bar{R}) \ [ B_1(\bar{r}) \Rightarrow \exists(\bar{s} \in \bar{S}(\bar{r})) \ B_2(\bar{r}, \bar{s}) ]$ the following rewrite is a chase step of $T$ with $d$:

$$T = \{ \bar{x} \in \bar{P}; C(\bar{x}) \} \xrightarrow{d} T' = \{ \bar{x} \in \bar{P}; \bar{s} \in \bar{S}(h(\bar{s})); C(\bar{x}) \text{ and } B_2(h(\bar{r}), \bar{s}) \}$$

provided that:

1. $T \xleftarrow{h'} \{ \bar{r} \in \bar{R}; B_1(\bar{r}) \}$ is a homomorphism, and
2. there is no homomorphism $T \xleftarrow{h'} T'$ such that $\forall(\bar{x} \in \bar{P}) \ [ C(\bar{x}) \Rightarrow \bar{x} = h'(\bar{x}) ]$ is trivial.

If conditions (1) and (2) above are satisfied we also say that $d$ is applicable to $T$. For a PC query $Q$ with tableau $T$ chasing $Q$ means chasing $T$. For an EPCD $\text{deg}(T, T')$ chasing $d$ means chasing $T$. A chase sequence with a set of EPCDs $D$ is a sequence of tableaux obtained by successive chase steps each with some dependency $d \in D$ (same $d$ can be used repeatedly). We say that a sequence starting with $T$ terminates if it reaches a tableau $T'$ that cannot be chased with any $d \in D$. Although in general $T'$ depends on the choice of terminating chase sequence, we shall denote the result of a terminating chase sequence by $\text{chase}_D(T)$ and extend the same notation to queries and dependencies.

**Relational vs. path-conjunctive chase.** Remark that the relational chase is just a particular case of the path-conjunctive case. Therefore all the negative results about the relational chase transfer to the path-conjunctive one as well. In particular, the chase may not terminate, and two terminating chase sequences may end up in different (non-isomorphic) tableaux. The rest of this section shows that the positive results about the relational chase still hold for the PC case. In particular, the chase is a complete proof procedure for dependency implication, and is terminating (and thus a decision procedure), confluent, and semantic invariant for full EPCDs. In section 3.6.2
we also show that the complexity of the PC chase with full EPCDs is the same as the complexity of the relational with full TGDs.

We discuss first two important properties of the chase:

- **Soundness.** Even in the absence of condition (2) the rewrite in the definition of the chase step is a valid sound rewrite, that preserves equivalence of queries/dependencies. This can be easily justified using semantic arguments (valuations). [PT98] gives a different proof of the soundness of chase in the context of an axiomatic equational theory of sets. The proof there will essentially be a simulation of the chase step in terms of more atomic rewrite steps based on a fundamental equivalence law for sets, the *idemloop* law.

- **Non-trivial rewriting.** Condition (2) of Definition 3.4.2 guarantees that a tableau is not to be chased unless it is changed in a non-trivial way. This ensures that the chase makes enough progress so that at the end (if the chase terminates) the resulting tableau (its canonical instance) satisfies all the dependencies that applied. This is an important idea used in proving completeness of the chase.

The following lemma makes more precise the above discussion.

**Lemma 3.4.3 (Chase properties)** (1a) If \( Q \) is a PC query s.t. \( Q \vdash^d Q' \) then \( Q = Q' \) is a consequence of \( d \) (i.e. \( Q \) and \( Q' \) are equivalent under all instances satisfying \( d \)). (1b) If \( d' \) is an EPCD s.t. \( d \vdash^d d' \) then \( d' = d'' \) is a consequence of \( d \). (2) If \( \text{Inst}(T) \not\models d \) then \( d \) is applicable to \( T \) (Here, \( B_1 \) is simple-type restricted).

**Non-simple type equality.** It is not hard to find a counterexample to Lemma 3.4.3, part (2), for the case when \( B_1(\vec{r}) \) involves non-simple type equality. Consider the following EPCD and tableau:

\[
d = \forall (a \in R) \forall (b \in S) \left[ R \equiv S \Rightarrow a = b \right]
T = \{ a \in R, a' \in R, b \in S, b' \in S : a = b \text{ and } a' = b' \}.
\]

Then it is easy to see that \( \text{Inst}(T) \not\models d \), since \( R \) and \( S \) are equal in \( \text{Inst}(T) \) (see figure 3.4b), but still there are elements in \( R \), respectively \( S \) that are not equal. Also, \( d \) is not applicable to \( T \) because there is no homomorphism to map the tableau \( \{ a \in R, b \in S : R \equiv S \} \) to \( T \). The problem seems to be related to the fact that \( R \) and \( S \) are equated "artificially" in \( \text{Inst}(T) \) (as a result of our enforcement of extensionality) even though their equality does not follow as a logical consequence from \( a = b \text{ and } a' = b' \) (this is in fact the reason why \( d \) is not applicable to \( T \)). The conclusion is that there are limitations on how useful is \( \text{Inst}(T) \) in characterizing dependencies in the presence of non-simple type equality. We will relax later (in section 3.6.3) our notion of instance to **weak instance** (not necessarily extensional) and we will draw some interesting parallels between the two of them. In particular, \( C\text{Inst}(T) \) is a weak instance and one can observe that \( C\text{Inst}(T) \models d \) (\( R \) and \( S \) are distinct, see figure 3.4a).

![Diagram](image)

**Figure 3.4:** Counter-example to Lemma 3.4.3 part 2 for non-simple type equality in \( B_1(\vec{r}) \)

**Terminating chase as a decision procedure.** Part (2) of the previous lemma allows us to observe that, for any **terminating** chase sequence \( T = I_0 \rightarrow \ldots \rightarrow T_n \) of \( T \) by a set of EPCDs \( D \), \( \text{Inst}(T_n) \models D \). The

\footnote{In the absence of condition (2), chasing with trivial constraints becomes possible and certainly does not terminate!}
main idea used in the proof of the next theorem is then the following. If \( D \models d \) (under all instances) then since \( \text{Inst}(T_n) \models D \) then it must be the case that \( \text{Inst}(T_n) \models d \). Thus we are able to infer the existence of a certain valuation/homomorphism\(^\text{17}\). The fact that Lemma 3.4.3, part 2, fails when dependencies in \( D \) may allow for non-simple type equality in \( B_1(r^\prime) \) implies that we cannot use \( \text{Inst}(T_n) \) as the model that we need for the proof. However, it is not a counterexample to the theorem itself. Other means for proving the theorem might exist, at least, in principle. We will limit ourselves here to simple type equality in \( B_1(r^\prime) \).

In the following we use the notation \( \text{chase}_D(Q) \) for a\(^\text{18}\) result of a terminating chase sequence applied to a PC query \( Q \) using dependencies from a set \( D \) (similar notation for EPCDs: \( \text{chase}_D(d) \)).

**Theorem 3.4.4 (Terminating chase)** Let \( D \) be a set of EPCDs.

1. Let \( Q_1, Q_2 \) be set-valued PC queries such that some chasing sequence of \( Q_1 \) with \( D \) terminates (with \( \text{chase}_D(Q_1) \)). Moreover, \( D \) is restricted as discussed above, while \( Q_1 \) and \( Q_2 \) are restricted as in Theorem 3.3.9. Then the following are equivalent:

   (a1) \( Q_1 \subseteq^{\text{unr}} D Q_2 \)
   (a2) \( Q_1 \subseteq^{\text{fin}} D Q_2 \)

   (b1) \( \text{chase}_D(Q_1) \subseteq^{\text{unr}} D Q_2 \)
   (b2) \( \text{chase}_D(Q_1) \subseteq^{\text{fin}} D Q_2 \)

2. Let \( d \) be an EPCD such that some chasing sequence of \( d \) with \( D \) terminates. Moreover, \( D \) is restricted as discussed above, while \( d \) is restricted as in Theorem 3.3.11. Then the following are equivalent:

   (a1) \( D \models^{\text{unr}} d \)
   (a2) \( D \models^{\text{fin}} d \)

   (b1) \( \text{chase}_D(d) \) is trivial (unr)
   (b2) \( \text{chase}_D(d) \) is trivial (fin)

**Full EPCDs.** This is a class of dependencies that generalizes the relational full dependencies [AHV95] (originally called total tgd’s and egd’s in [BV84b]). Since we work with “tuple” variables and we also have dictionaries, the definition needs a lot more care than in the first-order case. Before we define full EPCDs we need to consider a slight extension of path expressions that takes into account record constructors. Formally, an *extended path* \( EP \) is described by the following:

\[
EP ::= \text{Struct}(A_1 : EP_1, \ldots, A_n : EP_n) \mid P
\]

where \( P \) is a path expression as defined before. Note that the record constructor and the constructors that occur in \( P \) are not orthogonal, i.e. cannot be composed in an arbitrary way. This restricted form suffices for our purposes. We also note that we represent equality between extended paths, componentwise, as conjunction of equalities between paths. We say that an extended path \( EP \) is well-defined (see section 3.2) over some tableau \( T \) if every \( P \) occuring in \( EP \) is well-defined over \( T \).

**Definition 3.4.5 (Full dependencies)** Let \( d \) def \( \forall(r' \in \vec{R}) \exists(s \in \vec{S}(r')) \ B_1(r') \Rightarrow B_2(r', s) \) be an EPCD and \( T_r \) and \( T_{r^\prime} \) be the two tableaux such that \( d = \text{dep}(T_r, T_{r^\prime}) \). Then \( d \) is full if, for any path \( Q(r, s) \) over \( T_{r^\prime} \), there exists an extended path \( EP_Q(r') \) over \( T_r \) such that the following EGD is trivial:

\[
\forall(r' \in \vec{R}) \forall(s' \in \vec{S}(r')) \left[ B_1(r') \text{ and } B_2(r', s) \right] \Rightarrow Q'(r, s) = EP_Q(r')
\]

Informally, a full EPCD asserts the existence of elements \( s_1, \ldots, s_n \) such that all the new paths that are well-defined over \( T_{r^\prime} \) are determined (up to record constructors) in terms of \( r' \). We remark that the new paths may

\(^{17}\)The proof in the relational case uses the same idea [BV84b]

\(^{18}\)There may be more than one!
include paths accessible from both \( \hat{s} \) and \( \hat{r} \). For example, if \( r = s \) occurs in \( B_2(\hat{r}, \hat{s}) \) and \( s \in \text{dom} \mathcal{M} \) is a binding in the \( \exists \) part of \( d \) then \( \mathcal{M}[r] \) becomes well-defined over \( T_r \), although it may have not been so over \( T_r \).

Checking whether an EPCD is full is in PTIME (in the size of the dependency). This is because, as we observed earlier, there are polynomially many paths over \( T_r \) and triviality of EGDs is in PTIME. For each well-defined path over \( T_r \), if it is not a record, we only need to check whether its \( \equiv \)-equivalence class contains a path well-defined over \( T_r \), while if it is a record, we may need to go recursively on each attribute of it and check its corresponding \( \equiv \)-equivalence class.

We state here the main theorem regarding chase with full EPCDs: termination. The main idea in the relational case is that there are only finitely many records that can be built out of a finite number of possible values for their attributes, thus chasing with full TGDs is terminating ([BV84b]). We will also make use of this idea, although the proof is significantly more complicated in our nested/dictionary framework (see section 3.6).

**Theorem 3.4.6 (Termination)** If \( D \) is a set of full EPCDs and \( T \) is a tableau, both simple-type restricted, then any chase of \( T \) by \( D \) terminates.

**Corollary 3.4.7** Set-valued PC query containment/equivalence under full EPCDs and logical implication of EPCDs from full EPCDs are reducible to each other, their unrestricted and finite versions coincide, and both are decidable.

The chase with full EPCDs also enjoys the following nice properties:

**Theorem 3.4.8 (Confluence)** For any two terminal chase sequences of \( T \) with a set \( D \) of full EPCDs, \( T \xrightarrow{d_1} T_1 \xrightarrow{d_2} \ldots \xrightarrow{d_n} T_n \) and \( T \xrightarrow{d'_1} T'_1 \xrightarrow{d'_2} \ldots \xrightarrow{d'_m} T'_m \), it must be the case that \( \text{CInst}(T_n) \) and \( \text{CInst}(T'_m) \) are isomorphic (and therefore \( \text{Inst}(T_n) \) and \( \text{Inst}(T'_m) \) are isomorphic as well).

Note that we cannot hope that \( T_n \) and \( T'_m \) are "equal" (even modulo variable renaming) because the path-conjunctions may be different, although logically equivalent.

**Theorem 3.4.9 (Semantic invariance)** Let \( D \) and \( D' \) be two equivalent sets of full EPCDs. Assume \( T_n \) and \( T_m \) are the resulting tableaux of two arbitrary terminal chase sequences of a tableau \( T \) with \( D \) and, respectively, \( D' \). Then \( \text{CInst}(T_n) \) and \( \text{CInst}(T_m) \) are isomorphic.

**Non-terminating chase.** We also generalize the results of [BV84b] for non-terminating chase, that is, we show that in the PC case the chase is still a proof procedure. As opposed to the relational case where one can also invoke Gödel’s completeness theorem, the recursive enumerability of the PC problem was not obvious.

Let \( \text{dep}(T, T') \) be an EPCD where \( T = \{ \bar{x} \in \bar{P} \mid C(\bar{x}) \} \) and \( T' = \{ \bar{x} \in \bar{P}, \bar{y} \in \bar{R}(\bar{x}) \mid C(\bar{x}) \text{ and } D(\bar{x}, \bar{y}) \} \). Suppose \( T_m = \{ \bar{x}_m \in \bar{P}_m \mid C_m(\bar{x}_m) \} \) is the \( m \)th tableau in a chase sequence (not necessarily terminating) \( T = T_0 \rightarrow \ldots \rightarrow T_n \rightarrow \ldots \) of \( T \) by a set of EPCDs \( D \). We use the notation \( \text{chase}^m_D(d) \rightarrow_{d \in D} \text{dep}(T_m, T'_m) \) where \( T'_m = \{ \bar{x}_m \in \bar{P}_m, \bar{y} \in \bar{R}(\bar{x}) \mid C_m(\bar{x}_m) \text{ and } D(\bar{x}, \bar{y}) \} \). Similarly, for a PC query \( Q \) with tableau \( T \), we denote by \( \text{chase}^m_D(Q) \) the PC query obtained by replacing \( T \) with \( T_m \).

We show that if \( D \models \text{dep}(T, T') \) then for any infinite chase of \( T \) by \( D \) there is a tableau \( T_m \) (with \( m \) finite) in the chase sequence such that \( \text{dep}(T_m, T'_m) \) is trivial. A similar result holds for query containment/equivalence. The result generalizes the ones of [BV84b] regarding the relational case and the full proof is given in section 3.7.
Theorem 3.4.10 (Non-terminating chase) Let \( D \) be a set of EPCDs. In the following, \( D, d, Q_1 \) and \( Q_2 \) are restricted as in Theorem 3.4.4.

1. Let \( Q_1, Q_2 \) be PC queries and consider an arbitrary infinite chasing sequence of \( Q_1 \) with \( D \). The following are equivalent:

   (a) \( Q_1 \subseteq_D^\text{unr} Q_2 \)
   (b) there is a finite \( m \) such that:
   
   \[ \text{(1) } \text{chase}_{D}^m(Q_1) \subseteq_D^\text{unr} Q_2 \quad \text{and/or} \quad \text{(2) } \text{chase}_{D}^m(\text{cont}(Q_1, Q_2)) \text{ is trivial (unr)} \]
   (c) \( D \models_D^\text{unr} \text{cont}(Q_1, Q_2) \)

2. Let \( d \) be an EPCD and consider an arbitrary infinite chasing sequence of \( d \) with \( D \). The following are equivalent:

   (a) \( D \models_D^\text{unr} d \)
   (b) there is a finite \( m \) such that:
   
   \[ \text{(1) } \text{chase}_{D}^m(d) \text{ is trivial (unr) and/or } \text{(2) } \text{chase}_{D}^m(\text{front}(d)) \subseteq_D^\text{unr} \text{back}(d) \]
   (c) \( \text{front}(d) \subseteq_D^\text{unr} \text{back}(d) \)

3.5 Terminating Chase

The following lemma is easily verified (in fact we already proved a particular case of it in Lemma 3.2.2).

Lemma 3.5.1 Let \( T = \{ \bar{x} \in \bar{P} \ ; \ C(\bar{x}) \} \) and \( T' = \{ \bar{x} \in \bar{P}, \bar{y} \in \tilde{S}(\bar{x}) \ ; \ C(\bar{x}) \text{ and } D(\bar{x}, \bar{y}) \} \) be two tableaux, and let \( G, \simeq, \text{ and } G', \simeq' \) be the set of well-defined paths and the congruence closure for \( T \) and, respectively, \( T' \). Then \( G \subseteq G' \) and \( \simeq \subseteq \simeq' \).

For the following lemma, assume that \( T \xrightarrow{d} T' \), and \( d = \text{dep}(T_r, T_{rs}), T, T' \), and \( h : T_r \rightarrow T \) are as in the definition of the chase step. Let \( \text{id}_{\bar{x'}} \) be the identity mapping on \( \bar{s} \).

Lemma 3.5.2 \( h \cup \text{id}_{\bar{x'}} : T_{rs} \rightarrow T' \) is a homomorphism.

Proof. We will use \( G_{rs}, G', G_r, \) and \( G \) to denote the sets of well-defined paths for, respectively, \( T_{rs}, T', T_r \) and \( T \). Similar notation for \( \simeq \). Let \( h' \overset{\text{def}}{=} h \cup \text{id}_{\bar{x'}} \). First we prove, by induction on the derivation of \( Q : G_{rs} \) or \( Q_1 \simeq_{rs} Q_2 \), that \( Q : G_{rs} \) implies \( h'(Q) : G' \) and \( Q_1 \simeq_{rs} Q_2 \) implies \( h'(Q_1) \simeq' h'(Q_2) \). The interesting cases are (lookup-add) and (eq).

For (lookup-add) suppose \( Q \) is of the form \( Q'[u] \) where \( u \) is a variable in \( T_{rs} \) (either an \( r \) or an \( s \) ). Then to derive \( Q'[u] : G_{rs} \), we must have had two possibilities. The first case:

\[
\frac{u \simeq_{rs} r_i, \quad r_i \in R_i \in T_{rs}, \quad R_i \simeq_{rs} \text{dom } Q'}{Q'[u] : G_{rs}}
\]

Then, by the inductive hypothesis, we have \( h'(u) \simeq' h'(r_i) \) and \( h'(R_i) \simeq' \text{dom } h'(Q') \). On the other hand, \( h'(r_i) = h(r_i) = x_k \) for some \( x_k \in P_k \) in \( T \). Using condition (2) of Proposition 3.3.4, since \( h \) is a homomorphism,
we obtain that $P_k \simeq h(R_k)$ and thus $P_k \simeq h'(R_k)$. By Lemma 3.5.1, we must have $P_k \simeq h'(Q')$. Putting it all together, the following instance of (lookup-add) is applicable:

\[
\begin{align*}
\forall u, & \quad h'(u) \simeq h'(r_k), \quad h'(r_k) \in P_k \text{ in } T', \quad P_k \simeq h'(Q') \\
& \quad \therefore h'(Q')[h'(u)] : G'
\end{align*}
\]

thus concluding that $h'(Q'[u]) : G'$. The second case:

\[
\begin{align*}
u \simeq r, s_i, & \quad s_i \in S_i \text{ in } T_{rs}, \quad S_i \simeq r, \text{ dom } Q' \\
& \quad Q'[u] : G_{rs}
\end{align*}
\]

is simpler. Again, by the inductive hypothesis, $h'(u) \simeq h'(s_i)$ and $h'(S_i) \simeq \text{dom } h'(Q')$. But since $h'(s_i) = s_i$ and $h'(S_i) = S_i$ and $s_i \in S_i$ occurs in $T'$, we can immediately apply (lookup-add) to conclude that $h'(Q'[u]) : G'$.

For the (eq) rule, we have again two cases. The first case is when $Q_1 \simeq_{rs} Q_2$ is obtained by:

\[
\begin{align*}
Q_1 & = Q_2 \text{ occurs in } B_1(r), \quad Q_1 : G_{rs}, \quad Q_2 : G_{rs} \\
& \quad Q_1 \simeq_{rs} Q_2
\end{align*}
\]

It must be the case that $Q_1 = Q_1(r)$ and $Q_2 = Q_2(r)$, and $Q_1 : G_r, Q_2 : G_r$. Then, by (eq), $Q_1 \simeq_{r} Q_2$. Since $h$ is a homomorphism, it follows by Proposition 3.3.4 that $h(Q_1) \simeq h(Q_2)$, and therefore, $h'(Q_1) \simeq h'(Q_2)$. By Lemma 3.5.1, $h'(Q_1) \simeq h'(Q_2)$ as well. The second case:

\[
\begin{align*}
Q_1 & = Q_2 \text{ occurs in } B_2(r, s), \quad Q_1 : G_{rs}, \quad Q_2 : G_{rs} \\
& \quad Q_1 \simeq_{rs} Q_2
\end{align*}
\]

Then, since $h'(B_2(r, s)) = B_2(h(r), s)$, we have that $h'(Q_1) = h'(Q_2)$ occurs in $B_2(h(r), s)$. On the other hand, by the inductive hypothesis, $h'(Q_1) : G'$ and $h'(Q_2) : G'$. Thus, we can apply (eq) to conclude that $h'(Q_1) \simeq h'(Q_2)$.

We still need to show that $h'$ satisfies condition (2) of Proposition 3.3.4. We already showed it in the above proof, but we repeat it. Suppose $r_k \in R_k$ is a binding in $T_{rs}$. Then $h'(r_k) = h(r_k) = x_k$ for some $x_k \in P_k$ in $T$. By Proposition 3.3.4, $h$ is a homomorphism implies $h(R_k) \simeq P_k$ and thus $h'(R_k) \simeq P_k$ (by Lemma 3.5.1). For the case $s_i \in S_i$ we use the fact that $h'(s_i) = s_i$ and that $h'(S_i) = S_i$, plus reflexivity. End of Proof.

Proof of Lemma 3.4.3 part (2). We remind the statement here: If $T$ is a tableau and $d$ is an EPCD as in the chase step definition s.t. $B_1$ is simple-type restricted then, if $\text{Inst}(T) \nvdash d$ then $d$ is applicable to $T$. We prove the contrapositive. Assume the same notations as in the definition of the chase step and let $T_r$ and $T_{rs}$ be the two tableaux such that $d = \text{dep}(T_r, T_{rs})$. Assume $d$ is not applicable to $T$, and let $v : T_r \rightarrow \text{Inst}(T)$ an arbitrary valuation. We need to extend this to a valuation $v' : T_{rs} \rightarrow \text{Inst}(T)$ such that $v'(r) = v(r)$ and $v'(B_2(r, s)) = \text{true}$. By Corollary 3.3.7 there exists a homomorphism $h : T_r \rightarrow T$ such that $\text{val} \circ h = v$. Since $d$ is not applicable to $T$, it must be the case that there exists homomorphism $h' : T' \rightarrow T$ such that $\forall (x \in \tilde{P}) \quad [C(\tilde{x}) \Rightarrow \tilde{x} = h'(\tilde{x})]$. Now, observe that $h'' \overset{\text{def}}{=} h \cup \text{id}_{\tilde{T}}$ is a homomorphism from $T_{rs}$ into $T'$, by Lemma 3.5.2. Take $v' \overset{\text{def}}{=} \text{val} \circ h' \circ h''$.

It is easy to see that $v'$ is context-preserving. Moreover, $v'(r) = \text{val}(h'(h(r)))$. Since $\forall (x \in \tilde{P}) \quad [C(\tilde{x}) \Rightarrow \tilde{x} = h'(\tilde{x})]$ is trivial, it follows by the EGD Lemma, part 2, that $\text{val}(\tilde{x}) = \text{val}(h'(\tilde{x}))$ and thus $\text{val}(h'(r)) = \text{val}(h'(h(r)))$. Therefore $v'(r) = \text{val}(h'(r))$, and hence $v'(r) = v(r)$. Finally, $v'(B_2(r, s)) = B_2(\text{val}(h'(h(r))), \text{val}(h'(\tilde{s})))$.

Now, since $\forall (x \in \tilde{P}) \quad [C(\tilde{x}) \Rightarrow B_2(h'(h(r)), h'(\tilde{s}))]$ is trivial ($h'$ is a homomorphism), the last term equals $\text{true}$.
(again the EGD Lemma, part 2). We conclude that $\text{Inst}(T) \models d$. **End of Proof.**

**Proof of Theorem 3.4.4.** Consider part 2 first. If $\text{chase}_D(d)$ is trivial then $D \models d$, by repeatedly applying Lemma 3.4.3. For the interesting direction assume $D \models d$. Suppose $d = \forall(x \in \bar{P}) [ C(x) \Rightarrow \exists(y \in \bar{R}(x)) D(x, y) ]$, let $T_x$ and $T_{xy}$ be such that $d = \text{dep}(T_x, T_{xy})$. Let the result of chasing $T_x$ with $D$ be the tableau \( T = \{ \bar{x} \in \bar{P}, \bar{z} \in \bar{Q}(\bar{x}) ; C(\bar{x}) \text{ and } B(\bar{x}, \bar{z}) \} \). Then $\text{chase}_D(d) = \text{dep}(T, T')$ where $T' = \{ \bar{x} \in \bar{P}, \bar{z} \in \bar{Q}(\bar{x}), \bar{y} \in \bar{R}(\bar{x}) ; C(\bar{x}) \text{ and } B(\bar{x}, \bar{z}) \text{ and } D(\bar{x}, \bar{y}) \}$. To show that this is trivial, we show that there is a homomorphism from $T'$ into $T$ that is the identity on $\bar{x}, \bar{z}$.

$v = \text{val} \circ \text{id}_x$ is a valuation from $T_x$ into $\text{Inst}(T)$. Since $\text{Inst}(T) \models D$ and $D \models d$, we have $\text{Inst}(T) \models d$, too. Thus there exists a valuation $v' : T_{xy} \to \text{Inst}(T)$ such that $v'(\bar{x}) = v(\bar{x})$ (see figure 3.5a). Thus, $v'(\bar{x}) = \text{val}(\bar{x})$ where $\text{val}$ is the canonical valuation corresponding to $T$. Therefore, we can apply Lemma 3.3.10 and conclude that there exists a mapping $v'_c : T_{xy} \to C\text{Inst}(T)$ (see figure 3.5b) such that $v'_c(\bar{x}) = \text{val}_c(\bar{x}), v'_c = \text{collapse} \circ v'_c$ and, making use of the fact that $D(\bar{x}, \bar{y})$ is simple type restricted, $v'_c(\bar{x}, \bar{y}) = \text{true}$.

**Figure 3.5: Proof of Theorem 3.4.4**

Then, by Lemma 3.3.6 there exists homomorphism $h' : T_{xy} \to T$ such that $\text{val}_c \circ h' = v'_c$. Thus, $\text{val}_c((h'(\bar{x})) = \text{val}_c(\bar{x})$. Now, consider $h = h' \cup \text{id}_x$. Obviously, $h$ is a context-preserving mapping from $T'$ into $T$. Moreover, $\text{val}_c(h(\bar{x})) = \text{val}_c((h'(\bar{x})) = \text{val}_c(\bar{x})$, hence by the EGD Lemma part (2),

$$\forall(\bar{x} \in \bar{P}) \forall(\bar{z} \in \bar{Q}(\bar{x})) [ C(\bar{x}) \text{ and } B(\bar{x}, \bar{z}) \Rightarrow h(\bar{x}) = \bar{x} ]$$

is trivial. A similar triviality holds for $\bar{z}$ since $h$ is defined to be the identity on $\bar{z}$. Finally, $\text{val}_c(h(D(\bar{x}, \bar{y}))) = \text{val}_c((h'(D(\bar{x}, \bar{y})))) = v'_c(D(\bar{x}, \bar{y})) = \text{true}$. Again, by the EGD Lemma part (2),

$$\forall(\bar{x} \in \bar{P}) \forall(\bar{z} \in \bar{Q}(\bar{x})) [ C(\bar{x}) \text{ and } B(\bar{x}, \bar{z}) \Rightarrow D(h(\bar{x}), h(\bar{y})) ]$$

is trivial. (The case for $B(\bar{x}, \bar{z})$ is immediate.) We conclude that $h$ is a homomorphism from $T'$ into $T$, identity on the variables of $T$. Thus, $\text{chase}_D(d)$ is trivial.

Part 1 follows from Part 2 by applying the two Reducibility lemmas. First, $Q_1 \subseteq_D Q_2$ iff $D \models \text{cont}(Q_1, Q_2)$, by Lemma 3.4.1. But, by part 1, $D \models \text{cont}(Q_1, Q_2)$ if $\text{chase}_D(\text{cont}(Q_1, Q_2))$ is trivial. Now, we observe that $\text{chase}_D(\text{cont}(Q_1, Q_2)) = \text{cont}(\text{chase}_D(Q_1), Q_2)$. Then, $\text{cont}(\text{chase}_D(Q_1), Q_2)$ trivial iff $\text{chase}_D(Q_1) \subseteq Q_2$, by Lemma 3.3.8. **End of Proof.**
3.6 Chase with Full EPCDs

The following results are concerned with one step of chasing with a full EPCD \( d \). Let \( T = \{ \tilde{x} \in \tilde{P} \mid C(\tilde{x}) \} \) be a PC tableau. Let \( T \xrightarrow{d} T' \) be a chase step, where \( T' = \{ \tilde{x} \in \tilde{P}, \tilde{s} \in \tilde{S}(\tilde{r}) \mid C(\tilde{x}) \text{ and } B_2(\tilde{h}(\tilde{r}), \tilde{s}) \} \) for some homomorphism \( T \xleftarrow{\tilde{r}} \{ \tilde{r} \in \tilde{R} \mid B_1(\tilde{r}) \} \). Moreover, we assume that \( C(\tilde{x}), B_1(\tilde{r}) \) and \( B_2(\tilde{h}(\tilde{r}), \tilde{s}) \) are simple-type restricted. Then every well-defined path over \( T' \) is determined by a well-defined extended path over \( T \). The proof consists of several simple observations.

**Lemma 3.6.1** Let \( T = \{ \tilde{x} \in \tilde{P} \mid C(\tilde{x}) \} \) be any tableau such that \( C(\tilde{x}) \) is simple type restricted. Then \( P \simeq \text{dom } Q \) implies that \( P = \text{dom } P' \) for some \( P' \) such that \( P' \simeq Q \).

**Proof.** Induction on the derivation of \( P \simeq \text{dom } Q \) or \( \text{dom } Q \simeq P \). We give the proof for the case \( P \simeq \text{dom } Q \) (the other one is symmetric). The cases that may be possible, given the special form \( \text{dom } Q \), are as follows. First, (eq): this is not possible since \( C(\tilde{x}) \) is simple-type restricted. Second, (\text{dom} - cong): then \( P \) must be of the form \( \text{dom } P' \) and \( P' \simeq Q \). (refl) is trivial. For (trans), there exists some \( P_1 \) such that \( P \simeq P_1 \) and \( P_1 \simeq \text{dom } Q \). Applying the inductive hypothesis for \( P_1 \simeq \text{dom } Q \), it must be that \( P_1 = \text{dom } P_1' \) with \( P_1' \simeq Q \). Therefore we can use the inductive hypothesis again, this time for \( P \simeq \text{dom } P_1' \) to obtain that \( P = \text{dom } P' \) for some \( P' \simeq P_1' \). Then we apply transitivity. For (sym) we use the inductive hypothesis for \( \text{dom } Q \simeq P \) and (sym). **End of Proof.**

Coming back to the previous notations regarding the chase step, recall that, by Lemma 3.5.2, \( h' = h \cup \text{id}_r \) is a homomorphism from \( T_{rs} \) into \( T' \). As we did before, we will use \( G_{rs}, G', G_r, \) and \( G \) to denote the sets of well-defined paths for, respectively, \( T_{rs}, T', T_r \) and \( T \). Similar notation for \( \simeq \).

**Lemma 3.6.2** For any \( Q(\tilde{x}, \tilde{s}) : G' \) either \( Q \simeq^! Q_0 \), for some \( Q_0 : G \), or \( Q \simeq^! h'(Q_{rs}) \), for some \( Q_{rs} : G_{rs} \).

**Proof.** Induction on the derivation of \( Q : G' \). The base cases are trivial. For (var-add): if \( Q = x \) then \( Q : G \) and \( Q \simeq Q \), therefore \( Q \simeq^! Q \). If \( Q = s \) then \( Q = h'(s) \) and \( s : G_{rs} \). For (proj-add): \( Q.A : G' \) and \( Q : G' \). We apply the inductive hypothesis for \( Q \) and we have two cases. First, suppose that \( Q \simeq^! Q_0 \), for some \( Q_0 : G \). Then, by (proj-add), \( Q_0.A : G \) and therefore \( Q_0.A : G' \). Hence, by (proj-cong), \( Q.A \simeq^! Q_0.A \). Second case, suppose that \( Q \simeq^! h'(Q_{rs}) \), for some \( Q_{rs} : G_{rs} \). Then \( Q_{rs}.A : G_{rs} \). By (proj-cong), \( Q.A \simeq^! h'(Q_{rs}).A = h'(Q_{rs}.A) \). The case (\text{dom} - add) is similar. Finally, the interesting case is (lookup-add):

\[
\begin{align*}
 \quad u & \simeq^! v, \quad v \in R \text{ in } T', \quad R \simeq^! \text{dom } Q \\
 Q[u] & : G'
\end{align*}
\]

We know, by the previous lemma and by our simple-type restriction, that \( R = \text{dom } R' \) for some \( R' \simeq Q \). Then, by (lookup-add), \( R'[v] : G' \) and, by (lookup-cong), \( Q[u] \simeq^! R'[v] \). Now, we have two possible cases for the variable \( v \). First, \( v = x \) and \( x \in \text{dom } R' \) in \( T \). Then \( R'[v] : G \). Second, \( v = s \), \( \text{dom } R' \) is of the form \( \text{dom } S(\tilde{r}) \), and \( s \in \text{dom } S(\tilde{r}) \) in \( T_{rs} \). Thus, \( S(\tilde{r})[s] : G_{rs} \), and, moreover, \( R'[v] = S(\tilde{h}(\tilde{r}))[s] = h'(S(\tilde{r})[s]) \). **End of Proof.**

**Lemma 3.6.3** For any \( Q(\tilde{x}, \tilde{s}) \) over \( T' \) there exists an \( EP_Q(\tilde{x}) \) over \( T \) such that the following EGD is trivial:

\[
\forall(\tilde{x} \in \tilde{P}) \forall(\tilde{s} \in \tilde{S}(\tilde{r})) \quad [ C(\tilde{x}) \text{ and } B_2(\tilde{h}(\tilde{r}), \tilde{s}) \Rightarrow Q(\tilde{x}, \tilde{s}) = EP_Q(\tilde{x}) ]
\]

**Proof.** This is an immediate application of the previous lemma and of Definition 3.4.5. If \( Q \simeq^! Q_0 \), for some \( Q_0 : G \) then take \( EP_Q = Q_0 \). If \( Q \simeq^! h'(Q_{rs}) \) for some \( Q_{rs} : G_{rs} \), then there must exist some extended
path $EP_{Q_n}(\vec{r})$ well-defined over $T_n$ such that $EP_{Q_n} \simeq_{r_{n}} Q_{r_n} \ (\text{componentwise})$. It follows immediately ($h'$ is a homomorphism) that $h'(EP_{Q_n}) \simeq' h'(Q_{r_n})$, and thus, $h'(EP_{Q_n}) \simeq' Q$. Then take $EP_Q = h'(EP_{Q_n})$. \footnote{Here we suitably extend $h'$ to extended paths.} \textbf{End of Proof.}

The previous lemma generalizes to any sequence of chase steps, given that all the EPCDs involved are full. Let $T = \{ \vec{x} \in \vec{P} : C(\vec{x}) \}$ be a tableau and $T \xrightarrow{d_1} T_1 \xrightarrow{d_2} \ldots \xrightarrow{d_n} T_n$ be a chase sequence such that $d_i$ is full, for any $i$. $T_i$ must be of the form $\{ \vec{x} \in \vec{P}, \vec{y}_1 \in \vec{Q}_1(\vec{x}), \ldots, \vec{y}_n \in \vec{Q}_n(\vec{x}, \ldots, \vec{y}_{n-1}) \ ; \ C(\vec{x}) \text{ and } B_1(\vec{x}, \vec{y}_1) \text{ and } \ldots \text{ and } B_i(\vec{x}, \ldots, \vec{y}_i) \}$, for $i = 1, \ldots, n$.

\textbf{Corollary 3.6.4} For any path $Q(\vec{x}, \vec{y}_1, \ldots, \vec{y}_n)$ over $T_n$, there exists an extended path $EP_Q(\vec{x})$ over $T$ such that the following EGD is trivial:

$$\forall(\vec{x} \in \vec{P}) \ldots \forall(\vec{y}_n \in \vec{Q}_n(\vec{x}, \ldots, \vec{y}_{n-1})) \left[ \begin{array}{c} B_1(\vec{x}, \vec{y}_1) \text{ and } \ldots \text{ and } B_n(\vec{x}, \ldots, \vec{y}_n) \\ \Rightarrow \ Q(\vec{x}, \vec{y}_1, \ldots, \vec{y}_n) = EP_Q(\vec{x}) \end{array} \right]$$

\section{3.6.1 Chase with Full EPCDs: Termination}

We will look at a single chase step $T_i \xrightarrow{d} T_{i+1}$ within a chase sequence with full EPCDs starting with a tableau $T = \{ \vec{x} \in \vec{P} : C(\vec{x}) \}$ (using the notations introduced in the previous subsection). By Corollary 3.6.4 paths over $T_i$ (and $T_{i+1}$) are determined by extended paths $EP$ over $T$. The first remark is that, in the case of paths of set type, $EP$ must be paths as well. Second, we observe that the set of all possible extended paths $EP(\vec{x})$ over $T$ that can determine paths over any $T_i$, for a given schema, is finite. This is true since, according to our earlier observation, any path $Q$ has a type smaller than the type of some root name, thus any $EP$ equated to $Q$ must satisfy the same property. But there are only finitely many $EP$s with variables in $\vec{x}$ that satisfy this upper bound on the size of the type (we will call them \textit{bounded extended paths over $T$}). Let us denote by $\mathcal{E}$ the set of all bounded $EP$s over $T$ and $S$ the set of all paths $P$ over $T$ of set type (note that $S$ is a subset of $\mathcal{E}$).

We introduce a \textit{measure} on a tableau $T_i$ of the chase sequence, $m(T_i) = < p(T_i), s(T_i) >$, consisting of two components: \footnote{\textit{eval}$_t$ is defined componentwise on a bounded extended path $EP$.}

$$\begin{array}{lcl}
p(T_i) &=& \text{number of pairs } (EP, P) \in \mathcal{E} \times S \text{ such that } EP : \tau, P : \text{Set}(\tau), \text{ and either } \text{eval} \_t \text{EP} \text{ is not in } C\text{Inst}(T_i) \text{ or, if it is, then there is no } \varepsilon \text{-edge from } \text{eval} \_t \text{EP} \text{ to } \text{eval} \_t \text{EP} \text{ in } C\text{Inst}(T_i) \\
s(T_i) &=& \text{number of nodes in } C\text{Inst}(T_i)
\end{array}$$

As we discussed above, $p(T_i)$ is finite. The ordering that we consider on pairs $m(T_i)$ is the lexicographical one. We show next that, under the simple type restriction, if $T_i \xrightarrow{d} T_{i+1}$ then $m(T_{i+1}) < m(T_i)$. This reflects our intuition that each chase step must either assert some membership condition between some $EP$ in $\mathcal{E}$ and some $P$ in $S$ (a typical EPCD case) or, if not, must collapse some $\simeq$-equivalence classes in $C\text{Inst}(T_i)$ (EGD case).

\textbf{Lemma 3.6.5} If $T_i \xrightarrow{d} T_{i+1}$, and $T_i$, $T_{i+1}$, and $d$ are simple type restricted, then $m(T_{i+1}) < m(T_i)$.

\textbf{Proof.} It is quite obvious that $p(T_{i+1}) \leq p(T_i)$. This is because if an $EP$ is in $C\text{Inst}(T_i)$ and participates in an $\varepsilon$-edge then it will be also in $C\text{Inst}(T_{i+1})$ and it will participate in the same $\varepsilon$-edge (we use Lemma 3.5.1 here). If $p(T_{i+1}) < p(T_i)$ we are done. Now suppose that $p(T_{i+1}) = p(T_i)$. In other words, applying $d$ doesn’t produce any new $\varepsilon$-edges. We’ll show that either some $\simeq_t$-equivalence classes are collapsed in $C\text{Inst}(T_{i+1})$ and in this
case $s(T_{i+1}) < s(T_i)$ and we are done, or $\text{ClInst}(T_i)$ and $\text{ClInst}(T_{i+1})$ are isomorphic. In the latter case, we can immediately infer the existence of a homomorphism from $T_{i+1}$ into $T_i$, thus contradicting the definition of the chase step. The homomorphism comes, by Lemma 3.3.6, from the valuation obtained by composing $\text{eval}^{i+1}_C$ with the isomorphism.

**Sublemma.** If $p(T_{i+1}) = p(T_i)$ then for any path $Q : G_{i+1}$ there exists some path $Q_0 : G_i$ such that $Q \simeq_{i+1} Q_0$.

**Proof of sublemma.** Induction on the derivation of $Q : G_{i+1}$. The base cases are trivial. Now suppose $Q = y_{i+1}$ where $y_{i+1} \in Q_{i+1}$ is a binding in $T_{i+1}$. We know that $y_{i+1} \simeq_{i+1} \text{EP}(\bar{x})$ and $Q_{i+1} \simeq_{i+1} \text{P}(\bar{x})$ for some $\text{EP}$ and $\text{P}$ over $T$. Thus we have an $\varepsilon$-edge from $\text{P}$ into $\text{EP}$ in $\text{ClInst}(T_{i+1})$. Since $p(T_{i+1}) = p(T_i)$ it must be the case that $\text{EP}$ is also in $\text{ClInst}(T_i)$ (with the corresponding $\varepsilon$-edge), i.e. there exists $Q_0 : G_i$ such that $Q_0 \simeq_{i} \text{EP}$. Therefore $Q_0 \simeq_{i+1} \text{EP}$ and, by transitivity, $y_{i+1} \simeq_{i+1} Q_0$. The case when $Q = u$ for some variable occurring in $T_i$ is trivial.

(prj-add): $Q.A : G_{i+1}$ for some $Q : G_{i+1}$. Applying the inductive hypothesis, there exists $Q_0 : G_i$ such that $Q \simeq_{i+1} Q_0$. Then, $Q_A : Q_i$ by (prj-add) and $Q.A \simeq_{i+1} Q_0.A$ by (prj-cong). The case (dom-add) is similar. The (lookup-add) case (we apply Lemma 3.6.1 first):

\[
\begin{align*}
u \simeq_{i+1} v, & \quad v \in \text{dom } R \text{ in } T_{i+1}, \quad R \simeq_{i+1} Q \\
Q[u] : G_{i+1}
\end{align*}
\]

Case 1: $v$ occurs in $T_i$. In this case, $R[v] : G_i$ and by (lookup-cong) $Q[u] \simeq_{i+1} R[v]$. Case 2: $v$ is bound only in $T_{i+1}$. We know that $v \simeq_{i+1} \text{EP}$ and $R \simeq_{i+1} P$ for some $\text{EP}$ and $\text{P}$ over $T$. Thus we have an $\varepsilon$-edge from $\text{dom } P$ into $\text{EP}$ in $\text{ClInst}(T_{i+1})$. We apply the hypothesis $p(T_{i+1}) = p(T_i)$ to conclude that $\text{EP}$ must be in $\text{ClInst}(T_i)$ and an $\varepsilon$-edge exists between $\text{eval}^{i}_C(\text{EP})$ and $\text{eval}^{i+1}_C(\text{dom } P)$. Since $\varepsilon$-edges come from variables, it must be the case that there exists some $z \in \text{dom } P$ in $T_i$ such that $z \simeq_i \text{EP}$ and $P^z \simeq_i P$ (here we made again use of Lemma 3.6.1). We obtain $P^z[z] : G_i$. Also, by transitivity and Lemma 3.5.1, $z \simeq_{i+1} u$ and $P^z \simeq Q$. Hence, by (lookup-cong), $Q[u] \simeq_{i+1} P^z[z]$. And this concludes the proof of the sublemma.

Under the assumption that $p(T_{i+1}) = p(T_i)$, a direct consequence of the sublemma is that $s(T_{i+1}) \leq s(T_i)$. This is because the mapping $f : \text{ClInst}(T_i) \to \text{ClInst}(T_{i+1})$, $f(\text{eval}^{i}_C(Q)) = \text{eval}^{i+1}_C(Q)$ is surjective, as it can be easily verified. If $s(T_{i+1}) < s(T_i)$ we are done. If $s(T_{i+1}) = s(T_i)$ then $f$ is a bijection. Moreover, $f$ preserves the structure (since $\text{eval}^{i}_C$ and $\text{eval}^{i+1}_C$ are algebraic homomorphisms, see Lemma 3.2.7), and the $\varepsilon$-edges. Finally, since $p(T_{i+1}) = p(T_i)$, $\text{ClInst}(T_{i+1})$ doesn’t have more $\varepsilon$-edges than $\text{ClInst}(T_i)$. Thus, $f$ is an isomorphism. As discussed above, this yields a contradiction with Definition 3.4.5. **End of Proof.**

Since there cannot exist an infinitely decreasing sequence of $m(T_i)$, we proved Theorem 3.4.6.

### 3.6.2 Chase with Full EPCDs: Complexity Analysis.

Relational full/total tgds are full EPCDs. Note that EGDs are always full, provided that they don’t involve set/dictionary equality on the right-hand side. We show here that the complexity of the PC problem is the same as in the relational case ([BV84b, CLM81]), i.e. exponential. As we already said in section 3.2 we assume that the schema size is constant. However, some parameters of the schema will appear in our formulas.

Suppose $d = \text{dep}(T, T')$ is a full EPCD, $D$ is a set of full EPCDs, both such that the restrictions of Theorem 3.4.6 and of Theorem 3.4.4 are satisfied, and our problem is to decide whether $D \models d$. The following parameters will be used in the analysis:
\[ n = \text{number of variables in the tableau } T \]
\[ m = \text{number of variables in the tableau } T' \text{ that are not in } T \]
\[ S = \text{number of root names in the schema (including constants true and false)} \]
\[ h = \text{maximum height of a record type or dictionary type in the schema} \]
\[ w = \text{maximum width of a record type (number of attributes) in the schema} \]
\[ s = \text{maximum number of variables in any EPCD in } D \]
\[ d = \text{number of EPCDs in } D \]

**Lemma 3.6.6** (1) The number of paths \( P(x) \) over \( T \) is \( N = O((nwS)^{\alpha h}) \).

(2) The number of bounded extended paths \( EP(x) \) over \( T \) is \( |E| \leq N + c_1N^{w^h} = O((nwS)^{c_1^{h,w^h}}) \), where \( c_0, c_1 \) and \( c_1' \) are constants depending on the schema.

**Proof.** For the first part, we observe that all paths can be organized in a stratified way: a) on the first level are variables and root names, b) paths on level \( i \) are obtained from paths from level \( i - 1 \) by record projection, applying \( \text{dom} \) or dereferencing a dictionary \( P \) (with any of the \( n \) variables). There is no circularity between levels, due to our restriction that only variables can dereference dictionaries. The number of paths on level \( i \) is the number of paths on level \( i + 1 \) multiplied by \( nw + 1 \) (each \( P : \text{Struct}(A_1 : \tau_1, \ldots, A_k : \tau_k) \) on level \( i + 1 \) generates \( k \) \( w \) paths on level \( i \), and each dictionary \( M \) on level \( i \) may generate at most \( n \) paths of the form \( x \mapsto M \) on level \( i \). \( \text{dom} \) contributes with multiplicity \( 1 \). The number of levels is bounded by \( h \). Thus, the total number of paths can be estimated by:

\[ (n + S)(1 + (nw + 1) + \ldots + (nw + 1)^h) = O((n + S)(nw + 1)^{h+1}) = O((nwS)^{h+2}) \]

For the second part, we observe that there is a constant number of possible record types in the schema, and for each of them the number of bounded extended paths obtained by applying the record constructor is at most \( N^{w^h} \) (there are at most \( w^h \) subexpressions which are not record expressions). **End of Proof.**

By Corollary 3.6.4, the total number of nodes that can appear in any \( C\text{Inst}(T_i) \) in the chase sequence is at most \( |E| \). Thus the sequence \( m(T) > m(T_i) > \ldots \) has length at most \( |E \times S| + |E| \). This is an upper bound on the number of chase steps. Recall that \( S \) is the number of paths over \( T \) of set type and therefore its size is bounded by \( N \) from Lemma 3.6.6. Hence, the number of chase steps is bounded by

\[ (N + 1) |E| \leq (N + 1)(N + c_1N^{w^h}) \]

and replacing \( N \) with its upper bound from Lemma 3.6.6 we obtain

**Lemma 3.6.7** The number of chase steps is \( O((nwS)^{\alpha h,w^h}) \).

Notice that this is polynomial in \( n \) when schema is fixed. We focus next on the complexity of a single chase step. To check whether a dependency \( \text{dep}(T_r, T_r) \) applies to a tableau \( T_i \) in the sequence, we need to check the existence of a homomorphism from \( T_r \) into \( T_i \). Since the number of variables in \( T_r \) is at most \( s \) and the number of variables in \( T_i \) is bounded by \( |E| \), there can be at most \( |E| = O((nwS)^{\alpha h,w^h}) \) mappings. Checking whether any such mapping is actually a homomorphism takes time polynomial in the sizes of \( C\text{Inst}(T_r) \) and \( C\text{Inst}(T_i) \), which we can estimate as follows:

\[ \text{size of } C\text{Inst}(T_r) = O((swS)^{\alpha h}) \]

65
size of $C_{Inst}(T_i) = O(|\mathcal{E}|)$

Any polynomial in the first term is dominated asymptotically by the expression for the number of mappings ($s$ is at the exponent in the latter expression). Similarly, any polynomial in $|\mathcal{E}|$ is smaller than $|\mathcal{E}|^s$. We also observe that the time to check whether there is no homomorphism from $T_{i+1}$ into $T_i$ can be taken into account in the same manner (it is dominated by $|\mathcal{E}|^s$ as well). Finally, we have to multiply everything by $d$. Thus:

**Lemma 3.6.8** The complexity of a single chase step is $O(d(nwS)^{k_1k_2s})$.

Remark that this is exponential in $s$. Hence, the most expensive part of the chase procedure is actually the chase step itself and not the total number of steps. To decide whether $D \models dep(T, T')$, at the end of the chase we still have to check whether $chase_D(d)$ is trivial. Here we need to consider $O(|\mathcal{E}|^m)$ mappings and we obtain, in a similar way with the chase step analysis, the upper bound $O((nwS)^{k_1k_2s}m)$. We conclude as follows:

**Proposition 3.6.9** Let $D$ be a set of full EPCDs and $d = dep(T, T')$ an EPCD. Then:

1. The total number of chase steps and the size of $chase_D(T)$ (i.e. the number of variables in it) are both polynomial in $n : O(n^{k_1})$, respectively $O(n^{k_2})$,
2. The chase finishes in time exponential in $s : O(dn^{k_3s})$,
3. Deciding whether $D \models d$ can be done in time exponential in $s$ and $m : O(dn^{k_3s}) + O(n^{k_4m})$,

where $k_1, k_2, k_3$, and $k_4$ are constants depending on the schema.

### 3.6.3 Chase with Full EPCDs: Confluence and Semantic Invariance

**Definition 3.6.10** Given two instances $I_1$ and $I_2$, an instance homomorphism $ih : I_1 \rightarrow I_2$ is a type-preserving mapping from values of $I_1$ into values of $I_2$ such that:

1. $ih$ is an algebraic homomorphism: $ih(R^i) = R^j$, for any root name, $ih(\text{dom} P) = \text{dom} (ih(P))$, for any dictionary $P$ in $I_1$, $ih(P.k) = ih(P).k$, for any record $P$ in $I_1$ and any attribute $k$, and $ih(P_2[P_1]) = ih(P_2)[ih(P_1)]$ for any dictionary $P_2$ in $I_1$ and any key $P_1 \in \text{dom} P_2$.
2. $ih$ is context preserving: for any $P_1 \in P_2$ in $I_1$, $ih(P_1) \in ih(P_2)$ in $I_2$.

In the following, just for technical purposes, we will consider a slightly larger class of instances for which we don’t require to be extensional. We call these instances weak instances. This class includes the normal instances considered until now. However, structures like $Inst(T)$ are in this class as well. The notions of valuation, dependency satisfaction and instance homomorphism are defined in the same way and we will call them, respectively, weak valuation, weak satisfaction (we use the same symbol, $|$) and weak instance homomorphism. eval, is an example of a weak valuation. Also remark that any valuation is a weak valuation and any instance homomorphism is a weak instance homomorphism.

The first important observation is that Lemma 3.4.3, part (2), still holds when we replace $Inst(T)$ with $Cl_{Inst}(T)$. We also note that it would hold even in the presence of set/dictionary equality in $d$. We redo here the proof in the new context.

**Lemma 3.6.11** If $d$ is not applicable to $T$ then $Cl_{Inst}(T) \models d$. 
Proof. Assume \( d = \text{dep}(T_r, T_{rs}) \), and let \( v_c : T_r \to \text{Clinst}(T) \) be a weak valuation. By Lemma 3.3.6 there exists a homomorphism \( h : T_r \to T \) such that \( \text{val}_c \circ h = v_c \). Since \( d \) is not applicable to \( T \), there exists a homomorphism \( h' : T' \to T \) such that \( \forall (\bar{x} \in \bar{P}) \quad C(\bar{x}) \Rightarrow \bar{x} = h'(\bar{x}) \). As in the first version of the lemma, \( h' \triangleq h \cup \text{id}_c \) is a homomorphism from \( T_{rs} \) into \( T' \). Take \( v'_c \triangleq \text{val}_c \circ h' \).

\( v'_c \) is obviously context preserving. Since \( \forall (\bar{x} \in \bar{P}) \quad C(\bar{x}) \Rightarrow \bar{x} = h'(\bar{x}) \) is trivial, it follows by Lemma 3.3.1, part (2), that \( \text{val}_c(h'(\bar{x})) = \text{val}_c(h(\bar{x})) \), and hence \( v'_c(\bar{r}) = v_c(\bar{r}) \). Finally, \( v'_c(B_1(\bar{r}, \bar{s})) = B_2(\text{val}_c(h'(\bar{r})), \text{val}_c(h'(\bar{s}))) \). Now, since \( \forall (\bar{x} \in \bar{P}) \quad C(\bar{x}) \Rightarrow B_2(h'(\bar{r})), h'(\bar{s}) \) is trivial \((h' \) is a homomorphism\), the last term equals \text{true}, again by Lemma 3.3.1, part (2). We conclude that \( \text{Clinst}(T) \models d \). End of Proof.

The next two lemmas show the first one, that any weak valuation from a tableau \( T \) into a weak instance instance \( I \) is factored through a weak instance homomorphism from \( \text{Clinst}(T) \) into \( I \), and the second one, that such instances and weak instances are indistinguishable with respect to trivial EGDs. We also remark that the first lemma fails if we try to strengthen it by replacing \( \text{Clinst}(T) \) with \( \text{Inst}(T) \) and weak instance and weak valuation with instance and, respectively, valuation. This is one of the reasons why we need to develop several technical results with respect to weak instances and we cannot simply use what we have proved so far with respect to instances.

Lemma 3.6.12 Let \( T = \{ \bar{x} \in \bar{P} : C(\bar{x}) \} \) be a tableau and \( I \) a weak instance. Then, for any weak valuation \( v : T \to I \) there exists a weak instance homomorphism \( ih_c : \text{Clinst}(T) \to I \) such that \( v = ih_c \circ \text{val}_c \).

Proof. For any node \( Q \) in \( \text{Clinst}(T) \), pick any path expression \( P \) over \( T \) such that \( \text{val}_c(P) = Q \), and define \( ih_c(Q) \triangleq v(P) \). This is well-defined because for any \( P_1 \) and \( P_2 \) over \( T \), \( \text{val}_c(P_1) = \text{val}_c(P_2) \) implies \( v(P_1) = v(P_2) \) (we already proved this in Proposition 3.2.3). Also, we have that \( v = ih_c \circ \text{val}_c \). It is easy to see that \( ih_c \) is context preserving and algebraic homomorphism. End of Proof.

Lemma 3.6.13 An EGD (with set/dictionary equality) is trivial if and only if it is trivial under all weak instances.

Proof. We already observed in the previous lemma that if \( \text{val}_c(P_1) = \text{val}_c(P_2) \) then \( v(P_1) = v(P_2) \) for any weak valuation \( v : T \to I \). Thus, \( d \) trivial implies \( d \) weak trivial. The converse is obvious. End of Proof.

Lemma 3.6.14 (Technical Lemma) Let \( d = \forall (\bar{r} \in \bar{R}) \quad B_1(\bar{r}) \Rightarrow \exists (\bar{s} \in \bar{S}(\bar{r})) \quad B_2(\bar{r}, \bar{s}) \) be an EPCD, and let \( T = \{ \bar{x} \in \bar{P} : C(\bar{x}) \} \) be a tableau, where \( C, B_1, \) and \( B_2 \) are simple-type restricted. Assume that \( T \xrightarrow{h} T' \) where \( T' = \{ \bar{x} \in \bar{P}, \bar{s} \in \bar{S}(\bar{r}) \} \) for some homomorphism \( T \xleftarrow{h} \{ \bar{r} \in \bar{R} : B_1(\bar{r}) \} \). Let \( I \) be a weak instance such that \( I \models d \) and let \( v : T \to I \) be a weak valuation. Then:

1. there exists \( v' : T' \to I \) a weak valuation such that \( v'(\bar{x}) = v(\bar{x}) \).
2. if, in addition, \( d \) is full, then for any \( Q(\bar{x}, \bar{s}) \) over \( T' \)
   \( v'(Q(\bar{x}, \bar{s})) = v(EP_Q(\bar{x})) \) (\( v' \) is uniquely determined by \( v \))

where \( EP_Q(\bar{x}) \) is from Lemma 3.6.3.

Proof. Let \( T_r \) and \( T_{rs} \) such that \( d = \text{dep}(T_r, T_{rs}) \). Then \( v \circ h \) is a weak valuation from \( T_r \) into \( I \). Since \( I \models d \) there exists a weak valuation \( v' : T_{rs} \to I \) such that \( v'(\bar{r}) = v \circ h(\bar{r}) \). \( h \cup \text{id}_c \) is a homomorphism from \( T_{rs} \) into \( T' \). We show that there exists \( v' : T' \to I \) a weak valuation such that \( v'' = v' \circ (h \cup \text{id}_c) \). \( v' \) is defined
as follows: \( v'(\bar{x}) \overset{\text{def}}{=} v(\bar{x}) \), \( v'(\bar{s}) \overset{\text{def}}{=} v''(\bar{s}) \), and then we extend it to paths over \( T' \) such that it is an algebraic homomorphism. Thus, \( v''(r') = v(h(r)) = v'(h(r)) = v' \circ (h \cup \text{id}_D)(r) \), and \( v''(s) = v'' \circ (h \cup \text{id}_D)(s) \), and therefore, \( v'' = v' \circ (h \cup \text{id}_D) \). It is easily verified that \( v' \) is context preserving.

Suppose now that \( d \) is full. By Lemma 3.6.3, for any \( Q(\bar{x}, \bar{s}) \) over \( T' \) there exists \( EP_Q(\bar{x}) \) over \( T \) such that

\[
\forall(\bar{x} \in \bar{P})\forall(\bar{s} \in \bar{S}(h(\bar{r}))) \quad [C(\bar{x}) \text{ and } B_2(h(\bar{r}), \bar{s}) \Rightarrow Q(\bar{x}, \bar{s}) = EP_Q(\bar{x})]
\]

is trivial, and therefore weak trivial, by Lemma 3.6.13. Hence, \( v'(Q(\bar{x}, \bar{s})) = v'(EP_Q(\bar{x})) = v(EP_Q(\bar{x})) \). \textbf{End of Proof.}

We are now ready, by making use of Corollary 3.6.4, Lemma 3.6.11 and Lemma 3.6.14, to prove that chasing with full EPCDs is confluent.

\textbf{Proof of Theorem 3.4.8.} Assume \( T = \{ \bar{x} \in \bar{P}; \ C(\bar{x}); T_n = \{ \bar{x} \in \bar{P}, \bar{y} \in \bar{S}(\bar{x}); \ C(\bar{x}) \text{ and } B_1(\bar{x}, \bar{y}) \} \) and \( T'_n = \{ \bar{x} \in \bar{P}, \bar{y} \in \bar{S}(\bar{x}); \ C(\bar{x}) \text{ and } B_2(\bar{x}, \bar{y}) \} \). Obviously, \( \text{id}_D : T \rightarrow T'_n \) is a homomorphism. Thus,

\[
\text{val}_{\bar{x}} \circ \text{id}_D : T \rightarrow \text{ClInst}(T'_n)
\]

is a weak valuation, where \( \text{val}_{\bar{x}} : T_n \rightarrow \text{ClInst}(T'_n) \) is the canonical weak valuation on \( T'_n \). By Lemma 3.6.11, \( \text{ClInst}(T'_n) = D \). Then by repeatedly applying Lemma 3.6.14 for the chase sequence \( T \xrightarrow{d_1} T_1 \xrightarrow{d_2} \ldots \xrightarrow{d_n} T_n \) we obtain a weak valuation \( \text{val}_n : T_n \rightarrow \text{ClInst}(T'_n) \) such that \( v_n(\bar{x}) = \text{val}_{\bar{x}} \circ \text{id}(\bar{x}) \). In addition, by Corollary 3.6.4, for any \( Q(\bar{x}, \bar{y}) \) over \( T_n \) there exists \( EP_Q(\bar{x}) \) over \( T \) such that

\[
\forall(\bar{x} \in \bar{P})\forall(\bar{y} \in \bar{S}(\bar{x})) \quad [C(\bar{x}) \text{ and } B_1(\bar{x}, \bar{y}) \Rightarrow Q(\bar{x}, \bar{y}) = EP_Q(\bar{x})]
\]

is trivial, and therefore weak trivial. Thus \( v_n(Q(\bar{x}, \bar{y})) = v_n(EP_Q(\bar{x})) = \text{val}_{\bar{x}}(EP_Q(\bar{x})) \). Next, by Lemma 3.6.12, \( v_n \) induces a weak instance homomorphism \( \theta : \text{ClInst}(T_n) \rightarrow \text{ClInst}(T'_n) \) such that \( \theta \circ \text{val}_{\bar{x}} = v_n \), where \( \text{val}_{\bar{x}} : T_n \rightarrow \text{ClInst}(T_n) \) is the canonical weak valuation on \( T_n \). Then, \( \theta \) satisfies the following:

\[
\theta(\text{val}_{\bar{x}}(Q(\bar{x}, \bar{y}))) = \text{val}_{\bar{x}}(EP_Q(\bar{x})), \quad \text{for any } Q(\bar{x}, \bar{y}) \text{ over } T_n
\]

\[
\theta(\text{val}_{\bar{x}}(\bar{x}) = \text{val}_{\bar{x}}(\bar{x})
\]

Similarly, we can show the existence of a weak instance homomorphism \( \psi : \text{ClInst}(T'_n) \rightarrow \text{ClInst}(T_n) \) satisfying similar properties. Then, for any \( Q(\bar{x}, \bar{y}) \) over \( T_n \), we have:

\[
\psi \circ \theta(\text{val}_{\bar{x}}(Q(\bar{x}, \bar{y}))) = \psi(\text{val}_{\bar{x}}(EP_Q(\bar{x}))) = \text{val}_{\bar{x}}(EP_Q(\bar{x}))
\]

But \( \text{val}_{\bar{x}}(Q(\bar{x}, \bar{y}) = \text{val}_{\bar{x}}(EP_Q(\bar{x}) \beta) \) by the above mentioned triviality! Thus, \( \psi \circ \theta \) is the identity weak instance homomorphism on \( \text{ClInst}(T_n) \). Similarly, \( \theta \circ \psi \) is the identity weak instance homomorphism on \( \text{ClInst}(T'_n) \). We conclude that we have an isomorphism between \( \text{ClInst}(T_n) \) and \( \text{ClInst}(T'_n) \). \textbf{End of Proof.}

We will prove next that the final result of chase with full dependencies doesn’t depend on the syntax of dependencies but rather on their semantics, Theorem 3.4.9. Before doing that we need to go further into analyzing the behavior of dependencies with respect to weak instances. The first result generalizes Lemma 3.4.3, part (1), by stating the soundness of the chase step not only with respect to instances, but with respect to the larger class of weak instances. We only show the proof for queries (for dependencies a similar lemma and proof holds).

\textbf{Lemma 3.6.15} \textit{If } \( Q \xrightarrow{d} Q' \text{ then } Q = Q' \text{ under any weak instance } I \text{ such that } I \models d \).

\textbf{Proof.} Let \( d = \text{dep}(T_n, T_n') \) and suppose \( T = \{ \bar{x} \in \bar{P}; \ C(\bar{x}) \} \) and \( T' = \{ \bar{x} \in \bar{P}, \bar{s} \in \bar{S}(\bar{r}); \ C(\bar{x}) \text{ and } B_2(h(\bar{r}), \bar{s}) \} \) for some homomorphism \( h : T_n \rightarrow T' \).
Suppose \( Q = \textbf{select} \ O(\bar{x}) \textbf{from} \ \bar{P} \ \bar{x} \textbf{where} \ C(\bar{x}) \textbf{and} \ Q' = \textbf{select} \ O(\bar{x}) \textbf{from} \ \bar{P} \ \bar{x}, \ \bar{S}(h(\bar{x})) \textbf{where} \ C(\bar{x}) \textbf{and} \ B_2(h(\bar{x})) \) and let \( I \) be an arbitrary weak instance.

The direction \( Q'(I) \subseteq Q(I) \) is immediate. Indeed assume \( t \in Q'(I) \). Then there exist a weak valuation \( v' : T' \rightarrow I \) such that \( t = v'(O(\bar{x})) \). Then \( v = v' \circ \text{id}_{\bar{x}} \) is a weak valuation from \( T \) into \( I \). Moreover, \( v(O(\bar{x})) = v'(O(\bar{x})) = t \), thus \( t \in Q(I) \).

Conversely, suppose \( t \in Q(I) \), thus there exists \( v : T \rightarrow I \) a weak valuation such that \( t = v(O(\bar{x})) \). Then we can apply the Technical Lemma, part (1), to infer the existence of a weak valuation \( v' : T' \rightarrow I \) such that \( v'(\bar{x}) = v(\bar{x}) \), and thus \( t = v'(O(\bar{x})) \in Q'(I) \). \textbf{End of Proof.}

**Proposition 3.6.16** (1) An EGD (with set/dictionary equality) is trivial if and only if it is weak trivial.

(2) An EPCD (restricted as in Theorem 3.3.11) is trivial if and only if it is weak trivial.

(3) If \( D \) and \( d \) are restricted as in Theorem 3.4.4 and some chase of \( d \) by \( D \) terminates then:

\[ D \models d \text{ iff } D \models_{\text{weak}} d \]

**Proof.** (1) was already proved (Lemma 3.6.13). For (2), weak triviality implies obviously triviality. Now suppose \( d = v(\bar{x} \in \bar{P}) \mid C_1(\bar{x}) \Rightarrow \exists y \in \bar{P}_2(\bar{x}) \mid C_2(\bar{x}, y) \) and \( T_1 \) and \( T_2 \) are such that \( d = \text{dep}(T_1, T_2) \). If \( d \) is trivial then, by Theorem 3.3.11, there exists homomorphism \( T_1 \xrightarrow{h} T_2 \) such that the EGD \[ v(\bar{x} \in \bar{P}_1) \mid C_1(\bar{x}) \Rightarrow \bar{x} = h(\bar{x}) \]

is trivial. Let \( I \) be an arbitrary instance and \( v : T \rightarrow I \) an arbitrary weak valuation. Then \( v \circ h : T' \rightarrow I \) is a weak valuation. Moreover, by (1), the above EGD is also weak trivial. Thus, \( v(\bar{x}) = v(h(\bar{x})) \) and therefore \( v(\bar{x}) = v'(\bar{x}) \). We conclude that \( d \) is weak trivial.

For (3), again, one direction is immediate: \( D \models_{\text{weak}} d \) implies \( D \models d \). Now, suppose that \( D \models d \). Then, by Theorem 3.4.4, \( \text{chase}_D(d) \) is trivial, and therefore, by (2), weak trivial. Applying Lemma 3.6.15 for each chase step, we obtain that \( D \models_{\text{weak}} d \). \textbf{End of Proof.}

We can now observe that the confluence proof still works if one chase sequence uses dependencies from a set \( D \) and the other one uses dependencies from a different set \( D' \) provided that \( D \) and \( D' \) are weak equivalent, i.e. \( D \models_{\text{weak}} D' \) and \( D' \models_{\text{weak}} D \). But, by the previous proposition, weak implication and implication of full dependencies are the same. Thus, we obtain a proof of Theorem 3.4.9.

### 3.7 Non-Terminating Chase

As in [BV84b] we start by making the assumption that every EPCD that is applicable infinitely many times should be applied infinitely many times (non-starvation of dependencies).

Let \((T)\) be an infinite chase sequence of \( T \) by a set of EPCDs \( D \): \( T_0 \rightarrow \ldots \rightarrow T_n \rightarrow \ldots \). We define first a (countably) infinite tableau \( T^\infty = \{ \bar{x} \in \bar{P} : C(\bar{x}) \} \) that satisfies the following:

1. for any finite prefix \( \bar{x}_n \in \bar{P}_n \) of \( \bar{x} \in \bar{P} \) there exists a tableau \( T_m = \{ \bar{x}_m \in \bar{P}_m : C_m(\bar{x}_m) \} \) in \( (T) \) such that \( \bar{x}_n \in \bar{P}_n \) is a prefix of \( \bar{x}_m \in \bar{P}_m \)
2. \( C(\bar{x}) = \bigwedge_{T_m \in (T)} C_m(\bar{x}_m) \) (\( C(\bar{x}) \) is an infinite conjunction)
Next, we define the two canonical instances of $T^\infty$, $\text{Clnst}(T^\infty)$ and $\text{Inst}(T^\infty)$ as follows. $\text{Clnst}(T^\infty)$ is built as the limit of the sequence $(\text{Clnst}(T_n))_{n \geq 0}$: the set of well-defined paths over $T^\infty$ and the congruence closure of $T^\infty$ are defined by:

$$G^\infty \overset{\text{def}}{=} \bigcup_{n \geq 0} G_n \quad \text{and} \quad \simeq^\infty \overset{\text{def}}{=} \bigcup_{n \geq 0} \simeq_n$$

where $G_n$ denotes the set of well-defined paths over $T_n$ while $\simeq_n$ denotes the congruence closure of $T_n$. Recall (Lemma 3.5.1) that $G_n \subseteq G_{n+1}$ and $\simeq_n \subseteq \simeq_{n+1}$. It is easily verified that $G^\infty$ and $\simeq^\infty$ are closed under the add and equality rules of the section 3.2. Similar with the construction of $\text{Clnst}(T)$ in the case when $T$ was finite, we define $\text{Clnst}(T^\infty)$ by considering $\simeq^\infty$-equivalence classes and adding the appropriate edges. $\text{Clnst}(T^\infty)$ is a weak infinite instance (however, countable). As in the finite case, we denote by $\text{val}_1$ the canonical weak valuation.

$\text{Inst}(T^\infty)$ is defined out of $\text{Clnst}(T^\infty)$ via a construction similar with the one given in section 3.2 for the finite case, using the same set of extensionality rules. However, because sets and dictionaries may be now infinite, a finite induction is not enough. In particular, the rules (set-ext) and (dict-ext) are the ones causing the difficulty. For example, in (set-ext), each pair $e \sim e'$ may have a finite length derivation but still the derivation $S_1 \sim S_2$ may be infinite. This is because $S_1$ and $S_2$ themselves may be infinite sets and the sup of an infinite set of finite numbers is not necessarily finite. Thus, $\sim$ is built by transfinite induction:

$$\begin{align*}
\sim_0 & \overset{\text{def}}{=} \{(Q, Q) \mid Q \text{ node in } \text{Clnst}(T^\infty)\} \\
\sim_{\lambda+1} & \overset{\text{def}}{=} \sim_{\lambda} \cup \\
\sim_{\lambda} & \overset{\text{def}}{=} \bigcup_{\alpha < \lambda} \sim_{\alpha}, \quad \text{if } \lambda \text{ is a limit ordinal}
\end{align*}$$

Then we define $\sim \overset{\text{def}}{=} \bigcup_{\alpha} \sim_{\alpha}$ where the union is over all ordinals. Of course, since $\text{Clnst}(T^\infty)$ is countable, $\sim$ must be countable as well, therefore the induction must terminate at some countable ordinal. In other words, there exists a countable $\beta$ such that $\sim_{\beta+1} = \sim_{\beta}$. If $\beta_0$ is the least such $\beta$ then $\sim_{\beta_0} = \bigcup_{\alpha} \sim_{\alpha}$ and hence $\sim = \sim_{\beta_0}$.

One can easily verify that $\sim$ is closed under the extensionality rules of section 3.2. By straightforward transfinite induction, one can show that Lemmas 3.2.4, 3.2.5 and 3.2.6 still hold. To exemplify, we prove here (set-irm-ext). First case: $S_1 \sim_{\lambda+1} S_2$. Then either $S_1 \sim \lambda S_2$, in which case we apply directly the inductive hypothesis, or $S_1 \sim_{\lambda+1} S_2$ follows by some rule application. For (sym) and (trans) we apply the inductive hypothesis. The other case possible is (set-ext), but this follows immediately since it must be the case that $\forall e \in S_1, \exists e' \in S_2. e \sim_{\lambda} e'$ and the symmetric condition are true. Second case: $S_1 \sim_{\lambda} S_2$ where $\lambda$ is a limit ordinal. Then there must exist an $\alpha < \lambda$ such that $S_1 \sim_{\alpha} S_2$, in which case we apply the inductive hypothesis.

Thus, we can define $\text{Inst}(T^\infty)$ as in the finite case by taking $\sim$-equivalence classes and adding the necessary edges. We denote by $\text{collapse}$ the canonical mapping from nodes of $\text{Clnst}(T^\infty)$ to $\text{Inst}(T^\infty)$. The first important result is that Lemma 3.3.5 in which we replace $T_1$ with $T^\infty$ still holds. The proof is essentially the same (making use of Lemma 3.2.4).

**Lemma 3.7.1** Let $T_2 = \{[y] \in \mathcal{P}_2 \mid C_2([y])\}$ in which $C_2$ is simple type restricted. Then, for any valuation $v : T_2 \rightarrow \text{Clnst}(T^\infty)$ there exists a weak valuation $v_c : T_2 \rightarrow \text{Clnst}(T^\infty)$ such that $v = \text{collapse} \circ v_c$. Moreover, $v$ and $v_c$ satisfy the same set of formulas $Q = Q'$ over $T_2$ with $Q$ and $Q'$ of simple type.

We remark that for any tableaux $T_i, T_j$ in the chase sequence with $i < j$ the identity mapping on $T_i$ is a homomorphism from $T_i$ into $T_j$. We will denote it by $\text{id}_{ij}$. By Lemma 3.6.12 there exists an instance homomorphism $ih_{ij} : \text{Clnst}(T_i) \rightarrow \text{Clnst}(T_j)$ such that $ih_{ij} \circ \text{val}_2 = \text{val}_2 \circ \text{id}_{ij}$. These functions also exist when we replace $T_j$ with $T^\infty$. We will have in that case the identity mapping $\text{id} : T_i \rightarrow T^\infty$ and the weak instance homomorphism $ih_i : \text{Clnst}(T_i) \rightarrow \text{Clnst}(T^\infty)$ satisfying $ih_i \circ \text{val}_2 = \text{val}_2 \circ \text{id}$. Then we can prove the following lemma.
Lemma 3.7.2 Let $T_r = \{ r \in \mathcal{R} : \mathcal{B}_1(r) \}$. Then, for any weak valuation $v_r : T_r \rightarrow \text{Clinst}(T^\infty)$ there exists a finite $n$ and a homomorphism $h : T_r \rightarrow T_n$ such that $v_r = \text{val}_r \circ \text{id}_n \circ h$.

Proof. For any $m \geq 0$ we define the following two sets:

$$G^m_r = \{ Q : Q : G_r \text{ with derivation length } \leq m \}$$

$$\simeq^m_r = \{ (Q, Q') : Q \simeq_r Q' \text{ with derivation length } \leq m \}$$

We use the notations $Q : G^m_r$ for $Q \in G^m_r$ and $Q \simeq^m_r Q'$ for $(Q, Q') \in \simeq^m_r$. Note that, for any $m \geq 0$, $G^m_r$ and $\simeq^m_r$ are finite. Also, there exists $m_0$ finite such that $G_r = G_{r_0}^m$ and $\simeq_r = \simeq_{r_0}$ (in other words all derivations are finite. We already argued in section 3.2 that in fact all derivations are of polynomial length). We set the following inductive hypothesis:

For any $m \geq 0$ there exist $i_m \geq 0$ finite and $h_m : G^m_r \rightarrow G_{i_m}$ such that the following conditions are satisfied:

1. $h_m$ maps variables of $T_r$ into variables of $T_{i_m}$.
2. for any $Q : G^m_r$, $h_m(Q) : G_{i_m}$
3. $h_m$ is algebraic homomorphism on $G^m_r$
4. $h_m$ is context-preserving:
   for any $r \in R$ in $T_r$, if $h_m(r) = x$ with $x \in P$ in $T_{i_m}$ then $P \simeq_{i_m} h_m(R)$
5. for any $Q \simeq^m_r Q'$, $h_m(Q) \simeq_{i_m} h_m(Q')$
6. $v_r = \text{val}_r \circ \text{id}_{i_m} \circ h$

Then by taking $m$ to be $m_0$ and $n$ to be $i_{m_0}$ we obtain the lemma. To prove the above statement we proceed by induction on $m$. The inductive step will consider all possible cases for the last rule applied in the derivation of $Q : G^m_r$ or $Q \simeq^m_r Q'$ and will extend in each case $h_{m-1}$ to a new $h'_m$ that will satisfy conditions (1) - (6) for $m$. A new $i'_m \geq i_{m-1}$ is obtained in each case. Then $h_m$ will be the union of all $h'_m$ while $i_m$ will be taken as the maximum of all $i'_m$. The union, respectively, maximum, are taken over all $Q$ and $(Q, Q')$ that are in $G^m_r$, respectively $\simeq^m_r$, but not in $G^{m-1}_r$, respectively $\simeq^{m-1}_r$ (i.e. they have derivation length $m$). Since there sets are finite, $i_m$ will be also finite. (Of course we assume that for each $Q$ or $(Q, Q')$ only one derivation is considered, in case there is more than one. This ensures that the above union is disjoint.)

Base case: $m = 0$. The possible cases are: (root-add), (true-add) and (false-add). Define $h_0(R) = R$, $h_0(\text{true}) = \text{true}$ and $h_0(\text{false}) = \text{false}$. Conditions (1)-(6) are trivially satisfied with $i_0 = 0$.

Induction step: $m > 0$, $h_m(Q) = h_{m-1}(Q)$, for all $Q : G^{m-1}_r$. For $Q$ and $Q \simeq_r Q'$ with derivation length equal to $m$, we analyze the possible cases. First, (var-add):

$$r \in R \text{ in } T_r \quad , \quad R : G^{m-1}_r$$

$$r : G^m_r$$

Since $v_r$ is a weak valuation, $v_r(r) \in v_r(R)$ in $\text{Clinst}(T^\infty)$. From the way $\varepsilon$-edges are added to $\text{Clinst}(T^\infty)$ we can infer that there must exist $x \in P$ in some $T_k$ such that $\text{val}_r.x = v_r(r)$ and $\text{val}_r(P) = v_r(R)$. By the inductive hypothesis, $h_{m-1}(R) : G^{m-1}_{i_{m-1}}$ and $v_r(r) = \text{val}_r \circ \text{id}_{i_{m-1}} \circ h_{m-1}(R) = \text{val}_r(h_{m-1}(R))$. We infer that $\text{val}_r(P) = \text{val}_r(h_{m-1}(R))$ or $P \simeq_{i_{m-1}} h_{m-1}(R)$. Then, using the definition of $\simeq_{i_{m-1}}$, there exists some $j \geq \max(i_{m-1}, k)$ such that $h_{m-1}(R) \simeq_{i_{m-1}} P$. Take $h'_m(r) \overset{\text{def}}{=} x$ and $i'_m = j$. Condition (1) is automatically verified. Condition (2) is true: $h_m(r) : G_{i_m}$, while (3) and (5) are satisfied by the inductive hypothesis for $h_{m-1}$.
and hence for \( h'_m \) (which only extends \( h_{m-1} \) on \( r \)). Condition (4) is true as well: \( P \simeq_{h'_m} h'_m(R) \). Finally, for condition (6): \( v_r(r) = \text{val}_r \cdot x = \text{val}_r \cdot (h'_m(r)) = \text{val}_r \cdot \text{id}_{m} \circ h'_m(r) \).

The cases \((\text{proj}-\text{add})\), \((\text{dom}-\text{add})\) and \((\text{lookup}-\text{add})\) are all similar. We show here \((\text{lookup}-\text{add})\):

\[
\begin{align*}
\frac{r \simeq_{h'_m} \ r', \ r' \in R \in T_r, \ R \simeq_{h'_m} \ \text{dom} \ Q}{Q[r] : G^m_r}
\end{align*}
\]

Then, by the inductive hypothesis, \( h_{m-1}(r) \simeq_{i_{m-1}} h_{m-1}(r') \) and they are both variables. Also, \( h_{m-1}(R) \simeq_{i_{m-1}} h_{m-1}(\text{dom} \ Q) = \text{dom} \ (h_{m-1}(Q)) \) (the last equality follows from the fact that \( h_{m-1} \) is an algebraic homomorphism). Now, since \( h_{m-1} \) is context preserving, we have \( h_{m-1}(r') = x \) for some \( x \in P \) in \( T_{i_{m-1}} \), such that \( P \simeq_{i_{m-1}} h_{m-1}(R) \). By \((\text{trans})\), \( P \simeq_{i_{m-1}} \text{dom} \ (h_{m-1}(Q)) \). Thus, we can apply \((\text{lookup}-\text{add})\):

\[
\begin{align*}
\frac{h_{m-1}(r) \simeq_{i_{m-1}} h_{m-1}(r'), \ h_{m-1}(r') \in P \in T_{i_{m-1}}, \ P \simeq_{i_{m-1}} \text{dom} \ (h_{m-1}(Q))}{h_{m-1}(Q[h_{m-1}(r)]) : G^m_{i_{m-1}}}
\end{align*}
\]

and then \( h'_m(Q[r]) = \text{def} \ h_{m-1}(Q[h_{m-1}(r)]) \). We also set \( i'_m = i_{m-1} \). Conditions (1)-(6) are easily verified.

Among the equality rules, the \((\text{eq})\) rule is the more interesting one:

\[
\begin{align*}
\frac{Q_1 = Q_2 \in B_1(\vec{r}), \ Q_1 : G^m_{r_1}, \ Q_2 : G^m_{r_2}}{Q_1 \simeq r_1 Q_2}
\end{align*}
\]

By the inductive hypothesis, \( h_{m-1}(Q_1) : G_{i_{m-1}} \) and \( h_{m-1}(Q_2) : G_{i_{m-1}} \). Moreover, by the condition (6), \( v_r(Q_1) = \text{val}_r \cdot (h_{m-1}(Q_1)) \) and \( v_r(Q_2) = \text{val}_r \cdot (h_{m-1}(Q_2)) \). But \( v_r \) is a weak valuation, therefore \( v_r(Q_1) = v_r(Q_2) \). It follows that \( h_{m-1}(Q_1) \simeq_{i_{m-1}} h_{m-1}(Q_2) \). Then there must exist \( j \geq i_{m-1} \) such that \( h_{m-1}(Q_1) \simeq_j h_{m-1}(Q_2) \). Then we take \( i'_m = j \) and \( h'_m = h_{m-1} \). Condition (5) for the new pair \((Q_1, Q_2)\) is satisfied.

The last cases, \((\text{refl})\), \((\text{sym})\), \((\text{trans})\), \((\text{proj}-\text{cong})\), \((\text{dom}-\text{cong})\), \((\text{red-ext})\) and \((\text{lookup}-\text{cong})\) are simple. We show here \((\text{proj}-\text{cong})\):

\[
\frac{Q \simeq_{r_1} Q'}{Q.A \simeq_{r_1} Q'.A}
\]

Applying the inductive hypothesis, \( h_{m-1}(Q) \simeq_{i_{m-1}} h_{m-1}(Q') : G_{i_{m-1}} \). Therefore \( h_{m-1}(Q.A) \simeq_{i_{m-1}} h_{m-1}(Q'.A) \) by \((\text{proj}-\text{cong})\). Since \( h_{m-1} \) is an algebraic homomorphism, it follows that \( h_{m-1}(Q.A) \simeq_{i_{m-1}} h_{m-1}(Q'.A) \). Thus, it suffices to take \( h'_m = h_{m-1} \) and \( i'_m = i_m \). Condition (5) for the new pair \((Q.A, Q'.A)\) is satisfied. **End of Proof.**

**Lemma 3.7.3** \( C_{\text{Inst}}(T^\infty) \models D \).

**Proof.** Let \( d = \forall(\vec{r} \in \vec{R}) \ [ B_1(\vec{r}) \Rightarrow \exists(\vec{s} \in \vec{S}(\vec{r})) \ B_2(\vec{r}, \vec{s}) ] \) be an EPCD in \( D \) and let \( T_r \) and \( T_{r_s} \) be such that \( d = \text{dep}(T_r, T_{r_s}) \). Suppose \( v_r : T_r \rightarrow C_{\text{Inst}}(T^\infty) \) is a weak valuation. Then, by Lemma 3.7.2, there exists some finite \( n \) and homomorphism \( h : T_r \rightarrow T_n \) such that \( v_r = \text{val}_r \circ \text{id}_n \circ h \). Then for any \( m \geq n \), \( h_m = \text{id}_{i_m} \circ h : T_r \rightarrow T_m \) is a homomorphism. If \( T_m = \{ \vec{x}_m \in \vec{F}_m ; C_m(\vec{x}_m) \} \) then let \( T'_m = \{ \vec{x}_m \in \vec{F}_m ; \vec{s} \in \vec{S}(h_m(\vec{r})) ; C_m(\vec{x}_m) \text{ and } B_2(h_m(\vec{r}), \vec{s}) \} \). We claim that there exists a finite \( m \geq n \) such that either (1) there exists a homomorphism \( h'_m : T_m \rightarrow T_m \) which is the identity (modulo trivial equalities) on \( \vec{x}_m \), or (2) \( T_{m+1} = T'_m \) (in the latter case \( T_m \xrightarrow{d} T_{m+1} \)). Indeed, if this is not the case then \( d \) is applicable to any \( T_m \) with \( m \geq n \) and it is not applied, contradicting thus our earlier assumption regarding non-starvation of dependencies.

72
Let $m$ be as above. We know that $h_m \cup \text{id}_x : T_{rs} \to T_m$ is a homomorphism (Lemma 3.5.2). In case (1), $g = h'_m \circ (h_m \cup \text{id}_x) : T_{rs} \to T_m$ is a homomorphism. In case (2), $g = h_m \cup \text{id}_x : T_{rs} \to T_{m+1}$ is a homomorphism. Thus, in both cases, we obtain a homomorphism $g$ from $T_{rs}$ into some tableau at finite index $k \geq m$ in the chase sequence. Let $T_k = \{ \bar{x}_k \in \bar{P}_k : C_k(\bar{x}_k) \}$. We claim that the following EGD is trivial:

$$\forall(\bar{x}_k \in \bar{P}_k) \ [ \ C_k(\bar{x}_k) \Rightarrow g(\bar{x}) = h(\bar{x}) ]$$

Indeed, suppose we are in case (1). Then $\text{val}_m^m(h'_m(h_m(\bar{x}))) = \text{val}_m^m(h_m(\bar{x}))$ ($h'_m$ is the identity on $\bar{x}_m$, and we apply the EGD Lemma). Therefore, $\text{val}_m^k(g(\bar{x})) = \text{val}_m^k(h'_m(h_m(\bar{x}))) = \text{val}_m^k(h_m(\bar{x})) = \text{val}_m^k(h(\bar{x}))$ (we applied here Lemma 3.5.1 and the fact that $h_m = \text{id}_m \circ h$). In case (2), $\text{val}_m^k(g(\bar{x})) = \text{val}_m^k(h_m(\bar{x})) = \text{val}_m^k(h(\bar{x}))$. Hence, in both cases, using the EGD lemma, we conclude that the above EGD is trivial.

Next, define $v'_r : T_{rs} \to \text{Inst}(T^\infty)$ as $v'_r \overset{\text{def}}{=} \text{val}_c \circ \text{id}_r \circ g$. It is easily verified using the definition of $\simeq^\infty$ and Lemma 3.5.1 that $v'_r$ is a weak valuation. Moreover, $\text{val}_c^k(g(\bar{x})) = \text{val}_c^k(h(\bar{x}))$ implies $\text{val}_c^k(h(\bar{x})) = \text{val}_c^k(h(\bar{x}))$. Thus, $v'_r(\bar{x}) = \text{val}_c(\bar{x}) = \text{val}_c(\bar{x}) = v_r(\bar{x})$. We conclude that $\text{Inst}(T^\infty) \models d$, and since $d$ is arbitrary in $D$, $\text{Inst}(T^\infty) \models D$. **End of Proof.**

**Lemma 3.7.4** $\text{Inst}(T^\infty) \models D$.

**Proof.** Let $d = \text{dep}(T_r, T_{rs})$ be an EPCD in $D$ and let $v : T_r \to \text{Inst}(T^\infty)$ be an arbitrary valuation. Then by Lemma 3.7.1 there exists a weak valuation $v_s : T_r \to \text{Inst}(T^\infty)$ such that $v = \text{collapse} \circ v_s$. Since $\text{Inst}(T^\infty) \models D$ it follows that $\text{Inst}(T^\infty) \models d$. Thus there exists a valuation $v'_r : T_{rs} \to \text{Inst}(T^\infty)$ such that $v'_r(\bar{x}) = v_s(\bar{x})$. Define $v' : T_{rs} \to \text{Inst}(T^\infty)$ as $v' = \text{collapse} \circ v'_r$. Then $v'$ is a valuation and $v'(\bar{x}) = \text{collapse}(v'_r(\bar{x})) = \text{collapse}(v_s(\bar{x})) = v_r(\bar{x})$. **End of Proof.**

**Proof of Theorem 3.4.10.** We prove part 2 first. If $\text{chase}_n^m(d)$ is trivial for some $m$ then $D \models d$, by repeatedly applying Lemma 3.4.3. For the interesting direction assume $D \models d$. Suppose $d = \bar{x} \in \bar{P} \subseteq C(\bar{x}) \Rightarrow \text{Some}(\bar{y} \in \bar{R}(\bar{x})) D(\bar{x}, \bar{y})$, and let $T_x$ and $T_{xy}$ be such that $d = \text{dep}(T_x, T_{xy})$. Let $T_x = T_0 \rightarrow T_1 \rightarrow \ldots \rightarrow T_n \rightarrow \ldots$ be an infinite chase sequence of $T_x$ by $D$. Then $v = \text{val}_c \circ \text{id}_r : T_r \to \text{Inst}(T^\infty)$ is a valuation. We know by Lemma 3.7.4 that $\text{Inst}(T^\infty) \models D$, therefore, since $D \models \text{chase}_n^m d$, $\text{Inst}(T^\infty) \models d$. Hence there exists a valuation $v' : T_{xy} \to \text{Inst}(T^\infty)$ such that $v'(\bar{x}) = v(\bar{x})$, and therefore $v'(\bar{x}) = \text{val}_c(\bar{x})$. Now, by an analogous of Lemma 3.3.10 for the infinite case (the reader can verify it), there exists a weak valuation $v'_r : T_{xy} \to \text{Inst}(T^\infty)$ such that $v'_r(\bar{x}) = \text{collapse} \circ v'_r$ and $v'_r(\bar{x}) = \text{val}_c(\bar{x})$. Applying Lemma 3.7.2 we infer the existence of a tableau $T_n$ in the chase sequence and of a homomorphism $h_n : T_{xy} \to T_n$ such that $\text{val}_c \circ \text{id}_r \circ h_n = v'_r$. Thus we have $\text{val}_c(h_n(\bar{x})) = \text{val}_c(\bar{x})$, or $h_n(\bar{x}) \simeq^\infty \bar{x}$. By the definition of $\simeq^\infty$ there must exist $n \geq m$ such that $h_n(\bar{x}) \simeq_m \bar{x}$, or $\text{val}_c(\bar{x}) = \text{val}_c(\bar{x})$. We use the following notations for $T_n$ and $T_m$ ($T_n$ is a subtableau of $T_m$ since $m \geq n$):

$$T_n = \{ \bar{x}_n \in \bar{P}_n : C_n(\bar{x}_n) \}$$

$$T_m = \{ \bar{x}_n \in \bar{P}_n, \bar{z} \in \bar{Q} : C_n(\bar{x}_n) \text{ and } B(\bar{x}_n, \bar{z}) \}$$

Recall also that $T_x = T_0$ and thus $T_x = \{ \bar{x} \in \bar{P} : C(\bar{x}) \}$ is a subtableau of both $T_n$ and $T_m$. Consider now the tableau $T'$:

$$T' = \{ \bar{x}_n \in \bar{P}_n, \bar{z} \in \bar{Q}, \bar{y} \in \bar{R}(\bar{x}) : C_n(\bar{x}_n) \text{ and } B(\bar{x}_n, \bar{z}) \text{ and } D(\bar{x}, \bar{y}) \}$$

Let $h_m$ be the mapping from $T'$ into $T_m$ that is the same as $h_n$ on $\bar{x}$ and $\bar{y}$ and the identity on the rest. Since $h_m$ is a homomorphism, $h_m$ is also a homomorphism (we already used this idea in the proof of Theorem 3.4.4). We claim that the following is a trivial EGD:
\[ \forall (\bar{x}_n \in \bar{P}_n) \forall (\bar{z} \in \bar{Q}) \left[ C_n(\bar{x}_n) \textbf{ and } B(\bar{x}_n, \bar{z}) \Rightarrow \bar{x}_n = h_m(\bar{x}_n) \textbf{ and } \bar{z} = h_m(\bar{z}) \right] \]

The only part that we have to prove is the one concerning variables \( \bar{x} \) because for the rest \( h_m \) is defined to be the identity. But then we know that \( \text{val}^m_n(h_n(\bar{x})) = \text{val}^m_n(\bar{x}) \), therefore \( \text{val}^m_n(h_m(\bar{x})) = \text{val}^m_n(\bar{x}) \) and we can apply the EGD lemma. To conclude, \( h_m \) proves, by Theorem 3.3.11, that \( \text{chase}^m_D(d) \) is trivial. Part 1 follows from Part 2 by using the two Reducibility Lemmas. **End of Proof.**

**Corollary 3.7.5** Let \( D \) and \( d \) be EPCDs restricted as in Theorem 3.4.10. Then \( D \models_{\text{unr}} d \) if and only if \( D \models_{\text{unr}} d \) and both problems are r.e.
Chapter 4

A Completeness Result for the C&B Optimization

In this chapter we give our two main completeness results with regard to the C&B enumeration method. We first define the notion of a scan-minimal query. The first result, theorem 4.2.3 in section 4.2, states that, when the constraints used during the chase and backchase are only constraints characterizing materialized path-conjunctive (PC) views, the result of chasing an input PC query \( Q \) with those constraints (i.e. the universal plan) "contains" any scan-minimal rewriting of \( Q \) that is allowed to use the views. The second result, theorem 4.3.8 in section 4.3 states that the backchase minimization algorithm in which only one scan is eliminated at a time is a complete procedure for enumeration of scan-minimal subqueries, under a certain restriction. The two results put together guarantee that the C&B strategy (when the restrictions needed for the theorems are satisfied) prunes away queries that are not scan-minimal.

4.1 Preliminary Definitions.

We assume a schema \( S \) (logical and/or physical) with sets and dictionaries, and a set of EPCDs \( D \). All subsequent queries are assumed to be path-conjunctive queries over \( S \). Recall from Chapter 3, Theorem 3.3.9, that a containment mapping from a PC query \( Q_1 \) into a PC query \( Q_2 \) is a homomorphism from the tableau of \( Q_1 \) into the tableau of \( Q_2 \) that has the additional property that it maps the select clause of \( Q_1 \) into a record "equal" to the select clause of \( Q_2 \).

**Definition 4.1.1 (1-1 Minimal)** \( Q_1 \leq_{1-1} Q_2 \) if there exists a containment mapping \( h : Q_1 \to Q_2 \) such that \( h \) is one-to-one. \( Q_1 <_{1-1} Q_2 \) is \( Q_1 \leq_{1-1} Q_2 \) and \( Q_1 \) has strictly less scans than \( Q_2 \). \( Q \) is 1-1 minimal with respect to \( D \) if there is no \( Q' <_{1-1} Q \) such that \( Q' \equiv_D Q \).

We thus immediately relate the notion of \( \leq_{1-1} \) with that of subquery introduced in Chapter 2.

**Lemma 4.1.2** \( Q_1 \leq_{1-1} Q_2 \) if and only if \( Q_1 \) is a subquery of \( Q_2 \).

**Definition 4.1.3 (Scan-minimal)** A query
\[ Q = \text{select } O(\bar{x}) \text{ from } \bar{P} \bar{x} \text{ where } C(\bar{x}) \]

is scan-minimal with respect to \( D \) if:

1) \( Q \) is 1-1 minimal, and

2) for any query

\[ Q' = \text{select } O(\bar{x}) \text{ from } \bar{P} \bar{x} \text{ where } C(\bar{x}) \text{ and } C'(\bar{x}) \]

such that \( Q' \equiv_D Q \), it must be the case that \( Q' \) is 1-1 minimal.

Thus, scan-minimality is a strictly stronger condition than 1-1 minimality: any scan-minimal query must be 1-1 minimal, but there are 1-1 minimal queries that are not scan-minimal. To illustrate, recall Example 2.5.3. The query:

\[
(Q') \quad \text{select } \text{struct}(\text{title}: b.\text{title})
\text{ from Books } b, \text{ Books } b', b'.\text{copies } c \\
\text{where } b'.\text{bookId} = b.\text{bookId} \text{ and } c.\text{borrowed} = \text{false}
\]

is 1-1 minimal because there is no subquery that is equivalent to it. However, it is not scan-minimal, because we can find an equivalent query (the universal plan obtained by chasing with the key constraint on bookId):

\[
(U) \quad \text{select } \text{struct}(\text{title}: b.\text{title})
\text{ from Books } b, \text{ Books } b', b'.\text{copies } c \\
\text{where } b'.\text{bookId} = b.\text{bookId} \text{ and } c.\text{borrowed} = \text{false} \text{ and } b = b'
\]

which is not 1-1 minimal, because \( U \) has an equivalent strict subquery:

\[
(Q'_{m}) \quad \text{select } \text{struct}(\text{title}: b'.\text{title})
\text{ from Books } b', b'.\text{copies } c \\
\text{where } c.\text{borrowed} = \text{false}
\]

On the other hand, it is easy to see that \( Q'_{m} \) is scan-minimal (assuming no other constraints besides the key constraint).

### 4.2 Bounding Chase Theorem

In this section we take a closer look at the specific problem of optimizing queries in the presence of materialized views. We show that for a given logical query, although there may be infinitely many equivalent rewritings that use the views, there are only finitely many equivalent rewritings that are scan-minimal. Moreover, all these scan-minimal rewritings are subqueries of the universal plan obtained by chasing the input query with EPCDs characterizing the views. Our assumptions for this section are the following:

1. The logical schema contains only (nested) relations and classes modeled with dictionaries. We will collectively denote the logical schema names by \( \bar{R} \).
2. There are no dependencies over the logical schema (i.e. no semantic constraints).
3. The physical schema consists of all the relations and classes of the logical schema, \( \bar{R} \), and, in addition, of a set of path-conjunctive materialized views, denoted by \( \bar{V} \). Each view \( \bar{V} \) in \( \bar{V} \) is characterized by a definition \( \bar{V}^{(R)} \), i.e. a path-conjunctive query over the logical schema:
\[ V \overset{\text{def}}{=} \text{select } O(\bar{p}) \text{ from } \bar{P} \bar{p} \text{ where } B(\bar{p}) \]

However, as in section 2.3, in the optimizer this is replaced by a pair of EPCDs:

\[
\begin{align*}
\dot{d}_V &= \forall(\bar{p} \in \bar{P}) \ [B(\bar{p}) \Rightarrow \exists(v \in V) \ v = O(\bar{p})] \\
\ddot{d}_V &= \forall(v \in V) \ \exists(\bar{p} \in \bar{P}) \ B(\bar{p}) \quad \text{and} \quad v = O(\bar{p})
\end{align*}
\]

We collectively denote these dependencies by \( \dot{d}_V \) and \( \ddot{d}_V \).

In general, not all views contribute to a rewriting of a query \( Q(\bar{R}) \). We call a view \( V \) relevant to \( Q \) if there exists some query \( Q'(\ldots, V, \ldots) \) equivalent to \( Q \). (This equivalence is not with respect to all databases but with respect to all databases that satisfy the view definitions \( V \equiv V(\bar{R}) \). We denote this equivalence by \( \equiv_V \).) When enumerating query plans we are interested only in relevant views. The following lemma characterizes the class of relevant views, thus providing a first pruning strategy for the space of plans.

**Lemma 4.2.1 (Relevant Views)** Given a logical query \( Q(\bar{R}) \), a view \( V \) is relevant to \( Q \) iff there exists a homomorphism from the tableau of \( V \) (tableau of \( \dot{d}_V \)) into the tableau of \( Q \).

**Proof.** Let \( Q(\bar{R}) = \text{select } O(\bar{r}_1) \text{ from } \bar{R}_1 \bar{r}_1 \text{ where } C_1(\bar{r}_1) \). Suppose \( V \) is relevant to \( Q \), that is, there exists an equivalent rewriting \( Q' \) of \( Q \) that uses \( V \). Assume, for simplicity, that \( V \) is the only view that occurs in \( Q' \). Thus, \( Q' = \text{select } O_2(\bar{r}_2, v) \text{ from } \bar{R}_2 \bar{r}_2, \ V v \text{ where } C_2(\bar{r}_2, v) \). Let \( V = \text{select } O(\bar{p}) \text{ from } \bar{P} \bar{p} \text{ where } B(\bar{p}) \). Then the query \( Q'_V(\bar{R}) \) obtained from \( Q' \) by unfolding the view definition:

\[ Q'_V(\bar{R}) = \text{select } O_2(\bar{r}_2, O(\bar{p})) \text{ from } \bar{R}_2 \bar{r}_2, \bar{P} \bar{p} \text{ where } C_2(\bar{r}_2, O(\bar{p})) \text{ and } B(\bar{p}) \]

must be equivalent to \( Q \) (under all instances). Thus, by Theorem 3.3.9, there exists a containment mapping \( h \) from \( Q'_V \) into \( Q \). In other words, \( h \) is a homomorphism from the tableau of \( Q'_V \), \( \{ \bar{r}_2 \in \bar{R}_2, \bar{p} \in \bar{P}; \ C_2(\bar{r}_2, O(\bar{p})) \text{ and } B(\bar{p}) \} \) into the tableau of \( Q \), \( \{ \bar{r}_1 \in \bar{R}_1; \ C_1(\bar{r}_1) \} \) such that the following EGD is trivial:

\[ \forall(\bar{r}_1 \in \bar{R}_1) \ [ C_1(\bar{r}_1) \Rightarrow C_1(h(\bar{r}_1), O(h(\bar{p}))) \text{ and } B(h(\bar{p})) \text{ and } O_1(r_1) = O_2(h(\bar{r}_2), O(h(\bar{p}))) ] \]

But this immediately implies that the following EGD is trivial:

\[ \forall(\bar{r}_1 \in \bar{R}_1) \ [ C_1(\bar{r}_1) \Rightarrow B(h(\bar{p})) ] \]

therefore the restriction of \( h \) to \( \bar{p} \) is a homomorphism from the tableau of \( V \) into the tableau of \( Q \). **End of Proof.**

Thus whenever there exists a rewriting of \( Q \) that uses \( V \) there exists a chase step that rewrites \( Q \) into a query that uses \( V \). Moreover, the next lemma shows that for any scan-minimal rewriting of \( Q \) that uses \( n \) occurrences of \( V \) there exists a sequence of \( n \) chase steps that rewrites \( Q \) into a query with \( n \) occurrences of \( V \). As the proof of the lemma shows, the scan-minimality condition is necessary in order to ensure that each chase step doesn’t introduce a trivial occurrence of \( V \) (thus each rewrite step is a valid chase step, see definition 3.4.2 in Chapter 3).

**Lemma 4.2.2** Let \( Q(\bar{R}) \) be a logical query and let \( Q'(\bar{R}, \bar{V}) \) be a scan-minimal equivalent rewriting of \( Q \) such that the view \( V \) occurs \( n \) times in \( Q' \). Then

1. there exists a chase sequence: \( Q \xrightarrow{\dot{d}_V} Q_1 \xrightarrow{\ddot{d}_V} \ldots \xrightarrow{\ddot{d}_V} Q_n \)
2. \( Q' \) is a subquery of \( Q_n \)
**Proof.** We prove 1) first. Let, as before,

\[ Q(\bar{R}) = \textbf{select} \ O_1(r_1') \textbf{ from } \bar{R}_1 \ r_1' \textbf{ where } C_1(r_1') \]

\[ V = \textbf{select} \ O(\bar{p}) \textbf{ from } \bar{P} \ p \textbf{ where } B(\bar{p}) \]

and assume, for simplicity, that \( Q' \) has only two occurrences of \( V \) and no other views appear in \( Q' \). Thus,

\[ Q' = \textbf{select} \ O_2(r_2', v, v') \textbf{ from } \bar{R}_2 \ r_2', V, V \ v, v' \textbf{ where } C_2(r_2', v, v'). \]

Consider the unfolded version of \( Q' \):

\[ Q'_u(\bar{R}) = \textbf{select} \ O_2(r_2', O(\bar{p}), O(\bar{p}')) \textbf{ from } \bar{R}_2 \ r_2', \bar{P} \ p, \bar{P} \ p' \textbf{ where } C_2(r_2', O(\bar{p}), O(\bar{p}')) \and B(\bar{p}) \and B(\bar{p}') \]

As in the proof of the previous lemma, there exists a containment mapping \( h \) from \( Q'_u \) to \( Q \). By taking the two restrictions, \( h_1 \) and \( h_2 \), of \( h \) to \( \bar{p} \) and, respectively, \( \bar{p}' \), we obtain two homomorphisms from the tableau of \( V \) (and therefore of \( d_Y \)) into \( Q \). We show that there are two chase steps that use \( h_1 \) and \( h_2 \) to rewrite \( Q \) into a query with two occurrences of \( V \). The first chase step using the mapping \( h_1 \) from \( d_Y \) into \( Q \) rewrites \( Q \) into:

\[ Q_1 = \textbf{select} \ O_1(r_1') \textbf{ from } \bar{R}_1 \ r_1', V, V \ v \textbf{ where } C_1(r_1') \and v = O(h(\bar{p}')) \]

In the above we made use of the fact that \( h_1(\bar{p}) = h(\bar{p}) \). Since \( Q \) does not have any occurrence of \( V \) condition (2) in the definition of the chase step\(^1\) is automatically ensured (there is no way to map the new variable \( v \) occurring in \( Q_1 \) to any of the variables of \( Q \)). However, for the second chase step, using \( h_2 \), to rewrite \( Q_1 \) into:

\[ Q_2 = \textbf{select} \ O_1(r_1') \textbf{ from } \bar{R}_1 \ r_1', V, V \ v' \textbf{ where } C_1(r_1') \and v = O(h(\bar{p})) \and \ v' = O(h(\bar{p}')) \]

we need to prove that there is no homomorphism

\[ g : \{r_1' \in \bar{R}_1, v \in V, v' \in V' ; \ C_1(r_1') \and v = O(h(\bar{p})) \and \ v' = O(h(\bar{p}')) \} \]

\[ \longrightarrow \{r_1' \in \bar{R}_1, v \in V; C_1(r_1') \and v = O(h(\bar{p}'))\} \]

such that \((e_1) \ \forall (r_1' \in \bar{R}_1) \forall (v \in V) [ C_1(r_1') \and v = O(h(\bar{p})) \Rightarrow r_1' = g(r_1') \and v = g(v) ] \) is a trivial EGD. Suppose that such a homomorphism exists. Then it must be the case that \( g(v') = v \) since \( V \) doesn't occur in \( \bar{R}_1 \). We also know, from the fact that \( g \) is a homomorphism, that the following is also a trivial EGD:

\[ (e_2) \ \forall (r_1' \in \bar{R}_1) \forall (v \in V) [ C_1(r_1') \and v = O(h(\bar{p})) \Rightarrow g(v') = O(g(h(\bar{p}'))) ] \]

Since \( h(\bar{p}') \) is among \( r_1' \), using \((e_1) \) we can replace \( g(h(\bar{p}')) \) in the above EGD with \( h(\bar{p}') \). Replacing also \( g(v') \) with \( v \) we infer that

\[ (e_3) \ \forall (r_1' \in \bar{R}_1) \forall (v \in V) [ C_1(r_1') \and v = O(h(\bar{p})) \Rightarrow v = O(h(\bar{p}')) ] \]

is a trivial EGD as well.

**Sublemma.** The EGD \((e) \ \forall (r_1' \in \bar{R}_1) [ C_1(r_1') \Rightarrow O(h(\bar{p})) = O(h(\bar{p}')) ] \) is trivial.

**Proof of Sublemma.** By unfolding the view definition in \( e_3 \) we obtain the following trivial EGD:

\[ (e'_3) \ \forall (r_1' \in \bar{R}_1) \forall (\bar{p} \in \bar{P}) [ C_1(r_1') \and O(\bar{p}) = O(h(\bar{p})) \and B(\bar{p}) \Rightarrow O(\bar{p}) = O(h(\bar{p}'))) ] \]

On the other hand, the following is also a trivial EPCD:

\(^1\)The non-triviality condition.
(e₄) \( \forall (rₗ^1 ∈ \bar{R}_1) \ [ C_1(rₗ^1) \Rightarrow \exists (p ∈ \bar{P}) \ B(p) \ \text{and} \ O(p) = O(h(p)) \] 

because there exists a homomorphism satisfying the conditions of theorem 3.3.11, namely \( h \) itself. Then chasing \( e \) with \( e₃ \) and \( e₄ \) we obtain a trivial EGD. Thus \( e \) is a consequence of \( e₃ \) and \( e₄ \), and since \( e₃ \) and \( e₄ \) are trivial, \( e \) is trivial as well. End of proof of sublemma.

The immediate consequence of the sublemma is that the following query:

\[
Q'_e(\bar{R}) = \textbf{select } O_2(r'_2, O(p), O(p')) \textbf{ from } \bar{R}_2 \ r'_2, \ \bar{P} \ p, \ \bar{P} \ p' \\
\text{where } C_2(r'_2, O(p), O(p')) \ \text{and} \ B(p) \ \text{and} \ B(p') \ \text{and} \ O(p) = O(p')
\]

can be proven equivalent under all instances to \( Q \). Indeed, recall that \( Q'_e \) is equivalent under all instances to \( Q' \), one of the two containment mappings being \( h \) from \( Q'_e \) into \( Q \). Let \( h' \) be the other containment mapping from \( Q \) into \( Q'_e \). It is easy to see that \( h' \) is still a containment mapping when considered from \( Q \) into \( Q'_e \). On the other hand, \( h \) itself is still a containment mapping when considered from \( Q'_e \) into \( Q \) because of the EGD (e) proven trivial in the above sublemma. Now, consider the query:

\[
Q'' = \textbf{select } O_2(r'_2, v, v') \textbf{ from } \bar{R}_2 \ r'_2, \ \forall \ v, \ \forall \ v' \textbf{ where } C_2(r'_2, v, v') \ \text{and} \ v = v'
\]

It is not hard to see that \( Q'' \) is the unfolded version of \( Q'' \). Then, since \( Q'' \) is equivalent to \( Q'' \), it follows that \( Q'' \equiv Q \). Therefore \( Q'' \equiv Q' \). Now observe that \( Q'' \) is not 1-1 minimal. Indeed, the query:

\[
Q''_m = \textbf{select } O_2(r'_2, v, v) \textbf{ from } \bar{R}_2 \ r'_2, \ \forall \ v \textbf{ where } C_2(r'_2, v, v)
\]

is a subquery of \( Q'' \) and, moreover, equivalent to \( Q'' \). This contradicts the fact that \( Q' \) was assumed scan-minimal.

We prove next item 2) of the lemma. Recall the containment mapping \( h \) from \( Q'_e \) into \( Q \). We show first that \( h \) is one-to-one on \( r'_2 \) and then we extend \( h \) to be the identity on \( v \) and \( v' \) and we show that the extension is a one-to-one containment mapping from \( Q' \) to \( Q_2 \). Therefore, \( Q' \) is a subquery of \( Q_2 \). The general case for \( n \) is the same.

Suppose \( h \) is not one-to-one on \( r'_2 \). Thus there exist scans \( S_1 \ s_1 \) and \( S_2 \ s_2 \) among \( \bar{R}_2 \ r'_2 \) such that \( h(s_1) = h(s_2) \) = \( r \), where \( R \) is some scan of \( Q \). \( Q'_e \) has the following form:

\[
Q'_e(\bar{R}) = \textbf{select } O_2(r'_3, s_1, s_2, O(p), O(p')) \textbf{ from } \bar{R}_3 \ r'_3, \ S_1 \ s_1, \ S_2 \ s_2 \ \bar{P} \ p, \ \bar{P} \ p' \\
\text{where } C_2(r'_3, s_1, s_2, O(p), O(p')) \ \text{and} \ B(p) \ \text{and} \ B(p')
\]

From the conditions of \( h \) being a containment mapping we know that \( h(S_1) \) must be "equal" to \( R \), and similarly, \( h(S_2) \) must be "equal" to \( R \). Thus, the following EGD is trivial:

\[
\forall (r'_1 ∈ \bar{R}_1) \ [ C_1(r'_1) \Rightarrow h(S_1) = h(S_2) ]
\]

We can infer now that \( h \) is also a containment mapping from the following modified version of \( Q'_e \):

\[
Q'_e(\bar{R}) = \textbf{select } O_2(r'_3, s_1, s_2, O(p), O(p')) \textbf{ from } \bar{R}_3 \ r'_3, \ S_1 \ s_1, \ S_2 \ s_2 \ \bar{P} \ p, \ \bar{P} \ p' \\
\text{where } C_2(r'_3, s_1, s_2, O(p), O(p')) \ \text{and} \ B(p) \ \text{and} \ B(p') \ \text{and} \ s_1 = s_2 \ \text{and} \ S_1 = S_2
\]

Chasing with trivial constraints doesn’t satisfy condition 2) in definition 3.4.2. However, since we are only making use of the soundness of this more relaxed form of chase, this is OK.

79
into $Q$. Therefore, $Q^1$ and $Q$ are equivalent under all instances\(^3\). Folding the views in $Q^1$ we obtain:

$$
Q^1 = \textbf{select } O_2(r_3', s_1, s_2, v, v') \\
\textbf{from } R_0, r_3', S_1 s_1, S_2 s_2, \forall v, \forall v' \quad \text{where } C_2(r_3', s_1, s_2, v, v') \quad \text{and } s_1 = s_2 \quad \text{and } S_1 = S_2
$$

We have then that $Q^1 \equiv_q Q'$. But $Q^1$ has an equivalent strict subquery:

$$
Q^1_m = \textbf{select } O_2(r_3', s_1, s_1, v, v') \\
\textbf{from } R_0, r_3', S_1 s_1, \forall v, \forall v' \quad \text{where } C_2(r_3', s_1, s_1, v, v')
$$

and this contradicts the fact that $Q'$ was assumed to be scan-minimal. Thus, we conclude that $h$ is one-to-one on $r_2$. It remains to show that $h$ when extended to be the identity on $v$ and $v'$ is a containment mapping from $Q'$ into $Q_2$, i.e. we need to check that the following EGD is trivial:

$$\forall (r_1' \in R_1) \forall (v \in V) \forall (v' \in V) \quad [ \begin{align*}
& C_1(r_1') \ \text{and } v = O(h(\bar{p})) \ \text{and } v' = O(h(\bar{p}')) \\
& \Rightarrow C_2(h(r_2'), v, v') \ \text{and } O_2(h(r_2'), v, v') = O_1(r_1')
\end{align*} ]$$

On the other hand, we know that the following EGD is trivial:

$$\forall (r_1' \in R_1) \quad [ \begin{align*}
& C_1(r_1') \Rightarrow C_2(h(r_2'), O(h(\bar{p})), O(h(\bar{p}'))) \ \text{and } O_2(h(r_2'), O(h(\bar{p})), O(h(\bar{p}'))) = O_1(r_1')
\end{align*} ]$$

because $h$ is a containment mapping from $Q_1'$ into $Q$. By strengthening the universal part and the left-hand side of the $\Rightarrow$ as below we obtain an EGD still trivial:

$$\forall (r_1' \in R_1) \forall (v \in V) \forall (v' \in V) \quad [ \begin{align*}
& C_1(r_1') \ \text{and } v = O(h(\bar{p})) \ \text{and } v' = O(h(\bar{p}')) \\
& \Rightarrow C_2(h(r_2'), O(h(\bar{p})), O(h(\bar{p}'))) \ \text{and } O_2(h(r_2'), O(h(\bar{p})), O(h(\bar{p}'))) = O_1(r_1')
\end{align*} ]$$

But the above EGD immediately implies the one that we need. **End of Proof.**

Thus all occurrences of views in scan-minimal equivalent rewritings are part of the result of chasing $Q$ with $d_Q^-$, and moreover scan-minimal rewritings are subqueries of the result of chasing. Now, observe that $d_Q^-$ are full EPCDs. Therefore, the result of chasing $Q$ is finite and unique! (Theorems 3.4.6 and 3.4.8). Hence, all scan-minimal equivalent rewritings are subqueries of the same finite query, the universal plan $\text{chase}_{\overline{d}_Q}^-(Q)$. This is summarized by the next theorem.

**Theorem 4.2.3 (Bounding Chase)** Let $Q(\overline{R})$ be a PC query over logical schema with relations and dictionaries $\overline{R}$, and let $\overline{V}$ be a set of PC view definitions characterized by EPCDs $d_Q^-$ and $\overline{d}_Q^-$. Then any scan-minimal rewriting $Q'((\overline{R}, \overline{V})$ of $Q$ is a subquery of $\text{chase}_{\overline{d}_Q}^-(Q)$.

Remark that the worst-case size of the universal plan $\text{chase}_{\overline{d}_Q}^-(Q)$ is polynomial in the size of the query $Q$, and the number of views in the schema (see Proposition 3.6.9). Enumerating scan-minimal equivalent rewritings can be done by looking at subqueries of the universal plan and the complexity of the enumeration procedure is then exponential in the number of relevant views.

\(^3\)Notice that $S_1 = S_2$ is a set equality. However we are using only one direction of Theorem 3.3.9, namely that existence of a containment mapping implies containment, and this direction doesn’t require the set/dictionary equality restriction.
4.3 Complete Subquery Enumeration

In this section we focus on the enumeration of subqueries of the universal plan. This enumeration is a minimization procedure (the backchase algorithm) that produces essentially all scan-minimal equivalent subqueries of the universal plan.

**Complete backchase step.** The backchase step that was shown in Chapter 2 looks at subqueries of a query and checks equivalence by testing whether a certain constraint $\delta$ is implied by the constraints in the schema. This test is sufficient for equivalence of the two queries but not necessary. Thus, there may be equivalent subqueries of a query that are not explored by the backchase minimization algorithm because the backchase step fails to recognize them equivalent. In this section we rectify this problem and we give the complete version of the backchase step.

To illustrate, consider a relation $R(A,B,C)$, $D = \emptyset$ and the following two queries:

\[
(Q_2) \quad \text{select } \textbf{struct}(A = r_1.A, B = r_2.B) \\
\text{from } R \ r_1, R \ r_2, R \ t \\
\text{where } r_1.A = r_2.A \text{ and } t.B = r_2.B \text{ and } t.C = r_1.C
\]

\[
(Q_1) \quad \text{select } \textbf{struct}(A = r_1.A, B = r_2.B) \\
\text{from } R \ r_1, R \ r_2 \\
\text{where } r_1.A = r_2.A
\]

It is obvious that $Q_1$ is a subquery of $Q_2$. Thus, since there exists a containment mapping from $Q_1$ into $Q_2$ (the identity one), we have $Q_2 \subseteq Q_1$. We can also find an inverse containment mapping from $Q_2$ into $Q_1$ (for example, the one in which $r_1, r_2$ and $t$ are all mapped into $r_2$). Thus, $Q_1 \subseteq Q_2$, and $Q_1 \equiv Q_2$. However,

\[
(\delta) \quad \forall (r_1 \in R) \forall (r_2 \in R) \left[r_1.A = r_2.A \Rightarrow \exists (t \in R) t.B = r_2.B \text{ and } t.C = r_1.C\right]
\]

is not a trivial constraint: we cannot find any homomorphism from the tableau \{r_1 \in R, r_2 \in R, t \in R; r_1.A = r_2.A \text{ and } t.B = r_2.B \text{ and } t.C = r_1.C\} into the tableau \{r_1 \in R, r_2 \in R; r_1.A = r_2.A\} such that $h(r_1) = r_1$ and $h(r_2) = r_2$.

In general, the necessary and sufficient condition that guarantees the equivalence under $D$ of $Q_1$ and $Q_2$ when $Q_1$ is a subquery of $Q_2$ is, by Lemma 3.4.1, $D \models \text{cont}(Q_1, Q_2)$. The complete backchase step, that we are going to use from now on, checks for this condition. For our example, this translates into checking whether the following constraint is trivial:

\[
(\delta') \quad \forall (r_1 \in R) \forall (r_2 \in R) \left[r_1.A = r_2.A \Rightarrow \exists (r_1' \in R) \exists (r_2' \in R) \exists (t \in R) r_1'.A = r_2'.A \text{ and } r_1'.B = r_2'.B \text{ and } t.C = r_1'.C\right]
\]

And, of course, $(\delta')$ is trivial: the witness is the same homomorphism that we used as a containment mapping from $Q_2$ into $Q_1$.

**Decremental backchase.** In the following we show that, by considering only backchase steps that remove only one scan at a time, we are able enumerate, in a complete way, all equivalent subqueries of a given query. Informally speaking, the proof will consist of showing that, given a query $Q$ with variables $x_1, \ldots, x_k, \ldots, x_n$, for any equivalent subquery $Q_1$ of $Q$ with variables $x_{k+1}, \ldots, x_n$, there exists a sequence of $k$ backchase steps, each removing exactly one of the variables $x_1, \ldots, x_k$, each preserving equivalence, ending in $Q_1$. Thus a systematic enumeration of minimal equivalent subqueries of $Q$ can proceed in a top-down way, decrementally removing one scan at a time, without missing any equivalent subquery.

Recall, from Chapter 3, that the (non-extensional) canonical instance of a PC tableau $T$ consists of a pair $(G, \simeq)$ where $G$ contains all paths that are well-defined over $T$ while $\simeq$ is the congruence closure containing all equalities
between paths in $G$ that can be inferred from the equalities in $T$. In addition, the canonical instance has $\text{dom}$, $\emptyset$, $\wedge$, and $\in$-edges. For conciseness, we do not denote explicitly the edges when we talk about a canonical instance and we refer to it as simply a pair $(G, \simeq)$. The following definition is a variation on the notion of subquery.

**Definition 4.3.1 (Sub-instance)** Let $T_1$ and $T_2$ be two well-defined PC tableaux with canonical instances $(G_1, \simeq_1)$ and, respectively, $(G_2, \simeq_2)$. We say that $T_1$ is a subtableau of $T_2$ and $(G_1, \simeq_1)$ is a sub-instance of $(G_2, \simeq_2)$ if there exists a homomorphism $h : T_1 \rightarrow T_2$ such that $h$ is one to one.

Let $T$ be a PC tableau with canonical instance $(G, \simeq)$ and assume $x$ is some variable that occurs in $T$. We construct a pair $(G_{-x}, \simeq_{-x})$ in which $G_{-x}$ is a subset of $G$ consisting of all paths in $G$ that do not depend on $x$ while $\simeq_{-x}$ is essentially the restriction of $\simeq$ to paths in $G_{-x}$. Finding what are the paths in $G$ that do not depend on $x$ is not simple because we have to trace back the derivations of such paths in $G$ and check whether their derivation used $x$ or not. We choose to do this in a slightly different way by redoing the entire derivations of paths in $G$ taking care not to use $x$. The set of rules that we give below formally describe the construction.

We will call $(G_{-x}, \simeq_{-x})$ the canonical sub-instance of $(G, \simeq)$ that removes the variable $x$. We will show next that any sub-instance of $(G, \simeq)$ that doesn’t have the variable $x$ is necessarily a sub-instance of $(G_{-x}, \simeq_{-x})$. Thus, $(G_{-x}, \simeq_{-x})$ is the maximal sub-instance of $(G, \simeq)$ that removes $x$. Moreover, the rules provide us with an effective way of computing the maximal sub-instance. It will be easy from here to define (and compute) the maximal subquery of a query that removes a particular scan. Then the backchase enumeration algorithm will enumerate subqueries of a given query by reducing the problem to enumerating subqueries of its maximal subqueries.

Given tableau $T$ with canonical instance $(G, \simeq)$ and variable $x$ occurring in $T$, let $G_{-x}$ be the least set, and $\simeq_{-x}$ and $\in_{-x}$ be the least binary relations that are closed under the rules:

\[
\begin{align*}
\text{(prj-add-x)} & \quad Q : G_{-x} & \quad (\text{dom-add-x}) & \quad \text{dom } Q : G_{-x} & \quad (\text{root-add-x}) & \quad R \text{ in the schema} & \quad R : G_{-x} \\
\quad Q : A & \quad \Rightarrow Q : A_{-x} & \quad \text{dom } Q & \quad \Rightarrow \text{dom } Q_{-x} & \quad \Rightarrow R \text{ in the schema} & \quad \Rightarrow R : G_{-x} \\
\text{(var-add-x)} & \quad y \in S \text{ in } G, \quad y \neq x, \quad S \simeq S', \quad S' : G_{-x} & \quad \Rightarrow y : G_{-x} \\
\text{(in-add-x)} & \quad y \in S \text{ in } G, \quad y : G_{-x}, \quad S \simeq S', \quad S' : G_{-x} & \quad \Rightarrow y \in_{-x} S' \\
\text{(true-add-x)} & \quad \text{true} : G_{-x} & \quad (\text{false-add-x}) & \quad \text{false} : G_{-x} \\
\text{(lookup-add-x)} & \quad y \simeq_{-x} y', \quad y' \in_{-x} S', \quad S' \simeq_{-x} \text{ dom } Q & \quad \Rightarrow Q[ y ] : G_{-x} \\
\text{(eq-add-x)} & \quad Q_1 \simeq Q_2, \quad Q_1 : G_{-x}, \quad Q_2 : G_{-x} & \quad \Rightarrow Q_1 \simeq_{-x} Q_2
\end{align*}
\]
Then the canonical sub-instance of \((G, \simeq)\) that removes \(x\) is defined as follows. The well-defined paths and the congruence closure are \(G_{\simeq x}\) and \(\simeq_{\simeq x}\), respectively. The \(\ell, \emptyset\) and \(\text{dom}\) edges are defined in the obvious way. For the \(\varepsilon\)-edges we need first to observe that we may have many choices: it is possible to have \(y \in_{\simeq x} S_1\) and \(y \in_{\simeq x} S_2\) with \(S_1\) and \(S_2\) different paths. Then, we choose among such paths the one, call it \(S\), with the smallest derivation length (in \(G_{\simeq x}\)), and we add an \(\varepsilon\)-edge between \(y\) and \(S\). We denote such an \(\varepsilon\)-edge, by abuse of notation, with \(y \in_{\simeq x} S\). We remark that there may be other variables besides \(x\) that do not occur in \(G_{\simeq x}\).

The above construction is similar to the one given in section 3.2. The main difference lies in the fact that while in the construction of the canonical instance of a tableau \(T\) the choice of variables and of their \(\varepsilon\)-edges is given by the scans that occur in the tableau \(T\), here we need to infer them in an explicit way. First, we need to infer what are the variables \(y\) that do not depend on \(x\), and therefore belong to the canonical sub-instance. Second, for any such \(y\) we need to infer what is the set node \(S'\) such that \(y \in_{\simeq x} S'\). Although \(y\) was connected to some node \(S\) in the original canonical instance, \(S\) may depend on \(x\) and therefore not belong to the canonical sub-instance. Thus we need to find a replacement for \(S\). The way we find this replacement is by looking for an expression \(S'\) that is "equal" to \(S\) (in the original canonical instance) and is known to belong already to \(G_{\simeq x}\). This is summarized in the two rules above, \((\text{var-add-k})\) and \((\varepsilon\text{-add-x})\). By taking \(S'\) to be some expression "equal" to \(S\) we will be able to show that the identity mapping from \(G_{\simeq x}\) to \(G\) is a homomorphism. Moreover by taking \(S'\) to be the one with the smallest derivation in \(G_{\simeq x}\) we will be able to show that there exists a well-defined PC tableau \(T_{\simeq x}\) such that \(\text{ClInst}(T_{\simeq x})\) is exactly the canonical sub-instance.

**Lemma 4.3.2** \(G_{\simeq x} \subseteq G\) and \(\simeq_{\simeq x} \subseteq \simeq\).

**Proof.** Simple induction on the derivation of \(Q : G_{\simeq x}\) and \(Q \simeq_{\simeq x} Q'\). End of Proof.

**Lemma 4.3.3** \(\simeq_{\simeq x}\) is closed under rules \((\text{refl})\), \((\text{sym})\), \((\text{trans})\), \((\text{prj-cong})\), \((\text{dom-cong})\), \((\text{red-ext})\) and \((\text{lookup-cong})\) from section 3.2.

**Proof.** Straightforward case analysis. End of Proof.

**Proposition 4.3.4** There exists a well-defined PC tableau \(T_{\simeq x}\) such that its canonical instance is \((G_{\simeq x}, \simeq_{\simeq x})\).

**Proof sketch.** The scans that occur in \(T_{\simeq x}\) are all pairs \(y \in_{\simeq x} S\) occurring in the canonical sub-instance, while the path-conjunction can be the conjunction of all terms of the form \(Q_1 = Q_2\) with \(Q_1 \simeq_{\simeq x} Q_2\). We only need to check that we write the scans in \(T\) in such an order that if \(y_i \in_{\simeq x} S_i\) is the \(i\)th scan, then \(S_i\) does not depend on any of \(y_1, \ldots, y_{i-1}\). We do this by generating the scans \(y_i \in_{\simeq x} S_i\) in the order of the derivation length of \(y_i\).

Thus, \(T_{\simeq x}\) has the following form:

\[
T_{\simeq x} \overset{\text{def}}{=} \{y_1 \in S_1, \ldots, y_k \in S_k; \ \wedge\{Q_1 = Q_2 \mid Q_1 \simeq_{\simeq x} Q_2\}\}
\]

and it is syntactically well-formed. Let \((G', \simeq')\) be the canonical instance resulting from applying the construction of section 3.2 on \(T_{\simeq x}\). It is simple to show that \((G', \simeq') \subseteq (G_{\simeq x}, \simeq_{\simeq x})\). We show here only the other direction \((G_{\simeq x}, \simeq_{\simeq x}) \subseteq (G', \simeq')\). We prove by induction on the derivation of \(Q : G_{\simeq x}\) and \(Q_1 \simeq_{\simeq x} Q_2\). Thus we do a case analysis on the last rule applied in such a derivation.

**Case \((\text{eq-x})\).** We have \(Q_1 \simeq_{\simeq x} Q_2\) as a result of \(Q_1 \simeq Q_2\), \(Q_1 : G_{\simeq x}\) and \(Q_2 : G_{\simeq x}\) and we need to show that \(Q_1 \simeq Q_2\). By the inductive hypothesis, since the derivations of \(Q_1 : G_{\simeq x}\) and \(Q_2 : G_{\simeq x}\) are smaller, it must be

\footnote{Remark that it must be the case that \(S_i\) has a derivation length smaller than that of \(y_i\) (from our choice of \(S_i\)) thus \(S_i\) cannot depend on \(y_{i+1}, \ldots, y_k\) which have derivation lengths strictly larger than \(y_i\).}

83
the case that \( Q_1 : G' \) and \( Q_2 : G' \). Then, since \( Q_1 = Q_2 \) occurs in the path-conjunction of \( T_{-x} \), we can infer by rule (eq) that \( Q_1 \simeq Q_2 \).

Rules (proj-add-x) and (dom-add-x) require simple application of the induction hypothesis and the corresponding rules (proj-add) and (dom-add). Rules (root-add-x), (true-add-x) and (false-add-x) are obvious.

**Case (var-add-x).** We have \( y_i : G_{-x} \) as a result of \( y_i \in S \) in the original canonical instance, \( y_i \neq x \), \( S \simeq S' \), and \( S' : G_{-x} \). Since we can infer in one more derivation step that \( y_i \in S' \) \( S' \) and we know that what we have in \( T_{-x} \) is \( y_i \in S_i \) it must be the case that \( S_i : G_{-x} \) with a smaller derivation than that of \( S' : G_{-x} \). Thus we can apply the inductive hypothesis to conclude that \( S_i : G' \) and then by (var-add) it follows that \( y_i : G' \).

**Case (lookup-add-x).** We have \( Q \{ y \} : G_{-x} \) with a derivation of length \( n \) as a result of \( y \in S' \) \( y' \in S' \) and \( S' \simeq \text{dom } Q \) with derivation lengths of at most \( n - 1 \). By I.H. we must have \( y \simeq y' \) and \( S' \simeq \text{dom } Q \). What we have in \( T_{-x} \) is \( y \in S_j \) for some \( S_j : G_{-x} \) with derivation smaller than that of \( S' : G_{-x} \) and \( y' \in S_j \). Then the derivation of \( y' \in S' \), of length \( n - 1 \), is of the form:

\[
\begin{array}{c}
y' \in S \text{ in } G, \quad y' : G_{-x}, \quad S \simeq S', \quad S' : G_{-x}
y' \in S'
\end{array}
\]

with each derivation in the premise having length at most \( n - 2 \). On the other hand, the derivation of \( y' \in S_j \) is of the form:

\[
\begin{array}{c}
y' \in S \text{ in } G, \quad y' : G_{-x}, \quad S \simeq S_j, \quad S_j : G_{-x}
y' \in S_j
\end{array}
\]

and, since \( S_j : G_{-x} \) has a derivation smaller than \( S' : G_{-x} \), is of length at least \( n - 1 \). Thus, each derivation in the premise is of length at most \( n - 2 \). From \( S \simeq S' \) and \( S \simeq S_j \) we have \( S' \simeq S_j \). Together with \( S' : G_{-x} \) and \( S_j : G_{-x} \) we must have, by (eq-x), \( S' \simeq S_j \). Thus we can apply the inductive hypothesis to conclude that \( S' \simeq S_j \). We have already argued that \( S' \simeq \text{dom } Q \). Thus, by transitivity, \( S_j \simeq \text{dom } Q \). Using \( y \simeq y' \) and \( y' \in S_j \) in \( T_{-x} \), by (lookup-add), we obtain that \( Q \{ y \} : G' \). **End of Proof.**

**Lemma 4.3.5** The identity mapping \( \text{id} : (G_{-x}, \simeq_{-x}) \to (G, \simeq) \) is a one-to-one homomorphism.

**Proof.** By lemma 4.3.2 it is obvious that such an identity mapping exists. To show that it is a homomorphism we use again lemma 4.3.2 and the following observation. Suppose \( y \in S \). Then by (eq-add-x) rule it must be the case that \( S \simeq S' \). Then since \( \text{id}(S') = S' \) it follows that \( \text{id}(S) \simeq S \). **End of Proof.**

Thus, \( (G_{-x}, \simeq_{-x}) \) is a sub-instance of \( (G, \simeq) \) and \( T_{-x} \) is a subtableau of \( T \). We show next that any sub-instance /subtableau of \( (G, \simeq) / T \) that does not use \( x \) must be a sub-instance / subtableau of \( (G_{-x}, \simeq_{-x}) / T_{-x} \).

**Proposition 4.3.6 (Maximal sub-instance)** Let \( T \) be a tableau with canonical instance \( (G, \simeq) \), let \( x \) be a variable occurring in \( T \), and let \( (G_{-x}, \simeq_{-x}) \) be the canonical sub-instance of \( (G, \simeq) \) that removes \( x \). Let \( T' \) be a tableau with canonical instance \( (G', \simeq') \) such that there exists a one-to-one homomorphism \( h : (G', \simeq') \to (G, \simeq) \) with \( x \) not in the image of \( h \). Then there exists a one-to-one homomorphism \( h' : (G', \simeq') \to (G_{-x}, \simeq_{-x}) \) such that \( \text{id} \circ h' = h \).

**Proof.** We construct \( h' \) by induction on the derivation of \( Q : G' \) or \( Q_1 \simeq Q_2 \). Let \( G'_m = \{ Q \mid \text{Q : G' with derivation length} \leq m \} \) and let \( \simeq'_m = \{(Q_1, Q_2) \mid Q_1 \simeq Q_2 \text{ with derivation length} \leq m \} \). Then, for each \( m \geq 0 \) we define \( h'_m : G'_m \to G_{-x} \) such that the following conditions are satisfied:

\[
\text{ welche Bedingungen erfüllt werden.}
\]

84
1. \( h'_m \) maps variables of \( G'_m \) into variables of \( G_{-x} \)
2. for any \( Q : G'_m \), \( h'_m(Q) : G_{-x} \)
3. for any \( Q_1 \approx'_m Q_2 \), \( h'_m(Q_1) \approx_{-x} h'_m(Q_2) \)
4. \( h'_m \) is algebraic homomorphism on \( G'_m \)
5. for any \( y \in S \) in \( G' \) s.t. \( y : G'_m \) and \( h'_m(y) = y' \) with \( y' \in_{-x} S' \), we have \( S' \approx_{-x} h'_m(S) \)
6. \( h_{G'_m} = \text{id} \circ h'_m \)

**Base case:** \( m = 0 \). We define \( h'_0(R) = R \), for any name \( R \) in the schema, \( h'_0(\text{true}) = \text{true} \) and \( h'_0(\text{false}) = \text{false} \). It is easy to check that properties (1) - (6) are satisfied.

**Induction case:** \( m > 0 \). \( h'_m \) is defined to be the same as \( h'_{m-1} \) on any \( Q : G'_{m-1} \). For the rest, we do a case analysis on the last rule used in the derivation of \( Q : G'_m \) or \( Q_1 \approx'_m Q_2 \).

**Case (prj-add).** We have \( Q.A : G'_m \) as a result of \( Q : G'_{m-1} \). Then we define \( h'_m(Q.A) = h'_{m-1}(Q).A \). Since by inductive hypothesis we have \( h_{m-1}(Q) : G_{-x} \), it must be then the case, by (prj-add) that \( h_{m-1}(Q).A : G_{-x} \). Thus property (2) is ensured. Property (4) is ensured by definition of \( h'_m \) on \( Q.A \). We check property (6): \( h(Q.A) = h(Q).A \) since \( h \) is an algebraic homomorphism, \( h(Q).A = h'_{m-1}(Q).A \) by applying I.H. property (6), and \( h'_m(Q).A = h'_m(Q.A) \) by definition. Thus, \( h(Q.A) = h'_m(Q.A) \). The case (dom-add) is similar.

**Case (var-add).** We have \( y : G'_m \) as a result of \( y \in S' \) in \( G' \) and \( S' : G'_{m-1} \). Define \( h'_m(y) = h(y) \). We know that \( h(y) : G \) and \( h(y) \neq x \). By I.H. property (2), we must have \( h'_{m-1}(S') : G_{-x} \). On the other hand, since \( h(y) \) is a variable, there exists a node \( S \) in \( G \) such that \( h(y) \in S \). Moreover, \( h(S') \approx S \). Then \( h'_m(S') = h_{m-1}(S') \) and by I.H. property (6) \( h_{m-1}(S') = h(S') \). Therefore, it must be the case that \( h(S') : G_{-x} \). Putting them all together the following instance of (var-add-x) is applicable:

\[
\begin{align*}
  h(y) &\in S \text{ in } G, \quad h(y) \neq x, \quad S \approx h(S'), \quad h(S') : G_{-x} \\
  h(y) : G_{-x}
\end{align*}
\]

Thus, property (2) is verified. Properties (1) and (6) are automatically ensured by our definition. We check property (5). In \( G_{-x} \), \( h(y) \in_{-x} S \) such that \( S_0 \approx S \) and \( S_0 : G_{-x} \) and \( S_0 \) has the smallest derivation in \( G_{-x} \) among all similar \( S_0 \). Thus \( S_0 \approx h(S') \) by transitivity. Since both \( S_0 \) and \( h(S') \) are in \( G_{-x} \), by (eq-x) we have \( S_0 \approx_{-x} h(S') \) and therefore \( S_0 \approx_{-x} h'_m(S') \).

**Case (lookup-add).** The last rule applied has the form:

\[
\begin{align*}
  y &\approx_{m-1} y', \quad y' \in S \text{ in } G', \quad S \approx_{m-1} \text{ dom } Q \\
  Q[y] : G'_m
\end{align*}
\]

We define \( h'_m(Q[y]) = h'_m(Q)[h'_{m-1}(y)] \). Applying I.H. properties (3) and (4) we have \( h'_{m-1}(y) \approx_{-x} h'_{m-1}(y') \) and \( h'_{m-1}(S) \approx_{-x} h'_m(\text{dom } Q) = \text{dom } h'_{m-1}(Q) \). Applying I.H. properties (1) and (2), \( h'_{m-1}(y) \) and \( h'_{m-1}(y) \) are both variables in \( G_{-x} \). It must also be the case that \( h'_m(S) \approx_{-x} S_0 \) for some \( S_0 : G_{-x} \). By I.H. property (5) we have \( h'_{m-1}(S) \approx_{-x} S_0 \). Thus, by transitivity, \( S_0 \approx_{-x} \text{ dom } h'_{m-1}(Q) \). Then the following instance of (lookup-add-x) is applicable:
\[ h'_m(y) \cong \eta_x h'_{m-1}(y'), \quad h'_{m-1}(y') \in \pi_x S_0, \quad S_0 \cong \eta_x \text{dom } h'_{m-1} Q \]

\[ h'_m(Q) \{ h'_{m-1}(y) \} : G_{-x} \]

thus, proving property (2). The other properties are easily checked.

**Case (eq).** We have \( Q_1 \cong_{m} Q_2 \) as a consequence of \( Q_1 = Q_2 \) occurring in \( G' \), where \( G' \) is the path-conjunction of \( T', Q_1 : G'_{m-1} \) and \( Q_2 : G'_{m-1} \). Applying the I.H. property (2), we obtain \( h'_{m-1}(Q_1) : G_{-x} \) and \( h'_{m-1}(Q_2) : G_{-x} \). On the other hand, \( h \) is a homomorphism from \( T' \) into \( T \), thus it must be the case that \( h(Q_1) \cong h(Q_2) \). Applying I.H. property (6) we then have \( h'_{m-1}(Q_1) \cong h'_{m-1}(Q_2) \). Hence we set all the conditions needed for the applicability of \( (\text{eq-}x) \). We conclude that \( h'_{m-1}(Q_1) \cong_{-x} h'_{m-1}(Q_2) \). Finally, since \( Q_1 : G'_{m-1} \) and \( Q_2 : G'_{m-1} \), we have \( h'_m(Q_1) = h'_{m-1}(Q_1) \) and \( h'_m(Q_1) = h'_{m-1}(Q_2) \). Therefore, \( h'_m(Q_1) \cong_{-x} h'_m(Q_2) \).

The rest of the cases, i.e. (refl), (sym), (trans), (proj-cong), (eq-cong), (red-ext) and (lookup-cong), are simple: we use the inductive hypothesis and Lemma 4.3.3. We conclude the proof of the proposition as follows. Since any derivation in \( G_{-x} \) (and \( \cong_{-x} \)) has length bounded by some polynomial in the size of \( T \) it follows that \( G' = G_{m_0} \) and \( \cong' = \cong_{m_0} \) for some finite \( m_0 \). Thus it suffices to take \( h' = h'_{m_0} \). **End of Proof.**

It is easy to extend the above construction to work on queries and subqueries rather than tableaux and subtableaux. Given a query \( Q \) with tableau \( T \) and canonical instance \( (G, \cong) \), constructing the maximal subquery \( Q_{-x} \) of \( Q \) that removes variable \( x \) amounts to two steps:

1. construct the maximal canonical sub-instance of \((G, \cong)\) that removes \( x \), and construct as in Proposition 4.3.4 the tableau \( T_{-x} \). The from and where clause of \( Q_{-x} \) are then determined by \( T_{-x} \).

2. construct the select clause of \( Q_{-x} \). This consists in finding appropriate replacements for the paths that occur in the select clause of \( Q \) such that the replacements do not depend on \( x \) (i.e. they belong to \( G_{-x} \)) and moreover they are "equal" \( \cong \) with the original paths (under \( \cong \)). Notice that finding such replacements may fail, in which case there is no subquery of \( Q \) that removes \( x \).

Proposition 4.3.6 still holds when we work with queries and subqueries. Thus, any subquery of \( Q \) that removes \( x \) is necessarily a subquery of \( Q_{-x} \). This gives us a systematic way of enumerating subqueries of a given query \( Q \): for each variable \( x \) in \( Q \), construct maximal subquery \( Q_{-x} \), if it exists, output \( Q_{-x} \) and then repeat recursively with \( Q_{-x} \). In this way any subquery of \( Q \) that is maximal in the where clause will be enumerated. Of course, if we want, we could enumerate also for each maximal subquery all the subqueries that have the same from clause but less conditions in the where clause. However, we are focusing on eliminating scans rather than conditions.

In the optimizer we need to enumerate equivalent subqueries of \( Q \) rather than all subqueries. The next lemma will tell us that recursive enumeration of subqueries of \( Q \) in a top-down way, in which a recursive branch stops whenever a non-equivalent subquery is found, is complete.

**Lemma 4.3.7 (Pruning Lemma)** Let \( Q, Q_1 \) and \( Q'_1 \) be such that \( Q'_1 \) is a subquery of \( Q_1 \) and \( Q_1 \) is a subquery of \( Q \), and let \( D \) be an arbitrary set of dependencies.

1. If \( Q'_1 \equiv_D Q \) then \( Q_1 \equiv_D Q \).
2. If \( Q_1 \not\equiv_D Q \) then \( Q'_1 \not\equiv_D Q \).

**Proof.** It is enough to prove item 1) since item 2) is the counter-positive. We know that \( Q \subseteq Q_1 \subseteq Q'_1 \) (because of the subquery relationship between the three). We also know that \( Q \) and \( Q'_1 \) are equivalent (under \( D \)). Then it must be the case that \( Q_1 \) is also equivalent (under \( D \)) with \( Q \) and \( Q'_1 \). **End of Proof.**

The completeness of the top-down, decremental, backchase minimization algorithm (algorithm 2.2.1 of Chapter 2)
follows then immediately from the previous results. We also assume here that the chase always terminates while checking that the constraint $\delta = \text{cont}(Q_1, Q_2)$ required by a backchase step is implied by the schema. And this is the case usually (full EPCDs, dependencies that come only from views, etc). In Chapter 7 we will discuss different alternatives for implementing the backchase minimization such as bottom-up (which is complementary to the top-down approach presented here) or mixed bottom-up and top-down. We will also discuss other issues such as the use of dynamic programming and cost-based pruning, and how does our backchase algorithm relate to other optimization strategies.

**Theorem 4.3.8 (Complete Backchase)** The decremental backchase minimization of $Q$ enumerates all 1-1 minimal equivalent subqueries of $Q$ that are maximal in the `where` clause.

For the case of materialized PC views, by putting together the two main theorems of this chapter, we obtain the following corollary.

**Corollary 4.3.9 (Optimizing queries with materialized PC views)** Let $Q(\bar{R})$ be a PC query over logical schema with relations and dictionaries) $\bar{R}$, and let $\bar{V}$ be a set of PC view definitions characterized with EPCDs $\bar{d}_Q$ and $\bar{d}_P$. Then the C&B enumeration produces exactly all scan-minimal rewritings $Q'(\bar{R}, \bar{V})$ of $Q$ that are maximal in the `where` clause.

Finally, we conjecture that the above corollary can be strengthened to more general situations such as ones in which the physical schema has sources with limited access capabilities (modeled by us with materialized PC dictionaries). An even stronger variant of the above corollary would be one in which arbitrary semantic constraints are allowed in the logical schema. We leave this very interesting theoretical problem open. Nonetheless, in our practical approach, we are using the chase as a bounding search space for minimal plans even in such a general situation.
Chapter 5

Feasibility of the C&B Enumeration

In this chapter we describe implementation techniques used to make the C&B enumeration feasible and worthwhile, and experiments used to measure whether this goal is achieved [PDST00]. We do not assume, yet, any cost information. Cost issues, conceptually orthogonal to C&B enumeration, will be addressed in the next chapter. There we will see that, by mixing cost-based pruning with C&B enumeration, the performance of a C&B-based optimizer can be substantially improved. The current implementation of the C&B enumerator prototype is for the path-conjunctive (PC) queries and embedded path-conjunctive dependencies (EPCDs) of Chapter 3. Also, the stratification techniques discussed below are for the same PC language.

In this chapter we discuss the following:

Feasibility of the chase (section 5.1)
This is critical because the chase is heavily used both to build the universal plan and in order to check the validity of a constraint used in a backchase step. In section 5.4.2 we measure for several experimental configurations the time to obtain the universal plan as a function of the size of the query and the number of constraints. The results show that the cost of the efficiently implemented chase is negligible.

Feasibility of the backchase (section 5.2)
A full implementation of the backchase (FB) consists of backchasing with all the available constraints starting from the universal plan obtained by chasing also with all constraints. This implementation exposes the bottleneck of the approach: the exponential (in the size of the universal plan) number of subqueries explored in the back chase phase. A general analysis suggests using stratification heuristics: dividing the constraints in smaller groups and chasing/backchasing with each group successively. We examine two approaches to this:

- fragmenting the query and stratifying the constraints by relevance to each fragment (the On-line Query Fragmentation aka OQF technique, section 5.2.1);
- splitting the constraints independently of the query (the Off-line Constraint Stratification aka OCS technique, section 5.2.2)

In the important case of materialized views [Lev], we prove that OQF can be used without losing any plan that might have been found by the full implementation (theorem 5.2.3). To evaluate and compare FB, OCS and OQF strategies, we measure in Section 5.4.3, for various experimental configurations: (1) the number of plans generated, (2) the time spent per generated plan, and (3) the effect of fragment granularity. Finally, we address in section 5.4.4 the question whether the time spent in optimization is recovered by the gains in execution time.
5.1 Feasibility of the Chase

Each chase step of our algorithm includes searching for homomorphisms (see Chapter 3 for full definition) mapping a constraint into the query. Finding a homomorphism is NP-complete, but only in the size of the universal part of the constraint (always small in practice). However the basis of the exponent is the size of the query being chased which can become large during the chase. Our language is more complicated than a relational language because of dictionaries and nestings of sets. Therefore homomorphisms are more complicated than just simple mappings between goals of conjunctive queries, and checking that a mapping from a constraint into a query is indeed a homomorphism is not cheap (even though polynomial).

Here are several techniques that we use to speed-up and/or avoid unnecessary checks for homomorphisms:

- Congruence closure for fast checking if an equality is a consequence of the where clause of the query. To do this, we implemented the rules given in Chapter 3 for building the canonical instance of a tableau, using an extension of the algorithm of [NO80].
- Rule out because of redundancies homomorphisms previously used in the chase sequence\footnote{Without this, a check for non-redundancy must be done and this is also NP-complete (see Chapter 3).}
- Prune variable mappings that cannot become homomorphisms by reasoning early about equality. Instead of building the entire mapping and checking in one big step whether it is a homomorphism, this is done incrementally. The idea is the following: if $h$ is a mapping that is defined on variables $x$ and $y$ and $x.A = y.A$ occurs in the constraint then we check whether $h(x).A = h(y).A$ is implied by the where clause of the query. This works well in practice because the "good" homomorphisms are typically just a few among all possible mappings.
- Implementation of the chase as an inflationary procedure that evaluates the input constraints on the internal representation of the input query. The evaluation looks for homomorphisms from the universal part of constraints into the query “adds” (if not there already\footnote{this is translated as a check for trivial equivalence}) the result of each homomorphism applied to the existential part of the constraint to the internal query representation. The similarity between chase and query evaluation on a small database is another explanation of why chase is fast.

The experimental results about the chase shown in section 5.4.2 are very positive and show that even chasing queries consisting of more than 15 joins with more than 15 constraints is quite practical.

5.2 Feasibility of the Backchase

The following analysis of a simple but important case (just indexes) shows that a full implementation of the backchase can unnecessarily explore many subqueries.

**Example 5.2.1** Assume a chain query that joins $n$ relations $R_1(A,B), \ldots, R_n(A,B)$:

$$(Q) \quad \text{select } \text{struct}(A = r_1.A, B = r_n.B) \quad \text{from } R_1 r_1, \ldots, R_n r_n \quad \text{where } r_1.B = r_2.A \text{ and } \ldots \text{ and } r_{n-1}.B = r_n.A$$

and suppose that each of the relations has a primary index $I_i$ on $A$. Let $D = \{d_1, d_1', \ldots, d_n, d_n'\}$ be all the constraints defining the indexes (here $d_i$ and $d_i'$ are the constraints for $I_i$).
In principle, any of the $2^n$ plans obtained by either choosing the index $I_i$ or scanning $R_i$, for each $i$, is a plausible plan. One direct way to obtain all of them is to chase $Q$ with the entire set of constraints $D$ obtain the universal plan, of size $2n$, and then backchase it with $D$. If the backchase goes top-down from the universal plan, it inspects all possible subqueries of $2n-1$, ..., $n$ loops (it stops at $n$ because any subquery with less than $n$ loops cannot be equivalent to $Q$, in this case), for a total of: 

$$C_{2n-1}^n + \ldots + C_{2n}^n = 2^{2n-1} + \frac{1}{2}2^{2n-1} - 1.$$

Continuing the example, the same $2^n$ resulting plans can be obtained with the following different strategy, much closer to the one implemented by standard optimizers. For each $i$, handle the $i$th loop of $Q$ independently: chase then backchase the query fragment $Q_i$ of $Q$ that contains only $R_i$ with \{d_i, \ldots, \ldots, d_i\} to obtain two plans for $Q_i$, one using $R_i$ the other using the index $I_i$. At the end, assemble all plans generated for each fragment $Q_i$ in all possible combinations to produce the $2^n$ plans for $Q$.

The number of plans inspected by this “stratified” approach can be computed as follows. For each stage $i$ the universal plan for fragment $Q_i$ has only 2 loops (over $R_i$ and $I_i$) and therefore the number of plans explored by the subsequent backchase is 2. Thus the work to produce all the plans for all fragments is $2n$. The total work, including assembling the plans, is then $2n + 2^n$.

This analysis suggests that detecting classes of constraints that do not “interact”, grouping them accordingly and then stratifying the chase/backchase algorithm, such that only one group is considered at a time, can decrease exponentially the size of the search space explored.

The crucial intuition that explains the difference in efficiencies of the two approaches is the following. In the first strategy, for a given $i$, the universal plan contains both at the beginning of the backchase both $R_i$ and $I_i$. At some point during the backchase, since a plan containing both is not minimal, there will be a backchase step that eliminates $R_i$ and another backchase step, at the same level, that eliminates $I_i$ (see on the right). The minimization work that follows is exactly the same in both cases because it operates only on the rest of the relations. This duplication of work is avoided in the second strategy because each loop of $Q$ is handled exactly once. A solution that naturally comes to mind to avoid such situations is to use dynamic programming. Unfortunately, there is no straightforward way to do this and we leave the discussion of this issue in section 7. Instead, the next section gives a stratification algorithm that solves the problem for a restricted but common case.

### 5.2.1 On-line Query Fragmentation (OQF)

The main idea behind the OQF strategy is illustrated on the following example.

**Example 5.2.2** Consider a slightly more complicated version of example 1.23, shown in figure 5.1. The query graph is shaped like a chain of 2 stars, star $i$ having $R_i$ for its hub and $S_{ij}$ for its corners ($1 \leq i \leq 2$, $1 \leq j \leq 3$). The attributes selected in the output are the B attributes of all corners $S_{ij}$.

As suggested by the dotted polygonal lines, assume the existence of materialized views $V_m(K, B_1, B_2)$ ($1 \leq i \leq 2, 1 \leq l \leq 2$), where each $V_m$ joins the hub of star $i$ ($R_i$) with two of its corners ($S_{il}$ and $S_{l(i+1)}$). Each $V_m$ selects the B attributes of the corner relations it joins, as well as the K attribute of $R_i$.

90
If we apply the FB algorithm with all the constraints describing the views we obtain all possible plans in which views replace some parts of the original query. However it should be clear that \( V_{11} \) or \( V_{12} \) can only replace relations from the first star, thus not affecting any of the relations in the second star. If a plan \( P \) using \( V_{11} \) and/or \( V_{12} \) is obtained for the first star, such that it "reovers" the \( B \) attributes needed in the result of \( Q \), as well as the \( F \) attribute of \( R_{1} \) needed in the join with \( R_{2} \), then \( P \) can be joined back with the rest of the query to obtain a query equivalent to \( Q \). We say that \( V_{11} \) does not overlap with neither \( V_{21} \) nor \( V_{22} \). On the other hand this does not apply to \( V_{11} \) and \( V_{12} \), because the parts of the query that they cover overlap (and any further decomposition will in fact lose the plan that uses both \( V_{11} \) and \( V_{12} \)). \( Q \) can thus be decomposed into precisely two query fragments, one for each star, that can be optimized independently.

We give next the full description of the algorithm for query decomposition into fragments, Algorithm 5.2.1. The algorithm is based on computing the connected components of the interaction graph of constraints that map homomorphically into the query, and it is restricted to a class of physical access structures that we call skeletons, a class that includes indexes, materialized views, ASRs etc.

**Query Fragments.** Given a query \( Q \) as above, we define its closure as a query \( Q^* \) that has the same **select** and **from** clauses as \( Q \) while the **where** clause consists of all the equalities that occur in or are implied by the \( Q^* \)'s **where** clause. \( Q^* \) is computable from \( Q \) in PTIME and is equivalent to \( Q \). In fact \( Q^* \) without the **select** clause is nothing but an isomorphic representation, as a query, of the canonical instance introduced in Chapter 3.

Given a query \( Q \) and a subset \( S \) of its **from** clause bindings we define a query fragment \( Q' \) of \( Q \) induced by \( S \) as follows\(^3\):

1) The **from** clause consists of exactly the bindings in \( S \)

2) The **where** clause consists of all the conditions in the **where** clause of \( Q^* \) which mention only variables bound in \( S \), and

3) The **select** clause consists of all the paths \( P \) over \( S \) that occur in the **select** clause of \( Q \) or in an equality \( P = P' \) of \( Q^* \)'s **where** clause where \( P' \) depends on a binding that is not in \( S \). In the latter case, we call such \( P \) a link path of the fragment.

**Example 5.2.3** Recalling example 1.2.3 the query fragment of \( Q \) induced by \( S = \{ R_{1i} r_{1i}, S_{11} s_{11}, S_{12} s_{12} \} \) is the query:

\(^3\)This is not a formal definition. For a formal definition, we need to state the PC\(^0\) restriction of Chapter 6 under which query fragmentation is possible.
\[
\text{select } \text{struct}\left(B_{11} = s_{11} \cdot B, \ B_{12} = s_{12} \cdot B, \ L_{\left(r_1 F, r_2 K\right)} = r_1 F\right) \\
\text{from } \ B, s_{11}, s_{12} \\
\text{where } r_1 \cdot A_1 = s_{11} \cdot A_1 \text{ and } r_1 \cdot A_2 = s_{12} \cdot A_2
\]

Notice that \(r_1 F\) must occur in the \text{select} clause because it appears in an equality condition in \(Q\) with a path \((r_2 K)\) outside of the fragment (condition 3) above. Also \(s_{11} B\) and \(s_{12} B\) must occur in the \text{select} clause by condition 3 above. Essentially condition 3) will allow us to recover later a query from its query fragments by joining the fragments on the corresponding link paths and therefore we will be able to find a plan for the query by joining plans for the fragments. The label \(L_{\left(r_1 F, r_2 K\right)}\) for the link path \(r_1 F\) is generated so that it uniquely identifies the corresponding join condition.

\textbf{Skeletons.} While in general the chase/backchase algorithm can mix semantic with physical constraints, in the remainder of this section we describe a stratification algorithm that can be applied to a particular class of constraints which we call \textit{skeletons}. This class is sufficiently general to cover the usual physical access structures: indexes, materialized views, ASRs, GMAPs. As seen in section 2.3, each of these can be described by a pair of complementary inclusion constraints.

We define a skeleton as a pair of complementary constraints:

\[
d = \forall (\bar{x} \in \bar{R}) \left[ B_1(\bar{x}) \Rightarrow \exists (\bar{y} \in \bar{V}) \ B_2(\bar{x}, \bar{y}) \right] \\
d^- = \forall (\bar{y} \in \bar{V}) \exists (\bar{x} \in \bar{R}) \ B_1(\bar{x}) \text{ and } B_2(\bar{x}, \bar{y})
\]

such that all schema names occurring among \(\bar{V}\) belong to the physical schema, while all schema names occurring among \(\bar{R}\) belong to the logical schema. Note that while materialized views and primary indexes are described precisely by skeletons, secondary indexes require an additional non-emptiness constraint (see section 2.3).

\textbf{Algorithm 5.2.1 (Decomposition into Fragments.)} Given a query \(Q\) and a set of skeletons \(\mathcal{V}\):

\textit{Step 1:} Construct an interaction graph as follows: 1) there is a node labeled \((V, h)\) for every skeleton \(V = (d, d^-)\) in \(\mathcal{V}\) and homomorphism \(h\) from \(d\) into \(Q\); 2) there is an edge between nodes \((V_1, h_1)\) and \((V_2, h_2)\) whenever the intersection between the bindings of \(h(d_1)\) and \(h(d_2)\) is nonempty.

\textit{Step 2:} Compute the connected components \(\{C_1, \ldots, C_k\}\) of the interaction graph.

\textit{Step 3:} For each \(C_m = \{(V_1, h_1), \ldots, (V_n, h_n)\}\) (1 \(\leq m \leq k\) let \(S\) be the union of the sets of bindings in \(h_i(d_i)\), together with all their dependent bindings that are not in the image of any homomorphism, for all \(1 \leq i \leq n\). Compute \(F_m\) as the fragment of \(Q\) induced by \(S^4\).

\textit{Step 4:} The decomposition of \(Q\) into fragments consists of \(F_1, \ldots, F_k\) together with the fragment \(F_{k+1}\) induced by the set of bindings that are not covered by \(F_1, \ldots, F_k\).

The obtained fragments are disjoint, and \(Q\) can be reconstructed by joining them on the link paths. We are now ready to define the on-line query fragmentation strategy as follows:

\textbf{Algorithm 5.2.2 (OQF)} Given a query \(Q\) and a set \(\mathcal{V}\) of skeletons:

\textit{Step 1.} Decompose \(Q\) into query fragments \(\{F_1, \ldots, F_n\}\) based on \(\mathcal{V}\) using Algorithm 5.2.1.

\textit{Step 2.} For each fragment \(F_i\) find the set of all minimal plans by using the chase/backchase algorithm

---

\(^4\)The set expressions over which \(S\) range may need to be replaced by "equal" expressions from \(Q^*\)'s \text{where} clause, in order to avoid any dependent bindings across fragments. This is always possible, under the \textit{FC}\(^6\) restriction of Chapter 6.
**Step 3.** A plan for $Q$ is the "cartesian product" of sets of plans for fragments (cost-based refinement: the best plan for $Q$ is the join of the best plans for each individual fragment)

**Theorem 5.2.3** Let $Q$ be a minimal query (with non-redundant scans). Then, for a skeleton schema and a minimal (under trivial constraints) input query, OQF produces the same minimal query plans for $Q$ as the full backchase (FB) algorithm.

Another strength of OQF is that in the limit case when the physical schema contains skeletons involving only one logical schema name (obvious examples are primary/secondary indexes) it degenerates smoothly into a backchase algorithm that operates on each loop of the query individually in order to find the access method for the particular loop. One of the purposes of the experimental configuration $EC1$ is to demonstrate that OQF performs well in a typical relational setting. However, OQF can be used in more complex situations, like for example in answering/optimizing queries with materialized views. While in the worst case when the views are strongly overlapping, the fragmentation algorithm may result in one fragment (the query itself), in practice we expect to achieve reasonably good decompositions in fragments. Scalability of OQF in a setting with views that exhibits a reasonable amount of non-interaction between views is demonstrated by using the experimental configuration $EC2$.

### 5.2.2 Off-line Constraint Stratification (OCS)

One disadvantage of OQF is that it needs to find the fragments of a query $Q$. While this has about the same complexity as chasing $Q$ \(^5\) (and we have argued that chase itself is not a problem) in practice there may be situations in which interaction between constraints can be estimated in a pre-processing phase that examines only the constraints in the schema. The result of this phase is a partitioning of constraints into disjoint sets such that only the constraints in one set are used at one time by the backchase algorithm. As opposed to query fragmentation this method tries to isolate the independent optimizations that may affect a query by stratifying the constraints without fragmenting the query. During the optimization process the entire query is pipelined through stages in which the chase/backchase algorithm uses only the constraints in one set. At each stage different parts of the query are affected.

Similarly to OQF, this algorithm finds first the connected components in a constraint interaction graph which however is constructed in a different, query-independent way. The result of this stage is a partitioning of the set of initial constraints into disjoint sets of constraints (strata). The full details of the algorithm for stratification of constraints, algorithm 5.2.4, are given next.

**Algorithm 5.2.4 (Stratification of Constraints.)** Given a schema with constraints, do:

1. **Step 1:** Construct an interaction graph as follows:
   1) there is a node labeled $c$ for every constraint $c$ in the schema.
   2) there is an edge between nodes $c_1$ and $c_2$ whenever there is a homomorphism $c_1$ into the tableau of $c_2$, or vice versa. The tableau $T(c)$ of a constraint $c = \forall (\vec{u} \in \vec{U}) \ B_1(\vec{u}) \Rightarrow \exists (\vec{e} \in E) \ B_2(\vec{u}, \vec{e})$ is obtained by putting together both universally and existentially quantified variables and by taking the conjunction of all conditions: $T(c) = \forall (\vec{u} \in \vec{U}) \forall (\vec{e} \in E) \ B_1(\vec{u}) \land B_2(\vec{u}, \vec{e})$.

2. **Step 2:** Compute the connected components $\{C_1, \ldots, C_k\}$ of the interaction graph. Each $C_i$ corresponds to a constraint stratum.

\(^5\)The chase also needs to find all homomorphisms between constraints and the query.
The above algorithm makes optimistic assumptions about the non-interaction of constraints: even though there may not be any homomorphism between the constraints, depending on the query they might still interact by mapping to overlapping subqueries at run time. Therefore, the OCS strategy is subsumed by the on-line query fragmentation but it has the advantage of being done before query optimization.

Based on the above partitioning, the following refinement of the C&B strategy, the off-line constraint stratification backchase (OCS) uses only constraints from one stratum at a time.

**Algorithm 5.2.5 (OCS)** Given a query Q and set of constraints C:

1. **Step 1.** Partition C into disjoint sets of constraints \(\{S_i\}_{1 \leq i \leq k}\) by using algorithm 5.2.4.
2. **Step 2.** Let \(P_0 = \{Q\}\).
3. **Step 3.** For every \(1 \leq i \leq k\), let \(P_i\) be the union of the sets of queries obtained by chase/backchasing each element of \(P_{i-1}\) with the constraints in \(S_i\).
4. **Step 4.** Output \(P_k\) as the set of plans.

![Figure 5.2: Inverse Relationships](image)

**Example 5.2.4** To illustrate the algorithm, we consider 3 classes (see figure 5.2 with \(n = 3\)) described by dictionaries \(M_1, M_2, M_3\). Each \(M_i\) includes a set-valued attributed \(N\) ("next") and a set-valued attribute \(P\) ("previous"). For each \(i = 1, 2\), there exists a many-many inverse relationship between \(M_i\) and \(M_{i+1}\) that goes from \(M_i\) into \(M_{i+1}\) by following the \(N\) references and comes back from \(M_{i+1}\) into \(M_i\) by following the \(P\) references. The inverse is described by the following constraints:

\[
(INV_{1N}) \forall (k \in \text{dom} M_i) \forall (o \in M_i[k], N) \exists (k' \in \text{dom} M_{i+1}) \exists (d' \in M_{i+1}[k'], P) k' = o \text{ and } d' = k \\
(INV_{1P}) \forall (k' \in \text{dom} M_{i+1}) \forall (o' \in M_{i+1}[k'], P) \exists (k \in \text{dom} M_i) \exists (o \in M_i[k], N) k' = o \text{ and } d' = k
\]

By running algorithm 5.2.4 we obtain the following stratification of constraints into two strata: \(\{INV_{1N}, INV_{1P}\}\) and \(\{INV_{2N}, INV_{2P}\}\). Suppose now that the incoming query \(Q\) is a typical navigation following the \(N\) references from class \(M_1\) to class \(M_2\) and from there to \(M_3\):

```
select struct(F = k_1, L = o_2)
from dom M_1 k_1, M_1[k_1].N o_1, dom M_2 k_2, M_2[k_2].N o_2
where o_1 = k_2
```

By chase/backchasing \(Q\) with the constraints of the first stratum, \(\{INV_{1N}, INV_{1P}\}\), we obtain, in addition to \(Q\), the query \(Q_1\) shown below in which the sense of navigation from \(M_1\) to \(M_2\) following the \(N\) attribute is "flipped" to a navigation in the opposite sense: from \(M_2\) to \(M_1\) along the \(P\) attribute.
select struct(F = o_1, L = o_2)
from dom \mathcal{M}_2 k_2, \mathcal{M}_2[k_2].P o_1, \mathcal{M}_2[k_2].N o_2

In the stage corresponding to stratum 2, we chase/backchase \{Q, Q_1\} with \{INV_2, INV_2P\}, this time flipping in each query the sense of navigation from \mathcal{M}_2 to \mathcal{M}_3 via \mathcal{N} to a navigation from \mathcal{M}_3 to \mathcal{M}_2 via \mathcal{P}. The result of this stage consists of four queries: the original \(Q\) and \(Q_1\) (obtained by chasing and then backchasing with the same constraint), and the additional \(Q_2\) (obtained from \(Q\)) and \(Q_3\) (obtained from \(Q_1\) and shown below).

\[
(Q_3) \quad \text{select struct}(F = o_1, L = k_3)
\]

\[
\text{from dom} \mathcal{M}_3 k_3, \mathcal{M}_3[k_3].P o_3, \text{dom} \mathcal{M}_2 k_2, \mathcal{M}_2[k_2].P o_1
\]

where \(o_3 = k_2\)

The OCS strategy does not miss any plans for this example (see also the experimental results for OCS with \textbf{EC2}), but in general it is just a heuristic. Our algorithm 5.2.4 makes optimistic assumptions about the non-interaction of constraints, which depending on the input query, may turn out to be false, therefore there is no completeness guarantee. \textbf{EC2} is an example of such a case and we leave open the problem of finding a more general algorithm for stratification of constraints.

5.3 The Architecture of the Prototype

The implementation of the \texttt{C\&B} enumeration system has been done in Java (25,000 lines of code). The architecture of the system is shown in figure 5.3. The arrowed lines show the main flow of a query being optimized, constraints from the schema, and resulting plans. The thick lines show the interaction between modules. The main module is the \texttt{plan generator} which, when given a query, performs the two basic phases of the \texttt{C\&B}: chase and backchase. The backchase is implemented top-down by removing one binding at a time and minimizing recursively the subqueries obtained if they are equivalent. Checking for equivalence is performed by verifying that the dependency equivalent to one of the containments is implied by the input constraints\(^6\). The module that does the check, \textit{dependency implication} shown in the figure as \(D \Rightarrow d\), uses the chase (and therefore the chase module) and the \textit{triviality check} module.

The most salient features of the implementation are summarized below:

- queries and constraints are compiled into a (same!) internal congruence closure based canonical database representation (shown in the figure as \(DB(Q)\) for a query \(Q\), respectively \(DB(d)\) for a constraint \(D\)) that allows for fast reasoning about equality.
- compiling a query \(Q\) into the canonical database is implemented itself as a chase step on an empty canonical database with one constraint having no universal but one existential part isomorphic to \(Q\)'s \texttt{from} and \texttt{where} clauses put together. Hence, the query compiler, constraint compiler and the chase modules are basically one module.
- in addition to internal, a language for describing queries and constraints that is as user friendly as \texttt{OQL}.
- a script language that can control the constraints that are fed into the chase/backchase modules. This is how we implemented the off-line stratification strategy and various other heuristics.

\(^6\)The other containment is always true.
5.4 Experiments

In this section we present our experimental configuration and report the results for the chase and the backchase. Finally, we address in section 5.4.4 the question whether the time spent in optimization is gained back at execution time.

5.4.1 Experimental configurations

We consider for our experiments three different settings that exhibit the mix of physical structures and semantic constraints that we want to take advantage of in our optimization approach. We believe that the scenarios that we consider are relevant for many practical situations.

**Experimental Configuration EC1:**
The first setting is used to demonstrate the use of the C&B enumerator in a relational setting with indexes. This is a simple but frequent practical case and therefore we consider it as a baseline for which we want to demonstrate that our optimizer performs quite well under various strategies.

The schema includes $n$ relations, each relation $R_i$ having a key attribute $K$ on which there is a primary index $PI_i$, a foreign key attribute $N_i$, and some additional attributes. The first $j$ of the relations have secondary indexes $SI_i$ on $N_i$, thus the total number of indexes in the physical schema is $m = n + j$. As in Example 5.2.1 we consider chain queries (see figure 5.4) that join $R_i$ with $R_{i+1}$ on attributes $N_i$ and $K_i$ respectively. The attributes in the select clause are not very important here and we return all the key attributes of the relations involved. The two scaling parameters for our experiments are $n$ and $m$.

**Experimental Configuration EC2:**
The second setting is designed to illustrate experimental results in the presence of materialized views and key constraints that the optimizer can take advantage of in finding good plans.

We consider a generalization of the chain of stars query of examples 1, 2, 3 and 5.2.2 (see figure 5.1) in which we have $i$ stars with $j$ corner relations, $S_{i1}, \ldots, S_{ij}$, that are joined with the hub of the star $R_i$. The query returns
all the \( B \) attributes of the corner relations. For each we assume \( v \leq j - 1 \) materialized views \( v_{i1}, \ldots, v_{iv} \) each covering, as in the previous examples, three relations. We assume that the attribute \( K \) of each \( R_i \) is a primary key. The scaling parameters that are \( i, j \) and \( v \).

**Experimental Configuration EC3:**

The third experimental setting is an object-oriented configuration with classes obeying many-to-many inverse relationship constraints. We use it to show how we can mix semantic optimization based on the inverse constraints to discover plans that use access support relations (ASRs). The query that we consider is not directly "mappable" into the existing ASRs, and the first optimization phase of our experiments (semantic optimization) enables rewriting the query into equivalent queries that can map into the ASRs. The mapping into ASRs is done in the second phase (physical optimization).

We generalize here the scenario considered in example 5.2.4 by considering \( n \) classes with inverse relationships. The queries \( Q \) (see Figure 3.2) that we consider are long navigation queries across the entire database following the \( N \) references from class \( M_1 \) to class \( M_n \). In addition we consider as part of the physical schema access support relations (ASRs) that are materialized navigation joins across three classes going in the backwards direction (i.e. following the \( P \) references). Each ASR is a binary table storing oids from the beginning of the navigation path and the corresponding oids from the end of the navigation path. Plans obtained after the inverse optimization phase are rewritten in the second phase into plans that replace a navigation chain of size 2 with one navigation chain of size 1 that uses an ASR (thus being likely better plans).

The parameters of the configuration are the number of classes, \( n \), and the number of ASRs, \( m \).

**Experimental settings**

All the experiments have been realized on a dedicated commodity workstation (Pentium III, Linux Red Hat 6.0, 128MB of RAM, 6.4GB of hard-drive). The optimization algorithm (chase, backchase) is fully implemented in Java and is run using IBM runtime environment for Linux (alpha version 1.1.8). The database management system used to execute queries is IBM DB2 version 6.1.0 for Linux (out-of-the-box configuration). For EC2, materialized views have been produced by creating and populating tables. All times measured are elapsed times, obtained using the Unix shell time command.

### 5.4.2 Feasibility of the Chase: Experiments

We measured the complexity of the chase in all our experimental configurations varying both the size of the input query and the number of constraints in the schema. We did not consider any stratification of the query or constraints because the numbers for the full chase are fine.

In **EC1** (figure 5.5, left) the constraints used in the chase are the ones describing the primary (2 constraints/index) and/or secondary (3 constraints/index) indexes. For example, chasing with 10 indexes, therefore 20+ constraints, takes under 1s. For **EC2** (figure 5.5, middle) the variable is the number of relations in the from clause, giving a measure of the query size. The number of constraints comes from the number of views (2 constraints/view) and
the number of key constraints (1 constraint/star hub). For EC3 (figure 5.5, right) the variable is the number of classes $C$ (measuring both the size of the schema and that of the queries we use). The chase is done with the inverse relationship constraints (2 constraints/relationship, $2 \times (C - 1)$ total) and with the ASR constraints (2 constraints/ASR, $[(C - 1)/2]$ total). For example, chasing with 8 classes, therefore 20 constraints, takes 8s. Overall, we conclude that the normalized chase time grows significantly with the size of the query and the number of constraints. In comparison, numbers for the chase time are much smaller than those of the backchase.

![Figure 5.5: Effect on chase time of increasing schema and query parameters](image)

### 5.4.3 Feasibility of the Backchase: Experiments

To evaluate and compare the two stratification strategies (OQF and OCS) and the full approach (FB) we measure, in each of the experimental configurations, the following:

- **The number of plans generated** measures the completeness with respect to FB. We found that OQF was complete for all experimental configurations considered, beyond what theorem 5.2.3 guarantees. As expected, both OQF and FB outperformed OCS.

- **The time spent per generated plan** allows for a fair comparison between all three strategies. We measured the time per plan as a function of the query size and number of constraints. Moreover, we studied the scale-up for each strategy by pushing the values of the parameters to the point at which the strategy became ineffective. We found that OQF performed much better than OCS which in turn outperformed FB.

  **Remark.** Another possible measure would be the efficiency of the search (the useful work performed during the backchase) measured as the ratio between the number of generated plans and the number of explored subqueries. We expect that OQF would greatly outperform FB here but OCS would be difficult to compare because it does not generate the same number of plans. However, a pleasant experimental observation and an indicator of the robustness of the implementation is that the time per subquery explored stays relatively constant for all three strategies, in all experimental configurations, for various query sizes and various numbers of constraints. This means that the efficiency of the search can in fact be estimated as the inverse of the time per generated plan, mentioned above.

- **The effect of fragment granularity on optimization time** is measured by keeping the query size constant and varying the number of strata in which the constraints are divided. This evaluates the benefits of finding a decomposition of the query into minimal fragments. The OQF strategy performs best by achieving the minimal decomposition that doesn’t loose plans. The results also show that OCS is a trade-offs giving up completeness for optimization time.
Number of generated plans

This experiment compares for completeness the full backchase algorithm with our two refinements: OQF (section 5.2.1) and OCS (section 5.2.2), and measures the number of generated plans, as a function of the size of the query and the number of constraints.

We ran the experiment for all three configurations. For EC1, we varied the number $r$ of relations involved in the join (which equals the number of primary indexes) and the number $s$ of secondary indexes at our disposal. For EC2, we varied the query size by increasing the number $s$ of stars per query and the number $c$ of corners per star. The number of key constraints was fixed to the number of stars (one constraint for every star hub). We varied the overall number of constraints by varying the number $v$ of views applicable per star. The query size is given by $s(c + 1)$, the number of constraints by $s(1 + 2v)$ (two constraints per view). For EC3, we varied the query size by increasing the number $n$ of classes traversed during the navigation. The number of inverse constraints necessarily varied linearly with the size of the query.

The three strategies yielded the same number of generated plans in configurations EC1 and EC3. The table below shows the results for configuration EC2:

<table>
<thead>
<tr>
<th>$s$</th>
<th>$c$</th>
<th>$v$</th>
<th>FB</th>
<th>OQF</th>
<th>OCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>4</td>
<td>13</td>
<td>13</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

As expected, the complete FB strategy outperforms CQF, which in turn performs much better than OCS. Note that in the common case of index introduction, all three strategies generate all the plans. The same holds for the less conventional EC3 scenario. However, the time spent for generating the plans differs spectacularly among the three techniques, as shown by the next experiment.

Optimization time spent per generated plan

This experiment compares the three backchase strategies by optimization time. Because not all strategies are complete and hence output different numbers of plans, we ensured fairness of the comparison by normalizing the optimization time which was divided by the number of generated plans. This normalized measure is called time per plan (tpp) and was measured as a function of the size of the query and the number of constraints.

We ran the experiment for all three configurations, varying the parameters as described in the previous experiment and the results are shown in figures 5.6 and 5.7.

The purpose of running the experiment in configuration EC1 was to show that for the trivial, yet common case of index introduction, our algorithm's performance is comparable to that of standard relational optimizers. Indeed, figure 5.6 shows the results obtained for three query sizes: 3, 4 and 5. By varying the number of secondary indexes for each query size, we observed an exponential behavior of the time per plan for the FB strategy, but a negligible time per plan for both OQF and OCS.

For configuration EC3 it turns out that OQF degenerates into FB because the images of the inverse constraints
overlap. We show a comparison of FB (= OQF) and OCS. The missing FB bars for a number of traversed classes larger than 4 indicate that the total optimization time needed by FB exceeded our timeout threshold of 2 minutes and the experiment was interrupted. OCS outperforms the other two strategies on this example because each pair of inverse constraints ends up in its own stratum. This stratification results in a linear time per plan (each stratum flips one join direction).

The most challenging configuration is EC2, dealing with large queries and numerous constraints. For example, the point corresponding to 4 stars of 4 corners and 2 views each corresponds to a query of 19 joins to which 20 constraints apply! Figure 5.7 divides the points into 3 groups, each group corresponding to the same number of views per star. This value determines the size of the query fragments and constraint strata for OQF, respectively OCS, and turns out to be the most important factor influencing the complexity. Again, missing data corresponds to timeout for our experiments.

While all strategies exhibit exponential time per plan, OCS is fastest, while FB cannot keep pace with the other two strategies.

The effect of stratification on the optimization time

This experiment was run in configurations EC2 and EC3 by keeping the query size constant and varying the number of strata in which the constraints are divided. For EC3 we considered two queries, one navigating over 5 classes and one over 6 classes, with 8, respectively 10 applicable constraints. The query considered in configuration EC2 joins three stars of 3 corners each, with one view applicable per star (for a total of 9 constraints).

The results are shown in figure 5.8. We observe an exponential reduction of the optimization time with the reduction in strata size. Note that the point of stratum size 1 corresponds for EC3 to OCS. These results corroborate the analytical analysis of example 5.2.1: by decomposing a fixed query into fragments of decreasing size in a completeness-preserving way, we observe an exponential reduction of the optimization time. This result validates the OQF strategy which achieves the minimal decomposition that doesn't loose plans. Moreover, it suggests that by decomposing beyond the threshold of preserving completeness, heuristics such as OCS are trade-offs giving up completeness for optimization time.

---

7The inverse between \( M_i \) and \( M_{i+1} \) with that between \( M_{i+1} \) and \( M_{i+2} \) overlap on a binding involving \( \text{dom} \) \( M_{i+1} \)

8Note though that we only measure time per plan here, not the quality of the generated plans (OCS systematically misses the best plan, which uses all the views). For a comparison of the cost versus benefit in this configuration, see experiment 5.4.4

9Note that FB and OQF are obtained as the extremes of this spectrum of decompositions.
Comparison of backchase techniques in EC2

---

Figure 5.7: Comparison of FB, OQF, OCS for EC2

Effect of stratification granularity [EC2, EC3]

---

Figure 5.8: Effect of stratification on the optimization time
5.4.4 The Benefit of Optimization

In this section, we measure the total query processing time: optimization time plus execution time. Since we didn’t implement our own query execution engine, we made use of DB2 as follows. We use EC2 with materialized views and key constraints, as presented at the beginning of section 5.4. Queries are optimized using the OQF strategy and fed into DB2 for comparing their processing times.

<table>
<thead>
<tr>
<th>Plan #</th>
<th>Execution time [s]</th>
<th>Views used</th>
<th>Corner relations used</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.54</td>
<td>$V_{1,1}$, $V_{2,1}$, $V_{3,1}$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>66.39</td>
<td>$V_{1,1}$, $V_{2,1}$</td>
<td>$S_{1,1}$, $S_{1,2}$</td>
</tr>
<tr>
<td>3</td>
<td>33.13</td>
<td>$V_{1,1}$, $V_{2,1}$</td>
<td>$S_{2,1}$, $S_{2,2}$</td>
</tr>
<tr>
<td>4</td>
<td>143.75</td>
<td>$V_{2,1}$</td>
<td>$S_{2,1}$, $S_{2,2}$, $S_{3,1}$, $S_{3,2}$</td>
</tr>
<tr>
<td>5</td>
<td>105.82</td>
<td>$V_{2,1}$, $V_{3,1}$</td>
<td>$S_{1,1}$, $S_{1,2}$</td>
</tr>
<tr>
<td>6</td>
<td>61.45</td>
<td>$V_{2,1}$</td>
<td>$S_{1,1}$, $S_{1,2}$, $S_{3,1}$, $S_{3,2}$</td>
</tr>
<tr>
<td>7</td>
<td>43.54</td>
<td>$V_{3,1}$</td>
<td>$S_{1,1}$, $S_{1,2}$, $S_{3,1}$, $S_{3,2}$</td>
</tr>
<tr>
<td>8</td>
<td>332.90</td>
<td>$S_{1,1}$, $S_{1,2}$, $S_{2,1}$, $S_{2,2}$, $S_{3,1}$, $S_{3,2}$</td>
<td>[*] original query</td>
</tr>
</tbody>
</table>

# Stars: 3, # Corner relations per star: 2, # Views per star: 1, 8 plans generated. Time to generate all plans: 7.6s

Figure 5.9: A detail of the plans generated for one instance of EC2

**Parameters measured** We denote by Opt $T$ the time taken by C&B to optimize the query; by $ExT$ the execution time of the query given to DB2 in its original form (no C&B optimization); and by $ExT_{Best}$, the DB2 execution time of the best plan generated by the C&B optimization.

We have $ExT_{Best} \leq ExT$ since the original query is always part of the generated plans.

We assume that the cost of picking the best plan among those generated by the algorithm is negligible.

**Performance indices** We define and display in figure 5.10, for increasing complexity of the experimental parameters, the following performance indices:

- **Redux** represents the time reduction resulting from our optimization with respect to $ExT$ assuming that no heuristic is used to stop the optimization as soon as reasonable.

- **ReduxFirst** represents the time reduction resulting from our optimization with respect to $ExT$ assuming that a heuristic is used to return the best plan first and stop the optimization.

Our current implementation of OQF is able to return the best plan first for all the experiments presented in this paper. The implementation of OCS has the same property (see section 7 for a discussion).

$$Redux = \frac{ExT - (ExT_{Best} + OptT)}{ExT} \quad \text{and} \quad ReduxFirst = \frac{ExT - (ExT_{Best} + OptT)}{ExT}$$

Negative values of Redux are not displayed.

**Dataset used** These performance indices correspond to experiments conducted on a small size database with the following characteristics:

| $|R_i|$ | $|S_{i,j}|$ | $\sigma(R_i \bowtie S_{i,j})$ | $\sigma(R_i \bowtie R_{i+1})$ |
|-------|------------|-------------------------------|-------------------------------|
| 5,000 tuples | 5,000 tuples | 4%                           | 2%                           |

On a larger database, the benefits of C&B should be even more important.
We also give the details of all the plans generated (8 plans in this case) and their ExTBest values for one instance of the configuration parameters in figure 5.9. For each generated plan, we present the views used and the star corner relations that these views and the star hub relations are joined with.

Our current implementation of the C&B technique algorithm is not tuned for maximum performance, thus skewing the results against us. Clearly using C or C++ and embedding the C&B as a built-in optimization (e.g. inside DB2) would lead to even better performance. We obtain excellent results nevertheless, proving that the time spent in optimization is well worth the gained execution time.

Even without the heuristic of stopping the optimization after the first plan, the C&B posts significant time reductions (40% to 90%), up to optimizing chain of stars queries as complex as having $2 \times (4 + 1) = 10$ relations with 9 joins, using $2 \times 2 = 4$ views and $2 \times 4 + 2 = 10$ constraints (parameter $[2, 4, 2]$ in figure 5.10). The practicality range is extended even further when using the “best plan first” heuristic, with reductions of 60% to 95%, up to optimizing queries with $3 \times (4 + 1) = 15$ relations with 14 joins, using $2 \times 3 = 6$ views and $2 \times 6 + 3 = 15$ constraints (parameter $[3, 4, 2]$ in figure 5.10).

Note that these numbers correspond to one run of the query. The benefit is much higher when the cost of optimization is amortized over multiple runs (as is often the case, e.g. OLAP environments).
Chapter 6

Mixing the Chase and Backchase with Cost-Based Optimization

So far, we have been concerned with using the C&B framework as an enumeration procedure: given an input query $Q_0$ and a set of constraints describing the physical access structures (indexes, materialized views, etc.) as well as the semantic knowledge (referential integrity constraints, key constraints, etc.), we are able to produce a set of minimal queries (candidate plans) $C = \{Q_1, \ldots, Q_n\}$ that are equivalent to $Q_0$. In the absence of a cost model, any of these minimal queries can be the optimal one: they are presumably better\(^1\) than the non-minimal queries (because they have less scans) but they are incomparable among themselves. In order to be able to efficiently compare candidate plans, from a cost perspective, during the backchase exploration, we need to understand several important issues about cost evaluation. This chapter addresses the following issues:

- **Mapping physical queries into physical plans.** A candidate plan obtained with the chase and the backchase specifies the physical access structures that are to be used, but still it does not specify how are they going to be used. In general, we will make a clear distinction between physical queries and physical plans. By physical query (as often used in the previous chapters) we mean a PC query that uses elements of the physical schema. The candidate plans coming out of the C&B enumeration are such physical queries. By physical plan we will understand (in this chapter) a more detailed representation of a physical query that can be directly executed by a query engine. There may be many physical plans for one physical query, and mapping a physical query into physical plans requires specification of the following:
  
  - **scan order:** the order in which the physical access structures are to be accessed. Here we have the following difficulty that we need to handle:
    In the relational case, the scan order (or the join order) is simply the order in which the relations involved in the query are to be accessed, and any order can yield a valid execution. In our context, however, not all scan orders are possible. For example, a scan $S_i x_i$ of a query may depend on another scan $S_j x_j$ if the set $S_i$ depends on the variable $x_j$. Thus any scan order that implements such a query must satisfy the constraint that $S_j$ is scanned first, and then $S_i$ is scanned. We will call such a scan order viable. This is a typical scenario for OO queries and such a dependency between scans is called a dependent join. Similar dependent joins occur when accessing secondary indexes, as well. Thus, we need to be able to consider in a systematic way all viable scan orders.
  
  - **method for dictionary access:** there are two different ways in which we can access a dictionary: (1) via a scan over the domain of the dictionary together with a lookup to retrieve the entry for each key in

\(^1\)See later discussion in the chapter about the cost monotonicity assumption.
the domain, or (2) via an individual \textbf{lookup} in the dictionary when the key is known to be equal to some value (possibly given by other scans).

- placement of selections and projections: depending on the scan order, selections and projections can have different placement within a plan. In general, a good heuristic is to apply them as early as possible.

All these factors can highly influence the cost of execution, and they need to be completely specified by a physical plan. In section 6.1 we introduce physical plans and describe the possible physical plans, \( \mathcal{P}_i = \{ P_1, \ldots, P_k \} \), associated to a physical query \( Q_i \). We show in sections 6.1.2 and 6.1.4 that by using fragmentation techniques we are able to express any physical plan for a given query as a plan for the natural join of its fragments (as opposed to relations in the traditional approach). All dependent joins are pushed automatically inside fragments. The space of all viable scan orders becomes then the same as the space of all possible orderings between fragments. Also, selections and projections are pushed down to fragments, as it is done in the relational approach.

- \textbf{Cost estimation of a physical plan.} Quantitative information is needed in order to evaluate the execution cost of a physical plan. Such information consists of: sizes of physical schema elements, and their selectivities (i.e. if we fix the value of some attribute, how many elements are still there in a given physical structure). While in the relational case, this information is quite straightforward to describe and use, our data model (and query language) poses some important problems due to the existence of arbitrarily nested sets (and queries). Section 6.2 deals with these issues and gives a language for describing cost information in the presence of nested sets and dictionaries. The cost information can then be used to estimate properties (cardinality, size, selectivity) of fragments (rather than relations). Then, since a physical plan is a join of fragments, we are able to estimate the cost/cardinality of any physical plan.

The cost model has as a starting point a relational cost model, simplified as follows: (1) only I/O cost is considered, and (2) only block nested-loops joins and index-based joins are considered as join methods.

We believe that additional extensions can be easily incorporated: we did not include them, because they do not directly affect our method and goal of measuring the impact of cost information on the backchase.

- \textbf{Cost estimation of a physical query.} Here, the problem is to \textit{efficiently enumerate} the space \( \mathcal{P}_i \) of possible physical plans for a physical query \( Q_i \), in order to find its best physical plan. This is the focus of section 6.3 where we show that selecting the best physical plan for a physical query can be done by means of a global dynamic programming algorithm in the spirit of System R. The algorithm handles fragments instead of relations, and we call it global because it reuses its already computed plans over multiple calls. This is something specific to our approach: the backchase exploration requires a call to the dynamic programming algorithm for each subquery explored, and since there may be partial plans that are common among several subqueries, we want to reuse their computation.

- \textbf{Efficient mixing of the C&B phase with cost estimation of the explored physical queries.}

The goal here is to take advantage of the cost information in order to to avoid a full enumeration of all candidate plans, and in general to avoid exploration of subqueries of the universal plan that have no chance to be the optimal one. We give in section 6.4 a \textbf{bottom-up} backchase algorithm that can be efficiently combined with cost-based pruning and dynamic programming. Cost-based pruning is employed here in a very effective way: when the cost of a subquery explored by the backchase is found to be larger than the best cost found so far, not only the respective subquery is pruned but also all of its superqueries (because their costs will be larger). The validity of this pruning strategy requires a monotonicity of cost assumption\textsuperscript{2} that we discuss in section 6.2.4. We also show in section 6.4 that a slight modification of the above pruning strategy is needed in the presence of dictionaries. This modification allows us to find physical plans that are minimal (no redundant scans) even though their translations as PC queries are not minimal. Such plans include the plan discussed in the example 1.2.4 of Chapter 1.

\textsuperscript{2}This is the same assumption that allows us to ignore non-minimal queries in the backchase phase.
It is important to remark that cost-based pruning can only be used in conjunction with the bottom-up backchase. For the top-down version of the backchase algorithm, no pruning can be done because the cost of subqueries decreases with elimination of scans. Thus better and better solutions can be found as the search progresses, and there is no way to stop early the search.

The end result is a full-fledged optimizer for PC queries, and the experimental results in section 6.5 show that its performance improves significantly over the performance of pure C&B enumeration (for large queries with as much as 20 times). We also give comparisons with the stratified techniques (OQF) and with the classical dynamic programming-only\(^3\) technique used in many relational optimizers. Section 6.6 summarizes the salient features of the various techniques that we have studied (cost/no cost, stratified/not stratified).

**Simplifying Restrictions:**

1. **The PC\(^0\) query language.** The fragmentation technique that we use for mapping physical queries into physical plans requires, for a given physical query \(Q\), that \(Q\) is a restricted form of a PC query. The restricted PC language, called PC\(^0\)(section 6.1.1), is one in which dictionaries cannot appear nested within sets, records or other dictionaries. In other words, dictionaries can only occur as schema names. In the presence of nested dictionaries, the problem of decomposing a query into commutable fragments, used by dynamic programming-based algorithms, appears to be quite hard, and we leave it for future work. However, under the PC\(^0\) restriction we can still have, as usual, indexes, which are dictionary type elements of the physical schema, and OO classes, which are dictionary type elements of the logical (and physical) schema. Thus, the PC\(^0\) fragment can still express the conjunctive core of OQL (and SQL).

2. **Left-deep trees only.** For simplicity purposes, all the physical plans that we consider correspond to left-deep trees only rather than bushy trees. It is not difficult to extend the results and the implementation to take into account bushy trees. However, such an extension is not directly relevant to our goal of measuring the effect of interacting the backchase exploration with cost evaluation.

3. **Nested-loops joins and index-based joins only.** The only join methods considered are nested-loops joins and index-based joins. It is not difficult to extend the results and the implementation to take into account hash-based joins and sort-merge joins as well. As for the previous simplifying assumption, the choice of join methods shouldn’t affect our results regarding the interaction between the backchase and cost evaluation.

### 6.1 Physical Plans for Physical Queries

#### 6.1.1 PC\(^0\) Restriction

The goal of this subsection is to introduce a simple, but important, restriction on the path-conjunctive language, which will allow us to develop two fragmentation techniques (described in sections 6.1.2 and 6.1.4). The first one, decomposition into atomic fragments, is similar, technically with the OQF fragmentation of Chapter 5. The main differences are: (1) the fragments are computed in a different way (and have different granularity, in general), and (2) the fragmentation has a different purpose here: it will allow us to reduce the problem of cost-estimation of a PC query to classical relational methods for join ordering enumeration, selectivity and cardinality estimation, etc. We must emphasize that this reduction is not at all obvious, due to the nature of the PC language: nested sets, dependent joins (in OO style), dictionaries. The goal of the fragmentation will be to partition the scans

\(^3\)No chase/backchase involved here.
of a query into fragments such that no dependency between scans is carried across different fragments. Each fragment resulting from the partition will correspond to exactly one schema element (possibly nested), and the query can be recovered from the join of its fragments. In the absence of the restriction that we are about to give, such a decomposition into fragments seems to be quite hard. The reasons will become apparent in section 6.1.4.

For all the results and algorithms of this chapter we restrict ourselves to a subclass of the PC language, subclass that we call PC0 and we define by the following two restrictions on the PC language:

- **Restriction 1:** All the paths appearing in path conjunctions (in queries and constraints) must be of simple type (no set/dictionary equality). Also, the paths occurring in the select clause of a query are restricted so that they are of simple type as well. (This is the same restriction that we imposed on PC queries and constraints in the completeness theorems of Chapter 3.)

  In addition,

- **Restriction 2:** All dictionaries that appear in queries and constraints are schema names. (In other words, we do not allow for higher-order dictionaries.)

**PC0 normal form.** Recall the canonical instance construction of section 3.2 used to describe the set of well-defined paths over a PC tableau \( T \). There we used the notation \( P \simeq Q \) whenever the equality of two path expressions \( P \) and \( Q \) is provable from the set of equalities given in \( T \). Under the PC0 restriction, the only possible instance of the rule (lookup-add), defining a lookup expression with respect to \( T \), has the following form:

\[
\frac{x \simeq y \quad y \in \text{dom} M \quad \text{in } T}{M[x] : G}
\]

Thus, given a well-defined lookup expression \( M[x] \) over some PC0 tableau \( T \), there are only two possible cases: a) \( x \in \text{dom} M \) occurs in \( T \), or b) \( x \) is “equal” to some other variable \( y \) such that \( y \in \text{dom} M \) occurs in \( T \). A PC0 normal form of a query \( Q \) is an equivalent rewriting \( Q^0 \) in which all occurrences of a lookup expression \( M[x] \) defined by case b) before are replaced by \( M[y] \). Such an equivalent rewriting always exists, because \( M[x] \simeq M[y] \) whenever \( x \simeq y \) (rule (lookup-cong) applies here). Therefore a PC0 normal form satisfies a third restriction:

- **Restriction 3:** Every lookup expression \( M[x] \) in a PC0 normal form of a tableau/query/dependency is guarded by an occurrence of \( x \in \text{dom} M \) in the tableau/query/dependency.

In the rest of the chapter we will always work with PC0 normal forms for queries.

### 6.1.2 Decomposition into Atomic Fragments.

As already mentioned in the introduction of the chapter, we want to be able to explore systematically all viable scan orders for a given query. Here a viable scan order means that the dependencies between scans are satisfied: a scan cannot be executed before a scan on which it depends. In the relational case all the scan orders (join orders) are viable because there are no dependent scans. The technique that we employ to achieve our goal is one of decomposition of the given query into atomic fragments. The atomic fragments of a query \( Q \) are disjoint query fragments of \( Q \) (as in the OQF fragmentation of Chapter 5, see also further section 6.1.4 for additional formal details) satisfying the following main properties:
1. $Q$ is equivalent, under the $PC^0$ restriction, with the natural join of its atomic fragments,
2. dependent scans are grouped within the same atomic fragment,
3. scans that are not dependent on each other are grouped in different fragments,
4. the space of all viable scan orders is the same as the space of all orders between atomic fragments.

The fragmentation that we use here is the same, technically, as the OQF fragmentation of Chapter 5. The main difference is the granularity of the fragments. Here each fragment is centered around one schema element occurring in the query, while in OQF the fragments have larger granularity, in general, and are determined by the homomorphisms from the views into the query. In both cases, the original query is equivalent to a join of its fragments. Also, the purpose of the fragmentation is different in the two methods. While in OQF, the goal was to split the universal plan in several of smaller size, so that the back chase becomes tractable for large queries, here the goal is to be able to employ dynamic programming-like algorithms to find a plan for a query from partial plans for its fragments.

We postpone a formal treatment of atomic fragments and decomposability of queries into atomic fragments for the section 6.1.4, and we illustrate the method using the following example.

Example 6.1.1 Recall the schema of example 1.2.1, and let us add to that one more class describing employees. Each employee has a name, a department to which he belongs, and a set of projects he works in. The class is translated into our internal language as a dictionary $\text{Emp}$.

$$\text{Dept : Dict(Did, Struct{string DName; Set(string) DProjs; string MgrName})}$$
$$\text{Proj : Set(Struct{string PName; string CustName; string PDept; string Budget})}$$
$$\text{Emp : Dict(Eid, Struct{string EName; Set(string) EProjs; string EDept})}$$

Consider now the following query $Q$ joining the class $\text{Dept}$, the relation $\text{Proj}$ and the class $\text{Emp}$:

$$\text{select struct(EName = Emp[e].EName, PName = p.PName, DName = Dept[d].DName)}$$
$$\text{from dom Dept d, Dept[d].DProjs s, Proj p, dom Emp e, Emp[e].EProjs n}$$
$$\text{where Emp[e].EDept = p.PDept and p.CustName = "Citibank" and s = p.PName and s = n}$$

The set of scans in the from clause of $Q$ can be partitioned in three sets, such that no scan in one set depends on scans in the other sets: $\{d, s\}, \{p\}, \{e, n\}$. (We showed here only the corresponding variables rather than the entire scans.) The above partition is the most refined\(^4\) partition that can be constructed in this way. We construct next the following three (atomic) query fragments of $Q$, each induced by one of the three above sets of variables:

$$(F_1) \text{ select struct(Lnk_1 = s, DName = Dept[d].DName)}$$
$$\text{from dom Dept d, Dept[d].DProjs s}$$

$$(F_2) \text{ select struct(Lnk_1 = p.PName, Lnk_2 = p.PDept)}$$
$$\text{from Proj p}$$
$$\text{where p.CustName = "Citibank"}$$

$$(F_3) \text{ select struct(Lnk_1 = n, Lnk_2 = Emp[e].EDept, EName = Emp[e].EName)}$$
$$\text{from dom Emp e, Emp[e].EProjs n}$$

There are two link attributes $\text{Lnk}_1$ and $\text{Lnk}_2$ that were to the output of the fragments. They store the information that is needed to recover all the equalities in $Q$ when $Q$ is to be reconstructed by joining the atomic fragments.

\(^4\)That’s why we chose the name atomic.
The link attributes and the path expressions associated to the link attributes in each fragment are computed as follows. We use the congruence closure of the equalities in the original query $Q$. There are three equivalence classes in the original query $Q$:

$$
\begin{align*}
C_0 &= \{p.\text{CustName}, \text{"Citibank"}\} \\
C_1 &= \{s, p.\text{PName}, n\} \\
C_2 &= \{\text{Emp}[e].\text{EDept}, p.\text{PDept}\}
\end{align*}
$$

The first class contains a constant, "Citibank". No link attribute is needed in this case. In general, whenever an equivalence class contains a constant $c$, we do the following: for each path expression $e$ that occurs in the class and is distinct from $c$, the equality $e = c$ can be pushed within the fragment that contains the variable on which $e$ depends. In our example, the equality $p.\text{CustName} = \text{"Citibank"}$ will occur in $F_2$, the fragment that is built around the variable $p$.

The second class contains three expressions, one depending on the variable $s$, the second depending on $p$, and the third depending on $n$. Therefore, after fragmentation, each expression will end up in a different atomic fragment: $F_1$, $F_2$, and $F_3$, respectively. To be able to recover the equivalence class from the fragments we need to introduce a new link attribute, call it $\text{Lnk}_1$, which will occur in each of the fragments, storing the respective expressions. In general, for each equivalence class (not containing a constant) that after fragmentation is split over $k$ fragments, with $k > 1$, we need to introduce a link attribute. The link attribute will appear in the output of each of the $k$ fragments. For the third class, similarly with $C_1$ we need to introduce a link attribute $\text{Lnk}_2$. Since the class is split over the fragments $F_2$ and $F_3$, each of them will output $\text{Lnk}_2$.

In addition to taking care of the equalities in $Q$ we need to make sure that each expression that occurs in the output of $Q$ is “produced” by one of the fragments. This is straightforward: for each $e$ in the output of $Q$ we find the (unique) atomic fragment over which $e$ is defined and we add $e$ to the output of the respective fragment.\footnote{If it is not already there, as the expression associated to some link attribute.}

The original query $Q$ can then be recovered from its atomic fragments by performing the natural join of the atomic fragments and then projecting over the attributes that appear originally in the output of $Q$ (an additional renaming is needed, in general):

$$Q = \Pi_{\text{EName}, \text{Lnk}, \text{DName}}(F_1 \bowtie F_2 \bowtie F_3)$$

As it can be seen from the example, decomposition into atomic fragments brings us a step closer to having a physical plan for the input query, in the traditional sense. If the atomic fragments are evaluated by some means and stored into temporary tables, we only need to specify the order in which the temporary tables are to be joined, and we obtain a physical execution plan. This plan simply performs a nested loops join of the temporary tables. Remark that selections are automatically pushed down to their atomic fragments. Also, all dependent joins are pushed within atomic fragments.

In the next subsection, we describe in a more systematic way the possible kinds of physical plans that we can have, including the important case of lookup plans. Later in section 6.3 we will look at the problem of enumerating all possible plans.

### 6.1.3 Physical Plans

In order to describe physical plans, we will use a simple language centered around two physical access methods: \textit{scan} and \textit{lookup}. A scan operation reads in the tuples of a given set (possibly the domain of a dictionary), while a lookup retrieves, given a key, the corresponding entry (if any) in the dictionary. We use a language rather than a tree-based representation because it is more convenient to express the flow of execution when scans may...
temp T1 =
  scan (x0 in dom Dept)
lookup x1 = Dept[x0]
scan (x2 in x1.DProjs)
  proj Lnk1 = x2, DName = x1.DName

temp T2 =
  scan (x0 in Proj)
  sel x0.CustName = ‘Citibank’
  proj Lnk1 = x0.PName, Lnk2 = x0.PDept

temp T3 =
  scan (x0 in dom Emp)
lookup x1 = Emp[x0]
scan (x2 in x1.EProjs)
  proj Lnk1 = x2, Lnk2 = x1.EDept, EName = x1.EName

in
blocl:
  scan (x0 in T1)
  proj Lnk1 = x0.Lnk1, DName = x0.DName

blocl2:
  scan (x0 in T2)
  sel x0.Lnk1 = Lnk1
  proj Lnk1 = Lnk1, DName = DName, Lnk2 = x0.Lnk2

blocl3:
  scan (x0 in T3)
  sel x0.Lnk1 = Lnk1, x0.Lnk2 = Lnk2
  proj EName = x0.EName, PName = Lnk1, DName = DName

Figure 6.1: A physical plan for the example query of section 6.1.2.

depend on each other and because of the explicit lookup operation (as opposed to the relational case, where it is implicit in index-based joins).

An important remark here is that, for physical plans, the restriction that we had for PC queries\(^6\), requiring that a lookup expression is safe, will not be needed anymore. In other words, the key used in the lookup of a physical plan is not required to be in the domain of the corresponding dictionary. Whenever the key is not in the domain, the entry and any tuple in the output of a plan based on that entry will be ignored (i.e not output). This will allow us to consider a larger class of physical plans than what PC queries can express. In particular we will be able to express plans based on pointer-based joins in addition to plans based on value-based joins. (Recall from section 3.1 that OO navigation queries can only be translated as PC queries by breaking the navigation into value-based joins using OID equality.)

Let us start from the example of subsection 6.1.2. The plan \( P \) for \( Q \), shown in figure 6.1, evaluates first the atomic fragments \( F_1, F_2, F_3 \), stores them into temporary relations \( T_1, T_2, T_3 \), and then performs the join \( T_1 \bowtie T_2 \bowtie T_3 \) with three nested scan operations. We explain below the details regarding the syntax and the semantics of the plan.

- Each plan specifies a set of temporary tables to be used in the main plan. A temporary table is populated by evaluating a local plan. Because we only consider left-deep trees of atomic fragments, a local plan will always be an atomic plan: a plan for an atomic fragment. In general, it can be another non-atomic plan with its own temporary tables and main plan. This would let us consider plans that are non left-deep (i.e bushy trees).

\(^6\)Essential for our theory of containment and chase.
• The local plans corresponding to $T_1$ and $T_3$ are each evaluated in two nested scans, separated by a lookup. Consider the plan for $T_1$: in the first stage, domDept is scanned, then for each resulting key $x_0$, and corresponding entry $x_1 = \text{Dept}[x_0]$, the set $x_1.\text{D}Projs$ is scanned. This plan implements a dependent join, depicted graphically in figure 6.2. In general every atomic fragment corresponding to a nested schema element (either set or dictionary) will require such a dependent join.

![Figure 6.2: Dependent join.](image)

• The main plan implements a **pipelined sequence of nested bloc plans**. Each bloc plan $i$ receives input tuples from its predecessor bloc plan $i-1$. For each such tuple\(^7\), bloc $i$ scans its own input table(s) for tuples that match its selection condition. The selection condition is a conjunction of equalities of the form $e = A$, where $e$ is an expression local to bloc $i$ while $A$ is a constant, or an attribute of a tuple coming from bloc $i-1$.

Each bloc plan $i$ projects then the attributes that are still needed in the latter blocs (for join conditions) or in the output of the plan. This is done in the proj statement, which specifies a list of attributes $A_1 = e_1, \ldots, A_k = e_k$, where $A_l$ is the name that is given to attribute $l$, while $e_l$ represents the expression assigned to attribute $l$. $e_l$ can be either an expression local to bloc $i$, or the name of an attribute coming from the previous bloc $i-1$, or a constant.

The scope of the variables introduced in one bloc is only the bloc itself.

• In the above example, all bloc plans in the main plan have only one scan operator (over a temporary table). The main plan corresponds thus to a left-deep join operator tree with three leaves, depicted graphically in figure 6.3. We will see, shortly, that, as a result of applying plan transformations, a bloc plan can have more than one scan.

• All the projections are applied as early as possible: if an attribute is not needed after bloc $i$, then it does not occur in the list of projected attributes of bloc $i$. When translating queries into plans we will always follow this strategy\(^8\).

![Figure 6.3: Left-deep trees of $P$ and $P'$.](image)

\(^7\)Or, rather, block of tuples, as we discuss in the cost model section.

\(^8\)Same as in many relational optimizers.
Plan Transformation 1: UnfoldTemp. The plan above is not necessarily the best plan for Q. Evaluating first the atomic fragment \(F_2\) and storing it into \(T_2\) may reduce considerably the number of tuples that enter later in the main plan, because of the selection condition on \texttt{CustName}. However, for the atomic fragment \(F_1\) we can short-circuit building the temporary table \(T_1\) and certainly obtain a faster plan. Therefore, one capability that the plan language must offer is the possibility of unfolding any of its temporary tables. In our example, we can unfold\(^9\) \(T_1\) and obtain the following plan \(P'\):

\[
\text{temp } T_2 = \\
\text{ scan (x0 in Proj) } \\
\text{ sel \ "Citibank" = x0.CustName } \\
\text{ proj Lnki = x0.PName, Lnk2 = x0.PDept }
\]

\[
\text{temp } T_3 = \\
\text{ scan (x0 in dom Emp) } \\
\text{ lookup x1 = Emp[x0] } \\
\text{ scan (x1 in 1.EProjs) } \\
\text{ proj Lnki = x2, Lnk2 = x1.EDept, EName = x1.EName }
\]

\[
in \text{ bloc1: } \\
\text{ scan (x0 in dom Dept) } \\
\text{ lookup x1 = Dept[x0] } \\
\text{ scan (x2 in 1.DProjs) } \\
\text{ proj Lnki = x2, DName = x1.DName }
\]

\[
\text{bloc2: } \\
\text{ scan (x0 in T2) } \\
\text{ sel x0.Lnki = Lnki } \\
\text{ proj Lnki = Lnki, DName = DName, Lnk2 = x0.Lnk2 }
\]

\[
\text{bloc3: } \\
\text{ scan (x0 in T3) } \\
\text{ sel x0.Lnki = Lnki, x0.Lnk2 = Lnk2 } \\
\text{ proj ENName = x0.ENName, PName = Lnki, DName = DName }
\]

After the transformation, the main plan corresponds to a left-deep join tree with four leaf nodes, see figure 6.3. In general, each scan in the main plan will correspond to a leaf node in the join tree.

Similarly, we can unfold \(T_3\) and obtain \(P''\):

\[
\text{temp } T_2 = \\
\text{ scan (x0 in Proj) } \\
\text{ sel \ "Citibank" = x0.CustName } \\
\text{ proj Lnki = x0.PName, Lnk2 = x0.PDept }
\]

\[
in \text{ bloc1: } \\
\text{ scan (x0 in dom Dept) } \\
\text{ lookup x1 = Dept[x0] } \\
\text{ scan (x2 in 1.DProjs) } \\
\text{ proj Lnki = x2, DName = x1.DName }
\]

\[
\text{bloc2: } \\
\text{ scan (x0 in T2) } \\
\text{ sel x0.Lnki = Lnki } \\
\text{ proj Lnki = Lnki, DName = DName, Lnk2 = x0.Lnk2 }
\]

\[
\text{bloc3: } \\
\text{ scan (x0 in dom Emp) }
\]

---

\(^9\)Very similar with view expansion.
lookup \( x_1 = \text{Emp}[x_0] \)
select \( x_1.\text{EDep} = \text{Lnk2} \)
scan \( (x_2 \text{ in } x_1.\text{EProjs}) \)
select \( x_2 = \text{Lnk1} \)
project \( \text{EName} = x_1.\text{EName}, \text{PName} = \text{Lnk1}, \text{DName} = \text{DName} \)

Following the same principle as before, after the expansion, the selections are pushed as early as possible. For example, the selection condition \( x_1.\text{EDep} = \text{Lnk2} \) is pushed right after the lookup operation in bloc 3. The main plan corresponds now to a left-deep tree with 5 leaf nodes, see figure 6.4. The execution of \( P'' \) proceeds for the first two blocks as in the case of \( P' \). When the pipelined execution reaches bloc 3, it does the following: for each incoming tuple \( t \) (from bloc 2) it scans \( \text{dom Emp} \) and for each key \( x_0 \) in \( \text{dom Emp} \) it appends \( t \) with \( x_0 \). The resulting tuples\(^{10} \) are then passed to the last node in the tree. Here, for each such tuple, the lookup is executed first, and if the selection condition is true, the final scan is performed.

![Left-deep tree of \( P'' \).](image)

**Note.** The number of incoming tuples before the last lookup operator in \( P'' \) can be quite large (because of the cartesian product computation). Therefore \( P'' \) may be more expensive than \( P' \). (In \( T_0 \) of \( P' \) the lookup is performed “only” for each key in \( \text{dom Emp} \).) However, there are situations in which such UnfoldTemp may be worthwhile. If after unfolding, the value of \( x_0 \) in \( \text{dom Emp} \) becomes equated with one of the attributes of the incoming tuples (because of some selection condition) then the scan over \( \text{dom Emp} \) can be entirely removed. Then we need to perform the lookup only as many times as many incoming tuples are. If this number is smaller than the cardinality of \( \text{dom Emp} \), then the resulting plan has a smaller cost. By using this strategy we will be able to discover the traditional index-based join plans from the relational world, as well as the pointer-based joins from the OO world. Removal of scans over domains of dictionaries is the object of the next plan transformation.

**Plan Transformation 2: RemoveScanDom.** Consider now the following query, equivalent to \( Q^{11} \):

\[
(Q_1) \quad \text{select} \quad \text{struct}(\text{EName} = \text{Emp}[e].\text{EName}, \text{PName} = p, \text{DName} = \text{Emp}[e].\text{EDep}) \\
\quad \text{from} \quad \text{dom I p, dom Emp e, Emp}[e].\text{EProjs} \\
\quad \text{where} \quad \text{Emp}[e].\text{EDep} = I[p].\text{PDEp} \text{ and } I[p].\text{CustName} = "\text{Citibank}" \text{ and } n = p
\]

The following is a plan for \( Q_1 \):

\[
\text{temp} \quad T_1 = \quad \text{scan} \quad (x_0 \text{ in dom Emp}) \\
\quad \text{lookup} \quad x_1 = \text{Emp}[x_0]
\]

\(^{10}\text{They constitute the cartesian product of } \text{dom Emp} \text{ with the result of the lower tree.}\)

\(^{11}\text{\( Q_1 \) is one of the candidate plans generated by C&B when given } Q \text{ as input and the constraints of example 1.2.1. Recall that } I \text{ is a primary index for Proj on PName.}\)
scan (x2 in x1.EProjs)
    proj Lnk1 = x2, Lnk2 = x1.EDept, EName = x1.EName

temp T2 =
scan (x0 in dom I)
  lookup x1 = I[x0]
  sel x1.CustName = "Citibank"
  proj Lnk1 = x0, Lnk2 = x1.PDept
in bloc1:
  scan (x0 in T1)
  proj Lnk1 = x0.Lnk1, Lnk2 = x0.Lnk2, EName = x0.EName
bloc2:
  scan (x0 in T2)
  sel x0.Lnk1 = Lnk1, x0.Lnk2 = Lnk2
  proj EName = EName, PName = Lnk1, DName = Lnk2

By applying UnfoldTemp to T1 and T2 we obtain:

bloc1:
  scan (x0 in dom Emp)
  lookup x1 = Emp[x0]
  scan (x2 in x1.EProjs)
    proj Lnk1 = x2, Lnk2 = x1.EDept, EName = x1.EName
bloc2:
  scan (x0 in dom I)
  sel x0 = Lnk1
  lookup x1 = I[x0]
  sel x1.CustName = "Citibank", x1.PDept = Lnk2
  proj EName = EName, PName = Lnk1, DName = Lnk2

Notice that the selection $x_0 = \text{Lnk}_1$ in bloc 2 was pushed right after the scan over $\text{dom I}$. (Pushing such selections can be done very easily by using the congruence closure representation of the equalities within a query.) Now it is obvious that the scan over $\text{dom I}$ can be short-circuited! The RemoveScanDom transformations performs the removal of scan ($x_0 \in \text{dom I}$) whenever the variable $x_0$ is equated to a variable-free expression (a constant or an attribute of the incoming tuple). The resulting plan is an index-based join between Emp and Proj:

bloc1:
  scan (x0 in dom Emp)
  lookup x1 = Emp[x0]
  scan (x2 in x1.EProjs)
    proj Lnk1 = x2, Lnk2 = x1.EDept, EName = x1.EName
bloc2:
  lookup x0 = I[Lnk1]
  sel x0.CustName = "Citibank", x0.PDept = Lnk2
  proj EName = EName, PName = Lnk1, DName = Lnk2

Remark here that $I[Lnk_1]$ is not a safe lookup operation (in the PC sense). While in the previous plan (right before applying RemoveScanDom), we lookup $I$ only after we check that $Lnk_1$ is equal to some key $x_0$ in $\text{dom I}$, in the transformed plan, $Lnk_1$ is not guaranteed to be in $\text{dom I}$. If this happens then the entry is ignored (and any resulting tuples based on that entry). Thus, the semantics of the transformed plan is the same as the semantics of the untransformed plan. Nonetheless, its operational semantics is quite different, and in fact any of the two plans can have a better cost.
Another case in which the RemoveScanDom transformation can be applied is when there is a selection with a constant on some attribute for which there exists an index. For example, consider the following query, also equivalent to $Q^{12}$:

$$(Q_2) \text{select } \text{struct}(\text{EName} = \text{Emp}[e].\text{EName}, \text{PName} = t.\text{PName}, \text{DName} = \text{Emp}[e].\text{EDept})$

$$\text{from } \text{dom } \text{SI } k, \text{SI}[k] t, \text{dom } \text{Emp } e, \text{Emp}[e].\text{EProjs } n$$

$$\text{where } \text{Emp}[e].\text{EDept} = t.\text{PDept } \text{and } k = "\text{Citibank}" \text{ and } n = t.\text{PName}$$

We translate the query into a plan in the usual way:

```plaintext
temp T1 =
  scan (x0 in dom Emp)
  lookup x1 = Emp[x0]
  scan (x2 in x1.EProjs)
    proj Lnk1 = x2, Lnk2 = x1.EDept, EName = x1.EName

temp T2 =
  scan (x0 in dom SI)
  sel x0 = "Citibank"
  lookup x1 = SI[x0]
  scan (x2 in x1)
    proj Lnk1 = x1.PName, Lnk2 = x1.PDept

in bloc1:
  scan (x0 in T1)
    proj Lnk1 = x0.Lnk1, Lnk2 = x0.Lnk2, EName = x0.EName

bloc2:
  scan (x0 in T2)
    sel x0.Lnk1 = Lnk1, x0.Lnk2 = Lnk2
    proj EName = EName, PName = Lnk1, DName = Lnk2
```

Remark that in the translation the selection condition on CustName is pushed right after the scan of dom SI. Therefore we can directly apply RemoveScanDom (without the need to apply UnfoldTemp on T2). The resulting plan, after applying UnfoldTemp on T1, is:

```plaintext
temp T2 =
  lookup x0 = SI["Citibank"]
  scan (x1 in x0)
    proj Lnk1 = x1.PName, Lnk2 = x1.PDept

in bloc0:
  scan (x0 in dom Emp)
    lookup x1 = Emp[x0]
    scan (x2 in x1.EProjs)
      proj Lnk1 = x2, Lnk2 = x1.EDept, EName = x1.EName

bloc1:
  scan (x0 in T2)
    sel x0.Lnk1 = Lnk1, x0.Lnk2 = Lnk2
    proj EName = EName, PName = Lnk1, DName = Lnk2
```

**Conclusion.** In order to find the best plan for a query, we will exhaustively try all possible applications of UnfoldTemp and RemoveScanDom, as well as all possible join orderings. The systematic enumeration of physical plans will be addressed in section 6.3.

---

12 And, also, a candidate plan. Recall from example 1.2.1 that SI is a secondary index for Proj on CustName.
6.1.4 Decomposition into Atomic Fragments: Formal Details

This section describes formally the technique for query decomposition into atomic fragments. Based on this, we will reduce the problem of enumerating viable scan orders to the problem of enumerating all orders between atomic fragments. This reduction is complete (does not miss any viable scan order) if all atomic fragments are single-path fragments (defined below). In general, and specific to queries over OO classes and complex values, atomic fragments may not be single-path. For such case, we give a general method for decomposition of an atomic fragment into a join of single-path fragments. Then, to enumerate viable scan orders, we consider also all possible orders between single-path fragments within one atomic fragment.

**Variable-dependency forest.** For any path expression $P$ in a $PC^0$ normal form, there can be at most one variable $x$ occurring in $P$. This is true because of Restriction 2 of Section 6.1.1. Otherwise we could have expressions such as $x[y]$ where $x$ is a dictionary but not one in the schema.

Let $s y$ be some scan in a $PC^0$ normal form query. We say that $y$ directly depends on the variable $x$ if $x$ occurs $s$. We can then define a dependency graph associated to a query by taking as the set of nodes the set of all variables in the query, and adding directed edges between any two variables such that one directly depends on the other. The following lemma is a simple, but important for this chapter, consequence of the $PC^0$ restriction.

**Lemma 6.1.1** The variable-dependency graph of a normal form $PC^0$ query is a forest: a set of connected components each of them being a tree. The root of each tree is a variable $x$ ranging over a schema element $S$ of set type or over $\text{dom}_S$ where $S$ is a schema element of dictionary type.

**Example:** recall the query $Q$ of section 6.1.2. Its variable-dependency forest consists of three (linear, in this case) trees: $\{d \rightarrow s, p, e \rightarrow n\}$.

Query fragments for a given query were first introduced, rather informally, in Chapter 5 in the context of OQF fragmentation. Here is the complete definition:

**Definition 6.1.2 (Query Fragment)** Let $Q$ be a $PC^0$ normal form query. Its closure is a query $Q^*$ that has the same **select** and **from** clauses as $Q$ while the **where** clause consists of all the equalities that occur in or are implied by the $Q$’s **where** clause. $Q^*$ is computable from $Q$ in PTIME and is equivalent to $Q$. $Q^*$ without the **select** clause is the isomorphic representation, as a query, of the canonical instance introduced in Chapter 3.

Let $S$ be a subset of the variables in the **from** clause of $Q$, such that $S$ is closed under the direct dependency relation (defined above). A query fragment $F$ of $Q$ induced by $S$ is a query such that:

1) The **from** clause consists of exactly the variables in $S$, together with their sets (as they were in $Q$)

2) The **where** clause consists of all the conditions in the **where** clause of $Q^*$ which mention only variables in $S$, and

3) The **select** clause consists of all the paths $P$ over $S$ that occur in the **select** clause of $Q$ or in an equality $P = P'$ of $Q^*$’s **where** clause where $P'$ depends on some variable that is not in $S$. In the latter case, we call such $P$ a link path of the fragment. A unique attribute name, called link attribute is invented for each equivalence class in the congruence closure of $Q$. Every link expression in the output of a fragment will appear preceded by the link attribute name associated to the equivalence class in which the expression belongs.

**Definition 6.1.3 (Atomic Fragments)** Let $Q$ be a normal form $PC^0$ query with a variable-dependency forest consisting of trees $\{t_1, \ldots, t_n\}$. The set of atomic fragments associated to $Q$ is the set of query fragments $\{F_1, \ldots, F_n\}$, where each $F_i$ is the query fragment of $Q$ induced by the set of variables in $t_i$. 

116
Each atomic fragment corresponds to exactly one schema element occurring in the query. There may be more than one atomic fragment for one schema element (if the schema element occurs more than once in the query). The output of each fragment $F_i$ may contain, besides expressions needed in the final output of $Q$, link expressions needed for joins with the other fragments. The following proposition holds (the proof is obvious):

**Proposition 6.1.4 (Decomposition into Atomic Fragments)** Every normal form PCO query can be uniquely recovered by a natural join of the atomic fragments.

**Decomposition into Single-Path Fragments.** So far, in our examples of atomic fragments, the dependency trees between variables were linear. We call such atomic fragments single-path atomic fragments. In general, the dependency tree of an atomic fragment may be a non-linear tree, i.e., a tree consisting of more than one path. We will call such atomic fragments multiple-path atomic fragments. This section describes our next method for decomposing a multiple-path atomic fragment into a join of smaller fragments, each of them being single-path. The decomposition allows us to find, under a contiguity assumption for storage of nested sets, efficient physical plans for any multiple-path atomic fragment.

Let us illustrate by an example. Consider the following nested schema element and query:

$$S : \text{Set} \{ \text{Struct} \{ A : \text{int}, \}
B : \text{Set} \{ \text{Struct} \{ X : \text{int},
Y : \text{Set} \{ \text{Struct} \{ D : \text{string}, E : \text{int} \} \},
Z : \text{Set} \{ \text{Struct} \{ D : \text{string}, E : \text{int} \} \} \},
C : \text{Set} \{ \text{Struct} \{ D : \text{string}, E : \text{int} \} \}
\}\}$$

$$(Q) \quad \text{select} \quad \text{struct}(A = x.A)
\quad \text{from} \quad S x, x.B y, x.C z, y.Y u, y.Z v
\quad \text{where} \quad u.E = z.E \quad \text{and} \quad u.D = v.D \quad \text{and} \quad y.X = 100$$

The dependency tree of the query, shown in figure 6.5, consists of three distinct paths: $\{x \rightarrow y \rightarrow u, x \rightarrow y \rightarrow v, x \rightarrow z\}$. The fact that the same variable $x$ is used as the root of all the three paths means a join condition: when navigating the nested set $S$ the three paths must always start from within the same top-level tuple $x$ of $S$. Similarly, the fact that $y$ occurs, at the second level, in the first two paths means another join condition: while navigating, the two paths, not only must be within the same top-level tuple $x$, but they must be within the same tuple $y$ in $x.B$. $Q$ is thus equivalent to the natural join of three single-path queries:

$$(P_1) \quad \text{select} \quad \text{struct}(\text{Lnk}_1 = u.D, \text{Lnk}_2 = u.E, \text{Tid}_1 = x, \text{Tid}_2 = y, A = x.A)
\quad \text{from} \quad S x, x.B y, y.Y u
\quad \text{where} \quad y.X = 100$$

$$(P_2) \quad \text{select} \quad \text{struct}(\text{Lnk}_1 = v.D, \text{Tid}_1 = x, \text{Tid}_2 = y)
\quad \text{from} \quad S x, x.B y, y.Z v$$

$$(P_3) \quad \text{select} \quad \text{struct}(\text{Lnk}_2 = z.E, \text{Tid}_1 = x)
\quad \text{from} \quad S x, x.C z$$

Here $\text{Tid}_1$ and $\text{Tid}_2$ are used to store identities of tuples $x$ in $S$ and $y$ in $x.B$. In order to do the decomposition we must always assume that each element has an identity. To recover $Q$ from its fragments, we need to join on $\text{Tid}_1$ and $\text{Tid}_2$. This is of a different nature from the join on $\text{Lnk}_1$ and $\text{Lnk}_2$ (also required to recover $Q$): it is required by the topology of the dependency tree of $Q$, and not by explicit equalities of $Q$.

---

13This is something particular to the complex value model, which allows for arbitrarily nested sets
14Similar to a tuple id in a relation. Such identifiers always exist, in an implementation.
An execution for \( Q \) can then be obtained by evaluating in any order the three single-path queries \( P_1, P_2 \) and \( P_3 \), with the corresponding join conditions. The second assumption that we make is a natural one and it concerns the storage model of complex values: every nested set is assumed to be stored \emph{contiguously} on disk. In consequence, evaluating a single-path query such as any \( P_i \) above requires exactly one single scan over the disk pages storing the nested set.

Below is a nested loop join plan for \( Q \), that uses the above decomposition of \( Q \) into three single-path queries. Notice that we only need three nested scans over \( S \) (even though the original query \( Q \) has 5 variables).

\begin{verbatim}
scan (x in S) (y in x.B) (u in y.Y)
    sel y.X = 100
    proj A = x.A, Tid1 = x, Tid2 = y, Lnk1 = u.D, Lnk2 = u.E
scan (x in S) (y in x.B) (v in y.Z)
    sel x = Tid1, y = Tid2, v.D = Lnk1
    proj A = A, Tid1 = Tid1, Tid2 = Tid2, Lnk1 = Lnk2
scan (x in S) (z in x.C)
    sel x = Tid1, z.E = Lnk2
    proj A = A
\end{verbatim}

Similarly, we may want to decompose multiple-path atomic fragments into single-path fragments in the case of dictionaries with entries that are complex values. The same decomposition algorithm as described above applies. The only difference is that the complex value in this case is not a schema element, but rather the entry of a dictionary.

**Enumeration of viable scan orders.** We conclude the discussion of this section with the following:

**Fact.** Let \( Q \) be a \( PC^0 \) normal form query such that all of its atomic fragments are single-path. Then the space of all viable scan orders for \( Q \) is the space of all possible orders between the atomic fragments of \( Q \).

When an atomic fragment is not single-path then we have to consider all possible scan orders between its single-path fragments, as well. In general, for complete enumeration of viable scan orders, we must consider also all possible orders between single-path fragments across more than one atomic fragment. However, we choose not to do it, in order to reduce the complexity of the dynamic programming algorithm that implements the enumeration (see section 6.3). In fact, in most of our examples and experiments, the atomic fragments will be single-path.

### 6.2 A Cost Model for Nested Sets and Dictionaries

This section presents a cost model for the physical plans introduced in section 6.1. Subsection 6.2.4 gives the formulas that we use to calculate the cost of a plan. These formulas, in turn, use quantitative information that is associated to the elements of the physical schema. Thus, subsection 6.2.1 gives a language to describe such information (cardinality, selectivity). The information that we can express by using this language generalizes the
so-called database statistics used in in relational databases [Ram98] and some of the OO databases [GGT96]. The generalization is required mainly because of the presence of arbitrarily nested sets.

The decomposability of queries into atomic fragments, which are further decomposed into single-path fragments, plays a fundamental role in cost estimation of queries. The cardinality tables that we introduce in section 6.2.1 are built for single-path tableaux over one schema element. They generalize what in the relational case used to be the selectivities of the attributes of one relation. We use only single-path tableaux because there are only finitely many of them for a schema element. Estimating the cardinality, selectivity, or size of one single-path query becomes then simply a problem of matching. Estimating the cardinality, selectivity, or size of one multiple-path query can then be done by using its decomposition into a join of single-path fragments. Similarly, for a PCO query, we use then its decomposition into a join of atomic fragments. In the absence of the PCO restriction, a query could have dependencies between scans that form a graph rather than a tree, and decomposability into atomic fragments and/or single-path fragments may not be possible. In that case, cost estimation becomes a much harder problem.

6.2.1 Cardinality Information.

As seen in section 6.1, a physical plans is a mainly nested sequence of plans for atomic fragments, corresponding to a left-deep join tree. In order to evaluate the cost of such a left-deep tree, we need to be able to evaluate the cost of its left and right subtrees. In addition, to evaluate the cost of the join (either a nested loop join or a lookup-based join) between the two subtrees, we will need to evaluate the cardinalities, and sizes, of the results of the subtrees. The cardinality of the result of a subtree is the number of tuples produced by the subtree. The size of the result of a subtree is the size, bytes, of all the tuples produced by the subtree. The following is a language for describing the information needed to compute cardinalities of any intermediate results.

Definition 6.2.1 Let $S$ be a schema element (set or dictionary). A single-path tableau for $S$ is a PCO tableau $T = \{ x_1 \in S, x_2 \in e_2(x_1), \ldots, x_n \in e_n(x_{n-1}); \text{true} \}$, with no equality conditions and no other schema element besides $S$.

A single-path tableau for $S$ corresponds to exactly one navigation path starting at the root $S$ and following nested sets within $S$. Since the type of $S$ is finite, there are only finitely many single-path tableaux for $S$.

Example 6.2.1 Recall the schema element $S$ of section 6.1.4. The following are single-path tableaux for $S$:

- $T_1 = \{ x \in S; \text{true} \}$
- $T_2 = \{ x \in S, y \in x.B, u \in y.Y; \text{true} \}$
- $T_3 = \{ x \in S, y \in x.B, v \in y.Z; \text{true} \}$
- $T_4 = \{ x \in S, z \in x.C; \text{true} \}$

Definition 6.2.2 A cardinality table associated to a single-path tableau $T = \{ x_1 \in S, x_2 \in e_2(x_1), \ldots, x_n \in e_n(x_{n-1}); \text{true} \}$ is a pair consisting of:

1. $\text{card}^T$ = the cardinality of the set \{$(x_1, \ldots, x_n)$\}, and
2. a partial function $\{ (e_1 : \text{card}_{e_1}^T), \ldots, (e_n : \text{card}_{e_n}^T) \}$ where, for each $i$:
   a) $e_i$ is a well-defined path over $T$, and
   b) $\text{card}_{e_i}^T$ = the cardinality of the set \{(x_1, \ldots, x_n) | e_i = \text{constant} \}.

Example 6.2.2 A cardinality table for $T_1$ of the previous example is: $< \text{card}_{e_1}^T = 100, \{(x \in S, (x.A : 25)) \}>$. The first number gives us the number of tuples in $S$: 100. The pair $(x, 1)$ tells us that if we fix the value of $x$
then we remain with 1 tuple in \( S \). The second pair tells us that if we fix the value of \( x.A \) then we are left with 25
tuples in \( S \). The ratio between \( \text{card}_{T_1,A} \) and \( \text{card}_{T_1} \) gives us the selectivity of the attribute \( A \) within the set \( S \).

Notice that there are semantic conditions that a cardinality table need to satisfy in order to make sense. For
eexample, the pair \((x, 1)\) is a uniqueness condition: each element of \( S \) must be unique. No other value but 1 makes
sense there. On the other hand, we could have any real number between 0 and 100 for the value of \( \text{card}_{T_1,A} \).

Checking in general that cardinality tables satisfy the semantic conditions that hold in the database (set
semantics (as opposed to bag semantics), key constraints, referential integrity constraints, etc.) is an interesting
problem in itself. However, we do not address it here.

Examples of cardinality tables for \( T_2, T_3, \) and \( T_4 \) are given below:

\[
< 150,000, \{(x : 1, 500), (x.A : 100), (y : 3, 000), (y.X : 5, 000), (u.D : 500), (u.E : 800)\} > \\
< 150,000, \{(x : 1, 500), (x.A : 100), (y : 3, 000), (y.X : 5, 000), (v.D : 450), (v.E : 750)\} > \\
< 40,000, \{(x : 400), (x.A : 100), (z : 25), (z.D : 200), (z.E : 350)\} >
\]

### 6.2.2 Estimating Cardinals of Physical Plans.

Using the cardinality tables described in section 6.2.1 we are now able to estimate the cardinality of any \( PC^0 \)
query, i.e. the number of tuples in the result of the query. Since any physical plan implements some \( PC^0 \) query,
we are then able to estimate the number of tuples produced by any intermediate plan.

**Cardinality of a single-path query.** Consider the single-path query \( P_1 \) of section 6.1.4:

\[
(P_1) \quad \text{select} \quad \text{struct}(\text{Lnk}_1 = u.D, \text{Lnk}_2 = u.E, \text{Tid}_1 = x, \text{Tid}_2 = y, A = x.A) \\
\text{from} \quad S \times x.B \ y \ y.\text{u} \\
\text{where} \quad y.X = 100
\]

The single-path tableau to which \( P_1 \) matches is \( T_2 \) from section 6.2.1. \( \text{card}_{T_2} \) tells us that there are 150,000
tuples \((x, y, u)\) over the single-path tableau \( T_2 \). Since we are interested only in those tuples that satisfy the
condition \( y.X = 100 \) we must multiply \( \text{card}_{T_2} \) with the selectivity of this equality, which is the ratio between
\( \text{card}_{y.X} \). Therefore the number of qualified tuples \((x, y, z)\) is 5,000. The cardinality of \( P_1 \) is 5,000
as well.\(^{15}\)

**Cardinality of a multiple-path query.** Consider now the multiple-path query \( Q \) of section 6.1.4. \( Q \) is
equivalent to the natural join of three single-path queries, \( P_1, P_2, P_3 \). We rewrite it in the following form:

\[
\text{select} \quad \text{struct}(A = x.A) \\
\text{from} \quad P_1 \times x_1, P_2 \times x_2, P_3 \times x_3 \\
\text{where} \quad x_1.\text{Tid}_1 = x_2.\text{Tid}_1 \quad \text{and} \quad x_2.\text{Tid}_1 = x_3.\text{Tid}_1 \quad \text{and} \quad x_1.\text{Tid}_2 = x_2.\text{Tid}_2 \quad \text{and} \\
\text{x_1.Lnk}_1 = x_2.\text{Lnk}_1 \quad \text{and} \quad x_2.\text{Lnk}_2 = x_3.\text{Lnk}_2
\]

The cardinality of \( Q \) (ignoring duplicates) can be computed as follows:

\[
(*) \quad \text{card}(Q) = \text{card}(P_1) \times \text{card}(P_2) \times \text{card}(P_3) \times \text{selectivity (join conditions)}
\]

where the last term is the product of selectivities of each equality condition. To estimate the selectivity of an
equality \( e_1 = e_2 \), we follow [Ram98] and use the following formula:

\[
\text{sel}(e_1 = e_2) = \min\{\text{sel}(e_1), \text{sel}(e_2)\}
\]

\(^{15}\)Ignoring duplicates. In general, we must take into account what attributes are in the output of the query since this can affect
the number of duplicates that the query produces. This can be done using the cardinality information as well, and we don't show
it here.
Then \( \text{sel}(e_i) \) can be calculated by finding the single-path fragment from which \( e_i \) comes and using the cardinality table associated to the respective single-path tableau. For example,

\[
\text{sel}(x_1. \text{tid}_1 = x_2. \text{tid}_1) = \min\{\text{sel}(x_1. \text{tid}_1), \text{sel}(x_2. \text{tid}_1)\}
\]

\[
= \min\{\frac{\text{card}_{\text{tid}_2}}{\text{card}_{\text{tid}_1}}, \frac{\text{card}_{\text{tid}_1}}{\text{card}_{\text{tid}_2}}\}
\]

\[
= \min\{\frac{500}{1500}, \frac{1500}{500}\}
\]

\[
= \frac{1}{100}
\]

The reader can then verify that the cardinality of \( Q \) is 960 tuples (ignoring duplicates).

**Cardinality of a PC\(^0\) query.** In section 6.1.2 we described how a PC\(^0\) query can be decomposed into atomic fragments. The cardinality of a PC\(^0\) query can then be calculated with a formula similar with (*) above but in which all single-path fragments are replaced by atomic fragments.

### 6.2.3 Estimating Sizes

While evaluating the cost of a plan we will need to be able to estimate the sizes of the sets over which a scan operation will be performed. The sets will be either schema elements of set type, or domains of dictionaries, or entries of dictionaries. In the following we give a general method, that uses the cardinality information described in section 6.2.1, to estimate the size of any expression defined over a single-path tableau.

Let \( T = \{x_1 \in e_1, \ldots, x_n \in e_n(x_{n-1})\} \) be a single-path tableau for schema element \( S = e_1 \), and let \( e(x_n) \) be some well-defined path over \( T \). We will denote by \( \text{sz}(e \mid x_1 \in e_1, \ldots, x_n \in e_n(x_{n-1})) \) the size of \( e \) (in bytes) with respect to \( T^{10} \).

The following is an inductive definition for \( \text{sz}(e \mid x_1 \in e_1, \ldots, x_n \in e_n(x_{n-1})) \). The induction is on the type \( \tau \) of \( e \).

- \( \tau \) is a base type. Let \( B_\tau \) be the size in bytes of an element of type \( \tau \)\(^{11}\).

\[
\text{sz}(e \mid x_1 \in e_1, \ldots, x_n \in e_n) = B_\tau
\]

- \( \tau \equiv \text{Struct}\{A_1 : \tau_1, \ldots, A_k : \tau_k\} \).

\[
\text{sz}(e \mid x_1 \in e_1, \ldots, x_n \in e_n) = \sum_{i=1}^k \text{sz}(e.A_i \mid x_1 \in S, \ldots, x_n \in e_n)
\]

- \( \tau \equiv \text{Set}(\tau_0) \).

\[
\text{sz}(e \mid x_1 \in e_1, \ldots, x_n \in e_n) = \text{card} \{ (x_1, \ldots, x_n, x_{n+1}) \mid x_1 \in e_1, \ldots, x_n \in e_n, x_{n+1} \in e \}
\]

\[
\times \text{sel}(x_1 \mid x_1 \in e_1, \ldots, x_n \in e_n, x_{n+1} \in e)
\]

\[
\ldots
\]

\[
\times \text{sel}(x_n \mid x_1 \in e_1, \ldots, x_n \in e_n, x_{n+1} \in e)
\]

\[
\times \text{sz}(x_{n+1} \mid x_1 \in e_1, \ldots, x_n \in e_n, x_{n+1} \in e)
\]

In the last case, we define, inductively, the size of \( e \) as the product between how many elements are in \( e \) (all but the last term) and the size in bytes of each such element (the last term). The cardinality of \( e \) is computed as follows: Let \( T_{n+1} \) be the single-path tableau obtained by adding one more variable \( x_{n+1} \in e \). We compute first

\(^{10}\)We need to specify \( T \) because \( e \) is defined with respect to \( T \).

\(^{11}\)Average size, if the size is not constant.
the total number of tuples \((x_1, \ldots, x_m, x_{m+1})\) (the first term, also equal to \(\text{card}\{T_n+1\}\)), and we multiply it with the selectivity of each \(x_i\) for \(i = 1, n\) (since we are interested only in how many values for \(x_{m+1}\) are there). The selectivity of \(x_i\) can be obtained as the ratio between \(\text{card}\{T_n\}\) and \(\text{card}\{T_{n+1}\}\).

If \(S\) is a schema element of type set, then the size of \(S\) is \(sz(S | \cdot)\). If \(\mathcal{M}\) is a schema element of type dictionary then the size of \(\text{dom}\mathcal{M}\) is given by \(sz(\text{dom}\mathcal{M} | \cdot)\), while the size of each entry in the dictionary is given by \(sz(\mathcal{M} \mid k) | k \in \text{dom}\mathcal{M}\).

6.2.4 A Simple Model for Cost-Evaluation of Physical Plans.

In the following we present a cost model for the different physical plans introduced in section 6.1.3. In general, there are several different components of the cost: CPU cost, I/O cost, network cost (for distributed databases), etc. For simplicity, we will ignore any CPU or network cost, and focus on the I/O cost (the cost of read and write operations between memory and disk). We assume that:

- The execution always corresponds to a left-deep tree in which the intermediate results from the left subtree are pipelined to the next join node in the tree;
- (Nested) sets are stored contiguously (i.e. clustered), as already discussed in section 6.1.4.
- Entries of dictionaries are stored non-contiguously (i.e. non-clustered: each entry starts on a different page); however, each entry, if it is a set, is stored clustered (according to the previous assumption). The last two assumptions are a simplification of more complicated clustering strategies that occur in the practice.
- \(M\) is the amount of memory (in pages) available for processing at a given join node in the left-deep tree;
- \(p\) is the page size (in bytes)

Let \(P\) be a plan corresponding to a left-deep tree in which the left subtree is another left-deep tree \(P_1\), while the right subtree is an atomic plan, \(P_2\), for some atomic fragment \(F\) with schema element \(S\). In other words, \(P = P_1 \bowtie P_2\). For simplicity of presentation we assume that the atomic fragment \(F\) is a single-path fragment. Then the cost in I/Os of executing \(P\) can be estimated as follows, depending on the shape of \(P_2\). All the cost formulas below do not include, unless explicit, the cost of writing the output to disk (because most of the times the output of an operator is pipelined to the next operator.)

- **Case 1**: \(P_2\) is a scan over a temporary table \(T\) storing the result of evaluating a plan \(P_2^\prime\) for \(F\). Then \(P\) is a *nested-scan join* between \(P_1\) and \(T\). Its total cost is:

\[
\text{cost}(P) = \text{cost}(P_1) + \text{cost}(P_2^\prime) + \text{cost}(\text{write } T \text{ to disk}) + \left[\frac{szpg(P_1)}{M}\right] \times \text{cost}(\text{read } T \text{ from disk})
\]

\[
= \text{cost}(P_1) + \text{cost}(P_2^\prime) + szpg(T) + \left[\frac{szpg(P_1)}{M}\right] \times szpg(T)
\]

where

\[
\begin{align*}
\text{szpg}(T) &= \left\lceil \frac{\text{card}(T) \times \text{sz(tuples of } T)}{p} \right\rceil, \\
\text{szpg}(P_1) &= \left\lceil \frac{\text{card}(P_1) \times \text{sz(tuples of } P_1)}{p} \right\rceil
\end{align*}
\]

Here \(\text{card}(T)\) and \(\text{card}(P_1)\) are estimated using the techniques described in section 6.2.2. The above cost formula assumes that \(P_1\) is read in blocks, each equal to the entire amount of memory \(M\) available at the current node. We ignore the memory needs for the output buffer or for the input buffer for reading in \(T\) (these would amount to 1 more page each). For \(\text{cost}(P_2^\prime)\) we need to consider three possible shapes for \(P_2^\prime\). These are the same as in the three subcases considered in Case 2 below. The cost evaluation of \(P_2^\prime\) follows then the same spirit as there.

- **Case 2**: \(P_2\) is the result of applying the UnfoldTemp transformation to \(T\) from above (i.e. pipelined execution of the plan for \(F\)). Here we have three subcases:

1. \(S\) is of set type. Then \(P\) performs a *nested-scan join* between \(P_1\) and \(S\). Its cost is:
\[
\text{cost}(P) = \text{cost}(P_1) + \left\lceil \frac{\text{szpg}(P_1)}{M} \right\rceil \times \text{szpg}(S)
\]
where \(\text{szpg}(S)\) is estimated by using the techniques of section 6.2.3 to compute the size of a schema element of (nested) set type.

2. \(S\) is of dictionary type and \(P_2\) is a plan that scans \(\text{dom\hspace{0.1em}S}\). Then \(P\) performs a nested scan join between \(P_1\) and \(\text{dom\hspace{0.1em}S}\) followed, for each resulting tuple, by a lookup operation into \(S\). For each lookup operation another scan over the result of the lookup is in general needed\(^{18}\). The cost of \(P\) is:

\[
\text{cost}(P) = \text{cost}(P_1) + \left\lceil \frac{\text{szpg}(P_1)}{M} \right\rceil \times \text{szpg}(\text{dom\hspace{0.1em}S})
\]

\[
+ \text{card}(P_1) \times \text{szpg}(S[k] \mid k \in \text{dom\hspace{0.1em}S})
\]

3. \(S\) is of dictionary type and \(P_2\) is the result of applying the RemoveScanDom transformation. \(P\) performs in this case, for each tuple in the result of \(P_1\), a lookup into \(S\), followed by a scan over the result of the lookup. The cost of \(P\) is:

\[
\text{cost}(P) = \text{cost}(P_1) + \text{card}(P_1) \times \text{szpg}(S[k] \mid k \in \text{dom\hspace{0.1em}S})
\]

where the last term is computed using the techniques of section 6.2.3.

Notice that in the last two cases, each entry in the dictionary is scanned (looked at) for each tuple coming from either \(P_1 \supset \text{dom\hspace{0.1em}S}\) or \(P_1\). Depending on the cardinalities of the two intermediate sets, each of the two plans can be better. The two alternatives correspond to a value-based join and, respectively, a pointer-based join between \(P_1\) and the dictionary \(S\). See also [SC90] for a more thorough analysis of the performances of the two join algorithms. Finally, we note that we did not consider the effect of buffering on the cost of accessing the pages storing entries of \(S\); we assume that a page is reread from disk, even if it was read before. In general, for more accurate estimation, to the last two formulas we must add another factor: Yao’s formula [Yao77].

All the above cost formulas naturally generalize cost formulas for block nested-loop joins ([SMK97, Ram98]) and navigation joins([SC90, GGT96]). It is not very difficult to generalize this cost model to take into account the case when \(F\) is a multiple-path fragment (additional nested scans for each single-path fragment are needed then), and for the case when the space of plans consists of all bushy trees instead of all left-deep trees. Also, it is possible to use hash-based join algorithms instead of nested-scans joins.

**Monotonicity of Cost.** The following is an important property of the cost model that we have implicitly made use of, so far, in our C&B-based approach to optimization.

**Definition 6.2.3** A cost model \(C\) is monotonic with respect to a class \(\mathcal{P}\) of physical plans, if for any two queries \(Q_1\) and \(Q_2\) such that \(Q_1\) is a subquery of \(Q_2\) we have that \(C(Q_1) \leq C(Q_2)\). Here \(C(Q)\) denotes the cost of the best physical plan in \(\mathcal{P}\) implementing \(Q\) (we have choice of any join algorithms in \(\mathcal{P}\) and any join orders).

The first advantage of having a monotonic cost model is that C&B-minimization can be performed independent of the cost. Cost is then used only at the end to choose among the minimal forms generated by the minimization algorithm. Since the cost model is monotonic, the best plan must correspond to a minimal form. Secondly, monotonicity of cost is the fundamental assumption on which the cost-based pruning variant of backchase, that we give in section 6.4, will be based. The main idea there is that, during the bottom-up exploration of a subquery \(Q_1\), if the cost of \(Q_1\) is higher than the best cost so far, then all the superqueries \(Q_2\) of \(Q_1\) will be pruned, due to monotonicity of cost.

Our cost model (and others, as well) is essentially monotonic if we make an important restriction: the class of physical plans allowed do not include lookup-based joins. For example, the join of two sets can be implemented \(^{18}\)If the entries in the dictionary \(S\) contain elements of set type.
by a nested scans join, a sort-merge join, a hash-join but not by an index-based join. We justify our statement
below and then show in section 6.4 how we can take care of lookup-based plans.

When plans implement only nested scan joins, the monotonicity property is satisfied if the memory available for
join is large enough (this is a common situation). Consider, for instance, the join of two relations \( Q_1 = R \bowtie S \).
According to our cost formula above (Case 2.1), the cost of the join is:

\[
\text{cost}(R \bowtie S) = \text{szpg}(R) + \left( \frac{\text{szpg}(R)}{M} \right) \times \text{szpg}(S)
\]

When \( M \) is very large this becomes simply the sum \( \text{szpg}(R) + \text{szpg}(S) \). In general, we can safely assume that the
cost of a nested scans join is linear. Therefore, any superquery \( Q_2 \) of \( Q_1 \) (with some additional relations entering
the join) that is implemented by a nested scans join will have a higher cost than \( Q_1 \) (more terms entering the
sum). And even when \( M \) is not very large, and the cost of \( Q_2 \) may become smaller than the cost of \( Q_1 \), the
difference in cost will not be very large. In that situation the optimizer, even though will not find the optimal
plan, will find a good enough plan.

A similar kind of linearity is true for hash-based implementations of join. We illustrate by considering the
hybrid-hash join algorithm [Sha86]. Assume two queries \( Q_1 = R \bowtie S \) and \( Q_2 = T \bowtie R \bowtie S \), such that \( Q_1 \) is a
subquery of \( Q_2 \). We show that if both \( Q_1 \) and \( Q_2 \) are implemented using the hybrid-hash join, then in the worst
case the cost of \( Q_2 \) is smaller than the cost of \( Q_1 \) by a margin that is not too big. From [Sha86] the cost of
hybrid-hash join is:

\[
\text{cost}(R \bowtie S) = b_R + b_S + 2(b_R + b_S)(1 - q)
\]

where \( R \) is assumed to be the outer relation and \( q \) is a factor that computes the fraction of \( R \) whose hash table
fits in the main memory. \( b_R \) and \( b_S \) denote the sizes of \( R \) and \( S \) in number of disk pages. The cost of \( Q_1 \) is then
no larger than \( 3(b_R + b_S) \) (this is the limit case when \( q = 0 \) and corresponds to the case when \( R \) and \( S \) are both
so large that a very small fraction of them can fit in the main memory). The best case for evaluating \( Q_2 \) is when
\( T \) and \( T \bowtie R \) are very small so that their hash tables can fit entirely in main memory. Then we can evaluate \( Q_2 
\)
using \( b_T + b_R I/Os \) to perform \( T \bowtie R \) plus \( b_T + b_R I/Os \) to perform \( (T \bowtie R) \bowtie S \). Thus the cost of \( Q_2 \) is at
best \( b_T + b_R + b_T + b_T \bowtie R \). If \( b_T \) and \( b_T \bowtie R \) are negligible, then the cost of \( Q_2 \) is three times smaller than the cost
of \( Q_1 \). Even in this best case - worst case situation the cost of \( Q_2 \) is in about the same order of magnitude as
the cost of \( Q_1 \). In common situations it is usually the other way around: the cost of \( Q_1 \) is smaller than the cost
of \( Q_2 \). A similar analysis can be carried out for sort-merge join.

However, when we consider plans that are based on dictionary lookup (i.e index-based joins), the monotonicity
of cost does not hold anymore. Indeed, consider a query \( Q \) scanning a dictionary domain \( \text{dom} \mathbb{K} \). If we add
another relation \( R \) in a join with \( Q \) we may obtain a superquery \( Q' \) for which the best plan avoids scanning
\( \text{dom} \mathbb{K} \) but performs a dictionary lookup instead (based on values that are coming from \( R \)). The plan for \( Q' \) may
be significantly cheaper; an order of magnitude is not unusual. See section 6.4 for a detailed example.

Thus, backhash minimization, as presented in Chapter 2, may omit lookup-based plans with good cost that do
not correspond to minimal PC queries. However, in section 6.4, we show how we can deal with such lookup-based
plans. There, we give an algorithm for bottom-up backhash extended with cost-based pruning, that is able to
consider, in a systematic way, lookup-based plans that do not correspond to minimal PC queries. The resulting
algorithm goes therefore beyond the backhash enumeration of minimal PC queries.

124
6.3 Global Dynamic Programming

Dynamic programming is used in traditional optimization (System R, for example) to find the best join ordering for a query. The algorithm avoids the repeated recomputation of the best join plan for any subset of the tables mentioned in the input query. Given a query joining relations \( R_1, \ldots, R_n \), the algorithm starts by building the best access plans for individual relations and stores them in a data structure for further use. Then it computes, and stores in the same data structure, the best plans for joining any subsets of 2 relations: \( \{ R_1, R_2 \}, \{ R_1, R_3 \}, \ldots \), using in the process the best plans for \( \{ R_1 \}, \{ R_2 \} \ldots \) After \( n \) stages, corresponding to all possible cardinalities of subsets of size \( n \), it produces the best join plan for the entire query.

We generalize the above dynamic programming in two ways:

- **Handle atomic fragments instead of flat relations.** For any subset of the atomic fragments in which the input query is decomposed, we will generate the best join sequence. The best plan for a subset \( \mathcal{F} = \{ F_1, \ldots, F_k, F_{k+1} \} \) of \( k + 1 \) atomic fragments will be computed by using the already computed best plans for subsets of size \( k \) of \( \mathcal{F} \). When adding a \( k+1 \)th fragment to the join plan corresponding to a subset of size \( k \), the algorithm considers three choices for block plan corresponding to the \( k + 1 \)th fragment:
  1. scan over a temporary table \( T_{k+1} \), or
  2. the result of applying UnfoldTemp to 1), or
  3. the result of applying RemoveScanDom (whenever possible) to 2)

Thus we are able to enumerate not only all join sequences but all possible choices of nested-loop joins and lookup-based joins\(^{19}\).

The above extension to the traditional dynamic programming algorithm will be called from now on the **local dynamic programming (LDP)** algorithm. The reasons for this choice of name will become apparent when we explain the second extension and the subsequent algorithm (which will be global).

- **Integrate dynamic programming with backchase.** Our goal is to enumerate and cost more than one query, in fact, many subqueries of the universal plan, during the backchase exploration. Since subqueries of the universal plan often share subsets of the atomic fragments in which the universal plan is decomposed, we want to be able to avoid the recomputation of the best plan for any such shared subset. Therefore what we need is a **global** data structure that remains active between the different calls to the dynamic programming algorithm corresponding to different subqueries of the universal plan.

The resulting dynamic programming algorithm, incorporating both extensions above, will be called from now on the **global dynamic programming algorithm (GDP)** algorithm. We explain next the details of GDP.

**OptPlan:** A shared data structure for multiple dynamic programming calls. We explain the use of the global data structure on two simple relational examples. Suppose that we have the following universal plan (a star query):

\[
(U) \ \text{select} \ \text{struct}(A = r.B) \\
\text{from} \ R, r, S, s, T, t, U, u \\
\text{where} \ r.A = s.A \ \text{and} \ t.A = t.A \ \text{and} \ u.A = u.A
\]

and assume that during the backchase we explore the following two subqueries:

\(^{19}\)The plan language and the dynamic programming algorithm can be easily extended to include other join methods as well. This is similar to traditional relational dynamic programming, where several join methods are considered at each step.
(S1) \( \text{select} \: \text{struct}(A = r.B) \)
    \( \text{from} \: R, r, S, s, T t \)
    \( \text{where} \: r.A = s.A \: \text{and} \: r.A = t.A \)

(S2) \( \text{select} \: \text{struct}(A = r.B) \)
    \( \text{from} \: R, r, S, s, u u \)
    \( \text{where} \: r.A = s.A \: \text{and} \: r.A = u.A \)

Suppose we use dynamic programming to find the best join ordering for \( S_1 \). This requires, among others, computing the best plan for the join between \( R \) and \( S \). The output for this plan will necessarily consist of \( r.A \) and \( r.B \): the first value is needed for further join with \( T \), while the second value is needed in the output. Now suppose that we use a second call to the dynamic programming algorithm to find the best join ordering for \( S_2 \). Incidentally, this requires computing the best join plan for \{ \( R, S \) \}, with the same output as in the first case. Thus, if the dynamic programming table for the first call is still accessible, we can avoid the recomputation of this plan.

However, as the next example shows, there may be situations in which different calls to the dynamic programming algorithm, corresponding to different subqueries of the universal plan, may compute plans for the same subset of tables, but with \textit{different} output.

Consider the following universal plan and two of its subqueries:

\( (U) \: \text{select} \: \text{struct}(A = r.B) \)
\( \text{from} \: R, r, S, s, T t, u u \)
\( \text{where} \: r.A = s.A \: \text{and} \: s.C = t.C \: \text{and} \: s.D = u.D \)

\( (S_1) \: \text{select} \: \text{struct}(A = r.B) \)
\( \text{from} \: R, r, S, s, T t \)
\( \text{where} \: r.A = s.A \: \text{and} \: s.C = t.C \)

\( (S_2) \: \text{select} \: \text{struct}(A = r.B) \)
\( \text{from} \: R, r, S, s, u u \)
\( \text{where} \: r.A = s.A \: \text{and} \: s.D = u.D \)

Finding the best join plan for \( S_1 \) requires finding the best plan for joining \{ \( R, S \) \} with output consisting of \( r.B \) (needed in the output) and \( s.C \) (needed for further join with \( T \)). On the other hand, finding the best join plan for \( S_2 \) requires finding the best plan for joining \{ \( R, S \) \} with output consisting of \( r.B \) (as before) and \( s.D \) (needed for further join with \( U \)). Thus, we have a different output in the second case. The two plans are \textit{not} the same (may have different cost, cardinality, and size). This shows that, as opposed to the case of dynamic programming for a single query, a global table that is common across several dynamic programming calls must store one entry for each pair: (subset of tables, output).

Let us call \texttt{OptPlan} the global data structure used across multiple calls to dynamic programming during backchase. In an implementation, \texttt{OptPlan} may consist of \( 2^n \) entries, where \( n \) is the number of atomic fragments in which the universal plan is decomposed. Each entry corresponds to a subset of atomic fragments. Before the first call, each entry is empty. Then, during the global dynamic programming, each entry will contain a set of plans, one for each output encountered. Later reuse of a plan requires a test for subset equality and an additional test for tuple equality\(^{20}\).

The global dynamic programming algorithm is given in figure 6.6. Figure 6.7 gives the pseudocode for \texttt{bestAccessPlan}(\( F, \text{plan} \), a procedure that is used by \texttt{GDP}. \texttt{bestAccessPlan} computes a new left-deep join tree by adjoining a local plan for the atomic fragment \( F \) on top of the already existent left-deep tree \texttt{plan}. The local plan for

\(^{20}\)In general, this test requires checking for membership in the same congruence class in the canonical database of the universal plan.

126
Algorithm 6.3.1 GDP\((Q, \text{OptPlan})\)

**Input:** PC query \(Q\), OptPlan data structure  
**Output:** Best physical plan for \(Q\)

1. plan = OptPlan.get\((Q)\);
2. if (plan != null) return plan;
3. compute atomic fragments for \(Q\): \(\mathcal{F} = \{F_1, \ldots, F_n\}\);
4. for \(i = 1\) to \(n\) do
5.   if (OptPlan.get\((F_i)\) != null) {
6.     plan = bestAccessPlan\((F_i, \text{null})\);
7.     OptPlan.put \((F_i, \text{plan})\);
8.   }
9. for \(i = 2\) to \(n\) do
10.   for all subsets \(X = \{F_{k_1}, \ldots, F_{k_i}\} \subseteq \mathcal{F}\) of cardinality \(i\) do
11.     minCost = Max, bestPlan = null;
12.     for \(j = 1\) to \(i\) do
13.       plan = bestAccessPlan\((F_{k_j}, \text{OptPlan.get}(\{X - F_{k_j}\}))\);
14.       cost = cost (plan);
15.       if (cost < minCost) {
16.         minCost = cost, bestPlan = plan;
17.       }
18.     OptPlan.put \((X, \text{bestPlan})\);
19. return OptPlan.get\((Q)\);

*This is a slight abuse of notation meaning the query obtained by joining the fragments in \(X - F_{k_j}\). The output of this query consists of all values needed in the rest of \(Q\).*

Figure 6.6: The GDP algorithm.

bestAccessPlan \((F, \text{plan})\):

1. compute an initial local plan for \(F\) and allocate a temp \(T\) for it;
2. compute initPlan = new left-deep tree with subtrees plan and \(T\);
3. compute bestPlan = initPlan, minCost = cost (initPlan);
4. for any application of UnfoldTemp and RemoveScanDom to \(T\) do
5.   for all orders between the single-path fragments of \(F\) do
6.     plan = transform initPlan accordingly;
7.     cost = cost (plan);
8.   if (cost < minCost) {
9.     minCost = cost, bestPlan = plan;
10.   };
11. return bestPlan;

Figure 6.7: bestAccessPlan procedure used by GDP.
$F$ is evaluated into a temporary table $T$, as described in section 6.1.3. bestAccessPlan tries then to apply UnfoldTemp and RemoveScanDom to $T$. It also tries all possible orderings between the single-path fragments of $F$. The best plan among all these choices is then returned.

**Note.** In the limit case when the input query $Q$ is the same query that was used to initialize OptPlan, the GDP algorithm becomes a local dynamic programming algorithm.

### 6.4 Bottom-Up Backchase with Cost-Based Pruning

Now we are ready to give the full details of the algorithm that combines the bottom-up backchase with cost-based pruning. The algorithm, shown in figure 6.8, is mainly a breadth-first, bottom-up, exploration of the subqueries of the universal plan $U$. At each step, if the subquery explored, $S$, has a cost that is larger than the cost of the best plan so far\footnote{initialized with the cost of the original query}, then all superqueries of $S$ are pruned (cost-based pruning). If the cost is smaller then the best cost so far and $S$ is equivalent to $U$ then we update the best plan to be the plan for $S$. In this case, also, all superqueries of $S$ are pruned (the Pruning Lemma, part 2, of chapter 4 applies). The cost and the plan for each subquery $S$ is computed by applying the global dynamic programming algorithm of the previous section. The dynamic programming is global with respect to $U$. In this way, for any combination of atomic fragments (having the same output) that occurs in more that one subquery of $U$, we compute only once the best plan for it.

**Note.** A simple variation of the algorithm implements the bottom-up backchase enumeration: we remove lines 8, 9, 10, 12, and we replace line 14 with **output** $S$.

**Pruning Strategy in the Presence of Dictionaries.** The algorithm in figure 6.8 is guaranteed to find the optimal plan when all schema elements are sets and the cost monotonicity requirement discussed in section 6.2.4 is satisfied. Unfortunately, when we have dictionaries, important plans based on the lookup operation may be missed! The culprit is the pruning strategy:

For instance, suppose that the algorithm explores at some point a subquery $Q$ of the universal plan, and $Q$ has a variable $k$ ranging over $\text{dom} M$, for some dictionary $M$. Also, suppose that every plan for $Q$ must perform a scan over $\text{dom} M$ (because no matter what join order we choose, $k$ cannot be equated to any other expression). Thus the dynamic programming algorithm yields a plan $P$ for $Q$ that accesses $M$ via a scan operation. Now, if the cost of $P$ is higher than $\minCost$ (line 9) or if $Q$ is a minimal, equivalent, PC query (line 11), all superqueries of $Q$ will be pruned. However, by adding some additional binding to $Q$ (preferably bound to range over a set of small cardinality), it may be that in the resulting superquery $Q'$ the best plan for accessing $M$ uses a lookup. This may happen if in the larger query $Q'$ the variable $k$ becomes equated to some expression not defined in $Q$ (but defined over $Q'$). Therefore $Q'$ may have a smaller cost than the cost of $Q$ (and $\minCost$) even though we added one more binding. If $Q'$ also happens to be equivalent to the universal plan, this means that the optimizer, as described above, misses one of the better plans.

Let us illustrate first with a concrete example, and then we will show how we can modify the above algorithm to take into account such lookup-based plans.

**Example 6.4.1** Recall the example 1.2.4 from section 1.2, with a logical schema with relations $R(A, B)$ and $S(B, C)$. The physical schema consists of $R$ and $S$ too (direct mapping!), as well as a materialized view $V = \Pi_A(R \bowtie S)$ and secondary indexes $I_R$ and $I_S$ on attributes $A$ and $B$ of $R$ and $S$, respectively. We want to optimize the logical query $Q = R \bowtie S$. 

---

128
Algorithm 6.4.1 (BottomUpFB+Prune)

Input:
Universal plan $U$ with variables $\mathcal{V} = \{1, \ldots, n\}$,
Set of constraints $D$, and
Original input PC query $Q^*$

Output:
Best physical plan equivalent to $Q$
1. initialize $\text{OptPlan}(U, n)$ for global dynamic programming;
2. compute $\text{bestPlan} = \text{DP}(Q, \text{OptPlan}), \min\text{Cost} = \text{cost}(\text{bestPlan});$
3. for $i = 1$ to $n$
4. for all subsets $\mathcal{X} \subseteq \mathcal{V}$ of cardinality $i$ do
5. if ($\mathcal{X}$ was pruned) continue with step 4;
6. compute $S$ the maximal subquery of $U$ induced by $\mathcal{X}$;
7. if (no such subquery exists) continue with step 4;
8. compute plan = $\text{DP}(S, \text{OptPlan}), \text{cost} = \text{cost}(\text{plan});$
9. if ($\text{cost} \geq \min\text{Cost}$) {
10. prune supersets of $\mathcal{X}$ and continue with step 4;
11. if ($\text{chase}_D(S) \subseteq U$) { // $S$ is equivalent to $Q$
12. $\min\text{Cost} = \text{cost}, \text{bestPlan} = \text{plan};$
13. prune supersets of $\mathcal{X}$;
14. output $\text{bestPlan};$

$^*U = \text{chase}_D(Q)$

Figure 6.8: Basic Bottom-Up Backchase Algorithm with Cost-based Pruning.
Q itself is a valid plan (as a nested scans join, modulo join-reordering). However, we want to take advantage of V and of the two indexes and find possible better plans. In fact, if V is a small relation, such a plan exists (the plan P' of example 1.2.4): scan V first, then use for each tuple in V the value of the A attribute to lookup in the index IR for R, then lookup in the index for IS for S.

Let us use the chase/backchase approach. Chasing first Q with constraints relating R and S with V, IR and IS produces the following universal plan:

\[ (U) \quad \text{select} \quad \text{struct}(A: r.A, B: s.B, C: s.C) \]
\[ \text{from} \quad V v, R r, S s, (\text{dom} IR[k], IR[k][r'], (\text{dom} IS[p], IS[p][s']) \]
\[ \text{where} \quad v.A = r.A \quad \text{and} \quad r.B = s.B \quad \text{and} \quad k = r.A \]
\[ \quad \text{and} \quad r'.r = r \quad \text{and} \quad p = s.B \quad \text{and} \quad s' = s \]

In the second phase, we explore, bottom-up, subqueries of U. One of this subqueries uses just the two indexes IR and IS:

\[ (Q_1) \quad \text{select} \quad \text{struct}(A: r'.A, B: s'.B, C: s'.C) \]
\[ \text{from} \quad \text{dom}(IR), IR[k], IR[k][r'], (\text{dom} IS[p], IS[p][s']) \]
\[ \text{where} \quad r'.B = p \]

One of the possible plans for Q1 is a plan P1 that scans dom IR and then using the value of r'.B looks-up into the index IS. The lookup-based access into IS is enabled by the fact that p is equated with r'.B. However, no such access is possible for IR: there is no expression that equates k. Let’s assume that P1 is the best plan for Q1, found by the dynamic programming algorithm. Q1 is also equivalent to U, therefore all of its superqueries are pruned. One of these superqueries is the following:

\[ (Q_2) \quad \text{select} \quad \text{struct}(A: r'.A, B: s'.B, C: s'.C) \]
\[ \text{from} \quad V v, (\text{dom} IR[k], IR[k][r'], (\text{dom} IS[p], IS[p][s']) \]
\[ \text{where} \quad v.A = r.A \quad \text{and} \quad r'.B = p \]

For this superquery, the variable k is now equated with v.A. Thus there exists a plan for Q2 that accesses IR via a lookup: the plan P' discussed above! However, P' will be missed by the bottom-up backchase algorithm as presented above.

We also have to remark that although Q2 above is not a minimal PC query, its physical plan P' is minimal, in the sense that no scan of it is redundant. This is easy to see if we express P' as a plan in the plan language of this chapter or, alternatively, as a (non PC) query using the non-failing lookup operation of Chapter 1:

\[ (P') \quad \text{select} \quad \text{struct}(A: r.A, B: s.B, C: s.C) \]
\[ \text{from} \quad V v, IR[v.A] r, IS[r.B] s \]

We will show next how a simple extension of algorithm 6.4.1 can find such minimal physical plans even though their corresponding translations as PC queries are not minimal.

The remedy is at our hand: all the information that we need is contained in the universal plan! Rather than stoping the search when the cost of a subquery is larger than minCost, we replace line 10 in the above algorithm by the following lines:

10.1. \( \mathcal{Y} = \emptyset; \)
10.2. for every dictionary \( \mathcal{M} \) that \( \text{plan} \) accesses via a scan do
10.3. \( \quad \text{let} \ k \ be \ the \ variable \ ranging \ over \ \text{dom} \mathcal{M}, \ and \ C = \ the \ congruence \ class \ of \ k \ in \ U \)
10.4. \( \quad \text{for all} \ \text{expressions} \ c(y) \ in \ \mathcal{C} \ such \ that \ y \not\in \mathcal{X} \cup \mathcal{Y} \) do;
10.5. \( \quad \mathcal{Y} = \mathcal{Y} \cup \{y\}; \)
10.6. prune supersets \( \mathcal{Z} \) of \( \mathcal{X} \) unless \( \mathcal{Z} \subseteq \mathcal{X} \cup \mathcal{Y} \);
10.7. continue with step 4;

Similarly, we replace line 13 with the same lines as above. Thus, the search doesn't always stop at subqueries with higher cost than \( \text{minCost} \), or equivalent to \( U \). As long as there are still dictionaries accessed via a scan and which have the possibility of being accessed via lookup by adding more variables, the search continues. But we only keep supersets that have a chance to transform scans into lookups.

The modified algorithm (we will call it \textbf{BottomUpFB+Prune}) is then able to find plans such as the one in the previous example. It goes therefore beyond the normal backchase minimization algorithm, which was looking only at plans corresponding to minimal PC queries. \textbf{BottomUpFB+Prune} is related to existing algorithms for answering/optimizing queries using sources with limited capabilities([FLMS99, YLUGM99]). Our algorithm is more general, since it automatically takes advantage of arbitrary constraints. However, it will be interesting as future work to see how \textbf{BottomUpFB+Prune} compares with the mentioned algorithms, in terms of optimization time.

**Stratification and cost-based pruning.** We can combine OQF fragmentation with cost-based pruning backchase to produce an optimizer (call it \textbf{OQF+Prune}) working as follows. It uses the OQF stratification, but the backchase procedure applied for each stratum is \textbf{BottomUpFB+Prune}. Thus, for each stratum, \textbf{OQF+Prune} finds the best physical plan (independently of the other strata). The best physical plan for the entire query is then found by joining, in the best possible way, using dynamic programming, the best physical plans for the individual strata. \textbf{OQF+Prune} is always complete if the physical schema does not have dictionaries. In the presence of dictionaries (indexes, for example) \textbf{OQF+Prune} is just a heuristic:

For instance, given one stratum \( i \), there may be some plan \( P_i \) accessing a dictionary via a scan. If \( P_i \) is not the best plan for that stratum (considered in isolation) it will be pruned. However when considered together with plans from other strata, \( P_i \) may be transformed into a plan \( P'_i \) that accesses the dictionary via a lookup. The global resulting plan may be the best one, but it is missed.

### 6.5 Experimental Results

The goal of this section is to compare the performance of several optimizers that are obtained by combining in various ways the backchase phase with the cost-based phase. We will look at the optimization time as the basic performance parameter. Also we measure the combined effect of optimization + execution time. An important difference among the various optimizers will be completeness. By completeness we mean the ability of finding the cheapest plan among all minimal PC queries that can be produced by a full backchase enumeration. (As we have already seen, this does not guarantee optimality in the presence of dictionaries: the best plan may not necessarily correspond to a minimal PC query. Then only \textbf{BottomUpFB+Prune} is able to find the additional plans that do not correspond to minimal PC queries.)

**Optimizers.** We group all optimizers that we consider into 4 categories:

1. **Full C&B.** These optimizers do not make use of stratification, therefore they have to consider the entire universal plan obtained by chasing the input query.
   
   (a) \textbf{TopDownFB}. This optimizer works in two stages. In the first stage (pure C&B), the normal C&B enumeration procedure, with a top-down backchase, is applied, in order to produce the set of candidate plans. No cost information is used in this stage. In the second stage (cost-based), for each candidate
plan we apply the dynamic programming algorithm to produce its best physical plan. The best 
physical plan overall is then the final result. **TopDownFB** is complete.

(b) **BottomUpFB.** This optimizer is the same as the previous one, with the only difference that the 
backchase is done bottom-up. **BottomUpFB** is complete, too.

2. **Full C&B + cost-based pruning.** We have only one in this category:

  (a) **BottomUpFB+Prune.** This optimizer combines the bottom-up backchase enumeration with cost-
      based pruning (as described in section 6.4). It does not need to generate all candidate plans because
      it takes advantage early of the cost information. **BottomUpFB+Prune** is complete, too\(^{22}\).
      
The above **BottomUpFB+Prune** uses global dynamic programming (GDP). For comparison, we
also consider the variant of **BottomUpFB+Prune** with local dynamic programming (LDP). In
Experiment 4 below we compare the two variants of **BottomUpFB+Prune**. However, for the rest
of the experiments we always consider the variant with GDP, since it is the faster one.

3. **Stratified.** The next two optimizers make use of stratification\(^{23}\), therefore are able to work on several
universal plans of smaller size. They only make use of cost information after they enumerate all the
candidate plans.

  (a) **OQF.** This optimizer works in two stages. In the first stage (pure OQF), the OQF enumeration (as
      described in the previous chapter) is applied to produce the set of candidate plans. The backchase
      procedure used for each stratum is the bottom-up enumeration one. No cost information is used in
      the first stage. We choose the bottom-up backchase enumeration instead of the top-down, because
      the bottom-up tends to perform slightly better, as we’ll see shortly. The second stage of **OQF** is the
cost-based stage and is the same as for **TopDownFB** and **BottomUpFB**. the cheapest plan among
all candidate plans is selected. **OQF** is always complete (whenever it applies).

  (b) **OCS.** This optimizer, like **OQF**, works in two stages. In the first stage (pure OCS), the OCS
      enumeration (as described in the previous chapter) is applied to produce the set of candidate plans
      (no cost information used). As with **OQF**, we choose for the first stage the bottom-up backchase
      enumeration instead of the top-down. The second stage of **OQF** is the cost-based stage and is the
      same as for **TopDownFB** and **BottomUpFB**. In general, **OCS** is just a heuristic. However, for the
      experiment in which we use it (EC3), it is complete.

4. **Stratified + cost-based pruning.** There is only one optimizer in this category, based on OQF stratifi-
cation. We do not have a method for combining OCS stratification with cost-based pruning.

  (a) **OQF+Prune.** It uses the OQF stratification, but the backchase procedure applied for each stratum
      is **BottomUpFB+Prune** (as discussed at the end of section 6.4. **OQF+Prune** is always complete
      if the physical schema does not have dictionaries. In the presence of dictionaries **OQF+Prune** is
      just a heuristic.

5. **Traditional.** Again, as before, there is only one optimizer that enters in this category:

  (a) **DP.** It consists of only one stage: the cost-based one. Without any chase or backchase, the best
      physical plan for the input query is obtained by applying the dynamic programming algorithm de-
scribed in section 6.3. **DP** is clearly not optimal, since it does not try to find any additional physical
      structures that the chase may discover. **DP** is the closest to a traditional query optimization algo-
rithm\(^{24}\). We use it for two reasons: first, as a baseline for comparison with the other, more sophistica-
ted optimizers; and, second, because the global version of it (as explained in section 6.3) is intensively
used by the other optimizers for costing (sub)queries.

\(^{22}\)But, as we have just discussed, it finds more plans than required by the above completeness criterion.

\(^{23}\)Whenever stratification is possible, see the previous chapter.

\(^{24}\)A traditional optimizer also takes indexes into account. In our approach, this task is integrated into the C&B phase.
Experiments. We perform six main experiments with which we test the performances of all the above optimizers. (Even not explicitly said below, in most of the experiments we measure the performance of the traditional optimizer DP, as well).

1. We compare BottomUpFB+Prune with TopDownFB and BottomUpFB in a configuration using a star query, materialized views and key constraints (recall EC2 from the previous chapter that the key constraints are necessary in order to find rewritings other than the input query) and show the significant benefits of cost-based pruning.

2. We add indexes to the previous configuration and again measure the performance of BottomUpFB+Prune in comparison to that of TopDownFB and BottomUpFB.

In both cases (views, and views + indexes), the experiments show that BottomUpFB+Prune outperforms the other two full optimizers, sometimes by several orders of magnitude, and clearly emerges as the practical optimizer among the three.

3. This experiment shows the benefits of using global dynamic programming (GDP) as opposed to local dynamic programming (LDP). The comparison there is between BottomUpFB+Prune with LDP and BottomUpFB+Prune with GDP.

4. We compare BottomUpFB+Prune with the two stratified OQF optimizers (with cost-based pruning and no cost), for 2-star queries, in the presence of views. This is a heavy setting, with large queries. There, OQF+Prune is the faster optimizer (better than DP, even!), while BottomUpFB+Prune and OQF have similar, and acceptable, performances. OQF+Prune, however, is not complete in the presence of dictionaries (indexes), and this may considerably reduce its value.

5. Next we compare BottomUpFB+Prune with the full optimizers and the stratified optimizer OCS in EC3 (from Chapter 5). OQF stratification doesn’t apply in this situation. The setting is used for measuring the performance in a OO context: navigation queries, inverse relationship constraints, access support relations (ASRs). Again, recall, from the previous chapter, that only by considering the inverse constraints, one can find rewritings that use the ASRs.

6. Finally, we show that, for the same 2-star configuration, the combined optimization + execution time\textsuperscript{25} of the optimized query can be significantly smaller than the execution time of the unoptimized query. Here, the optimizer that we use is either OQF+Prune or BottomUpFB+Prune.

All of the above optimizers were implemented in Java: about 20,000 lines of code, including fragmentation, translation into physical plans, cost-estimation, dynamic programming, pruning. This is on top of the implementation for the pure C&B (full top-down and stratified) enumeration described in Chapter 5. The implementation is not tuned for maximum performance, thus skewing the results against us. All the experiments have been realized on a dedicated commodity workstation (Pentium III, 500 MHz, Linux Red Hat 6.0, 256MB of RAM, 6.4GB of hard-drive). The optimization algorithms were run using IBM Java runtime environment for Linux (alpha version 1.1.8). The database management system used to execute queries is IBM DB2 version 6.1.0 for Linux (out-of-the-box configuration).

Experiment 1: Comparison of the full optimizers in the presence of views. We compare the three full optimizers in EC2 when no stratification is possible: for star-queries. Figure 6.9 shows the the performance (optimization time) as a function of the number of views, when the input query is of size 6 and, respectively, 7. (We measure the size of a query as the number of variables in its from clause). Recall from EC2 of Chapter 5 that a star query of size 6 joins one hub relation $R$ with five corner relations $S_i$, $i = 1..5$.

\textsuperscript{25}With DB2.
Figure 6.9: BottomUp Backchase with Cost-based Pruning vs Full Backchase.

More comprehensive numbers are shown in table 6.10. There \( q \) is the size of the star query, \( u \) is the size of the universal plan obtained after chasing with the constraints for all the relevant views\(^{20}\), while \( p \) is the number of the candidate plans. Besides the optimization time, we show for each of the three full optimizers the pruning ratio, that is the ratio between the number of the subqueries that are not explored during the backchase and the total number of the subqueries of the universal plan (the latter number is \( 2^u \)). The performance of each optimizer is highly influenced by its pruning ratio. For TopDownFB and BottomUpFB the pruning ratio is independent of the cost, and it depends only on the number of equivalent, respectively, non-equivalent subqueries (recall the two Pruning Lemmas for the two backchase strategies). For BottomUpFB + Prune the number of pruned subqueries is the number of subqueries that are not explored due to the Pruning Lemma (therefore not explored by BottomUpFB) plus the number of subqueries that are pruned because they have higher cost than the best cost so far. Therefore the pruning ratio for BottomUpFB + Prune is always higher than the pruning ratio for BottomUpFB.

Remarks:

- **BottomUpFB** performs in general better than TopDownFB even though TopDownFB has a higher pruning ratio. This is explained by the fact that TopDownFB explores the subqueries of larger size while BottomUpFB explores the subqueries of smaller size. The advantage of BottomUpFB becomes more substantial when the input query and the candidate plans have small size compared to the universal plan. (In practice we also expect not too large input queries, but relatively large universal plans). This trend is true for BottomUpFB + Prune as well. The difference is that BottomUpFB + Prune has even higher pruning ratio.

\(^{20}\)Therefore the number of relevant views is \( u - q \).
<table>
<thead>
<tr>
<th>q</th>
<th>u</th>
<th>p</th>
<th>TopDownFB</th>
<th>BottomUpFB</th>
<th>BottomUpFB+Prune</th>
<th>DP</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>2</td>
<td>1.43 (42.5%)</td>
<td>1.32 (4.6%)</td>
<td>1.1 (7.8%)</td>
<td>0.7</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>4</td>
<td>2.85 (27.4%)</td>
<td>2.38 (10%)</td>
<td>1.85 (15.6%)</td>
<td>0.7</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>7</td>
<td>14.8 (9%)</td>
<td>11.18 (21%)</td>
<td>2.85 (33.9%)</td>
<td>0.7</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>2</td>
<td>1.95 (57%)</td>
<td>1.7 (2.3%)</td>
<td>1.4 (6.25%)</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>4</td>
<td>4 (44%)</td>
<td>3.7 (5%)</td>
<td>2.1 (30.9%)</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>7</td>
<td>17.35 (23%)</td>
<td>13.65 (10.5%)</td>
<td>13.65 (10.5%)</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>13</td>
<td>108.1 (8.3%)</td>
<td>82.2 (19.9%)</td>
<td>6.4 (37.9%)</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 6.10: Optimization times (in seconds) and pruning ratio for the full optimizers. q is the query size (number of variables), u is the universal plan size, p is the total number of candidate plans produced by a full C&B enumeration.

- **BottomUpFB+Prune** clearly outperforms the other two full optimizers. When the input query and the universal plan become large the gain in performance can be of an order of magnitude. This is also reflected in the difference between the pruning ratios of **BottomUpFB+Prune** and **BottomUpFB** (e.g. 19.4% vs 9.9% for the entry [7, 11, 13] in Figure 6.10). We can conclude that, for star queries and views, **BottomUpFB+Prune** performs well on the entire range of tested queries.

Note. During the experiments, the cost information was kept fixed. However, at each size, we added to the input query a random number of selection conditions (in addition to the join conditions imposed by the star configuration), since the amount of cost-based pruning depends in general on the number of selections that the query may have. The measured times for **BottomUpFB+Prune** were then averaged over all queries generated, for each query size.

**Experiment 2: Comparison of the full optimizers in the presence of views and indexes.** Here we use star queries as in the previous experiment but for which we add indexes (primary, in the experiment). Indexes are added one by one, for fixed query size and number of views, for each of the corner relations of the star. The indexes are on the join attribute with the hub of the star. For example, for a query of size 6, a configuration point with 3 indexes means that the first three corner relations have indexes on the join attribute while the last two corner relations do not have indexes.

Figure 6.11 shows the the performance (optimization time) as a function of the number of indexes, when the input query is of size 6 and, the number of views is fixed: 2 and, respectively, 4. More comprehensive numbers are shown in the figure 6.12. There, q and v are the query size, and the number of views, respectively, while idx is the number of indexes.

Remarks:

- For the case when we have only 2 views, **BottomUpFB** and **TopDownFB** become rapidly inefficient: when the number of indexes is 3, their optimization time is over 1 minute. When the number of indexes
Figure 6.11: The performance of BottomUpFB+Prune in the presence of indexes.

is increased to 4 then the time is between 3 and 4 minutes (see also Figure 6.12). For the case of 4 views, both BottomUpFB and TopDownFB cannot be used anymore!

- On the other hand, BottomUpFB+Prune performs excellently for the case of 2 views, and relatively well for the case of 4 views. An average-sized configuration such as [6, 2, 4] has an optimization time of 20.2 seconds.

- The trend that we observed in the previous experiment is confirmed here: the more we increase the size of the universal plan (i.e. adding views, indexes), while the query size remains fixed, the pruning becomes more effective! This is reflected in the increasing values for the pruning ratio. Thus, BottomUpFB+Prune is probably best fit for the common case of not so large queries (up to 5-6 joins), but large universal plan (3-4 views, 5-6 indexes).

- Pruning is still quite effective despite the fact that BottomUpFB+Prune explores more subqueries of the search space than in a situation without dictionaries.

- In all cases, the best plan reported by TopDownFB and BottomUpFB is the same as the best plan reported by BottomUpFB+Prune. This is a situation (common, we believe) in which inspecting minimal PC queries is enough to find the best plan. In general, as discussed in section 6.4, minimality does not guarantee optimality, in the presence of dictionaries, and BottomUpFB+Prune is the only optimizer able to find lookup-based plans that do not correspond to minimal PC queries.

- Increasing the number of indexes increases the optimization time of BottomUpFB+Prune. However, the quality of the best plan found increases as well. In our experiment, the best plan is always the plan obtained in the case with the maximum number of indexes available to the optimizer.
For configuration point [6, 4, 4], the best plan (obtained in 53.2s), accesses the first three corner relations using the first two of the four views available, while the last two corner relations are accessed via two index lookup operations. On the other hand, for configuration point [6, 4, 0] the best plan (obtained in 6.4s) is forced to use all four views, yielding a significantly more expensive plan. The difference in optimization time is not small (tens of seconds) but it could be easily outweighed by the difference in execution time (tens of minutes or much more, since our query is quite large.). See also the last experiment showing the benefits of optimization when execution time is taken into account.

<table>
<thead>
<tr>
<th>q</th>
<th>v</th>
<th>idx</th>
<th>TopDownFB</th>
<th>BottomUpFB</th>
<th>BottomUpFB+Prune</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>2</td>
<td>0</td>
<td>3.56 (64.0%)</td>
<td>3.32 (5.0%)</td>
<td>2.1 (10.9%)</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>1</td>
<td>8.92 (59.9%)</td>
<td>7.65 (64%)</td>
<td>3.3 (13.3%)</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>24.9 (58.7%)</td>
<td>21.4 (7.1%)</td>
<td>5.7 (14.4%)</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
<td>68.35 (54.1%)</td>
<td>63.6 (8.5%)</td>
<td>10.7 (17.0%)</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>4</td>
<td>253.97 (35.7%)</td>
<td>201.3 (31.0%)</td>
<td>20.2 (36.1%)</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>34.7 (39.8%)</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>6.4 (37.9%)</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>10.1 (45.2%)</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>19.7 (48.3%)</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>31.7 (52.5%)</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>53.2 (55.2%)</td>
</tr>
</tbody>
</table>

Figure 6.12: Optimization times (sec) and pruning ratios in the presence of views and indexes. q is the query size, v is the number of relevant views, idx is the number of relevant indexes.

**Experiment 3: The effect of global dynamic programming on the performance of BottomUpFB+Prune.**
Here we use the same experimental configuration as before: star query, with materialized views and indexes. We compare the two variants of BottomUpFB+Prune: one using global dynamic programming (GDP), the other using local dynamic programming (LDP). Figure 6.13 shows the performance of the two optimizers for query size 6, with increasing number of indexes, in two cases: with 3 views and, respectively, with 4 views. We observe that the savings in optimization time become more significant when the number of indexes becomes larger (more than 20s for configuration point [6, 3, 5]).

As a general remark, cost-evaluation of subqueries is an important component of BottomUpFB+Prune, and the larger the query/universal plan is the more time is spent on dynamic programming (there are more calls to DP, and also its input becomes larger). Thus any improvement on the dynamic programming algorithm becomes more beneficial when the query/universal plan is larger, and the experiment shows this. For large queries, it will be probably interesting to see how the use of other methods for cost-evaluation of queries affect BottomUpFB+Prune: for instance, iterative dynamic programming [KS99].

**Experiment 4: Comparison of BottomUpFB+Prune with the stratified OQF optimizers.** Here we use experimental configuration EC2 (views and key constraints27) where the number of stars is fixed at 2. This is a heavy experiment in which both the input query and the equivalent rewritings are relatively large. The results are shown in figure 6.14.

27Recall from Example 1.2.3 that the key constraints on the hubs of the stars are needed in order to find equivalent rewritings.
Figure 6.13: The effect of global dynamic programming on the performance of BottomUpFB + Prune.

Figure 6.14: Comparison of BottomUpFB + Prune with the other optimizers, for 2-star queries.
Remarks:

- **BottomUpFB+Prune** and OQF have similar performances for all the configuration points, and are much better than TopDownFB and BottomUpFB. OQF takes advantage of stratification during enumeration of candidate plans. However, the explosion of candidate plans (16 for $|10,4|$) during the assembly and cost-evaluation phase makes OQF no faster than **BottomUpFB+Prune** which takes advantage of the cost information to avoid costing all candidate plans. The candidate plans are large, and costing each of them is significant in this experiment (see also the last remark).

- **OQF+Prune** is significantly faster than all the other optimizers, due to the fact that both stratification and cost-based pruning are used simultaneously. DP is slower than OQF+Prune; even though OQF+Prune uses DP (internally)! This is explained by the fact that, in the experiment, DP has the whole query as its input (size 10 for the last two points), while OQF+Prune divides first the query into 2 strata (size 5 each, for the last two points). Thus, for answering/optimizing queries in the presence of views only (no indexes or limited capability sources), OQF+Prune is definitely the best optimizer. It is an interesting question to find a hybrid between OQF and OQF+Prune that is optimal.

- The cost of DP becomes significant when the input query is large (14.6s for size 10). Therefore, the performance of **BottomUpFB+Prune**, which uses intensively DP, becomes ultimately affected by the performance of the DP algorithm (in addition to the exponential blow-up of the search space).

**Experiment 5: Comparison of BottomUpFB+Prune with the stratified OCS optimizer.** This experiment uses the OO configuration EC3, in which a chain query traversing $n$ classes is optimized in the presence of constraints describing inverse relationships (INVs) and in the presence of access support relations (ASRs). Comparative results for all applicable optimizers, for $n$ varying from 2 to 5, are shown in figures 6.15 and 6.16.

As before, TopDownFB and BottomUpFB are outperformed by BottomUpFB+Prune. While both TopDownFB and BottomUpFB cannot be used when $n$ is larger than 4, BottomUpFB+Prune works for $n = 5$ as well. We must remark that the configuration point $n = 5$ is significantly heavier than $n = 4$: the increase in the size of the universal plan is from 11 to 15, see figure 6.16, due to the increase in query size, number of INVs, and number of ASRs.

For small values of $n$, **BottomUpFB+Prune** is slightly better than OCS. However, when $n$ becomes large, OCS performs better. In fact, for $n$ larger than 5 (universal plan of size 18 or more!), OCS is the only choice that we are left with. This shows (again) that stratification is the only viable approach for large queries and universal plans.

**Experiment 6: Benefit of optimization.** In this experiment we measure the total query processing time: optimization time + execution time, and we show the benefits that can be obtained, in general, by applying optimization based on chase and backchase. For our results, we use the same configuration points that we used in Experiment 4 (2-star queries with materialized views and key constraints). We perform the measurements for two optimizers based on chase/backchase that are applicable: **BottomUpFB+Prune** and **OQF+Prune**. The optimization times (denoted by OptT) are the same as shown in figure 6.14. Using DB2, for each configuration point and for each of the two optimizers, we measure:

- Ext, the execution time of the original query (no views mentioned in the query$^{28}$)

$^{28}$Apparently, DB2 v6.1.0 can only make use of a limited class of materialized views, with bag semantics only. Thus, the original query is executed as it is, without using any of the views.
Figure 6.15: OO Configuration with INVs and ASRs (EC3).

<table>
<thead>
<tr>
<th>n</th>
<th>q</th>
<th>ASR</th>
<th>u</th>
<th>TopDownFB</th>
<th>BottomUpFB</th>
<th>BottomUpFB + Prune</th>
<th>OCS</th>
<th>DP</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>1.2 (18.6%)</td>
<td>1.1 (18.7%)</td>
<td>1.0 (29.2%)</td>
<td>1.15</td>
<td>0.75</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
<td>8</td>
<td>5.1 (26.9%)</td>
<td>3.0 (57.3%)</td>
<td>1.6 (63.7%)</td>
<td>2.4</td>
<td>1.14</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>1</td>
<td>11</td>
<td>39.1 (53.5%)</td>
<td>26.5 (27%)</td>
<td>4.2 (44.3%)</td>
<td>6.1</td>
<td>1.5</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>2</td>
<td>15</td>
<td>–</td>
<td>483.4 (37.6%)</td>
<td>31.4 (65.8%)</td>
<td>22.7</td>
<td>2.4</td>
</tr>
</tbody>
</table>

Figure 6.16: Optimization times (sec) and pruning ratios: OO configuration with INVs and ASRs (EC3). n is the number of classes that the input query traverses, q is the query size, ASR is the number of relevant ASRs, u is the size of the universal plan.
• **ExT**Opt, the execution time of the best\(^{29}\) query that the C&\(B\) based optimizer finds.

• **Reduction**, the reduction in processing time, calculated as:

\[
\text{Reduction} = \frac{\text{ExT}}{\text{ExT} \text{Opt} + \text{OptT}}
\]

We also measure (see figure 6.19) how much of the total processing time is spent on optimization and how much is spent on execution (of the optimized query).

**Dataset used.** The measurements were made on a medium size database with the following characteristics:

| \(|R_i|\) | \(|S_{i,j}|\) | \(\sigma(R_i \bowtie S_{i,j})\) | \(\sigma(R_i \bowtie R_{i+1})\) |
|---|---|---|---|
| 15,000 tuples | 15,000 tuples | 4\% | 2\% |

The views were materialized by creating and populating tables. The raw numbers for ExT and ExT\(\text{Opt}\) can be seen in figure 6.17. The values of Reduction for the two optimizers are shown in figure 6.18.

<table>
<thead>
<tr>
<th>q</th>
<th>ExT</th>
<th>views used</th>
<th>ExT(\text{Opt})</th>
<th>OptT</th>
<th>ExT(\text{Opt} + \text{OptT})</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>20min 40s</td>
<td>2</td>
<td>2min 16s</td>
<td>1.8s</td>
<td>2min 18s</td>
<td>9.0</td>
</tr>
<tr>
<td>8</td>
<td>24min 30s</td>
<td>2</td>
<td>5min 4.4s</td>
<td>7.3s</td>
<td>5min 5.1s</td>
<td>4.2</td>
</tr>
<tr>
<td>10</td>
<td>28min 10s</td>
<td>2</td>
<td>7min 52s</td>
<td>30s</td>
<td>8min 22s</td>
<td>3.4</td>
</tr>
</tbody>
</table>

Figure 6.17: Execution times: unoptimized ExT vs. optimized ExT\(\text{Opt}\). Also, optimization times and total processing times (BottomUpFB+Prune only). The last column shows the reduction factor (by how many) in total processing time (again, BottomUpFB+Prune only; for OQF+Prune the numbers are even higher).

**Remarks:**

• In all cases the best query reported by any of the two optimizers was the one that used the maximum number of views available.

• Since DB2 only accepts SQL as input, it is not the best physical plan obtained by our optimizer that is fed into DB2, but rather the physical query with the best physical plan. Thus, some of the decisions that are made by our optimizer may be cancelled by the decisions that the DB2 optimizer itself makes (regarding the join ordering, for example). Nonetheless, DB2 is forced to use the views that our optimizer chooses, and the benefits are significant.

• The execution time decreases dramatically by taking views into account (here, our optimizers consider the key constraints, as well). For instance, the execution time for the original query of size 8 is more than 24 minutes. By adding 2 materialized views and investing about 8s with BottomUpFB+Prune (or about 3.5s with OQF+Prune) we reduce the execution time down to 5.44 minutes. By adding 4 materialized views and investing about 20s with BottomUpFB+Prune (or about 4.5s with OQF+Prune) we reduce the execution time down to 19 seconds! Here, 20s and 4.5s are total optimization times and not time per plan as in Chapter 5. In the end, the total processing time is reduced by 37 times, when BottomUpFB+Prune is used, and by more than 60 times when OQF+Prune is used (see figure 6.18).

\(^{29}\) according to the cost information
Time Reduction (by how many) in Total Processing Time
(2 star queries)

Reduction Factor - logscale

BottomUpPrune
PruneOQF

Optimization Time vs. Total Processing Time
(BottomUpPrune - 2 star queries)

Execution time
Optimization Time

Figure 6.18: Reduction in total processing time, when BottomUpPrune, respectively, OQF+Prune, is used.

Figure 6.19: Optimization Time vs. Execution Time.
• For all the configuration points, the cost of optimization (either with \texttt{BottomUpFB+Prune} or with \texttt{OQF+Prune}) is quite small, when compared to the execution time of the original query. This is reflected in the large values for \texttt{Reduction}. We notice, in the experiment, that the more views are applicable the reduction is larger. This is in spite of the fact that more time is spent on optimization. In the end, the total processing time, which is the one that counts, is much shorter.

• The time spent on optimization, relative to total processing time, becomes significant. Figure 6.19 shows how the total processing time is divided into its two components: optimization and execution (of the optimized query). Only \texttt{BottomUpFB+Prune} is considered there. (For \texttt{OQF+Prune} the ratio between optimization and execution is smaller.) In general, we expect that, for any C&B optimizer, optimization time will be an important component of the total processing time (much more than in a traditional optimizer). Nonetheless, from a user point of view, this should be fine, as long as the total processing time is shorter than in the case of not doing any C&B optimization.

6.6 Conclusion

Based on the experimental results of the last section, we conclude the following. When no stratification is possible, \texttt{BottomUpFB+Prune} is clearly the best optimizer to use. Its improvement over \texttt{TopDownFB} and \texttt{BottomUpFB} is significant. When OQ stratification is possible and no dictionaries are present, \texttt{OQF+Prune} yields additional improvement. This optimizer is very useful for the particular, but important, case of optimizing queries with materialized views, since it offers a very good scalability in terms of both query size and number of views. However, in situations in which we have to consider indexes as well, \texttt{BottomUpFB+Prune} should be the preferred one, because \texttt{OQF+Prune} may prune the good plans. The other stratified optimizers do not bring a significant improvement over the performance of \texttt{BottomUpFB+Prune}, unless the universal plan is quite large. Finally, we have to remark that OCS and OQF stratification do not always apply, while \texttt{BottomUpFB+Prune} is always applicable.

In conclusion, \texttt{BottomUpFB+Prune} is the best optimizer to use in the majority of the situations that we have studied (universal plan size up to 15-16). However, when the queries are quite large, stratification should be employed (if applicable). At large queries/universal plans, even the dynamic programming algorithm becomes expensive, and stratification has the advantage of working with relatively smaller queries. \texttt{OQF+Prune} offers very good scalability for the important case of materialized views.

Figure 6.20 summarizes the main features of the C&B-based optimizers that we have studied.
<table>
<thead>
<tr>
<th>Category</th>
<th>Full</th>
<th>Full + cost pruning</th>
<th>Stratified</th>
<th>Stratified + cost pruning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimizer</td>
<td>TopDownFB, BottomUpFB</td>
<td>BottomUpFB+Prune</td>
<td>OQF, OCS</td>
<td>OQF+Prune</td>
</tr>
<tr>
<td>Strategy</td>
<td>Explore all equivalent (non-equivalent, respectively) subqueries. Pruning applies only due to syntactic minimality.</td>
<td>Uses cost to cut through the search space</td>
<td>Use stratification to divide the search space</td>
<td>Uses stratification together with cost-based pruning inside each stratum</td>
</tr>
<tr>
<td>Scalability</td>
<td>Can be used for univ. plans of size up to 9-10</td>
<td>Can be used for univ. plans of size up to 15-16</td>
<td>Can be used for large univ. plans if stratification applies. However, cost-evaluation of a large number of candidate plans can become prohibitive</td>
<td>Can be used for large univ. plans if stratification applies. No explosion in number of candidate plans occurs, because of cost-based pruning within each stratum</td>
</tr>
<tr>
<td>Completeness</td>
<td>Complete</td>
<td>Complete + Finds lookup-based plans that do not correspond to minimal PC queries</td>
<td>OQF is complete, OCS is a heuristic</td>
<td>Not complete, in the presence of dictionaries</td>
</tr>
<tr>
<td>When is the first plan produced</td>
<td>Only at the end (the best plan it finds)</td>
<td>Outputs better and better plans the more it runs (can be stopped early if a good enough plan is produced)</td>
<td>Only at the end (the best plan it finds)</td>
<td>Only at the end (the best plan it finds)</td>
</tr>
<tr>
<td>Applicability</td>
<td>Always applicable (even when no cost info is available)</td>
<td>Always applicable</td>
<td>Only when stratification is possible</td>
<td>Only when stratification is possible</td>
</tr>
</tbody>
</table>

Figure 6.20: Features of the C&B-based optimizers.
Chapter 7

Conclusion and Future Work

7.1 Related Work

There are many papers that discuss semantic query optimization for relational systems. An incomplete list includes [CGK+99, GGMR97, LS95, CGM90, SO89] and the references therein. The techniques most frequently used are [SO89] index introduction, join elimination, scan reduction, join introduction, predicate elimination and detection of empty answers. Of these, scan reduction, predicate elimination and empty answers use boolean and numeric bounds reasoning of a kind that we have left out of our optimizer for now. We have shown examples of index and join introduction in sections 1.2 and 2.5 and [GGMR97] contains a nice example of join introduction. The C&B technique covers index and join introduction and in fact extends them by trying to introduce any relevant physical access structure. The experiments with \textbf{EC2} and \textbf{EC3} are already more complex than the examples in sections 1.2 and 2.5 and [GGMR97]. It also covers join elimination (at the same time as tableau-like minimization) as part of subquery minimization during the backchase. The work that comes closest to ours in its theoretical underpinnings is [JCV84] where chasing with functional dependencies, tableau minimization and join elimination with referential integrity constraints are used. Surprisingly, very few experimental results are actually reported in these papers. [SO89] contains one experiment each for index introduction and join elimination, both with queries and schemas of lesser complexity than what we have considered. [CGK+99] reports on join elimination in star queries that are still less complex than our experiments with \textbf{EC2}.

Examples of SQO for OO systems appear in [SZ89a, CD92, Ch91, BK93, FM95b, FM95a, GGMR97, CZ98]. Use of referential integrity constraints to eliminate dependent joins is implicit in [JWKL90, CD92, KM90a, KM90b]. A general framework for SQO using rewrite rules expressed using OQL appears in [FR96, Flo96].

Techniques for using materialized views in query optimization are discussed in [YL87, TSI94, CKPS95, Flo96, FR96, TS96, Ba98]. A survey of the area appears in [Lev]. For us, the most important work here is probably that of [LMSS95] in which a finite search space for rewritings in the context of answering queries with materialized conjunctive views was found. There it was proved that any "minimal" rewriting is bounded in size by the size of the original query, thus providing a complete procedure for enumerating minimal rewritings. From our perspective, the work on join indexes [Val87] and precomputed access support relations [KM90a, KM90b] belongs here too. The general problem is forced by data independence: how to reformulate a query written against a "user"-level schema into a plan that also/only uses physical access structures and materialized views efficiently. A related topic is optimizing queries in the presence of data sources with limited access capabilities [RSU95, LRO96, FLMS99]. We are able to model such sources by using dictionaries and our \textbf{BottomUpFB+Prune} algorithm can be successfully used for such optimization (in the presence of additional semantic or physical constraints, as well!). However, a detailed comparison, at the physical plan level as well as in terms of performance time, with
the work in [FLMS99] is needed.

Recent work concerned with scalable algorithms for answering queries using views appears in [PL00]. Their Minicon algorithm uses a technique slightly related to our OQF fragmentation. One main difference is that their stratification has a finer granularity (and therefore the potential of having a better scalability). For example, in our EC2 configuration, Minicon further fragments each star, while OQF doesn’t. A consequence of this is, also, that the resulting rewritings of Minicon are not always minimal, therefore additional elimination of redundant goals may be needed. In contrast, OQF always produces minimal rewritings. A more detailed comparison, based on experiments, between the two approaches is needed.

The GMAP approach [TSI94, TSI96] works with a special case of conjunctive queries (PSJ queries). In contrast to the query plans obtained by our rewriting process, the output of the GMAP rewriting is a family of plans represented by a PSJ query. The burden of choosing a specific plan is shifted on the next phase of the optimizer. The core algorithm is exponential but the restriction to PSJ is used to provide polynomial algorithms for the steps of checking relevance of views and checking a restricted form of query equivalence. Both checks are made more flexible by taking certain restricted integrity constraints into account. However, the results we report here on using the chase show that there is no measurable practical benefit from all these restrictions. In the end, the exponential behavior of the GMAP algorithm and the difficulties we had to resolve for the backchase phase are closely related.

Our experiments include schemas, views and queries of significantly bigger complexity than those reported in [YL87, TSI94, TSI96, CKPS95]. These experiments show that using views can be done and in the case of [TSI94, TSI96] that it can produce faster plans. But [YL87] measures only optimization time and [TSI94, TSI96] does not separate the cost of the optimization itself, so they do not offer any numbers that we can compare with our figures time reduction (sections 5.4.4 and 6.5). [CKPS95] shows a very good behavior of the optimization time as a function of plans produced, but cannot be compared with our figures because the bag semantics they use restricts variable mappings to isomorphisms thus greatly reducing the search space.

The idea of representing constraints as equivalences between boolean-valued (OQL actually) queries already appears in [FRV96, Flo96]. We intend to make a comprehensive study of the algorithms presented in [Flo96]. The chase/backchase technique and the monad algebra laws given in [LT97, PT99, PT98] prove almost the entire variety of proposed algebraic query equivalences beginning with the standard relational algebraic ones, and including [SZ89a, SZ89b], [CD92, Chu91, FM95b, FM95a] and the very comprehensive work by Beeri and Kornatzky [BK93]. Our PC queries are less general than COQL queries [L97], by not allowing alternations of conditionals and BigU. However we are more general in other ways, by incorporating dictionaries and considering constraints. Containment of PC queries is in NP while a double exponential upper bound is provided for containment of COQL queries. In [Bl87] it is shown that containment of conjunctive queries for the Verso complex value model and algebra is reducible to the relational case. Other studies include semantic query optimization for unions of conjunctive queries [CM98], containment under Datalog-expressible constraints and views [DS96], and containment of non-recursive Datalog queries with regular expression atoms under a rich class of constraints [CGL98]. We are not aware of any extension of the chase to complex values and oodb models. Hara and Davidson [HD99] provide a complete intrinsic axiomatization of generalized functional dependencies for complex value schemas without empty sets. [BF99] examines the un/decidability of logical implication for path constraints in various classes of co-typed semistructured models. The maps of [ALPR91], the treatment of object types in [BK93] and in [DHP97], that of views in [dSLDA94], and that of arrays in [LMW96] are related to our use of dictionaries. An important difference is made by the operations on dictionaries used here.

Comparison with rule-based optimizers. The C&B strategy is extensible in the sense that one can add constraints to the logical and/or physical schema and the optimizer need not be modified. This extensibility is in the same spirit with the extensibility of rule-based systems [CZ96, GCD+94, HFLPS9]. However, in their case, one has to add rules, possibly with code (thus complex), to extend the capabilities of the optimizer. Extensibility in our case means just the addition of new constraints and is thus at a higher-level of abstraction.
(easier to use). Moreover, the C&B rewriting is able to take into account not only the constraints that are stated in the schema but also their logical consequences. This is in contrast with rule-based systems in which the rewriting is performed according to only the rules that are explicitly stated.

7.2 Summary of Contributions

We have proposed a new optimization framework integrating in an uniform way fundamental techniques such as semantic optimization and physical data independence, previously considered in isolation. The fundamental concept that we use to link these techniques is that of a constraint. By capturing physical access structures with constraints having the same syntactic form as the semantic ones, semantic optimization and physical data independence optimization are reduced to rewriting with chase and backchase. The search space for rewritings is the universal plan obtained by chasing the input query with constraints from the logical schema and constraints describing the physical schema definitions. Then minimal equivalent subqueries of the universal plan are enumerated via backchase.

We also use the chase as the procedure for checking equivalence of the explored rewritings with the original query. Our theoretical results show that the chase is a complete proof procedure for equivalence of path-conjunctive queries, a language that generalizes to nested sets and OO classes the language of relational conjunctive queries. We also show that other classical results from the relational theory of conjunctive queries generalize to path-conjunctive queries: NP-completeness of containment (under all instances), decidability of containment (under full dependencies) by chase, confluence of chase (for full dependencies). In addition, a completeness result of a different nature is given: when limiting the physical schema to path-conjunctive materialized views and in the absence of logical constraints, the universal plan is a complete search space for minimal rewritings.

Our experimental results are promising. The chase itself is very efficient. For the backchase, in the case when no cost information is available, we had to implement several stratification techniques that are shown to make the whole approach scalable and practical. For the case when cost is available, we showed that by mixing the backchase phase with cost evaluation, the overall performance improves significantly. Therefore, cost is very important for the efficiency of a C&B optimizer. The C&B with cost-based pruning performs well in many common situations even when no stratification is applicable. The whole approach becomes then even more practical and worthwhile. Further mixing of stratification and cost-based pruning yields additional improvement for the case of path-conjunctive materialized views. For that case, such mixing offers a very good scalability with the query size and the number of views.

We find the technique very valuable when only the presence of semantic integrity constraints enables the use of physical access structures or materialized views. The total processing time when C&B optimization is employed can become significantly smaller in such situations (in spite of the fact that the amount of time spent on optimization, relative to total processing time, is more significant than when traditional optimization is used).

7.3 Limitations of Our Approach

- **PC limitations.** The following limitations are a consequence of the PC restriction:
  - We do not address the problems resulting specifically from the nesting in the select clause. Our prototype can have such nested queries as input but the inner queries and the outer query are handled independently, i.e. we do not have any optimization techniques yet that work across nested queries. A particular and important case of such queries are queries using the group-by operator.
  - We did not address yet the problem of handling union of queries or of queries with disjunctions in the
where clause. The only optimizations that we do right now are ones that optimize individually each query within a union. Handling union in a more comprehensive way would make our framework more amenable to data integration applications and distributed query optimization. We plan to add such features to our current system. We also have in mind extensions to situations in which physical access structures contain just a subset of their definition (like in Information Manifold [LRO96]) in which case the question of finding maximally contained answers becomes relevant. It would be interesting to see whether the chase based techniques can be used successfully in such a context.

- We did not include any negation in the queries and constraints considered in the theory and implementation of C&B. This is in general a major difficulty for results such as completeness and decidability. In practice, the optimizer could handle negation by identifying the positive fragments of queries and optimizing them in isolation.

- **Additional limitations.**

  - Our entire approach based on chase and backchase is valid under a set semantics assumption. Under bag semantics, the resulting rewritings may have different number of duplicates. This difference becomes particularly important when aggregates for bags, such as count and sum, are further applied to such rewritings.

  - There are still many physical access plans inexpressible in our framework, in particular plans that use sorting and take advantage of order. By extending the physical data model to include lists or sequences it might be possible to capture in some substantial way algorithms with sorted values.

### 7.4 Future Work Items

- Extend the PC language for queries and constraints to include union and disjunction. This would make the C&B technique applicable for optimization of queries over distributed data, as discussed in the previous section.

- Handle bag semantics. As we have seen, the C&B approach works in its actual form only for queries with set semantics. For bag semantics, one must reason about the number of duplicates that are additionally generated at each chase step. Conceivably one could extend the chase in this direction, but it is not obvious to how extent this can be done, or whether it is the right approach. Reasoning about queries under bag semantics (and their aggregates) has foundations based on isomorphisms, different from the ones on which reasoning about queries under set semantics is based (homomorphisms). Reconciling the two directions of research is an interesting and important open problem.

- Further explore classes of applications in which the C&B method can make a significant impact. We have in mind environments in which logical constraints are frequent and/or there is a large variety of physical access structures describable through constraints. Moreover, optimization must be essential for such systems, e.g. large area networks in which the cost of unoptimized queries could be prohibitive. We believe that future performant data integration systems will be in request for advanced optimization tools such as the C&B method.

- Theoretical study of constraint interaction (extensions to OQF, OCS). The two stratification techniques introduced here are a first promising step in the direction of a deeper understanding of how the interference of constraints affects the chase/backchase rewrites. This is an attractive theoretical problem which we believe to be more tractable than the study of interference of rules in arbitrary rewrite systems.

- Investigate one remaining open question regarding the completeness of the universal plan: does the universal plan remain complete when considering equivalence under arbitrary EPCDs, assuming that the chase terminates? This would be a nice strengthening of theorem 4.2.3.
• A language for guiding in a rule-based manner the alternative plans that the optimizer generates. In our implementation, we have already taken a small step towards this direction by being able to express both OQF and OCS strategies in a high-level language that essentially manipulates lists of plans, lists of constraints, and query fragments. We intend to investigate how to add primitives able to deal with cost, in a parametric way, and high-level declarative rules to express the various search strategies. The final goal of this is that one is able then to design, fine-tune, modify and extend, in a high-level language, a C&B based optimizer.

• Investigate other decidable classes for EPCD implication and PC query containment/equivalence. One direction here is to try to generalize to our context the special classes of dependencies and queries for which implication/containment was proven decidable in the relational case [JK84, CKV90].
Bibliography


