Physical Data Independence, Constraints, and Optimization with Universal Plans

Alin Deutsch Lucian Popa Val Tannen *

University of Pennsylvania

Abstract

We present an optimization method and algorithm designed for three objectives: physical data independence, semantic optimization, and generalized tableau minimization. The method relies on generalized forms of chase and "backchase" with constraints (dependencies). By using dictionaries (finite functions) in physical schemas we can capture with constraints useful access structures such as indexes, materialized views, source capabilities, access support relations, gmaps, etc.

The search space for query plans is defined and enumerated in a novel manner: the chase phase rewrites the original query into a "universal" plan that integrates all the access structures and alternative pathways that are allowed by applicable constraints. Then, the backchase phase produces optimal plans by eliminating various combinations of redundancies, again according to constraints.

This method is applicable (sound) to a large class of queries, physical access structures, and semantic constraints. We prove that it is in fact complete for "path-conjunctive" queries and views with complex objects, classes and dictionaries, going beyond previous theoretical work on processing queries using materialized views.

1 Introduction

Physical data independence strives to free the query formulation process from needing to know the complex techniques that make the implementation efficient. This is a very desirable property for traditional DBMS and an essential one for information integration systems where the implementations are distributed and hidden. However, traditional DBMS still need techniques for a more radical decoupling of the logical schema from the physical implementation, while in information integration systems most difficulties come from heterogeneity.

There have been several research efforts investigating physical data independence as the central issue [45, 20] or investigating closely related problems [48, 16, 27, 15, 30, 39, 38]. All of them recognize physical data independence as an optimization problem: rewrite a query $Q(\Lambda)$ written against a *logical* schema Λ into an equivalent query *plan* $Q'(\Phi)$ written against a physical schema Φ, given a semantic relationship between Λ and Φ. The question is how to define, broadly but precisely, this relationship and what meaning to give to "equivalent". There are two main approaches to this (see figure 1). The first one is to assume an *abstraction mapping* $\mathcal{A}$ that expresses the instances of the logical schema Λ in terms of those of the physical schema Φ and then

$$\text{define} \quad Q' \stackrel{\text{def}}{=} Q \circ \mathcal{A}$$

and the second one is to assume an *implementation mapping* from Λ to Φ, then

$$\text{solve} \quad X \circ \mathcal{I} =_\Lambda Q \quad \text{for } X \quad \text{then define} \quad Q' \stackrel{\text{def}}{=} X$$

(Here $=_\Lambda$ means equality in the presence of the constraints of the logical schema Λ). The abstraction mapping approach is the one taken in [20], while the implementation mapping approach is the one taken in [45] and "solving for X" above is related to what is often called "answering queries using views" [30]. The second approach is mathematically and computationally harder but it has a clear advantage from the optimization perspective: the equation $X \circ \mathcal{I} =_\Lambda Q$ typically has more than one solution, even more

*Contact: val@cis.upenn.edu, http://db.cis.upenn.edu

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

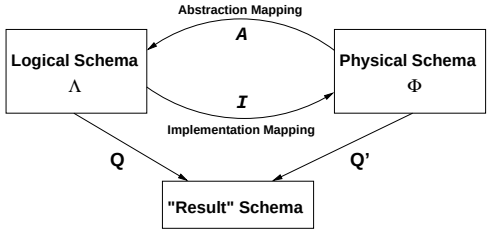

Figure 1: Logical and Physical Schema: two approaches towards rewriting

so because it takes into consideration the *constraints* of the logical schema Λ. In this paper we also take the second approach, but in a richer data model.

The physical data model Both [45] and [20] have some special constructs and types for representing physical structures but the operations on them that can be used in a query plan (e.g., joins or comprehensions) do not explicitly distinguish them from relations/complex values. It is assumed implicitly that the query engine will evaluate the joins and comprehensions over these special constructs in way that takes advantage of their physical efficiency. In constrast, we represent such structures explicitely, mainly using *dictionary* data structures (functions with a finite domain expressible in the language). This is a construct that reflects directly the efficiency of its representation through a fast lookup operation that appears in query plans. It turns out that dictionaries represent in a natural fashion physical structures such as primary and secondary indexes, extent-based representations of OO classes, join indexes [46], path indexes [34], access support relations [28], gmaps [45], etc. The physical level is represented just like the logical level is, with a typed data definition language and with constraints.

Constraints In a previous paper [37] we have generalized the classical relational tableau chase procedure [9] to work for the object-oriented model and dictionaries and for dependencies that capture a large class of semantic constraints including referential integrity constraints, inverse relationships, nested functional dependencies, etc. Moreover, we have shown that classical tableau minimization [14, 5] can be generalized correspondingly, as chasing with "trivial" (always true) constraints [1] In this paper we show that the elements of the implementation mapping (physical access structures, materialized views, etc.) are uniformly captured by the same kind of constraints and that we can use the chase (forwards and backwards) to find the solutions of the equation $X \circ \mathcal{I} =_\Lambda Q$ mentioned above.

Universal plans The constraints that capture the implementation mapping are of two kinds. The first kind apply the chase to the original query introducing explicitly the physical schema structures. Some semantic constraints work in the same way introducing structures that are alternatives to the ones mentioned in the original query. Chasing with these constraints [2] results in a query *plan* that we call *universal* because it is an amalgam of all the query plans allowed by the constraints. In a second phase we chase *backwards* from the universal plan trying to simplify the plan by removing structures, in particular some or all of the structures mentioned in the original query. The soundness of each such *backchase* step relies again on a constraint and we must test if this constraint is implied by the existing ones. This is where the second kind of constraints capturing the implementation mapping are used. This is also where we perform minimization, by testing for trivial constraints.

Applications An important contribution of this work is the systematic procedure for considering *all* alternate plans enabled by indexes and other physical access structures. Conventional relational optimization methods have long relied on ad-hoc heuristics for introducing indexes into a plan. Gmaps [45] have been proposed as an alternative but this work goes beyond gmaps, while for object-oriented data independence it goes beyond the approach of [28]. In fact, we have originally been motivated by our interest in distributed, mediator-based systems [47] for information integration, where it turns out that the techniques presented in [15, 30, 39, 38] are neither general enough nor flexible enough to be adapted to the problems we wish to solve. Moreover, we present our technique in a form that is easy to integrate in the rule-based paradigm [17], and easy to combine with conventional optimization techniques [41] such as selection pushing and join reordering.

Theoretical aspects We prove that our method is complete, i.e., finds the query plans that are minimal in a precise sense, for *path-conjunctive*(PC) queries and physical access structures (implementation mappings). An important restriction is that no constraints beyond those describing the implementation mappings are allowed. Still, PC queries and PC physical structures are more general and expressive than those considered in previous work. The main result of [30] is a particular case of ours.

About the language Our understanding of these results started with a different formalism [3] than the one used in this paper and in fact an efficient internal representation of the queries would be different yet (see [6]). However, to facilitate the presentation, we

[1] In fact, [37] applies the chase to deciding query containment and equivalence under constraints, to constraint derivation and to constraints holding in views.

[2] See [6] for termination of this process

[3] One in which it was easier to see the interaction between queries and constraints and the equivalence laws that govern it [37]

use throughout this paper the well-known syntax of ODMG/ODL and ODMG/OQL [12] (extended with a few constructs) for both logical and physical schema and queries. ODL already has a type of dictionaries $\mathsf{Dict}\langle T_1, T_2\rangle$, with keys of type T_1 and T_2 of type T_2, and OQL already has $M[k]$, the **lookup** operation that returns the entry corresponding to the key k in the dictionary M, provided that M is defined [4] 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 physical operation by $\mathtt{M}[\![k]\!]$. To this we add the operation $\underline{\mathsf{dom}}\, M$ that returns the **domain** of the dictionary M, i.e., the set of keys for which M is defined and a dictionary construction operation in section 2.

```
Proj: Set<Struct{     class Dept
 string PName;         (extent depts  key DName){
 string CustName;     attribute string DName;
 string PDept;        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 2: The `Proj-Dept` schema in extended ODMG

An example; logical schema and query Consider the *logical* schema in figure 2. 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` and a relation `Proj`. The schema has referential integrity (RIC), inverse relationship, and key constraints whose meaning can be specified by the following assertions.

(RIC1) $\forall(d \in \mathtt{depts})\ \forall(s \in d.\mathtt{DProjs})$
$\exists(p \in \mathtt{Proj})\ s = p.\mathtt{PName}$

(RIC2) $\forall(p \in \mathtt{Proj})\ \exists(d \in \mathtt{depts})\ p.\mathtt{PDept} = d.\mathtt{DName}$

(INV1) $\forall(d \in \mathtt{depts})\ \forall(s \in d.\mathtt{DProjs})\ \forall(p \in \mathtt{Proj})$
$(\ s = p.\mathtt{PName}\ \Rightarrow\ p.\mathtt{PDept} = d.\mathtt{DName}\ )$

(INV2) $\forall(p \in \mathtt{Proj})\ \forall(d \in \mathtt{depts})$
$(\ p.\mathtt{PDept} = d.\mathtt{DName}\ \Rightarrow$
$\exists(s \in d.\mathtt{DProjs})\ p.\mathtt{PName} = s\ )$

(KEY1) $\forall(d \in \mathtt{depts})\ \forall(d' \in \mathtt{depts})$
$(\ d.\mathtt{DName} = d'.\mathtt{DName}\ \Rightarrow\ d = d'\ )$

(KEY2) $\forall(p \in \mathtt{Proj})\ \forall(p' \in \mathtt{Proj})$
$(\ p.\mathtt{PName} = p'.\mathtt{PName}\ \Rightarrow\ p = p'\ )$

[4] Otherwise, lookup will fail. We will be careful to avoid this in the case of path-conjunctive queries, see section 5.

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

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

We deal only with set semantics in this paper, thus we omit writing the keyword $\underline{\mathtt{distinct}}$ from now on.

Example continued; physical schema 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` in figure 3; 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 [1]. The syntax of queries and that of query plans are very close: for example, if d is an oid in `depts` the implicit dereferencing in d.`DName` corresponds to the dictionary lookup in `Dept`$[d]$.`DName`. The relation `Proj`, stored as a table (a set of records), is also part of the physical schema, who therefore is not disjoint from the logical; 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 `Proj` and a secondary index `SI` on `CustName` of relation `Proj`(we could have also added an index between the key `DName` and the extent of `Dept` but we don't need it for the example). Both indexes are represented by dictionaries (see figure 3). For example, `I`$[s]$ returns the record r in `Proj` such that $r.\mathtt{PName} = s$. Similarly, `SI`$[c]$ gives back the set of records[5] r in `Proj` such that $r.\mathtt{CustName} = c$. Finally, the physical schema materializes the physical access structure defined by:

(JI) $\underline{\mathtt{select}}$ $\underline{\mathtt{struct}}(\mathtt{DOID}: d,\ \mathtt{PN}: p.\mathtt{PName})$
$\underline{\mathtt{from}}$ depts d, $d.\mathtt{DProjs}\ s$, $\mathtt{Proj}\ p$
$\underline{\mathtt{where}}$ $s = p.\mathtt{PName}$

Note that `JI` is both a generalized access support relation [28] and a generalized join index [46] since it involves a relation and a class.

Example continued; query plans With this physical schema, with the implementation mapping understood from the partly informal discussion above,

[5] 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 3: The physical schema

and especially with the constraints specified in the logical schema, we give four examples of query plans for the query Q we saw earlier.

(P_0) select struct(PN : s, PB : p.Budg,
DN : Dept[d].DName)
from dom Dept d, Dept[d].DProjs s, Proj p
where $s = p$.PName and
p.CustName = "CitiBank"

(P_1) select struct(PN : p.PName, PB : p.Budg,
DN : p.PDept)
from Proj p
where p.CustName = "CitiBank"

(P_2) select struct(PN : p.PName, PB : p.Budg,
DN : p.PDept)
from SI [["CitiBank"]] p

(P_3) select struct(PN : j.PN, PB : I[j.PN].Budg,
DN : Dept[j.DOID].DName)
from JI j
where I[j.PN].CustName = "CitiBank"

P_0 just introduces the representation of the class as a dictionary and its cost is essentially that of Q, but the other three are potentially significantly better. Depending on the cost model (especially in a distributed heterogeneous system), either one of P_1, P_2, and P_3 may be cheaper than the other two. As we shall see, although they are quite different in nature, our optimization algorithm generates all three.

Overview of the remainder of the paper. In section 2 we describe how we model with constraints physical structures such as primary and secondary indexes, materialized views, access support relations, join indexes, and gmaps. Section 3 presents our optimization algorithm. In section 4 we give two examples of relational scenarios, one on index access paths and one on using materialized views. The completeness results are in section 5. Related work is discussed in section 6.

2 Physical Structures as Constraints

We show here how typical physical access structures captured by constraints. For illustration, we also wish to be able to write down implementation mappings involving dictionaries. OQL does not have an operation that constructs a dictionary so we extend it with the following syntax dict x in Q $\Rightarrow$ $Q'(x)$ denotes the dictionary with domain Q and that associates to an arbitrary key x the entry $Q'(x)$. The notation $Q'(x)$ reflects the fact that Q' is an expression in which the variable x may occur free.

Indexes and classes The operation we just introduced allows us to define explicitly *primary* and *secondary indexes* such as I and SI:

$$\mathtt{I} \stackrel{\text{def}}{=} \underline{\text{dict}}\ k\ \underline{\text{in}}\ \Pi_{\mathtt{PName}}(\mathtt{Proj}) \Rightarrow \underline{\text{element}}(\underline{\text{select}}\ p\ \underline{\text{from}}\ \mathtt{Proj}\ p\ \underline{\text{where}}\ p.\mathtt{PName} = k)$$

$$\mathtt{SI} \stackrel{\text{def}}{=} \underline{\text{dict}}\ k\ \underline{\text{in}}\ \Pi_{\mathtt{CustName}}(\mathtt{Proj}) \Rightarrow (\underline{\text{select}}\ p\ \underline{\text{from}}\ \mathtt{Proj}\ p\ \underline{\text{where}}\ p.\mathtt{CustName} = k)$$

Here $\Pi_{\mathtt{A}}(\mathtt{R})$ is a shorthand for the query that projects relation R on A and 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**, eg., for I we use (PI1, PI2) and for SI we use (SI1, SI2, SI3) where

(PI1) $\forall(p \in \mathtt{Proj})\ \exists(i \in \underline{\text{dom}}\ \mathtt{I})\ i = p.\mathtt{PName}\ \underline{\text{and}}\ \mathtt{I}[i] = p$

(PI2) $\forall(i \in \underline{\text{dom}}\ \mathtt{I})\ \exists(p \in \mathtt{Proj})\ i = p.\mathtt{PName}\ \underline{\text{and}}\ \mathtt{I}[i] = p$

(SI1) $\forall(p \in \mathtt{Proj})\ \exists(k \in \underline{\text{dom}}\ \mathtt{SI})\ \exists(t \in \mathtt{SI}[k])\ k = p.\mathtt{CustName}\ \underline{\text{and}}\ p = t$

(SI2) $\forall(k \in \underline{\text{dom}}\ \mathtt{SI})\ \forall(t \in \mathtt{SI}[k])\ \exists(p \in \mathtt{Proj})\ k = p.\mathtt{CustName}\ \underline{\text{and}}\ p = t$

(SI3) $\forall(k \in \underline{\text{dom}}\ \mathtt{SI})\ \exists(t \in \mathtt{SI}[k])\ \underline{\text{true}}$

Notice that each of (PI1, PI2, SI1, SI2) is an *inclusion* constraint while (SI3) is a *non-emptyness* constraint. In fact, taken together, the pairs of inclusion constraints also state *inverse relationships* between the dictionaries and Proj. Similarly, we can represent the relationship between the class Dept and the dictionary implementing it, Dept, with two constraints. We show one of them (the other is "inverse"):

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

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. We leave the details out due to lack of space.

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 [48, 16, 15, 30, 38]. We consider the more general form

$$\mathtt{V} \stackrel{\text{def}}{=} \underline{\text{select}}\ O(\vec{x})\ \underline{\text{from}}\ \vec{P}\ \vec{x}\ \underline{\text{where}}\ B(\vec{x})$$

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

$$\delta_{\mathtt{V}} \stackrel{\text{def}}{=} \forall(\vec{x} \in \vec{P})\ [\ B(\vec{x})\ \Rightarrow\ \exists(v \in V)\ O(\vec{x}) = v\ ]$$

$$\delta'_{\mathtt{V}} \stackrel{\text{def}}{=} \forall(v \in V)\ \exists(\vec{x} \in \vec{P})\ [\ B(\vec{x})\ \ \underline{\text{and}}\ \ O(\vec{x}) = v\ ]$$

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

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

$$(\delta_{\mathtt{JI}})\ \ \forall(d \in \mathtt{depts})\ \forall(s \in d.\mathtt{DProjs})\ \forall(p \in \mathtt{Proj})$$
$$(\ s = p.\mathtt{PName}\ \Rightarrow\ \exists(j \in \mathtt{JI})\ j.\mathtt{DOID} = d$$
$$\underline{\text{and}}\ \ j.\mathtt{PN} = p.\mathtt{PName}\ )$$

Source capabilities often used in information integration systems can be described by either such materialized views or by dictionaries modeling the binding patterns of [39].

Join indexes [46] 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. 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 Dept with Proj is (Dept, I, JI).

Access support relations [28, 29] generalize **path indexes** [34, 10, 11] 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 [45] specify physical access structures as materialized PSJ views over logical schema. [45] 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:

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

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.

3 Optimization

The optimization algorithm starts with a query Q against a logical schema Λ and produces a query plan Q' against the physical schema Φ. Q' will be equivalent to Q under all the constraints and it will be selected according to a cost model. In addition to optimization for physical data independence, the algorithm performs semantic optimizations allowed by the constraints of the logical schema and eliminates superfluous computations (as in tableau minimization [2]).

The algorithm has two main phases: the first one, called the **chase**, introduces all physical structures in the implementation that are *relevant* for Q and rewrites Q to a **universal plan** U that explicitly uses them. The second phase, that we call the **backchase** searches for a minimal plan for Q among the "subqueries" of U. We believe that this is a novel approach. It was in fact inspired by our use of constraints as rewrite rules [37] and it is motivated by the completeness result we prove in section 5. For the following let us denote by D the dependencies on the logical schema and by D' the dependencies between the logical and physical schemas that model the implementation mapping (as we have shown in section 2).

Phase 1: chase. Given a constraint of the form

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

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

$$\begin{array}{l}\underline{\text{select}}\;\; O(\vec{r})\\ \underline{\text{from}}\;\; \ldots, R_1\, r_1, \ldots, R_m\, r_m, \ldots\\ \underline{\text{where}}\;\; \cdots\;\; \underline{\text{and}}\;\; B_1\;\; \underline{\text{and}}\;\; \cdots\end{array}$$

$$\Downarrow$$

$$\begin{array}{l}\underline{\text{select}}\;\; O(\vec{r})\\ \underline{\text{from}}\;\; \ldots, R_1\, r_1, \ldots, R_m\, r_m,\; S_1\, s_1, \ldots, S_n\, s_n, \ldots\\ \underline{\text{where}}\;\; \cdots\;\; \underline{\text{and}}\;\; B_1\;\; \underline{\text{and}}\;\; B_2\;\; \underline{\text{and}}\;\; \cdots\end{array}$$

Example. On our Proj-Dept schema, the logical query Q chases in one step using $\delta_{\mathtt{JI}}$ to the following. Note how new loops and conditions are being added to the ones already existing in Q.

$$\begin{array}{l}\underline{\text{select}}\;\; \underline{\text{struct}}(\mathtt{PN} : s,\; \mathtt{PB} : p.\mathtt{Budg},\; \mathtt{DN} : d.\mathtt{DName})\\ \underline{\text{from}}\;\; \text{depts}\; d,\;\; d.\mathtt{DProjs}\; s,\;\; \mathtt{Proj}\; p,\;\; \mathtt{JI}\; j\\ \underline{\text{where}}\;\; s = p.\mathtt{PName}\;\; \underline{\text{and}}\;\; p.\mathtt{CustName} = \text{“CitiBank”}\\ \qquad \underline{\text{and}}\;\; j.\mathtt{DOID} = d\;\; \underline{\text{and}}\;\; j.\mathtt{PN} = p.\mathtt{PName}\end{array}$$

The *chase phase* consists of applying repeatedly chase steps w.r.t. any applicable constraint from the logical schema and from the characterization of the physical structures (see section 2), i.e. $D \cup D'$. "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 [37] that the classical relational chase [9] is indeed a particular case of this. We also show that while the chase does not always terminate, it does so for certain classes of constraints and queries, yielding an essentially unique result U whose size is polynomial [6] in that of Q. Sometimes we denote U by $chase(Q)$.

Example. We illustrate the first phase of the algorithm on our example. By chasing with δ_{JI}, then with $\delta_{\mathtt{Dept}}$, INV1, SI1 and PI1, U is obtained as follows. None of the other dependencies are applicable.

$$\begin{array}{l}\underline{\text{select}}\;\; \underline{\text{struct}}(\mathtt{PN} : s, \mathtt{PB} : p.\mathtt{Budg}, \mathtt{DN} : \mathtt{Dept}[\,d\,].\mathtt{DName})\\ \underline{\text{from}}\;\; \text{depts}\; d,\; d.\mathtt{DProjs}\; s,\; \mathtt{Proj}\; p,\; \mathtt{JI}\; j\\ \qquad \underline{\text{dom}}\,\mathtt{Dept}\; d',\; \mathtt{Dept}[\,d'\,].\mathtt{DProjs}\; s',\\ \qquad \underline{\text{dom}}\,\mathtt{SI}\; k,\; \mathtt{SI}[\,k\,]\; t,\; \underline{\text{dom}}\,\mathtt{I}\; i\\ \underline{\text{where}}\;\; s = p.\mathtt{PName}\;\; \underline{\text{and}}\;\; p.\mathtt{CustName} = \text{“CitiBank”}\\ \qquad \underline{\text{and}}\;\; j.\mathtt{DOID} = d\;\; \underline{\text{and}}\;\; j.\mathtt{PN} = p.\mathtt{PName}\\ \qquad \underline{\text{and}}\;\; d = d'\;\; \underline{\text{and}}\;\; s = s'\;\; \underline{\text{and}}\;\; p = t\\ \qquad \underline{\text{and}}\;\; p.\mathtt{CustName} = k\;\; \underline{\text{and}}\; i = p.\mathtt{PName}\\ \qquad \underline{\text{and}}\;\; p = \mathtt{I}[\,i\,]\;\; \underline{\text{and}}\;\; d.\mathtt{DName} = p.\mathtt{PDept}\end{array}$$

For the optimization algorithm, the role of the chase phase is to bring, in a systematic way, all the relevant physical structures into the logical query. For example, chasing with (PI1) and (SI1) adds to the query the accessing of the corresponding primary and secondary index. The result of the chase, U, is the universal plan that holds in one place essentially all possible physical plans expressible in our language. However, U still references elements of the logical schema, and the role of the next phase is to uncover the physical plans.

Phase 2: backchase. The *backchase step* is the rewrite

$$\begin{array}{l}\underline{\text{select}}\;\; O(\vec{x}, y)\\ \underline{\text{from}}\;\; R_1\, x_1, \ldots, R_m\, x_m,\; R\, y\\ \underline{\text{where}}\;\; C(\vec{x}, y)\end{array}$$

$$\Downarrow$$

[6] This bound could be used a heuristic for stopping the chase when termination is not guaranteed.

$$\begin{array}{l}\underline{\text{select}}\ \ O'(\vec{x})\\ \underline{\text{from}}\ \ \ R_1\ x_1, \ldots, R_m\ x_m\\ \underline{\text{where}}\ \ C'(\vec{x})\end{array}$$

provided that: (1) the conditions C' are implied by C, (2) the equality of O and O' is implied by C, and (3) the following constraint is implied by $D \cup D'$:

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

Thus, the purpose of a backchase step is to eliminate (if possible) a **binding** $R\ y$ from the from clause of the query. [7] For any two queries Q and Q' as above such that conditions (1) and (2) are satisfied, we say that Q' is a **subquery** of Q. For computing O' and C' we have a procedure defined for a large class of queries that is sound when it succeeds and that always succeeds for the queries for which the algorithm is complete. The idea is to build a database instance out of the syntax of Q grouping terms in congruence classes according to the equalities that appear in C. Then, we can take C' to be a *maximal* set of equalities implied by C (maximality is needed here for completeness). We can check then by looking at the canonical database whether we can replace O with an equivalent (i.e. in the same congruence class) O' that doesn't depend on y. If we reduce the setting to that of conjunctive relational tableaux, our notion of subquery coincides with the notion of sub-tableau. The only difference is that in our language variables range over tuples rather than over individuals and the equalities (implicit in tableaux!) are explicit.

While the first two conditions ensure that the backchase reduces a query to a subquery of it, condition (3) guarantees that it reduces it to an *equivalent* subquery. This is true because its reverse is just the chase step with constraint (δ) followed by a simplification given by (1) and a replacement of equals given by (2). Sometimes the backchase can apply just by virtue of constraints (δ) that hold in all instances (so-called *trivial* constraints). Relational tableau minimization [2] is precisely such a backchase. To illustrate, if $\mathtt{R}(\mathtt{A}, \mathtt{B})$ is a relation then query

$$\begin{array}{l}\underline{\text{select}}\ \ \underline{\text{struct}}(\mathtt{A}:\ p.\mathtt{A},\ \mathtt{B}:\ r.\mathtt{B})\\ \underline{\text{from}}\ \ \ \mathtt{R}\ p,\ \mathtt{R}\ q,\ \mathtt{R}\ r\\ \underline{\text{where}}\ \ p.\mathtt{B} = q.\mathtt{A}\ \ \underline{\text{and}}\ \ q.\mathtt{B} = r.\mathtt{B}\end{array}$$

rewrites, by backchase, to

$$\begin{array}{l}\underline{\text{select}}\ \ \underline{\text{struct}}(\mathtt{A}:\ p.\mathtt{A},\ \mathtt{B}:\ q.\mathtt{B})\\ \underline{\text{from}}\ \ \ \mathtt{R}\ p,\ \mathtt{R}\ q\\ \underline{\text{where}}\ \ p.\mathtt{B} = q.\mathtt{A}\end{array}$$

[7] We show here a simplified form of backchase. In the case when there are bindings $R_i\ x_i$ depending on the variable y, we need to modify the rule so that either these dependent bindings are eliminated together with $R\ y$ or they can be replaced with bindings that do not depend on y.

It is obvious to see that conditions (1) and (2) are satisfied, while condition (3) is true because the following constraint is trivial:

$$(\delta)\ \ \forall(p \in \mathtt{R})\ \forall(q \in \mathtt{R})\ [\, p.\mathtt{B} = q.\mathtt{A}\ \Rightarrow\\ \exists(r \in \mathtt{R})\ p.\mathtt{B} = q.\mathtt{A}\ \ \underline{\text{and}}\ \ q.\mathtt{B} = r.\mathtt{B}\,]$$

Minimal queries We call a subquery Q_1 of Q_2 a *strict* subquery if Q_1 has strictly fewer bindings than Q_2. We say that a query Q is **minimal** if there does not exist a strict subquery Q' of Q such that Q' is equivalent to Q. In other words, we cannot remove any bindings from Q without losing equivalence. (It turns out that this is a generalization of the minimality notion of [30].) In general, we can think of the backchase as minimization for a larger (than just relational tableaux) class of queries, and under constraints. Trying to see whether (δ) of condition (3) is implied by the existing constraints can actually be done with the chase presented above when constraints are viewed as boolean-valued queries [37]. Again, this is a decidable problem in the case for which the algorithm is complete.

The backchase phase consists of applying backchase steps until this is not possible anymore. Clearly this phase always terminates and the original query must be among those it *could* produce (!), but the obvious strategy for the optimizer is to attempt to remove whatever is in the logical schema but not in the physical schema. For the case in which the algorithm is complete, any query that results from the backchase phase is minimal (as defined above), and, any minimal subquery of a given query Q is guaranteed to be produced by a backchase sequence from Q.

We can now put these together, and add conventional optimization techniques such as "algebraic" rewriting (e.g., pushing selections towards the sources) and cost-based dynamic programming for join reordering [41]. Without elaborating, we mention that by ignoring nesting it is possible to apply these techniques to the queries we consider here.

Algorithm 3.1 (Optimization)
Input: Logical schema Λ with constraints D,
Constraints D' characterizing physical schema Φ,
Cost function $\mathcal{C}$,
Query $Q(\Lambda)$
Output: Cheapest plan $Q'(\Phi)$ equivalent to Q under $D \cup D'$

1. **for each** $U(\Lambda, \Phi) \leftarrow chase_{D,D'}(Q)$
2. **for each** $p(\Phi) \leftarrow backchase_{D,D'}(U)$
3. do cost-based conventional optimization, keep cheapest plan so far p_m
4. $Q' \leftarrow p_m$

The first for loop (chase) enumerates all possible results of chasing (there may be more than one in general). For each such result, the second for loop (backchase) enumerates all possible backchase se-

quences (again there may be more than one result), each producing a plan p. In step (3) conventional optimization techniques, including mapping into physical operators different than those index-based, are applied to p. If the cost of p is smaller than the current minimum cost plan p_m then update p_m to be p. In step (4) the best query plan p_m is the final result.

The reader can check by backchasing the universal plan U shown previously that P_0, P_1, P_2, and P_3 are obtained as minimal queries in this algorithm. Steps (3) and (4) choose the cheapest plan among them.

Rule-based implementation In an implementation, the conceptual search of algorithm 3.1 can be specified implicitly by configuring a rule-based optimizer ([17, 23]) with the two rewrite rules *chase* and *backchase*, and requesting that the application of the chase rule always takes precedence over that of the backchase rule. Depending on the search strategy implemented by the optimizer, the search space may not be explored exhaustively but rather pruned using heuristics such as in [25, 44].

There is, however, a fundamental difference between our optimization framework and a rule-based optimizer as in Volcano [23]. While in Volcano's optimizer algebraic and physical transformations are mixed and the search is guided by a cost model, steps (1) and (2) of algorithm 3.1 are cost-independent and performed before the phase of (cost-driven) mapping into physical operator trees (other than index-based plans). This is more in the spirit of Starburst optimizer [33] which also had a clear separation between the two kinds of transformations. However, the query rewriting phase in Starburst did not include indexes nor logical constraints, and was heuristics-based.

4 Relational Examples

Our approach extends beyond the relational model, but it also proposes improvements over previous approaches to relational optimization. An important contribution of this work is the *systematic* procedure for considering all alternate plans enabled by indexes, as opposed to the ad-hoc heuristics proposed previously. Consider for example a *logical* schema with one relation $\mathtt{R}(\mathtt{A},\mathtt{B},\mathtt{C})$ and a *physical* schema containing secondary indexes SA and SB on attributes A and B of R. Then our algorithm will discover for the logical query

$$\begin{array}{l}\underline{\mathsf{select}}\ r.\mathtt{C}\\ \underline{\mathsf{from}}\ \ \mathtt{R}\ r\\ \underline{\mathsf{where}}\ r.\mathtt{A} > 5\ \underline{\mathsf{and}}\ r.\mathtt{B} = 20\end{array}$$

the following *index-only access path* plan ([40]):

$$\begin{array}{l}\underline{\mathsf{select}}\ r.\mathtt{C}\\ \underline{\mathsf{from}}\ \ \underline{\mathsf{dom}}\,\mathtt{SA}\ x,\ \mathtt{SA}[\,x\,]\ r_1,\ \mathtt{SB}[\![20]\!]\ r_2\\ \underline{\mathsf{where}}\ x > 5\ \underline{\mathsf{and}}\ r_1 = r_2\end{array}$$

Notice how the scan of R is replaced by a scan of index SA (which can be filtered using condition $x > 5$) interleaved with non-failing lookups in index SB.

Our algorithm considers exhaustively combinations of materialized views, indexes, and semantic constraints, thus generating plans which are not captured in frameworks such as [30]. Assume a *logical* schema with relations $\mathtt{R}(\mathtt{A},\mathtt{B})$ and $\mathtt{S}(\mathtt{B},\mathtt{C})$, and a *physical* schema that has R and S too (direct mapping!), as well as materialized view $V = \Pi_{\mathtt{A}}(\mathtt{R} \bowtie \mathtt{S})$ and secondary indexes $\mathtt{I}_{\mathtt{R}}$ and $\mathtt{I}_{\mathtt{S}}$ on attributes A and B of R and S, respectively. We want to optimize the logical query $Q = R \bowtie S$.

Q itself is a valid plan (modulo join-reordering and various implementations for the join). However, the view V can be used to produce the following equivalent query (again we ignore here the join order):

$$\begin{array}{ll}(P) & \underline{\mathsf{select}}\ \underline{\mathsf{struct}}(\mathtt{A}:\ r.\mathtt{A},\ \mathtt{B}:\ s.\mathtt{B},\ \mathtt{C}:\ s.\mathtt{C})\\ & \underline{\mathsf{from}}\ \ \mathtt{V}\ v,\ \mathtt{R}\ r,\ \mathtt{S}\ s\\ & \underline{\mathsf{where}}\ v.\mathtt{A} = r.\mathtt{A}\ \ \underline{\mathsf{and}}\ \ r.\mathtt{B} = s.\mathtt{B}\end{array}$$

This is obtained as a first step of the chase phase, by rewriting Q with one of the two constraints characterizing V (namely $\delta_{\mathtt{V}}$, see section 2). The techniques used by [30] for answering/optimizing queries using views can also be used to produce query P in a first phase, similar to our chase. Now, if V is small, P can be *implemented*, in a typical relational system, much better than Q because of the two indexes. (V is the only relation that is scanned while the relations R and S are accessed via indexes.) However, in the approach of [30], P is thrown away because Q is a subquery of P, thus P is not minimal. Minimality, in their case as well as in our case, is essential for bounding the search space for optimal plans. The problem in [30] is that Q and P are the only *expressible* plans using the conjunctive relational language. There is no way of expressing and taking advantage of the indexes at the language level. The language used for gmaps in [45] suffers of the same limitation.

Here is how we can overcome this problem and be able to produce a plan that corresponds to the good physical implementation hinted earlier. In our approach, P is still not a minimal plan, thus it will be thrown away, too, in the backchase phase. But the chase phase doesn't stop with P: we can still bring in the two indexes by chasing with constraints relating R and S with, respectively, $\mathtt{I}_{\mathtt{R}}$ and $\mathtt{I}_{\mathtt{S}}$:

$$\begin{array}{ll}(U) & \underline{\mathsf{select}}\ \underline{\mathsf{struct}}(\mathtt{A}:\ r.\mathtt{A},\ \mathtt{B}:\ s.\mathtt{B},\ \mathtt{C}:\ s.\mathtt{C})\\ & \underline{\mathsf{from}}\ \ \mathtt{V}\ v,\ \mathtt{R}\ r,\ \mathtt{S}\ s,\ (\underline{\mathsf{dom}}\ \mathtt{I}_{\mathtt{R}})\ k,\ \mathtt{I}_{\mathtt{R}}[\,k\,]\ r',\\ & \quad (\underline{\mathsf{dom}}\ \mathtt{I}_{\mathtt{S}})\ p,\ \mathtt{I}_{\mathtt{S}}[\,p\,]\ s'\\ & \underline{\mathsf{where}}\ \ v.\mathtt{A} = r.\mathtt{A}\ \ \underline{\mathsf{and}}\ \ r.\mathtt{B} = s.\mathtt{B}\ \ \underline{\mathsf{and}}\ \ k = r.\mathtt{A}\\ & \quad \underline{\mathsf{and}}\ \ r' = r\ \ \underline{\mathsf{and}}\ \ p = s.\mathtt{B}\ \ \underline{\mathsf{and}}\ \ s' = s\end{array}$$

Backchasing twice with the "inverse" constraints re-

lating $\mathtt{I_R}$ and $\mathtt{I_S}$ with, respectively, R and S:

$$\begin{array}{l}\underline{\text{select}}\ \underline{\text{struct}}(\mathtt{A}:\ r'.\mathtt{A},\ \mathtt{B}:\ s'.\mathtt{B},\ \mathtt{C}:\ s'.\mathtt{C})\\ \underline{\text{from}}\ \ \mathtt{V}\ v,\ (\underline{\text{dom}}\ \mathtt{I_R})\ k,\ \mathtt{I_R}[k]\ r',\\ \qquad (\underline{\text{dom}}\ \mathtt{I_S})\ p,\ \mathtt{I_S}[p]\ s'\\ \underline{\text{where}}\ k = v.\mathtt{A}\ \underline{\text{and}}\ p = r'.\mathtt{B}\end{array}$$

Using the equality $k = v.\mathtt{A}$ and the inclusion constraint $\mathtt{V[A]} \subseteq \mathtt{R[A]}$ that is inferred in our system as a consequence of $\delta_{\mathtt{V}}$, we can backchase one final step and eliminate the loop over $\underline{\text{dom}}\ \mathtt{I_R}$:

$$\begin{array}{l}\underline{\text{select}}\ \underline{\text{struct}}(\mathtt{A}:\ r'.\mathtt{A},\ \mathtt{B}:\ s'.\mathtt{B},\ \mathtt{C}:\ s'.\mathtt{C})\\ \underline{\text{from}}\ \ \mathtt{V}\ v,\ \mathtt{I_R}[v.\mathtt{A}]\ r',\ (\underline{\text{dom}}\ \mathtt{I_S})\ p,\ \mathtt{I_S}[p]\ s'\\ \underline{\text{where}}\ p = r'.\mathtt{B}\end{array}$$

The last transformation replaced a value-based join with a navigation join and the resulting query reflects almost entirely the navigation join implementation hinted earlier, except for the loop over $\underline{\text{dom}}\,\mathtt{I_S}$. This loop together with the condition $p = r'.\mathtt{B}$ is only a guard that ensures that the lookup of $r'.\mathtt{B}$ into $\mathtt{I_S}$ doesn't fail. It is not redundant, for without it we would lose equivalence (recall that the original query Q never fails). However, using the non-failing lookup, the last query is equivalent to the plan:

$$\begin{array}{l}\underline{\text{select}}\ \underline{\text{struct}}(\mathtt{A}:\ r'.\mathtt{A},\ \mathtt{B}:\ s'.\mathtt{B},\ \mathtt{C}:\ s'.\mathtt{C})\\ \underline{\text{from}}\ \ \mathtt{V}\ v,\ \mathtt{I_R}[v.\mathtt{A}]\ r',\ \mathtt{I_S}[\![r'.\mathtt{B}]\!]\ s'\end{array}$$

5 Completeness

We describe first the *path-conjunctive (PC) language* (mainly the one introduced in [37]), after which we give our main theoretical results: the *bounding chase* theorem and the *completeness of backchase* theorem. Completeness of algorithm 3.1 follows immediately from them. These results hold for PC queries when the logical schema has arbitrary classes and (nested) relations, but no constraints, while the physical schema has materialized PC views, but no arbitrary indexes (only dictionaries implementing classes). The two results are a generalization to a richer model of the results of [30].

The path-conjunctive fragment of the ODL / OQL language that we have used so far is defined below. It includes the relational conjunctive queries of [14, 5] but is more general because it includes dictionaries and nested relations.

Paths: $P ::= x \mid c \mid \mathtt{R} \mid P.\mathtt{A} \mid \underline{\text{dom}}\,P \mid P[x]$

Path-Conjunctions:

$$B ::= P_1 = P_1'\ \underline{\text{and}}\ \cdots\ \underline{\text{and}}\ P_k = P_k'$$

PC Queries:
$$\begin{array}{l}\underline{\text{select}}\ \underline{\text{struct}}(A_1 : P_1', \ldots, A_n : P_n')\\ \underline{\text{from}}\ \ P_1\ x_1, \ldots, P_m\ x_m\\ \underline{\text{where}}\ B\end{array}$$

Here x stands for variables, c denotes constants at base types, and R stands for schema names (relation or dictionary names). The following **restrictions** are imposed on a PC query Q.

(1) Keys of dictionaries, equalities in the where clause, and the expression in the select clause are not allowed to be/contain expressions of set/dictionary type.

(2) A lookup operation can only be of the form $P[x]$ with the additional condition that there must exist a binding of the form $\underline{\text{dom}}\,P\ x$[8] in the from clause. The reason for not allowing an arbitrary lookup is mainly technical: all our definitions including query equivalence would need to be extended with explicit null values, and tedious reasoning about partiality. With this restriction a lookup operation never fails.

Restriction (2) implies that we cannot express in the PC fragment navigation-style joins (involving chains of lookup operations). However, these kind of joins can be rewritten as value-based joins (as seen in section 4) and vice-versa provided that certain integrity constraints hold. In the value-based counterpart of a navigation join a chain of lookups is replaced by explicit joins involving equality of oids. Value-based joins are guaranteed not to touch any dangling oids and therefore are easier to reason with them.

Path-conjunctive constraints. *Embedded path-conjunctive dependencies* (EPCDs) defined in [37] are a generalization for the complex value and dictionary model of the relational tgds and egds ([4, 9]). EPCDs play a fundamental role in rewriting of PC queries by chase and they have the logical form:

$$\begin{array}{ll}\textit{EPCD:} & \forall(x_1 \in P_1)\ldots\forall(x_n \in P_n)\ [\,B_1(\vec{x})\ \Rightarrow\\ & \qquad \exists(y_1 \in P_1')\ldots\exists(y_k \in P_k')\ B_2(\vec{x},\vec{y})\ ]\end{array}$$

P_i and P_i' are paths, while B_1 and B_2 are path-conjunctions (as defined before, with the same restrictions). Each P_i may refer to variables $x_1, \ldots, x_{i-1}$, while P_j' may refer to $x_1, \ldots x_n, y_1, \ldots y_{j-1}$, thus an EPCD is not a first-order formula. A special class of EPCDs are constraints in which there are no existential quantifiers, EGDs. Functional dependencies, like (KEY1) and (KEY2), and the constraints typically involved in conditions (1) and (2) of the backchase step are examples of EGDs.

Main theorems Our assumptions for the rest of this section are that the logical schema contains only relations and classes (no dependencies) and the physical schema contains only (nested) relations (including materialized PC views) and dictionaries implementing class extents (no dictionaries implementing indexes). These restrictions are needed for the completeness result of theorem 5.1 below. We conjecture that the result holds even in the presence of indexes and a certain class of *full* dependencies (introduced

[8] Or, more general, a binding $\underline{\text{dom}}\,P\ y$ such that the equality $x = y$ is implied by the conditions in the `where` clause. This is a PTIME-checkable condition (see [37]).

in [37]).

Completeness follows from the *finiteness* of the space of minimal plans. We provide two upper bounds for this search space: In [6], we show how to generalize to PC queries the upper bound result obtained in [30] for conjunctive relational queries, thus justifying a procedure which enumerates equivalent plans bottom-up by building subsets of at most as many views, relations and classes as the number of bindings in the from clause of logical query Q, combining them by setting appropriate conditions in the where clause, then checking equivalence with Q.

In view of a rule-based implementation however, a top-down enumeration procedure implemented as step-by-step rewriting is better suited, and our algorithm uses a different, novel characterization of the search space of query plans:

Theorem 5.1 (Bounding Chase) *Any minimal plan $Q'(\Phi)$ for logical query $Q(\Lambda)$ is a subquery of the universal plan $chase(Q)(\Lambda, \Phi)$.*

Here $chase(Q)$ means the result of chasing Q with the set of all dependencies of the form δ_V (see section 2) associated with the view definitions. Since these are full dependencies(see [37] for definition and theorem), $chase(Q)$ exists and is unique.

Theorem 5.1 allows the enumeration of all minimal plans of Q by enumerating those subqueries of $chase(Q)$ which mention only the physical schema Φ (as seen in section 3. Conceptually, the enumeration proceeds by first listing the largest subqueries of $chase(Q)$ which involve only the physical schema Φ, pruning away those subqueries which are not equivalent to $chase(Q)$ and then applying itself recursively to each non-pruned subquery. The equivalence check can be done by unfolding the view definitions. It follows easily from the definition of subqueries that the pruning step doesn't compromise completeness, since whenever a subquery of $chase(Q)$ is not equivalent to the latter, neither are its subqueries.

The following theorem states that the desired enumeration and pruning of equivalent subqueries can be implemented in a rule-based optimizer by rewriting with the backchase rule introduced in section 3:

Theorem 5.2 (Complete Backchase) *The minimal equivalent subqueries of a query $Q(\Lambda, \Phi)$ are exactly the normal forms of backchasing $Q(\Lambda, \Phi)$.*

The use of the chase as upper bound for the space of minimal plans leads to an enumeration procedure that remains *sound* even in the presence of constraints on the logical schema and of indexes, which are not dealt with in [30].

Corollary 5.3 (Completeness of algorithm 3.1) *If Λ contains no dependencies and Φ contains no indexes, algorithm 3.1 is complete for PC queries.*

Our algorithm takes exponential time: each chase step is exponential, but in the case of chasing with $\vec{d}_V$, and more generally, as shown in [37], when chasing with arbitrary full dependencies, the chase rule applies only polynomially many times, resulting in a query whose size is polynomial in the size of the chased query. The second phase of algorithm 3.1 preserves the exponential complexity: each backchase step is exponential (is uses the chase to check the applicability of the rule) but it eliminates a binding, so the backchase process always reaches a normal form after at most as many steps as there are bindings in the result of the chase. The NP-completeness results given in [30] for the particular case of answering queries with conjunctive relational views tell us that there is little hope to do better than exponential if we want a complete enumeration.

6 Related work

Relevant work on integrating information systems includes [35, 31, 3, 36]. Arrays, as dealt with in [32] can be formalized as dictionaries, given some arithmetic and operations that produce integer intervals. The maps of [7] and the treatment of object types in [8] are related to our dictionaries. An important difference is made by the operations on dictionaries used here.

The framework that we use for optimization is quite comprehensive as it is possible to represent almost the entire variety of equivalences stated in various papers, beginning with the standard relational "algebraic" optimizations, continuing with OODB optimizations as in the work of Cluet, Zdonik, Maier, Fegaras and others [42, 43, 19, 18], and in fact including the very comprehensive work by Beeri and Kornatzky [8].

Use of referential integrity constraints to eliminate dependent joins is implicit in Orion optimizations [26] and the type-based approach of [19]. This, and the use of precomputed ASR's appear in [28, 29]. Precomputed join indexes are proposed in [46]. An approach to semantic query optimization using a translation into Datalog appears in [13, 24]. The idea of using semantic constraints as rewrite rules is introduced and exploited systematically in [21, 22].

When the physical schema contains only materialized relations, finding an execution plan is a generalization of the problem of answering queries using views ([30], [38]). At the opposite extreme, if the physical schema materializes all relations and classes in the logical schema, the original query is directly ex-

ecutable but the optimizer has to look for (better) execution plans. This is sometimes called the problem of optimizing queries using views [15].

The GMAP approach [45] solves the problem of physical data independence for a special case that is subsumed by our work which applies to a more general class of physical storage structures, queries against the logical schema and dependencies. 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.

7 What we do *not* do, but we'd like to

- We do not address the problems resulting specifically from the nesting of the queries.
- Our physical data model does not reflect information related to storage organization issues such as paging or clustering.
- By extending the physical data model to include lists, it might be possible to capture in some substantial way algorithms using sorted values.
- We expect that the algorithm proposed here will be used in conjunction with good cost models and good heuristics for pruning the search space, but we have not yet examined how these issues relate to the nature of the algorithm itself. An implementation is under way in order to help us understand these relationships and the feasibility of the whole approach.

Acknowledgements We thank Dan Suciu, Anthony Bonner, Rick Hull and Jerome Simeon for helpful discussions.

References

[1] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 159–173, Portland, Oregon, 1989.

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

[3] S. Adali, K. Selcuk Candan, Y. Papakonstantinou, and V.S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *ACM SIGMOD*, pages 137 – 148, 1996.

[4] A. V. Aho, C. Beeri, and J. D. Ullman. The theory of joins in relational databases. *ACM Transactions on Database Systems*, 4(3):297–314, 1979.

[5] A. V. Aho, Y. Sagiv, and J. D. Ullman. Efficient optimization of a class of relational expressions. *ACM Transactions on Database Systems*, 4(4):435–454, December 1979.

[6] Alin Deutsch and Lucian Popa and Val Tannen. Optimization for Physical Independence in Information Integration Components. Technical report, University of Pennsylvania, 1999.

[7] M. Atkinson, C. Lecluse, P. Philbrow, and P. Richard. Design issues in a map language. In *Proc. of the 3rd Int'l Workshop on Database Programming Languages (DBPL91)*, Nafplion, Greece, August 1991.

[8] Catriel Beeri and Yoram Kornatzky. Algebraic optimisation of object oriented query languages. *Theoretical Computer Science*, 116(1):59–94, August 1993.

[9] Catriel Beeri and Moshe Y. Vardi. A proof procedure for data dependencies. *Journal of the ACM*, 31(4):718–741, 1984.

[10] E. Bertino. *Query Processing for Advanced Database Systems*, chapter A Survey of Indexing Techniques for Object-Oriented Database Management Systems, pages 383–418. Morgan Kauffmann, San Mateo, CA, 1994.

[11] E. Bertino and W. Kim. Indexing techniques for queries on nested objects. *IEEE Trans. on Knowledege and Data Engineering*, 1(2), 1989.

[12] R. G. G. Cattell, editor. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, San Mateo, California, 1997.

[13] U. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems*, 15(2):162–207, 1990.

[14] Ashok Chandra and Philip Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of 9th ACM Symposium on Theory of Computing*, pages 77–90, Boulder, Colorado, May 1977.

[15] S. Chaudhuri, R. Krishnamurty, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of ICDE*, Taipei, Taiwan, March 1995.

[16] C.M. Chen and N. Roussopoulos. The implementation and performance evaluation of the ADMS query optimizer. In *Proceedings of the International Conference on Extending Database Technology*, 1994.

[17] M. Cherniack and S. B. Zdonik. Rule languages and internal algebras for rule-based optimizers. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages ??–??, Montreal, Quebec, Canada, 1996.

[18] S. Cluet. *Langages et Optimisation de requetes pour Systemes de Gestion de Base de donnees orienteobjet*. PhD thesis, Universite de Paris-Sud, 1991.

[19] Sophie Cluet and Claude Delobel. A general framework for the optimization of object oriented queries. In M. Stonebraker, editor, *Proceedings ACM-SIGMOD International Conference on Management of Data*, pages 383–392, San Diego, California, June 1992.

[20] L. Fegaras and D. Maier. An algebraic framework for physical oodb design. In *Proc. of the 5th Int'l Workshop on Database Programming Languages (DBPL95)*, Umbria, Italy, August 1995.

[21] D. Florescu. *Design and Implementation of the Flora Object Oriented Query Optimizer*. PhD thesis, Universite of Paris 6, 1996.

[22] D. Florescu, L. Rashid, and P. Valduriez. A methodology for query reformulation in cis using semantic knowledge. *International Journal of Cooperative Information Systems*, 5(4), 1996.

[23] Goetz Graefe, Richard L. Cole, Diane L. Davison, William J. McKenna, and Richard H. Wolniewicz. Extensible query optimization and parallel execution in Volcano. In Johann Christoph Freytag, David Maier, and Gottfried Vossen, editors, *Query Processing for Advanced Database Systems*, chapter 11, pages 305–335. Morgan Kaufmann, San Mateo, California, 1994.

[24] J. Grant, J. Gryz, J. Minker, and L. Raschid. Semantic query optimization for object databases. In *Proc. of the 13th Int'l. Conference on Data Engineering*, April 1997.

[25] Y. E. Ioannidis and E. Wong. Query optimization by simulated annealing. In Umeshwar Dayal and Irv Traiger, editors, *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 9–22, San Francisco, May 1987.

[26] P. Jeng, D. Woelk, W. Kim, and W. Lee. Query processing in distributed orion. In *Proc. EDBT*, Venice, Italy, March 1990.

[27] A.M. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, 1994.

[28] A. Kemper and G. Moerkotte. Access support relations in object bases. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 364–374, 1990.

[29] A. Kemper and G. Moerkotte. Advanced query processing in object bases using access support relations. In *Proc. VLDB*, Brisbane, Australia, 1990.

[30] A. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proceedings of PODS*, 1995.

[31] A. Levy, D. Srivastava, and T. Kirk. Data model and query evaluation in global information systems. *Journal of Intelligent Information Systems*, 1995.

[32] L. Libkin, R. Machlin, and L. Wong. A query language for multidimensional arrays: Design, implementation and optimization techniques. In *SIGMOD Proceedings, Int'l Conf. on Management of Data*, 1996.

[33] Guy M. Lohman, Bruce Lindsay, Hamind Pirahesh, and K. Bernhard Schiefer. Extension to Starburst: Objects, types, functions, and rules. 34(10):94–109, October 1991.

[34] D. Maier and J. Stein. Indexing in an object-oriented dbms. In *Proceedings of 2nd International Workshop on Object-Oriented Database Systems*, pages 171–182, Asilomar, CA, September 1986.

[35] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogenous information sources. In *Proceedings of IEEE International Conference on Data Engineering*, pages 251–260, March 1995.

[36] Yannis Papakonstantinou, Ashish Gupta, and Laura M. Haas. Capabilities-based query rewriting in mediator systems. *Distributed and Parallel Databases*, 6(1), 1998.

[37] Lucian Popa and Val Tannen. An equational chase for path-conjunctive queries, constraints, and views. In *Proceedings of ICDT*, Jerusalem, Israel, January 1999.

[38] X. Qian. Query folding. In *Proceedings ICDE*, pages 48–55, 1996.

[39] A. Rajaraman, Y. Sagiv, and J.D. Ullman. Answering queries using templates with binding patterns. In *Proc. 14th ACM Symposium on Principles of Database Systems*, pages 105–112, 1995.

[40] Raghu Ramakrishnan. *Database Management Systems*. McGraw-Hill, 1998.

[41] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979. Reprinted in *Readings in Database Systems*, Morgan-Kaufmann, 1988.

[42] G. Shaw and S. Zdonik. Object-oriented queries: equivalence and optimization. In *Proceedings of International Conference on Deductive and Object-Oriented Databases*, 1989.

[43] G. Shaw and S. Zdonik. An object-oriented query algebra. In *Proc. DBPL*, Salishan Lodge, Oregon, June 1989.

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

[45] O. Tsatalos, M. Solomon, and Y. Ioannidis. The gmap: A versatile tool for physical data independence. In *Proc. of 20th VLDB Conference*, Santiago, Chile, 1994.

[46] P. Valduriez. Join indices. *ACM Trans. Database Systems*, 12(2):218–452, June 1987.

[47] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, pages 38–49, March 1992.

[48] H.Z. Yang and P.A. Larson. Query transformation for psj queries. In *Proceedings of the 13th International VLDB Conference*, pages 245–254, 1987.