Chase & Backchase: A Method for Query Optimization with Materialized Views and Integrity Constraints

Alin Deutsch Lucian Popa Val Tannen

University of Pennsylvania IBM Almaden University of Pennsylvania

Abstract

We have previously proposed *chase and backchase* as a novel method for using materialized views and integrity constraints in query optimization. In this paper, we show that the method is usable in realistic optimizers by extending it to bag and mixed (i.e. bag-set) semantics as well as to grouping views and by showing how to integrate it with standard cost- based optimization. We understand materialized views broadly, including user-defined views, cached queries and physical access structures (such as join indexes, access support relations, and gmaps). Moreover, our internal query representation supports object features hence the method applies to OQL and (extended) SQL:1999 queries. Chase and backchase supports a very general class of integrity constraints, thus being able to find execution plans using views that do not fall in the scope of other methods. In fact, we prove completeness theorems that show that our method will find the best plan in the presence of common and practically important classes of constraints and views, even when bag and set semantics are mixed. We report on a series of experiments that demonstrate the practicality of our new ideas.

1 Introduction

Optimizers are characterized by three dimensions: the search space for execution plans, the strategy for exploring it, and the cost model used to compare plans to each other. This work addresses the first two dimensions.

Rather than comparing optimizers by exhibiting single examples of plans that one may find while another may not, it would be desirable to have mathematical results guaranteeing that at least under certain conditions the optimal plan is not missed (so-called *completeness* results). Such results are only possible if there is a clear definition of the search space for alternate plans, which is conceptually independent of the exploration strategy. Even when completeness can only be shown under restrictions, a clearly defined search space improves the overall understanding of the tradeoff between plan quality and optmization speed.

For early optimizers, the search space was well-understood: consider all available access methods (file scan or index lookups) per relation, all possible (usually only "left-deep") join orderings, and all join algorithms (from a small set of well-established ones). Exploration strategies included exhaustive search enhanced with cost-based pruning (e.g. dynamic programming [23]), and forms of non-exhaustive search trading plan quality (i.e. closeness to optimality) for optimization speed-up: rule-based [10], heuristic, randomized [14].

Subsequent optimizers extended the search space in two different directions, in order to improve plan quality, but also to increase physical data independence. One direction was semantic optimization, that is rewriting a query according to integrity constraints in the logical schema[1]. The other consisted in

[1] There is a large body of work on this topic. Since we have reviewed it in [8] and given the space constraints of this submission, we will omit here those references.

rewriting the query to also use, eg., materialized views/cached queries [2, 13, 3, 27] or physical storage structures [26, 15, 25]. In both cases, the rewriting was combined with standard cost-based pruning.

Completeness results were provided in the absence of integrity constraints [4, 17]. However, previous attempts to combine the use of views and integrity constraints all have the same problem: an ad-hoc search space for rewritings, usually defined implicitly by the exploration strategy, in terms of a particular set of transformations that are attempted. This makes it hard to find conditions that guarantee the completeness of the optimizer.

In a VLDB'99 paper [8] we have introduced the **chase & backchase** method in answer to some of these problems. Since then we have made significant progress in making this approach practical. We report this progress here. The following motivating example will help us summarize some of our new contributions.

1.1 Motivating Example

Consider the following ODMG/ODL schema definition, in which `Teams` associates projects with names of their members, `Payroll` associates departments with employees and `Dept` is a class whose set-valued attribute `DProjs` contains the projects run by each department. We will need to consider both **set** and **bag** semantics hence the `Set/Bag`, and we use a b superscript for the bag semantics: $\texttt{Teams}^b$, $\texttt{Payroll}^b$.

```
class Dept                    Teams/Teams^b: set/bag<struct{    Payroll/Payroll^b: set/bag<struct{
  (extent depts key DName){                  string TProj;                         string PDept;
  attribute string DName;                    string TMember;}>                     string Empl;}>
  attribute set<string> DProjs;}
```

Consider also an ODMG/OQL query Q that returns the members of projects run by the `"Security"` department. Here too we consider set and bag semantics: Q is *with* <u>distinct</u> , while Q^b is without.

$$
\begin{array}{lll}
Q/Q^b: & \underline{\text{select}} & [\underline{\text{distinct}}]\ \underline{\text{struct}}\ (\texttt{E}: t.\texttt{TMember}) \\
 & \underline{\text{from}} & \texttt{depts}\ d,\ d.\texttt{DProjs}\ pn,\ \texttt{Teams}\ t \\
 & \underline{\text{where}} & pn = t.\texttt{TProj}\ \underline{\text{and}}\ d.\texttt{DName} = \texttt{"Security"}
\end{array}
$$

Assume that all employees must receive reports about all projects run by their department. This is facilitated by materializing the view V_1 below. Assume also the materialized view V_2 below that joins `Teams` and `Payroll`. We consider both set and bag semantics for these views. Note also that the schema does not require that project members also be employees.

$$
\begin{array}{lll}
V_1/V_1^b: & \underline{\text{select}} & [\underline{\text{distinct}}]\ \underline{\text{struct}}\ (\texttt{D}: d.\texttt{DName},\ \texttt{P}: pn,\ \texttt{E}: p.\texttt{Empl}) \\
 & \underline{\text{from}} & \texttt{depts}\ d,\ d.\texttt{DProjs}\ pn,\ \texttt{Payroll}\ p \\
 & \underline{\text{where}} & d.\texttt{DName} = p.\texttt{PDept}
\end{array}
$$

$$
\begin{array}{lll}
V_2/V_2^b: & \underline{\text{select}} & [\underline{\text{distinct}}]\ \underline{\text{struct}}\ (\texttt{E}: t.\texttt{TMember},\ \texttt{D}: p.\texttt{PDept},\ \texttt{P}: t.\texttt{TProj}) \\
 & \underline{\text{from}} & \texttt{Teams}\ t,\ \texttt{Payroll}\ p \\
 & \underline{\text{where}} & t.\texttt{TMember} = p.\texttt{Empl}
\end{array}
$$

We will use this schema, query and views to illustrate two important aspects that have guided us:
1. Integrity constraints can help in answering queries using views. A query may not in general be equivalent to another query that uses given views. But additional equivalences may hold in the presence of integrity constraints. Hence an equivalent query that uses the views might be found. We explain this aspect in **set** semantics. For example, V_1, V_2 cannot in general be used to answer Q: if there are some team members of projects run by `"Security"` that are not on the `Payroll` (outside consultants), then any combination of V_1 and V_2 will miss these, because both views only store persons on the `Payroll`.

The situation changes if we assume the integrity constraint (*insider*) that states that `"Security"` uses only its own employees on the projects it runs. Then, Q is equivalent to R:

$$
(insider)\ \ \forall(d \in \texttt{depts})(pn \in d.\texttt{DProjs})(t \in \texttt{Teams})\ \ pn = t.\texttt{TProj} \wedge d.\texttt{DName} = \texttt{"Security"} \rightarrow \exists(p \in \texttt{Payroll})\ p.\texttt{PDept} = d.\texttt{DName} \wedge p.\texttt{Empl} = t.\texttt{TMember}
$$

$$
\begin{array}{lll}
R/R^b: & \underline{\text{select}} & [\underline{\text{distinct}}]\ \underline{\text{struct}}\ (\texttt{E}: v_1.\texttt{E}) \\
 & \underline{\text{from}} & V_1\ v_1,\ V_2\ v_2 \\
 & \underline{\text{where}} & v_1.\texttt{D} = \texttt{"Security"} \\
 & & \underline{\text{and}}\ v_1.\texttt{P} = v_2.\texttt{P}\ \underline{\text{and}}\ v_1.\texttt{E} = v_2.\texttt{E} \\
 & & \underline{\text{and}}\ v_1.\texttt{D} = v_2.\texttt{D}
\end{array}
$$

Our approach is the first that considers answering queries using views in the presence of integrity constraints

general enough to include for example (*insider*). In particular the methods in [4, 17, 12, 11, 24, 3, 27] will not discover the rewriting of Q to R.

2. When the schema, query, or views have bag semantics, we may need stronger integrity constraints. This is because the *multiplicities* (number of copies) of tuples may be different depending on whether the views are used or not. Stronger integrity constraints can insure this does not happen. For instance, if $\texttt{Payroll}^b$ contains the duplicate [2] entry (PDept : "Security", Empl : "John"), and "John" is a member of project "p1" run in the "Security" department, V_1^b will return two tuples (D : "Security", P : "p1", E : "John") and so will V_2^b. Hence, R^b will return four tuples (E : "John"), as opposed to only one returned by Q^b. The integrity constraint (*insider*) is not strong enough to prevent this. What we would like to say in this constraint is not just that $\ldots\exists(p \in \texttt{Payroll})\ldots$ but that there exists a *unique* such p! A possible notation for this would be:

$$(insider^b) \;\; \forall(d \in \texttt{depts})(pn \in d.\texttt{DProjs})(t \in \texttt{Teams}^b)pn = t.\texttt{TProj} \;\wedge d.\texttt{DName} = \texttt{"Security"} \\ \to \exists\,!\,(p \in \texttt{Payroll}^b)p.\texttt{PDept} = d.\texttt{DName} \wedge p.\texttt{Empl} = t.\texttt{TMember}$$

However, we have to be careful because $\texttt{Teams}^b$ and $\texttt{Payroll}^b$ are bags, not sets, and therefore standard logic does not give a meaning to such an assertion. This leads us to consider in this paper a new class of integrity constraints for bag semantics that we call *Unique Witness Dependencies* (UWDs). Their definition and the exact meaning of ($insider^b$) is given in section 2.

It turns out that the common practice of asserting key and referential integrity (foreign key) constraints in bag semantics schemas corresponds to a UWD. Our method handles answering queries using views in the presence of such constraints and finding query rewritings that previous approaches miss, for example rewriting Q^b to R^b above in the presence of ($insider^b$) (more examples in section 2).

1.2 Contributions and Some Related Work

Mixed semantics In practice, schema elements are often sets, while views and queries are often bags, defined without using the distinct keyword (bag-set semantics). We develop here techniques for bag semantics, bag-specific constraints (UWDs), and for handling bag queries over arbitrary mixes of bag and set schema elements and views (this includes bag-set semantics), building on our previous results on set semantics [8].

The main novelty of the C&B method (we use C&Bas a shorthand for chase & backchase) is in the way it constructs the search space for rewritings, namely as the result of *chasing* the query with constraints that capture materialized views as well as integrity constraints [8]. We understand materialized views broadly, including user-defined views, cached queries and physical access structures (such as join indexes [26], access support relations [15], and gmaps [25]). The chase result, a query we called the "universal plan", gathers redundantly all the ways to answer the original query. The search space is defined by the subqueries of the universal plan and explored by *backchase*. Thus we do rewriting with views, semantic optimization, and query minimization all in one.

Completeness The theoretical contributions of this paper are theorems that state that the C&B method is *complete*, i.e., will always find an optimal plan if one exists, using given views and under given constraints. A preliminary result in [8] showed completeness only in the absence of integrity constraints and only for set semantics. Here we prove completeness for mixed semantics and both set and bag integrity constraints.

Of course, such theorems are limited by standard undecidability and incompleteness barriers. Still, ours appear to be the first completeness results that assume very general and practically relevant classes of integrity constraints. Algorithms that are complete in the absence of constraints are given in [4] for bag

[2] Schema relations are often duplicate-free. But the problem really stems from the bag semantics of the *views*. Assume that $\texttt{Payroll}^b$ has additional attributes (eg., ContractNo). Even if $\texttt{Payroll}^b$ is duplicate-free, if its projection on PDept and Empl has duplicates then we have the same multiplicity problem.

semantics and in [17] for set semantics. The algorithm in [9] is complete in the presence of just functional dependencies.

This paper continues [21, 8, 20, 19] in using for queries, views and plans a language which uses *dictionaries* to express object/relational and object-oriented features, as in ODMG and in SQL:1999 and its extensions, as well as a variety of physical access structures, including indexes. The integrity constraints are expressed in a logic that corresponds to the same language, capturing common OO/relational integrity constraints, such as functional dependencies/key constraints, referential integrity (foreign key), lossless join and inverse relationship constraints. The C&B algorithm is defined on the *path-conjunctive* restriction of this language, while the completeness theorems require constraints to be *full* (so that the chase is guaranteed to terminate). To avoid cluttering the early sections, we postpone the precise definitions of the plan and constraint language to section 7.

The structure of the optimizer we proposed in [8] separated C&B optimization as a stage preceding standard cost-based query optimization. In fact, it was not clear that the chase or backchase were feasible at all, given the theoretical intractability of the chase [1]. Subsequently, we developed implementation techniques for the chase and the backchase and reported on their feasibility in [20]. That work still kept the C&B stage separate from any cost-based processing. However, there were strong indications that the C&B search space exploration could itself exploit cost information.

Backchase with cost-based pruning The main practical contribution of this paper is that we show how to restructure the exploration of the search space to allow for cost-based pruning using dynamic programming and we report on experimental results showing the benefit of combining our technique with System R-style cost-based optimization. As an additional improvement we show that we can postpone primary and secondary index selection to the cost-based phase, reducing the number of constraints used in chasing while preserving completeness.

The complete (no constraints, pure bag semantics) algorithm in [4] includes cost-based dynamic programming and, in fact, in the absence of integrity constraints our algorithm does the same work (same efficiency and same simplicity in implementation). Minicon [22] builds on the ideas of [17] and improves scalability with the number of views, however the algorithm is designed for a different purpose (data integration) and it doesn't consider any integrity constraints. The algorithm in [25], is also integrated with cost-based dynamic programming but uses a restricted class of constraints, is restricted to SPJ queries and views with set semantics in which every relation appears at most once, and is incomplete [3].

Grouping In this paper we extend our theoretical and practical work to cover a class of *grouping* views, with both set and bag semantics and with the same classes of set and bag constraints. For the completeness theorems we still have an important restriction: the grouping views are indeed nested (at arbitrary depth in fact) but the queries themselves have flat output. Also, we can decide equivalence of grouping queries in the presence of the same integrity constraints.

Even though we do not know if the method is complete, we can still apply the C&B to a nested query by rewriting each block. Then, the method can be combined with some of the techniques for handling aggregates in [24, 3, 27] and thus be applied to certain queries and views with aggregation. Space does not allow us to get into details here.

Algorithms that perform query rewriting with views in Oracle and DB2 are described in [3] (with some aggregation and some classes of constraints) and in [27] (concentrates on aggregation). A decision procedure for so-called *weak equivalence* of nested set queries but in the absence of integrity constraints is given in [18]. Both [24, 7] rewrite aggregate queries using aggregate views, but no integrity constraints are exploited and no queries with nested output are considered. [24] also gives a completeness result but for views without grouping and aggregation.

[3] Purposefully so, since they propose a PTIME algorithm for an NP-hard problem.

Overview of the rest of the paper. In section 2 we define and justify the (new) class of bag constraints that we can exploit systematically in C&B. In section 3 we review (following [8]) the C&B method for just set semantics. In section 4 we discuss the (new) chase and backchase of bag queries with bag integrity constraints and bag views. The (new) completeness theorem for quite general classes of queries, views and constraints, and with both bag and set semantics, is in section 5. The (new) algorithm that performs the backchase phase in a bottom-up fashion in conjunction with dynamic-programming cost-based pruning is described in section 6. Section 7 gives formal details on the query and constraint languages and the chase in bag semantics. In section 8 we discuss a (new) implementation improvement that allows us to postpone the consideration of indexes to the cost-based phase. In section 9 we report on (new) experiments that demonstrate the benefit of cost-based pruning in conjunction with the backchase, for both relational and object-oriented configurations. We also show how our method scales in practical cases and we discuss the tradeoff between plan quality and optimization time. In section 10 we sketch our (new) extension of the C&B method to grouping views. Conclusions and further work complete the paper.

2 Integrity Constraints in Bag Semantics

Previous work on rewriting using views in bag semantics does not give any systematic way of exploiting integrity constraints. However, SQL schemas do assert some constraints and these sometimes enable the use of views. For instance, revisiting the motivating example in section 1.1, the bag query Q^b can be rewritten as:

$$\begin{array}{lll} R_1^b: & \underline{\text{select}} & \underline{\text{struct}}\,(\texttt{E}: v_2.\texttt{E}) \\ & \underline{\text{from}} & \texttt{depts}\ d, d.\texttt{DProjs}\ pn, V_2\ v_2 \\ & \underline{\text{where}} & d.\texttt{DName} = \text{“}Security\text{''}\ \underline{\text{and}}\ pn = v_2.\texttt{P}\ \underline{\text{and}}\ d.\texttt{DName} = v_2.\texttt{D} \end{array}$$

provided that $\texttt{Payroll}^b$ contains no duplicates, `Empl` is a key in $\texttt{Payroll}^b$, and `TMember` is a foreign key in $\texttt{Teams}^b$ referencing `Empl` in $\texttt{Payroll}^b$. In fact, it suffices for $\texttt{Payroll}^b$ to not contain duplicates of just the tuples mentioning project team members (in addition to the key and foreign key constraints).

Of course, SQL itself does not provide a *general* notation for stating assertions about bags. However, the C&B method relies on such a notation. For a bag M, let us denote by $\underline{\text{dom}}\, M$ the *set* of values of its elements and by $M[m]$ the number of occurrences of m (m's *multiplicity* in M). Now we can state the key [4] and the foreign key constraints as

$$\begin{array}{ll} (fk) & \forall(t \in \underline{\text{dom}}\,\texttt{Teams}^b)\exists(p \in \underline{\text{dom}}\,\texttt{Payroll}^b)t.\texttt{TMember} = p.\texttt{Empl} \\ (key) & \forall(p_1 \in \underline{\text{dom}}\,\texttt{Payroll}^b)(p_2 \in \underline{\text{dom}}\,\texttt{Payroll}^b)p_1.\texttt{Empl} = p_2.\texttt{Empl} \to p_1 = p_2 \end{array}$$

and the constraints about no duplicates in $\texttt{Payroll}^b$ and in just the tuples mentioning project team members as

$$\begin{array}{ll} (noDup_{\texttt{Payroll}}) & \forall(p \in \underline{\text{dom}}\,\texttt{Payroll}^b)\texttt{Payroll}^b[p] = 1 \\ (noDup_{\texttt{PayrollTeams}}) & \forall(t \in \underline{\text{dom}}\,\texttt{Teams}^b)(p \in \underline{\text{dom}}\,\texttt{Payroll}^b)t.\texttt{TMember} = p.\texttt{Empl} \to \texttt{Payroll}^b[p] = 1 \end{array}$$

Note that $(noDup_{\texttt{Payroll}})$ implies $(noDup_{\texttt{PayrollTeams}})$ and that $(key) + (fk) + (noDup_{\texttt{PayrollTeams}})$ suffice to justify rewriting Q^b as R_1^b. A similar combination of three assertions captures what was intended by $(insider^b)$ in section 1.1 to justify R^b. Our analysis of constraints in bag semantics reveals this pattern of three assertions as occurring often. We therefore introduce a name and a notation which bundles together such a combination of assertions. Section 4 will show that this choice is not arbitrary: it turns out that these allow sound rewriting of queries with bag semantics in an analogous way to set semantics.

Definition 2.1 (Unique Witness Dependencies (UWDs)) *Given an instance I in which M, N are* bags*, we say that the dependency d denoted*

$$(d)\ \forall(m \in M)\ B_1(m) \to \exists!\,(n \in N)\ B_2(m,n) \qquad (M,\ N\ are\ bags!)$$

[4] For us a key constraint is just a functional dependency and it does not imply duplicate-freeness as opposed to the UNIQUE constraint in SQL, which is actually the combination of our (key) and $(noDup_{\texttt{Payroll}})$ below.

is satisfied by I if and only if the following are true in I:

$$\begin{array}{ll} \textit{(witness)} & \forall(m \in \underline{\mathsf{dom}}\, M)\ [B_1(m) \rightarrow \exists(n \in \underline{\mathsf{dom}}\, N)\ B_2(m,n)] \\ \textit{(functionality)} & \forall(m \in \underline{\mathsf{dom}}\, M)\ (n \in \underline{\mathsf{dom}}\, N)\ (n' \in \underline{\mathsf{dom}}\, N)\ [B_1(m) \wedge B_2(m,n) \wedge B_2(m,n') \rightarrow n = n'] \\ \textit{(multiplicity)} & \forall(m \in \underline{\mathsf{dom}}\, M)\ (n \in \underline{\mathsf{dom}}\, N)\ [B_1(m) \wedge B_2(m,n) \rightarrow N[n] = 1] \end{array}$$

B_1, B_2 *are conjunctions of equalities of variables or their projections on attributes* [5]*. The existentially quantified variable may be absent, in which case the meaning of* $\forall(m \in M)B_1 \rightarrow B_2$ *is the simplification of (witness) above.*

Observe that $(fk), (key), (noDup_{\mathtt{PayrollTeams}})$, are respectively the $(witness), (functionality), (multiplicity)$ of the UWD $(fk^b) : \forall(t \in \mathtt{Teams}^b)\exists!\,(p \in \mathtt{Payroll}^b)\ t.\mathtt{TMember} = p.\mathtt{Empl}$.

UWDs also allow us to handle applications mixing bags and sets, by considering all database relations as bags by default, and stating for some of them that they allow no duplicates. For $\mathtt{Payroll}^b$, this is expressed by the UWD $\forall(p \in \mathtt{Payroll}^b)\exists!\,(p' \in \mathtt{Payroll}^b)p' = p$. Indeed, the corresponding (witness) and (functionality) dependencies are trivially satisfied, while the (multiplicity) dependency is trivially equivalent to $(noDup_{\mathtt{Payroll}})$ above. As a particular case, we use this kind of "no-duplicates" constraints to model so-called *bag-set semantics* (bag queries over set relations [5]).

We show in section 4 how we use UWDs to rewrite bag queries. Our approach automatically detects and exploits *all* UWDs implied by common integrity constraints (like (fk^b)). Because of space limitations, we do not elaborate on this point.

Remark. The fact that $(insider)$ enables the rewriting R and $(insider^b)$ enables R^b does not mean that in general we can reduce the problem of rewriting with bag semantics to set semantics. To see this, construct V_2' from V_2 by dropping the D attribute from its output, and R' from R by using V_2' instead of V_2 and dropping $v_1.\mathtt{D} = v_2.\mathtt{D}$ from R's where clause. Now $(insider)$ enables the rewriting R', but $(insider^b)$ does not enable R'^b, even if $\mathtt{Payroll}^b$ contains no duplicates! This is because an employee may be on the payroll of several departments, again affecting the multiplicities of some tuples in R'^b's answer: suppose that Payroll contains two entries for John, (PDept : "Retail", Empl : "John") and (PDept : "Security", Empl : "John"). Then V_1^b returns (D : "Security", P : "p1", E : "John"), and $V_2'^b$ returns both this tuple and (D : "Retail", P : "p1", E : "John"). R'^b hence returns (E : "John") twice, as opposed to only once by Q^b. We must have an additional "bag key constraint" for $\mathtt{Payroll}^b$ on its Empl attribute to infer that R'^b is an equivalent rewriting. This is expressed as the UWD (bk) $\forall(p_1 \in \mathtt{Payroll}^b)\ (p_2 \in \mathtt{Payroll}^b)\ p_1.\mathtt{Empl} = p_2.\mathtt{Empl} \rightarrow p_1 = p_2$, and our algorithm obtains R'^b in its presence.

3 Review of the C&B Algorithm for Set Semantics

We are given a set V of views (in the broad sense), each individual view V_i from V being defined by a query QV_i. In the C&B approach, each V_i is captured with a pair of inclusion constraints: $V_i \subseteq QV_i$ and $QV_i \subseteq V_i$. Denote the set of all such pairs of constraints with Σ_V. Then a rewriting of a query Q against the logical schema S in the presence of the integrity constraints in Σ_C is found as a query written in terms of the views in V, equivalent to Q on every $S \cup V$-instance satisfying $\Sigma_C \cup \Sigma_V$.

Example. Recalling the motivating example from section 1.1, the constraints capturing V_1 are

$$\begin{array}{ll} (c_{V_1}) & \forall(d \in \mathtt{depts})(pn \in d.\mathtt{DProjs})(p \in \mathtt{Payroll}) \\ & \quad [d.\mathtt{DName} = p.\mathtt{PDept} \rightarrow \exists(v_1 \in V_1)v_1.\mathtt{D} = d.\mathtt{DName} \wedge v_1.\mathtt{P} = pn \wedge v_1.\mathtt{E} = p.\mathtt{Empl}] \\ (b_{V_1}) & \forall(v_1 \in V_1)\exists(d \in \mathtt{depts})(pn \in d.\mathtt{DProjs})(p \in \mathtt{Payroll}) \\ & \quad d.\mathtt{DName} = p.\mathtt{PDept} \wedge v_1.\mathtt{D} = d.\mathtt{DName} \wedge v_1.\mathtt{P} = pn \wedge v_1.\mathtt{E} = p.\mathtt{Empl} \end{array}$$

For reasons that will become apparent shortly, (c_{V_1}) is called V_1's *chase-in dependency*, and (b_{V_1}) is called its *backchase dependency.* •

[5]We actually allow equalities of *path expressions*, which generalize projections and are detailed in section 7. Also, we omit the straightforward generalization to multiple quantified variables.

The algorithm has two phases: the first is called the **chase**, and it rewrites Q with the semantic constraints in Σ_C and the chase-in dependencies in Σ_V, obtaining a query called the **universal plan** UP, which explicitly mentions all physical access paths that can be used to answer Q. The second phase is the **backchase**, which searches for P among the subqueries (defined shortly) of UP.

Phase 1: Chase. The constraints we use in rewriting belong to the class of *Embedded Path-Conjunctive Dependencies (EPCDs)*, which is defined in section 7 and only illustrated through examples here. EPCDs are logical assertions of the form $\forall(s_1 \in S_1)\dots\forall(s_k \in S_k)[B_1 \rightarrow \exists(t_1 \in T_1)\dots\exists(t_l \in T_l)B_2]$. The corresponding *chase step* (in its simplest form) with an EPCD d is the rewrite

$$\text{(1)}\quad \begin{array}{ll} \underline{\text{select}} & \underline{\text{distinct}}\ O(s_1,\dots,s_k) \\ \underline{\text{from}} & \dots, S_1\ s_1,\dots,S_k\ s_k,\dots \\ \underline{\text{where}} & \dots\ \underline{\text{and}}\ B_1\ \underline{\text{and}}\ \dots \end{array} \quad \xrightarrow{d} \quad \begin{array}{ll} \underline{\text{select}} & \underline{\text{distinct}}\ O(s_1,\dots,s_k) \\ \underline{\text{from}} & \dots, S_1\ s_1,\dots,S_k\ s_k,\ T_1\ t_1,\dots,T_l\ t_l,\dots \\ \underline{\text{where}} & \dots\ \underline{\text{and}}\ B_1\ \underline{\text{and}}\ B_2\ \underline{\text{and}}\ \dots \end{array}$$

Example. Our example query Q chases in one step using the EPCD (*insider*) to the following query (notice how the new variable binding `Payroll` p and the conditions involving p are added to Q):

$$Q_1:\ \begin{array}{ll} \underline{\text{select}} & \underline{\text{distinct}}\ \underline{\text{struct}}\ (\texttt{E}: t.\texttt{TMember}) \\ \underline{\text{from}} & \texttt{depts}\ d, d.\texttt{DProjs}\ pn, \texttt{Teams}\ t,\ \texttt{Payroll}\ p \\ \underline{\text{where}} & pn = t.\texttt{TProj}\ \underline{\text{and}}\ d.\texttt{DName} = \text{“}\textit{Security}\text{''}\ \underline{\text{and}}\ p.\texttt{PDept} = d.\texttt{DName}\ \underline{\text{and}}\ p.\texttt{Empl} = t.\texttt{TMember} \end{array}$$

The *chase phase* consists in repeatedly performing chase steps with any applicable constraint from $\Sigma_C \cup \Sigma_V$. “Applicability” must be defined carefully to avoid unnecessary duplication of variable bindings and to allow for chasing even when query and constraint do not match syntactically as in the example, but are related by the existence of a *homomorphism* from $s_1,\dots,s_k$ to the variables of the query. This notion is defined in section 7 as introduced in [21], which shows that the resulting chase procedure is a generalization of the classical relational chase [1] to our richer query language and constraints. [21] also shows that, while in general the chase may not terminate, it does so for the class of *full* EPCDs (see section 7), yielding a unique result UP whose size is polynomial in that of Q.

Example. (*insider*) is a full EPCD. By bringing the variable binding and associated conditions for p into Q_1, the chase step with (*insider*) enables further chase steps of Q_1 with c_{V_1} and c_{V_2} to obtain UP (no other chase steps apply).

$$UP:\ \begin{array}{ll} \underline{\text{select}} & \underline{\text{distinct}}\ \underline{\text{struct}}\ (\texttt{E}: t.\texttt{TMember}) \\ \underline{\text{from}} & \texttt{depts}\ d, d.\texttt{DProjs}\ pn, \texttt{Teams}\ t,\ \texttt{Payroll}\ p,\ V_1\ v_1,\ V_2\ v_2 \\ \underline{\text{where}} & pn = t.\texttt{TProj}\ \underline{\text{and}}\ d.\texttt{DName} = \text{“}\textit{Security}\text{''}\ \underline{\text{and}}\ p.\texttt{PDept} = d.\texttt{DName}\ \underline{\text{and}}\ p.\texttt{Empl} = t.\texttt{TMember} \\ & \underline{\text{and}}\ v_1.\texttt{D} = d.\texttt{DName}\ \underline{\text{and}}\ v_1.\texttt{P} = pn\ \underline{\text{and}}\ v_1.\texttt{E} = p.\texttt{Empl} \\ & \underline{\text{and}}\ v_2.\texttt{D} = p.\texttt{PDept}\ \underline{\text{and}}\ v_2.\texttt{E} = t.\texttt{TMember}\ \underline{\text{and}}\ v_2.\texttt{P} = t.\texttt{TProj} \end{array}$$

Notice that the effect of chasing with c_{V_1} and c_{V_2} is to explicitly bring into UP the views V_1, V_2, as well as the join conditions that relate them to the rest of the query. This wouldn't be possible in the absence of (*insider*), and rightfully so, because V_1, V_2 wouldn't be usable for answering Q. •

Phase 2: Backchase. Conceptually, the backchase phase considers all **subqueries** of the universal plan and keeps those which are equivalent to Q and for whom the removal of any variable binding (called *scan*) would compromise this equivalence. We call such queries **scan-minimal** rewritings of Q in the presence of Σ_C. They are important because under a **monotonic cost assumption** (i.e. when the execution cost of a query is always greater than the cost of any of its subqueries), a scan-minimal query is always cheaper than its non-minimal superqueries [6]. While this assumption is reasonable, there are practical examples which violate it. Section 8 shows how we deal with them.

A subquery SQ is obtained by picking a subset SV of UP's variable bindings, as well as all equalities implied by the UP's where clause and involving solely the picked variables [7]. The subquery's select clause must depend only on the its variables, and this is possible only if UP's conditions imply the equality of its own select clause with that of SQ. In this case we say that SV **induces** SQ.

[6]There is also a technical reason: there are infinitely many non-minimal rewritings (just duplicate some scan repeatedly), hence no hope for completeness for them. In focussing on scan-minimal rewritings, we follow [17].

[7]Implication reduces to membership in the reflexive, symmetric, transitive congruence closure of the equalities in UP, which is computed in PTIME [21].

Example. R from section 1.1 is a subquery of UP induced by $V_1\ v_1, V_2\ v_2$ since for instance $v_1.\texttt{D} = v_2.\texttt{D}$ is implied by $v_1.\texttt{D} = d.\texttt{DName}$, $d.\texttt{DName} = p.\texttt{PDept}$ and $p.\texttt{PDept} = v_2.\texttt{D}$. R_1 from section 2 is induced by $\texttt{Dept}\ d, d.\texttt{DName}\ pn, V_2\ v_2$ and there is a previously unmentioned rewriting found as the subquery induced by $V_1\ v_1, \texttt{Teams}\ t$: R_2 : select distinct struct ($\texttt{E} : t.\texttt{TMember}$) from $V_1\ v_1, \texttt{Teams}\ t$ where $v_1.\texttt{D} =$ "*Security*" and $v_1.\texttt{E} = t.\texttt{TMember}$ and $v_1.\texttt{P} = t.\texttt{TProj}$ •

A subquery SQ of the universal plan is tested for equivalence to Q by chasing both with $\Sigma_C \cup \Sigma_V$ and looking for *containment mappings* (see section 7) between the chase results [21].

4 The C&B Algorithm for Bag Semantics

Phase 1: Chase. We ask if there exists a transformation on bag queries that is analogous to the chase of a set query with a constraint d (recall (1) in Section 3)? If so, its simplest form would look like this (notice the absence of the distinct keyword):

$$(2)\quad \begin{array}{ll} \underline{\text{select}} & O(m) \\ \underline{\text{from}} & \ldots, M\ m, \ldots \\ \underline{\text{where}} & \ldots\ \underline{\text{and}}\ B_1\ \underline{\text{and}}\ \ldots \end{array} \quad \xrightarrow{d} \quad \begin{array}{ll} \underline{\text{select}} & O(m) \\ \underline{\text{from}} & \ldots, M\ m,\ N\ n, \ldots \\ \underline{\text{where}} & \ldots\ \underline{\text{and}}\ B_1\ \underline{\text{and}}\ B_2\ \underline{\text{and}}\ \ldots \end{array}$$

The analogy stops when we realize that d cannot be just the assertion $\forall(m \in \underline{\text{dom}}\, M)[B_1 \to \exists(n \in \underline{\text{dom}}\, N)B_2]$ (recall the notation from section 2). For example, if for some x in $\underline{\text{dom}}\, M$ for which $B(x)$ holds there are distinct tuple values y_1, y_2 in $\underline{\text{dom}}\, N$ satisfying $B_2(x, y_1)$ and $B_2(x, y_2)$, then the multiplicity of $O(x)$ in the right hand side of (2) is at least twice its multiplicity in the left hand side. Worse, (2) is not sound even if there is actually a unique value y in $\underline{\text{dom}}\, N$ satisfying $B_2(x, y)$, but its associated multiplicity $N[y]$ is greater than 1. On the other hand, requiring uniqueness of the value y in $\underline{\text{dom}}\, N$ that satisfies $B_2(x, y)$, *and* unit multiplicity for y will suffice to make (2) sound for all $O(x)$.

Chasing with UWDs. We recognize from the above discussion that the conditions under which transformation (2) is sound are equivalent to the (witness), (functionality) and (multiplicity) constraints (as introduced in definition 2.1) for the UWD $(d)\ \forall(m \in M)[B_1 \to \exists!(n \in N)B_2]$. Transformation (2) is hence an example of a new kind of chase, namely that of a bag query with the UWD (d).

The full definition of the chase step with UWDs is more involved than transformation (2) above. It is inspired by the definition of UWDs (2.1) and given in section 7. We show there that the chase step is a *sound* transformation, i.e. it preserves equivalence under UWDs. Moreover, we show how to use this new chase for *deciding* equivalence of bag queries under UWDs.

The main difference from chasing under set semantics is that, as we chase a bag query Q, we infer that some of its variables are guaranteed to range only over unique tuple values, of multiplicity 1, in any instance satisfying the UWD. We denote these variables with $\mathrm{UW}(Q)$. Initially, $\mathrm{UW}(Q)$ is the empty set, but the chase step can update both Q and $\mathrm{UW}(Q)$.

Example. Recall the example from section 1.1. 2. Starting with an empty $\mathrm{UW}(Q^b)$, Q^b chases with $(insider^b)$ to

$$\begin{array}{lll} Q_1^b : & \underline{\text{select}} & \underline{\text{struct}}\ (\texttt{E} : t.\texttt{TMember}) \\ & \underline{\text{from}} & \texttt{depts}\ d, d.\texttt{DProjs}\ pn, \texttt{Teams}\ t,\ \texttt{Payroll}\ p \\ & \underline{\text{where}} & pn = t.\texttt{TProj}\ \underline{\text{and}}\ d.\texttt{DName} = \text{"}Security\text{"}\ \underline{\text{and}}\ p.\texttt{PDept} = d.\texttt{DName}\ \underline{\text{and}}\ p.\texttt{Empl} = t.\texttt{TMember} \end{array}$$

and the fact that p is known to range only over unique witnesses is recorded: $\mathrm{UW}(Q_1^b) = \{p\}$. •

We now focus on the search space for rewritings. Recall from section 3 that, for set semantics, the views that are relevant to answering the query are brought into the universal plan as a result of chasing with their chase-in dependecies from Σ_V. For bag semantics we only know how to chase with UWDs, which unfortunately do not capture the views in general. To see why this is the case, consider the view $V \stackrel{\text{def}}{=} \underline{\text{select}}\ O\ \underline{\text{from}}\ M\ m\ \underline{\text{where}}\ B$ and, analogously to the set semantics case, the chase-in dependency $c_V^b \stackrel{\text{def}}{=} \forall(m \in M)\ B \to \exists!(v \in V)\ v = O$. In general, (c_V^b) is not satisfied because the unit multiplicity requirement on the tuples bound by v is not guaranteed to hold, even if M is a set (just consider that O projects on some

non-key attribute). (c^b_V) would hold if V were defined using select distinct , as is the case for set semantics. We therefore introduce the concept of chasing a bag query with a bag view:

Chasing with a view; Universal Plan. The result of chasing a query Q with a view V is the result of chasing Q with (c^b_V), without updating UW(Q). The formal definition is given in section 7, and it contains subtleties on the applicability of this chase step with a view. For the moment, we only note that the effect of chasing with a view is that of bringing it into the universal plan, just like in the set case. We emphasize that since (c^b_V) is not guaranteed to hold, this transformation is not equivalence-preserving. Still, it enables us to define the search space for scan-minimal rewritings of Q as being the subqueries of the **universal plan** obtained by chasing Q with the semantics constraints expressed as the set of UWDs Σ_C, and then with the views. Theorem 5.1 shows that for the important class of *full* UWDs, this universal plan is unique, polynomial in the size of Q, and contains all scan-minimal rewritings.

Example. Revisiting the motivating example, we can easily check that the universal plan UP^b obtained by first chasing Q^b with ($insider^b$) to Q^b_1, and then chasing Q^b_1 with the views V^b_1, V^b_2, is the same as UP after dropping the distinct keyword. R^b and R^b_1 are subqueries of UP^b (there are others, such as R^b_2, the bag version of R_2 from the example in section 3, which is also a rewriting). •

Phase 2: Backchase. In this phase, we must find those subqueries of the universal plan which are equivalent to Q. Section 3 showed that for set semantics, the equivalence check is reducible to checking equivalence under the constraints capturing the views. While we know how to check equivalence under UWDs (shown below in theorem 7.1), remember that bag views are not captured by them.

Unfolding. The problem of checking a subquery for equivalence to Q can nevertheless be reduced to deciding bag equivalence under Σ_C, by using the notion of *unfolding* of R w.r.t. to the set of views $V = (V_1, \ldots, V_n)$ defined by the queries Q_i. The result is denoted $\mathsf{unfold}_V(R)$ and obtained as follows: for every variable binding $V_i\ v_i$ in R, (i) rename the variables in Q_i to fresh ones, obtaining Q'_i, (ii) construct R_1 by substituting $V_i\ v_i$ in R with Q_i's from clause, (iii) construct R_2 by adding Q_i's where clause to R_1's and (iv) for every component C : *expr* in Q'_i's select clause, substitute *expr* for v_i.C in R_2.

Example. The unfolding of R^b w.r.t. V_1, V_2 is

```
UFR^b : select struct (E : p1.Empl)
        from   depts d, d.DProjs pn, Payroll p1, Teams t, Payroll p2
        where  d.DName = p1.PDept and t.TMember = p2.Empl and d.DName = "Security"
               and pn = t.TProj and p1.Empl = t.TMember and d.DName = p2.PDept •
```

The following result is used to check that R^b is indeed a rewriting:

Proposition 4.1 *R is a rewriting of Q using the views in V in the presence of the UWDs in Σ_C if and only if* $\mathsf{unfold}_V(R)$ *is equivalent to Q under Σ_C.*

The equivalence of $\mathsf{unfold}_V(R)$ under Σ_C is checked by chasing, according to theorem 7.1. A detailed example for R^b is deferred until after theorem 7.1.

5 Completeness of the C&B Method

The following justifies why it is enough to restrict our search to the subqueries of the universal plan, for both set and bag semantics. The result shows that the C&B algorithm handles also the general case in which some of the database relations are sets, and some of the views have set semantics (were defined using select distinct). We state the result in its full generality for path-conjunctive (PC) queries and (full) EPCD and UWD dependencies, which are all defined in section 7.

We are given a schema with bag and set elements. Let the corresponding semantic constraints be expressed by a set Σ_C of full EPCDs, and by a set Σ^b_C of full UWDs. Also let V^b, V^s be sets of views with bag, respectively set semantics.

Given a PC query Q, we obtain the universal plan UP as follows: if Q has set semantics, interpret all schema elements as sets and the bag views as set views. Capture *all* views with the EPCDs Σ_V and

translate all UWDs to their corresponding (witness) and (functionality) EPCDs Σ_{transl}. UP is the result of chasing Q with $\Sigma_C \cup \Sigma_V \cup \Sigma_{transl}$. If Q has bag semantics, add for every set schema element the "no-duplicates" UWD to Σ^b_{nodup}. Let $\Sigma^b_{implied}$ be all UWDs implied from Σ_C and the "no-duplicate" constraints (as in section 2) [8] . Capture the set views with the UWDs $\Sigma^b_{V^s}$ (recall from section 4 that this can be done). UP is the result of chasing Q with $\Sigma^b_C \cup \Sigma^b_{nodup} \cup \Sigma^b_{implied} \cup \Sigma^b_{V^s}$, then with the bag views in V^b.

Theorem 5.1 *All scan-minimal rewritings of Q can be found among the subqueries of the universal plan UP, which is unique and polynomial in the size of Q.*

A particular case for set semantics (no V^b, or bag schema elements) in the absence of semantic constraints (no Σ^{EPCD} or Σ^{UWD}) was given in [8]. The completeness and decidability results of [4, 17] follow from our theorem when we restrict it to conjunctive queries in the absence of semantic constraints.

6 Backchase with Cost-Based Pruning

Backchase with cost-based pruning is our algorithm for exploring the subqueries of the universal plan. The algorithm takes as input a query Q, a universal plan UP (for the search space), a set of views V and a set of integrity constraints Σ and returns the optimal plan for Q. It works for all semantics (set, bag, mixed), and has three parameters.

The first parameter is the cost model (procedure *computeCost* below), which we don't further specify, requiring only that it be *monotonic* (recall page 7) The second parameter is a traditional optimizer (extended as in section 8) which, given a rewriting, picks the best execution plan by performing classical index selection and join reordering (procedure *computePlan* below). The third parameter is the procedure $areEquivalent(Q, SQ, V, \Sigma)$, which decides equivalence of query Q and subquery SQ of the universal plan under the views in V and the constraints in Σ. Recall from section 3 that, for set semantics, this equivalence is decided by exhibiting a containment mapping (section 7) from Q to the result of chasing SQ with Σ and Σ_V (the dependecies we use to capture the views). As we shall see shortly in theorem 7.1, for bag and bag-set semantics the equivalence is decided by exhibiting special containment mappings between the results of chasing Q and $\mathsf{unfold}_V(SQ)$ with the UWDs in Σ.

One more bit of notation before showing the algorithm: Each scan over a view $V\ v$ is uniquely identified by the mapping h used to chase it into the universal plan. We say that the scans in the image under h of V's chase-in dependency are *covered* by $V\ v$, and denote with $\mathtt{NUMS}(V\ v)$ the covered scans whose variables are not it UW(UP) (we call them non-unit multiplicity scans, which explains the notation). For example, recalling Q^b_1 from section 4, $\mathtt{NUMS}(V_2\ v_2) = \{\mathtt{Teams}\ t\}$. Recalling our discussion at the end of section 4, note that for set and bag-set semantics, all "no-duplicates" UWDs are present, hence there are no non-unit multiplicity scans and $\mathtt{NUMS}(V\ v)$ is always empty.

backchase with cost-based pruning
input: query Q, universal plan UP, views V, dependencies Σ
param: *areEquivalent*, *computePlan*, *computeCost*
output: cheapest scan-minimal rewriting according to *computeCost*.

```
    for i = 1 to number of scans in UP
(1)     for each non-pruned subset S of cardinality i of UP's scans
            if S contains a scan over a view Vv such that NUMS(Vv) ∩ S ≠ ∅
(2)            prune all supersets of S and continue
            if S induces no subquery of UP^b continue
            let SQ be the subquery induced by S, plan = computePlan(SQ), cost = computeCost(plan) in
(3)             if plan is not cheapest so far, prune all supersets of S and continue
```

[8]In practice, we do not compute these, but rather achieve their effect by translating UWDs constraints to their witness, functionality and multiplicity EPCDs, so they can interact with Σ_C. Space does not allow us to elaborate on this subtle point!

if $areEquivalent(SQ, Q, V, \Sigma)$
(4) record *plan* and *cost* as best so far and prune all supersets of S
return best plan so far

Because the universal plan contains all scan-minimal rewritings among its subqueries and because we explore all non-pruned subqueries (see step (1)), it is enough to show that no pruning step misses the optimal plan. Then completeness of our algorithm follows.

Step (3) performs cost-based pruning, and it is correct only under the monotonic cost assumption. Step (4) prunes only those rewritings that are not scan-minimal, because they already contain the rewriting induced by S. The most subtle pruning step is step (2). As discussed above it never applies for set and bag-set semantics. This step rules out subqueries containing both a scan over a view and some of the `NUMS` it covers. We omit the proof of why this pruning preserves optimality, but we illustrate it to give some intuition: a subquery of UP^b which is pruned in step (3), is the one induced by $S = \{\texttt{Teams}\ t,\ V_2\ v_2\}$:
SQ : select struct (E : t.TMember) from Teams t, $V_2\ v_2$ where v_2.D = "*Security*" and v_2.E = t.TMember and v_2.P = t.TProj
pruned because $\texttt{NUMS}(V_2\ v_2) \cap S = \{\texttt{Teams}\ t\}$. It is easy to see that neither it nor its superqueries can be rewritings, because SQ iterates over `Teams` tuples twice (once hidden in the definition of V_2) as opposed to only once in Q^b. The multiplicities in SQ's result are therefore higher than those in Q^b's and they will be even higher for SQ's superqueries.

It turns out that the subqueries that are not pruned in (2) could alternatively be obtained by "substituting" view scans for the `NUMS` they cover, in all possible ways. Remarkably, for bag semantics, in the absence of integrity constraints, this operation reduces to that of *substituting a view* from [4], and our backchase with cost-based pruning behaves just like their algorithm.

7 Formal Details

[21] presents the theory of chasing a class of queries with set semantics called *path-conjunctive (set PC)* with a class of dependencies called *embedded path-conjunctive dependencies (EPCDs)*. In this section, we develop the theory of chasing *bag PC* queries with UWDs.

Paths. The PC language (formally defined in [21]) extends conjunctive queries to complex values and dictionaries, introducing two operations: dom M returns the **domain** of the dictionary M, i.e. the set of its keys, and $M[k]$, is the result of looking up a key k in a dictionary M. We define *paths* as $P ::= x \mid c \mid SN \mid P.\mathtt{A} \mid \mathrm{dom}\, P \mid P[x]$ where x stands for variables, c for constants at base types, SN, `A` for schema, respectively attribute names. Wherever OQL allows a schema name, we allow a set/bag-typed path. The expressions in equalities of the where clause and in the select clause are paths whose type cannot be set/bag or dictionary. [8] shows how dictionaries can model indexes, which we omit here for simplicity sake. For the same reason, all queries in our examples are set/bag PC queries without dictionaries.

EPCDs and UWDs; Fullness. EPCDs generalize relational embedded dependencies [1] to our data model, and they have the logical form $\forall(m_1 \in M_1)\ldots(m_k \in M_k)\ B_1 \rightarrow \exists(n_1 \in N_1)\ldots(n_l \in N_l)B_2$ where all M_i, N_j are paths, and B_1, B_2 are conjunctions of path equalities (same restrictions as in the from, respectively where clauses of PC queries). Recalling definition 2.1, the (witness),(functionality) and (multiplicity) dependencies are examples for EPCDs [9] UWDs have the same syntax as EPCDs, except for the special quantifiers. We say that an EPCD or a UWD is **full** if B_1 implies that all components of the existentially quantified variables (or the variables themselves if not of record type) are equal to paths over the universally quantified variables. Fullness can be checked in PTIME [21]. The EPCDs and UWDs in our examples are full, and so are many common integrity constraints.

Homomorphism. The definition of the chase step with a UWD relies on the notion of *homomorphism*: Given UWD $(d)\forall(m_1 \in M_1)\ldots(m_k \in M_k)\ B_1 \rightarrow \exists!(n_1 \in N_1)\ldots(n_l \in N_l)B_2$ and bag PC query

[9] This is not accidental: we model bags using dictionaries as well, and exploit this in implementation.

$Q =$ select $O(s_1, \ldots, s_q)$from $S_1\ s_1, \ldots, S_q\ s_q$where $C(s_1, \ldots, s_q)$, a *homomorphism* from d's universal part into Q [10] is a mapping $h : \{m_1, \ldots, m_k\} \rightarrow \{s_1, \ldots, s_q\}$ such that (i) $h(M_i)\ h(m_i)$ is among the bindings of Q for every i and (ii) $h(B_1)$ is implied by Q's conditions C $(C \Rightarrow h(B_1))$. We say that h *extends* to d's existential part, if it extends to $n_1, \ldots, n_l$ such that (i) and (ii) above hold even when substituting n_i, N_i, B_2 for m_i, M_i, B_1.

Chase step. The chase step's definition is inspired by definition 2.1. The idea is that, as we chase a bag PC query Q, we infer that some of its variables are guaranteed to range only over unique tuple values, of multiplicity 1, in any instance satisfying the UWD. We denote these variables with UW(Q). Initially, UW(Q) is the empty set, but it can be updated during the chase.

The *chase step* of Q with d using the homomorphism h from d's universal part is *applicable* if either of d's *witness* $(w(d))$, *functionality* $(f(d))$, or *multiplicity* $(m(d))$ rules below apply. The result of applying the chase step is denoted $chase_X(Q)$, with $X \in \{w(d), f(d), m(d)\}$.

rule	applies if	$chase_X(Q)$	UW($chase_X(Q)$)
w(d)	h has no extention to d's existential part	select O from $S_1\ s_1, \ldots, S_q\ s_q,$ $h(N_1)\ n_1, \ldots, h(N_l)\ n_l$ where C and $h(B_2)$	UW(Q)
f(d)	h has extensions h', h'' such that $C \not\Rightarrow h'(n_i) = h''(n_i)$ for some $1 \leq i \leq l$	select O from $S_1\ s_1, \ldots, S_q\ s_q$ where C and $h'(n_1) = h''(n_1)$ $\cdots$ and $h'(n_l) = h''(n_l)$	UW(Q)
m(d)	h has an extension h', but $h(n_i) \notin$ UW(Q) for some $1 \leq i \leq l$	Q	UW(Q) $\cup \{h(n_1), \ldots, h(n_l)\}$

The functionality and multiplicity rules are necessary: examples can be given in which chasing with the witness rule alone cannot decide equivalence of bag queries. Before showing how the chase with UWDs is used to decide equivalence, we need to generalize the notion of containment mapping from [21].

MP mapping. Let Q_1, Q_2 be bag PC queries. A mapping h from the variables of Q_1 to those of Q_2 is *multiplicity-preserving* (MP) if and only if (i) h is a homomorphism from Q_1 into Q_2, (ii) all variables in UW(Q_1) are mapped into UW(Q_2), (iii) h is surjective on Q_2's variables that are not in UW(Q_2), and (iv) for any variables $x, y \in$ UW(Q_1) with the same image under h, the equality $x = y$ is implied by the conditions in Q_1's where clause. A mapping h from Q_1 to Q_2 is an *MP containment mapping*, if it is an MP mapping and if the conditions in Q_2 imply the equality of its select clause with the image of Q_1's select clause.

In particular, if UW(Q_1) is empty and UW(Q_2) contains all of Q_2's variables, h degenerates to a homomorphism. If both UW(Q_1), UW(Q_2) are empty, h degenerates to a surjective homomorphism and if only UW(Q_2) is empty, there is no MP-mapping from Q_1 to Q_2.

Theorem 7.1 (Bag chase theorem) *Let Q_1, Q_2 be bag PC queries with empty UW(Q_1) and UW(Q_2). Let Σ be a set of UWDs such that the chase of Q_i terminates, yielding CQ_i. Then Q_1 and Q_2 are equivalent under Σ if and only if there exist MP containment mappings $h_{1,2} : CQ_1 \rightarrow CQ_2$ and $h_{2,1} : CQ_2 \rightarrow CQ_1$. Moreover, the existence of $h_{1,2}$ implies the containment of Q_2 in Q_1 under Σ.*

If all UWDs in Σ are full, the chase terminates in polynomial time in the size of Q_i. Still, finding the MP mappings is as hard as graph isomorphism.

Notice that in the absence of constraints we get UW(CQ_1) = UW(CQ_2) = $\emptyset$ and we recover the corresponding results of [5] extended to PC queries: Q_1, Q_2 are bag equivalent if and only if they are isomorphic, and surjective containment mappings imply containment .

Example. We remarked at the end of section 2 that in order for Q^b to allow the rewriting R'^b, we needed the "bag key constraint" (bk) to hold on the `Empl` attribute of `Payroll`: $\forall (p_1 \in$ `Payroll`$)$ $(p_2 \in$ `Payroll`$)$ p_1.`Empl` $= p_2$.`Empl` $\rightarrow p_1 = p_2$ which is trivially implied by the corresponding key constraint on the

[10]The notion can be easily adapted to that of homomorphism h from a query Q_1 into a query Q_2 by considering a dependency whose universal part is constructed from the from and where clauses of Q_1. A homomorphism h from Q_1 to Q_2 is a *containment mapping*, if the conditions in Q_2 imply the equality of its select clause with the image of Q_1's select clause under h.

underlying set dom Payroll (yet another example of a UWD that is implied from common dependencies on sets). Recall from proposition 4.1 that R'^b is equivalent to its unfolding

UFR'^b : select struct (E : p_1.Empl)
from depts d, d.DProjs pn, Payroll p_1, Teams t, Payroll p_2
where d.DName = p_1.PDept and t.TMember = p_2.Empl and d.DName = "*Security*"
and $pn = t$.TProj and p_1.Empl = t.TMember

It therefore suffices to check the equivalence of UFR'^b and Q^b. Q^b chases with $w(insider^b)$ and $m(insider^b)$ to Q^b_1 with $\mathrm{UW}(Q^b_1) = \{p\}$:

Q^b_1 : select struct (E : t.TMember)
from depts d, d.DProjs pn, Teams t, Payroll p
where $pn = t$.TProj and d.DName = "*Security*" and p.PDept = d.DName and p.Empl = t.TMember

UFR'^b chases with $w(bk)$, then twice with $m(insider^b)$ to Q^b_2 with $\mathrm{UW}(Q^b_2) = \{p_1, p_2\}$:

Q^b_2 : select struct (E : p_1.Empl)
from depts d, d.DProjs pn, Payroll p_1, Teams t, Payroll p_2
where d.DName = p_1.PDept and t.TMember = p_2.Empl and d.DName = "*Security*"
and $pn = t$.TProj and and p_1.Empl = t.TMember and p_1.PDept = p_2.PDept

We exhibit MP containment mappings $h_{2,1} : Q^b_2 \to Q^b_1$ and $h_{1,2} : Q^b_1 \to Q^b_2$ as the identity mapping on d, pn, t extended with $\{p_1 \mapsto p, p_2 \mapsto p\}$, respectively $\{p \mapsto p_1\}$. Notice that, in the absence of (bk), $\mathrm{UW}(Q^b_2)$ would contain only p_1, and $h_{1,2}$ wouldn't be an MP mapping because of violating condition (ii) in the definition: p_2 is not in $h_{1,2}$'s image. R'^b would therefore not be a rewriting of Q^b. •

By chasing with the UWDs in Σ and the "no-duplicates" UWDs (full as well!) for every relation in the schema, we decide bag-set equivalence under Σ, thus extending the corresponding result of [5].

Chasing with a bag view. We define here the concept of chasing with a view, introduced in section 4. A **view mapping** from a bag PC view V, into a bag PC query Q, is a homomorphism h from the universal part of V's chase-in dependency (c^b_V), which is injective on the variables not mapped into $\mathrm{UW}(Q)$. The *result* of the **view chase step** of query Q with V using h is $chase_{w(c^b_V)}(Q)$, the result of chasing Q using h with the witness rule for (c^b_V) (not surprisingly, the functionality and multiplicity rules are disabled since the corresponding statements are not true). However, as opposed to the regular witness rule, the chase step with V is *applicable* as long as it wasn't previously performed using *the same* view mapping h.

8 Combining Traditional Cost-Based Optimization with C&B.

This section addresses further details about the interaction between cost estimation of subqueries and the cost-based backchase algorithm. For cost estimation of subqueries we extended a traditional cost-based optimization method. The extensions are with respect to the path-conjunctive language that we use for queries as well as with respect to the interaction with the backchase enumeration.

Improved handling of indexes in the integrated cost-based C&B. The treatment of traditional indexes that we adopt in this approach differs from that within the "pure" chase & backchase approach as introduced in [8]. The C&B method can perform rewriting with indexes by chasing and including them in the universal plan. However, we found that it is more efficient (and still equivalent: no plans are missed) to push the selection of index-based plans for a given subquery down to the cost-based phase (as in traditional optimizers). Thus indexes do not need to appear in the universal plan, and the backchase avoids exploring many redundant combinations of relations and indexes. This cannot be done[11] for the more complex indexes such as join indexes, access support relations and in general for materialized views, which must appear in the universal plan, in order to be considered in a *complete* way.

Non-monotonic cost. The backchase as introduced in [8] is, essentially, a minimization technique: the universal plan UP is transformed into several minimal subqueries among which the best query plan can

[11] Without substantially modifying the cost-based optimizer to handle views, and our work is precisely such a modification.

be found. Minimization is applied there independent of cost, because of a *cost-monotonicity* assumption that is made: adding scans to a query increases its cost (or, in other words, minimization decreases cost). Accordingly, the pruning steps (3) and (4) in the cost-based algorithm of section 6 are based on this assumption, as well. While cost-monotonicity is basically true in most situations, there are scenarios in which minimization can actually result in a worse plan. We present first the problem and then show how we modified the pruning steps (3) and (4) to address this.

A plan that scans a relation R can be transformed into a better plan by scanning first an additional relation (or view) S of small cardinality and then accessing R via an index lookup using values given by tuples of R, provided that an index for S exists. Thus we transform a scan into an index-based join where S is the outer relation. The transformation is sound provided some semantic constraint between R and S exists or if S is a view over R.

To capture such transformations (opposite of minimization), we perform the following greedy modification of the backchase algorithm of section 6: after computing the best plan $\mathcal{P}$ for the currently explored subquery, do the following: 1) find any index I for a relation R mentioned in the subquery such that I is not used for lookup in $\mathcal{P}$; 2) find any additional relation S in the UP for which there is some join condition with R, such that 3) the join is over the attribute(s) on which I is defined. Then, for any such I and S, pruning (according to steps (3) and (4) of the backchase algorithm) of any superquery that contains S is postponed.

During our experiments we found that the additional overhead is not too high because only few relations S usually qualify for the above conditions. The result is that we perform minimization in a controlled way, based on physical level information. It is an example of a tight interaction between the C&B and cost-based optimization: the cost-based component provides some information (the access plan and the relevant indexes) based on which the C&B algorithm decides whether minimization is a good transformation (in which case a superquery is pruned) or maybe not (in which case it postpones the pruning step). This idea, although not implemented there, was advocated by [6] in the context of semantic optimization.

Other extensions to a traditional cost-based optimization method: Because of space limitations, we must omit details about the following: 1) dynamic programming enumeration of join orders and methods for path-conjunctive queries (thus including relational and OO conjunctive queries, with arbitrary levels of nesting in the input data); 2) reuse of computation of shared subplans not only within the current subquery explored by the backchase, but also within other subqueries of the universal plan, later explored; 3) a cost model that extends a standard object-relational one to include the presence of arbitrary nested sets and bags in the input database. Further details can be found in [19]. We will call the extended cost-based optimizer the **DP** optimizer (from dynamic programming).

9 Experiments

In this section we denote backchase with cost-based pruning as BottomUp+Prune, in order to distinguish it from an alternative, top-down backchase implementation which we also evaluated.

We compare **BottomUp+Prune**, with the (previous) implementation of C&B in which the cost-based optimizer is separated. In the latter case, C&B is used for *enumeration* of all scan-minimal rewritings, *after* which each of these is costed in order to select the best plan. The enumeration is either top-down (the resulting optimizer is called **TopDown**) or bottom-up (**BottomUp**). We also compare **BottomUp+Prune** with the cost-based optimizer alone, **DP**, which doesn't consider views or constraints[12].

Independent of cost-based pruning, stratification techniques, introduced for the C&B enumeration in [20], offer a significant improvement in optimization time, as well. Combining stratification (when applicable) with cost-based pruning can further decrease optimization time. However the resulting method may miss the optimal plan at times. Even so, we believe it to be a reasonably good heuristic that has the

[12]There are other experiments not shown here [19].

advantage of excellent scalability. Due to space limit, we chose not to show any results along this line and focus instead on the method that is always applicable (including worst-case scenarios such as star queries), and *complete*: **BottomUp+Prune**. Details and experiments regarding stratification[13] can be found in [19].

We focus only on experiments for the bag-set case: input relations and classes are sets, queries and views are bags. In the case when some (or all) of the input are not known to be sets, there are less subqueries of UP to consider and **BottomUp+Prune** performs better. Thus, the bag-set case is a worst case, as well.

The experimental setups that we used are parameterized by the size of the queries, number of views and constraints, in order to study how well the method scales with increasing schema complexity. All the optimizers considered were implemented in Java and run using IBM Java runtime environment for Linux (alpha version 1.1.8). The experiments have been realized on a commodity workstation (Pentium III, 500 MHz, Linux Red Hat 6.0, 256 MB of RAM).

Experiment 1. The relational case: star query, materialized views and indexes. The input query is a star query (of size n) joining one hub relations with $n-1$ corner relations. There are $n-1$ indexes, one for each corner relation, on the attribute that is used to join with the hub. In addition, there are several materialized views, each storing a three-way join between the hub and two of the corners. The hub satisfies a key constraint that is *necessary* to ensure that rewritings that use views exist. Figure 1 shows the the optimization time as a function of the number of views, when the input query is of size 6 and, respectively, 7. (The size of a query is the number of scans in its from clause.)

Comments. BottomUp+Prune outperforms **TopDown** and **BottomUp**. When the input query and the universal plan increase, the performance gain due to cost-based pruning can be of an order of magnitude! We measured, for each C&B optimizer, a *pruning ratio*: the number of subqueries that are *not* processed (due to pruning[14]) divided by the total number of subqueries of UP (2^u for UP of size u). As an example, for query size 7, with 4 views, **BottomUp+Prune** has a pruning ratio of 19.4% while **BottomUp** has a pruning ratio of only 9.9%. Thus, 9.5% of the search space is pruned based on cost alone. The difference in optimization time is then much higher (12.3s vs 94.6s) because the pruned subqueries are the large ones (supersets), and the cost of processing a subquery is more than linear in the size of the subquery[15].

Even when the C&B optimizer is not integrated with the cost-based one, it is better to use a bottom-up strategy for search. While both **TopDown** and **BottomUp** stop when they reach subqueries that are equivalent and scan-minimal, **TopDown** explores the subqueries of larger size while **BottomUp** explores the subqueries of smaller size. Thus the cost of processing a subquery (equivalence check by chase, etc.) in the bottom-up case is much smaller than in the top-down case.

To give an idea of how many rewritings are in the search space considered by the C&B optimizers, for the case of query size is 6 and 4 views, we have 13 scan-minimal rewritings,one of which is the original query, while the rest use views. For the case of query size 7 and 5 views, this number becomes 49, and 48 of them use views (and therefore are not considered by **DP**).

Experiment 2. The object-oriented case: navigation queries, access support relations and inverse relationship constraints. The input query is a chain query traversing n classes, by following references from on class to the next. The query is optimized in the presence of $2(n-1)$ constraints describing inverse relationships (INVs) and in the presence of access support relations (ASRs). By considering the INVs the C&B optimizer (any of them) can find rewritings that use ASRs. (If the INVs are not known to be satisfied then there is no rewriting using ASRs.) The number of ASRs that are usable is 0, 1, 1, and 2, respectively, when $n = 2, 3, 4, 5$, while the total number of INV constraints is 2, 4, 6, and 8, respectively.

As shown in figure 2, **TopDown** and **BottomUp** are outperformed by **BottomUp+Prune**. While the former two cannot be used when n is larger than 4, **BottomUp+Prune** works for $n = 5$ as well. 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, due to increase in query size, number of INVs, and number of ASRs. This results in an

[13] For the set semantics case only. But the same techniques are applicable for bag and bag-set semantics.

[14] Based on cost or not. Recall the algorithm of Section 6.

[15] Here, the cost of chase is a polynomial of degree equal to the maximum size of a view: 3. The cost of **DP** is exponential.

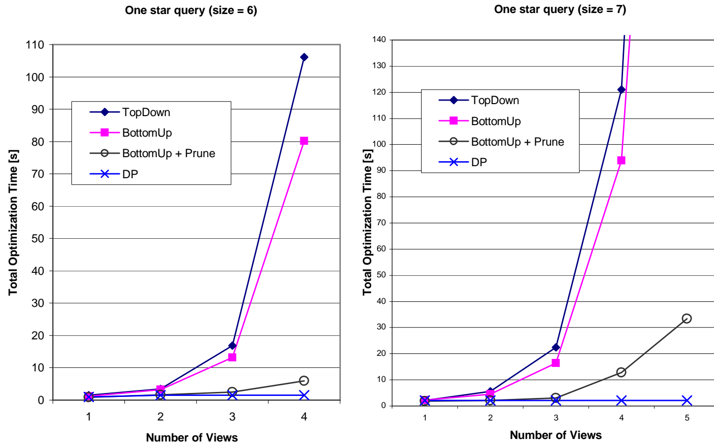

Figure 1: Optimization times: the relational case.

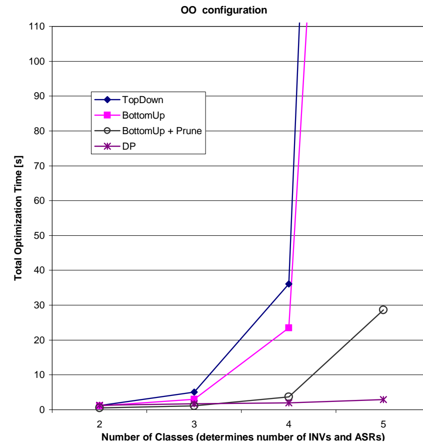

Figure 2: Optimization times: the OO case.

increase of the search space by a factor of 2^4!

Conclusion of our experiments. We summarize the main ideas that we learned from the experiments.

1. When the size of the universal plan is no more than 10 then the difference between **DP** and **BottomUp+Prune** is insignificant (see, for example, query of size 7, and 3 views, in figure 1). Then it is better to use **BottomUp+Prune**, because it may find better plans.

2. When the universal plan size is in the range 10–15, **DP** performs better. This is expected, since **DP** doesn't consider any views or constraints. Still, **BottomUp+Prune** doesn't take longer than few tens of seconds[16]. A universal plan in this range can be large enough to cover, say, 6-way joins, quite a few relevant views and constraints, and as many indexes as **DP** can handle.

Moreover, we argue that in many applications the resulting execution plans can be substantially better and the total processing (optimization + execution) time can be much smaller when **BottomUp+Prune** is used. This is especially true when queries have increased number of joins (e.g. 5, 6 or 7) and the execution is expected to take minutes. For example, queries $\mathcal{J}_2, \mathcal{J}_5, \mathcal{J}_6, \mathcal{J}_7$ reported in [6], for an OLAP setting, all have execution times between 8 and 10 minutes with DB2 . After applying semantic optimization (join elimination in that case) the execution times greatly reduce ($\mathcal{J}_2$ takes under 1 minute). **BottomUp+Prune** can perform such optimization and more, in a comprehensive way, based on cost, and the benefit of using it can then easily outweigh the cost of optimization.

3. If the universal plan is larger than 15, then **BottomUp+Prune** as it is becomes unacceptably slow. Heuristics based on stratification must be then used, as much as iterative or greedy techniques must be used for improving the scalability of **DP** itself [16]. Although not shown here, **BottomUp+Prune** has the feature that a first plan is obtained fast, and the longer the algorithm runs, better plans are produced. Thus the algorithm can be stopped early if a good enough plan is found.

4. In the case of bag semantics (no known sets in the database), the bag pruning step (step (2) in the algorithm of section 6) starts applying and the performance of **BottomUp+Prune** increases dramatically. Thus, its range of applicability increases as well (more than 15 scans in `UP`). In fact, in the absence of constraints, **BottomUp+Prune** degenerates into an algorithm with similar features as that of [4] and good experimental results were already shown there.

[16]With a Java implementation that is not tuned for performance.

10 Extension to Grouping Views

Assume that instead of V_1 from our motivating example, a more storage-efficient nested view N_1 is materialized. Then instead of R, we obtain the equivalent rewriting NR.

N_1 : select distinct struct (D : d.DName,
G : select distinct struct (P : pn, E : p.Empl)
from d.DProjs pn, Payroll p
where d.DName = p.PDept)
from depts d

NR : select distinct struct (E : m.E)
from N_1 n, n.G m, V_2 v_2
where n.D = "*Security*" and m.P = v_2.P
and m.E = v_2.E and n.D = v_2.D

When trying to capture N_1 by constraints, here is the one corresponding to the chase-in dependency:

$$(c_{N_1})\ \forall(d \in \texttt{depts})\ \exists(n \in N_1)\ d.\text{DName} = n.\text{D} \wedge$$
$$\forall(pn \in d.\texttt{DProjs})(p \in \texttt{Payroll})d.\text{DName} = p.\text{PDept} \rightarrow \exists(m \in n.\text{G})m.\text{P} = pn \wedge m.\text{E} = p.\texttt{Empl}$$

This is not an EPCD (it has more than one alternation of quantifiers) and we don't know how to chase with it. In general, nested views cannot be captured by EPCDs. We can nevertheless prove that this capture is possible whenever the flat typed attributes of the output tuples (d.DName for N_1) functionally determine all of the inner query's path expressions (d.DProjs and d.DName) involving its free variables (d). We omit a formal definition, but we exemplify on N_1, for which we ask that the functional dependency below be implied by the integrity constraints (the key on DName for class Dept in this case):

$$(FD_{N_1})\ \forall(d_1 \in \texttt{depts})(d_2 \in \texttt{depts})d_1.\text{DName} = d_2.\text{DName} \rightarrow d_1.\texttt{DProjs} = d_2.\texttt{DProjs} \wedge d_1.\text{DName} = d_2.\text{DName}$$

We call queries whose associated functional dependency FD is satisfied **grouping queries**, because they contain as particular case all SQL-like queries with a group-by clause (but which, against SQL syntax, do not aggregate the groups), for whom FD is trivially satisfied. In general, we decide whether a query is grouping by using the chase to decide FD's implication [21].

The reader can check that c_{N_1} is equivalent to $\{c^o_{N_1}, c^i_{N_1}\}$ below, but *only* in the presence of FD_{N_1}:

$$(c^o_{N_1})\ \forall(d \in \texttt{depts})\ \exists(n \in N_1)\ d.\text{DName} = n.\text{D}$$
$$(c^i_{N_1})\ \forall(d \in \texttt{depts})(n \in N_1)(pn \in d.\texttt{DProjs})(p \in \texttt{Payroll})$$
$$d.\text{DName} = n.\text{D} \wedge d.\text{DName} = p.\text{PDept} \rightarrow \exists(m \in n.\text{G})m.\text{P} = pn \wedge m.\text{E} = p.\texttt{Empl}$$

Now $\{c^o_{N_1}, c^i_{N_1}\}$ chases $(n \in N_1)$ and $(m \in n.\text{G})$ into the universal plan, which contains NR as a subquery.

Theorem 10.1 *The* C&B *algorithm remains complete even when we add grouping views with bag and set semantics, generalized to arbitrarily many nesting levels. Moreover, we can decide equivalence of set (bag) grouping queries in the presence of full EPCDs (UWDs).*

11 Conclusions and Future Work

This work reports on several extensions of the chase & backchase (C&B) method for optimizing queries with materialized views and integrity constraints. The theoretical contributions include the extension of the method to bag semantics, as well as to a class of grouping views. We also give completeness theorems for both set and bag case. On the practical side, we have integrated the C&B method with a standard cost-based optimization method. The experimental results show a great improvement in performance over the case when the two methods are implemented as separate phases. The resulting optimization algorithm is the only systematic cost-based optimizer, that works for both sets and bags, takes advantage of both materialized views and integrity constraints, and has precise optimality guarantees, that we are aware of.

Future Work. We would like to extend our comprehensive method to include, in a systematic way, queries and views with common aggregates and union, as well as grouping queries. On the practical side, we plan to investigate to what extent the cost-based chase & backchase algorithm can be implemented on top of an already existing commercial optimizer.

References

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

[2] 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.

[3] Randall G. Bello, Karl Dias, Alan Downing, James Feenan Jr., William D. Norcott, Harry Sun, Andrew Witkowski, and Mohamed Ziauddin. Materialized views in oracle. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 659–664. Morgan Kaufmann, 1998.

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

[5] Surajit Chaudhuri and Moshe Y. Vardi. Optimization of real conjunctive queries. In *Proceedings of 12th ACM Symposium on Principles of Database Systems*, pages 59–70, Washington, D. C., May 1993.

[6] Qi Cheng, Jarek Gryz, Fred Koo, T. Y. Cliff Leung, Linqi Liu, Xiaoyan Qian, and Berni Schiefer. Implementation of two semantic query optimization techniques in db2 universal database. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 687–698. Morgan Kaufmann, 1999.

[7] Sara Cohen, Werner Nutt, and A. Serebrenik. Rewriting aggregate queries using views. In *Proceedings of PODS*, pages 155–166, 1999.

[8] Alin Deutsch, Lucian Popa, and Val Tannen. Physical Data Independence, Constraints and Optimization with Universal Plans. In *International Conference on Very Large Databases (VLDB)*, September 1999.

[9] Oliver M. Duschka and Michael R. Genesereth. Answering recursive queries using views. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS-97*, May 1997.

[10] 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.

[11] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In Stanley Y. W. Su, editor, *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, pages 152–159. IEEE Computer Society, 1996.

[12] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-query processing in data warehousing environments. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 358–369. Morgan Kaufmann, 1995.

[13] Alon Halevy. *Answering Queries Using Views: a survey.* Available from `http://www.cs.washington.edu/homes/alon/site/files/view-survey.ps`, 2000.

[14] 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.

[15] 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.

[16] Donlad Kossman and Konrad Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *TODS*, 25(1), 2000.

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

[18] Alon Levy and Dan Suciu. Deciding containment for queries with complex objects. In *Proc. of the 16th ACM SIGMOD Symposium on Principles of Database Systems*, Tucson, Arizona, May 1997.

[19] Lucian Popa. *Object/Relational Query Optimization with Chase and Backchase.* PhD thesis, University of Pennsylvania, CIS Department, 2000.

[20] Lucian Popa, Alin Deutsch, Arnaud Sahuguet, and Val Tannen. A Chase Too Far? In *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 2000.

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

[22] Rachel Pottinger and Alon Y. Levy. A scalable algorithm for answering queries using views. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 484–495. Morgan Kaufmann.

[23] 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.

[24] Divesh Srivastava, Shaul Dar, H. V. Jagadish, and Alon Y. Levy. Answering queries with aggregation using views. In *Proceedings of VLDB*, pages 318–329, 1996.

[25] 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.

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

[27] Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, and Monica Urata. Answering complex sql queries using automatic summary tables. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, volume 29, pages 105–116. ACM, 2000.