A Chase Too Far?

Lucian Popa* **Alin Deutsch** **Arnaud Sahuguet** **Val Tannen**
University of Pennsylvania

Abstract

In a previous paper we proposed a novel method for generating alternative query plans that uses chasing (and back-chasing) with logical constraints. The method brings together use of indexes, use of materialized views, semantic optimization and join elimination (minimization). Each of these techniques is known separately to be beneficial to query optimization. The novelty of our approach is in allowing these techniques to interact systematically, eg. non-trivial use of indexes and materialized views may be enabled only by semantic constraints.

We have implemented our method for a variety of schemas and queries. We examine how far we can push the method in term of complexity of both schemas and queries. We propose a technique for reducing the size of the search space by "stratifying" the sets of constraints used in the (back)chase. The experimental results demonstrate that our method is practical (i.e., feasible *and* worthwhile).

1 Introduction

In [9] we proposed a new optimization technique aimed at several heretofore (apparently) disparate targets. The technique captures and extends many aspects of semantic optimizations, physical data independence (use of primary and secondary indexes, join indexes, access support relations and gmaps), use of materialized views and cached queries, as well as generalized tableau-like minimization. Moreover, and most importantly, using a uniform representation with *constraints* the technique makes these disparate optimization principles *cooperate* easily. This presents a new class of optimization opportunities, such as the non-trivial use of indexes and materialized views enabled only by the presence of certain integrity constraints. In section 2 we motivate the technique and some of the experimental configurations we use with two such examples.

We will call this technique the C&B **technique** from *chase* and *backchase*, the two principal phases of the optimization algorithm. The optimization is completely specified by a set of constraints, namely schema integrity constraints together with constraints that capture physical access structures and materialized views. In the first phase, the original query is chased using applicable constraints into a *universal* plan that gathers all the pathways and structures that are relevant for the original query and the constraints used in the chase. The search space for optimal plans consists of subqueries of this universal plan. In the second phase, navigating through these subqueries is done by chasing *backwards* trying to eliminate joins and scans. Each backchase step needs a constraint to hold and the algorithm checks if it follows from the existing ones. Thus, everything we do is captured by constraints, and only two (one, really!) generic rules.

The chase transformation was originally defined for conjunctive (tableau) queries and embedded implicational dependencies. We are using a significant extension of the chase to *path-conjunctive* queries and dependencies [19] that allows us to capture object-oriented queries, as well as queries against Web-like interfaces described by dictionary (finite function) operations. Dictionaries also describe many physical access structures giving us succinct declarative descriptions of query plans, in the same language as queries.

While sound and *complete* for the important case of path-conjunctive materialized views [9, 16], the C&B technique is sound for a larger class of queries, physical structures and constraints. We describe here the performance of a first prototype that uses path-conjunctive query graphs internally. The optimizations on which we concentrate here are increasingly relevant as more queries are generated automatically by mediator tools in heterogenous applications, while materialized views are increasingly used in dealing with source capabilities, security, encapsulation and multiple layers of logical/physical separation.

*Contact author. Email: lpopa@gradient.cis.upenn.edu

Contributions Our previous paper was promising on the potential of the C&B technique but raised the natural question: is this technique *practical*? This means two sets of issues:

1. Are there *feasible* implementations of the technique? In particular:
   (a) Is the chase phase feasible, given that even determining if a constraint is applicable requires searching among exponentially many variable mappings?
   (b) Is the backchase feasible, given that even if each chase or backchase step is feasible, the backchase phase may visit exponentially many subqueries?
2. Is the technique *worthwhile*? That is, when you add the significant cost of C&B optimization, is the cost of an alternative plan that only the C&B technique would find still better than the cost of the plan you had without C&B?

In this paper we show the following:

1. The technique is definitely feasible, for practical schemas and queries, as follows:
   (a) By using congruence closure and a homomorphism pruning technique, we can implement the chase very efficiently in practice.
   (b) The backchase quickly becomes impractical if we increase both query complexity and the size of the constraint set. But we have designed several *stratification* strategies that reduce the size of either the query or the constraint set by partitioning them into subparts that can be dealt with independently, in a dynamic programming style. Both strategies work well in common situations and one of them is *complete* for the case of path-conjunctive materialized views [9, 16].
2. We find the technique very valuable when only the presence of semantic integrity constraints enables the use of physical access structures or materialized views. This situation clearly justifies the original intuition for this research direction [9, 19].

Experiments We have built a prototype implementation of the C&B technique for path-conjunctive queries and constraints. With this implementation, we have used three experimental configurations to answer the above questions, repeating the experiments on families of queries and schemas of similar structure but of increasing complexity. This allows us to find out *how far* (as the title of the paper asks) the technique can take us and to show that the applicability range of the implementation likely includes many practical queries. For one of the configurations where we can use a conventional execution engine, we have also measured the global benefit of the C&B technique by measuring the reduction in total processing (optimization + execution) time, as a function of the complexity of the queries and the schema.

Overview of the paper Section 2 presents two motivating examples that support the goals of the C&B technique. Section 3 describes the implementation techniques we have designed to make C&B feasible and worthwhile. The architecture of our prototype is shown in section 4. Section 5 describes our experimental configurations and results. We survey related work in section 6. Section 7 discusses some possible improvements and extensions.

The rest of this paper requires familiarity with some concepts in [9], such as dictionaries, constraints, chase, universal plan, backchase, minimal plans.

2 Motivating Examples

In this section, we illustrate with two examples certain optimizations that one would like to see performed automatically in a database system.

Example 2.1 This is a very simple and common relational scenario adapted from [1], showing the benefits of exploiting referential integrity constraints. Consider a relation $\mathtt{R}(\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{E})$ and a query that selects all tuples in R with given values for attributes B and C:

(Q) select struct $(\mathtt{A} = r.\mathtt{A},\ \mathtt{E} = r.\mathtt{E})$ from R r
where $r.\mathtt{B} = b$ and $r.\mathtt{C} = c$

The relation is very large, but the number of tuples that meet the where clause criteria is very small. However, the SQL engine is taking a long time in returning an answer. Why isn't the system using an index on R ? Simply because there is no index on the attributes B and C. The only index on R that includes B and C is an index on ABC. There is no index with B and/or C in the high-order position(s), and the SQL optimizer chooses to do a table scan of R. The only way of forcing the SQL optimizer to use the index on ABC is to rewrite Q into an equivalent query that does a join of R with a small table S on attribute A knowing that there is a foreign key constraint from R into S on A:

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

Although we have not selected any attributes from S, the join with S is of a great benefit. The SQL optimizer chooses (only now!) to use S as the outer table in the join and while scanning S, as each value a for A is retrieved, the index is used to lookup the tuples corresponding to a, b, c.

Example 2.2 Integrity constraints also create opportunities for rewriting queries using materialized views. Consider the query Q given below, which joins relations $\mathtt{R}_1(\mathtt{K},\mathtt{A}_1,\mathtt{A}_2,\mathtt{F},\ldots)$, $\mathtt{R}_2(\mathtt{K},\mathtt{A}_1,\mathtt{A}_2,\ldots)$ with $\mathtt{S}_{ij}(\mathtt{A}_i,\mathtt{B},\ldots)$ $(1 \le i \le 2, 1 \le j \le 2)$. Figure 1 depicts Q's join graph, in which the nodes represent the query variables and the edges represent equijoins between them.

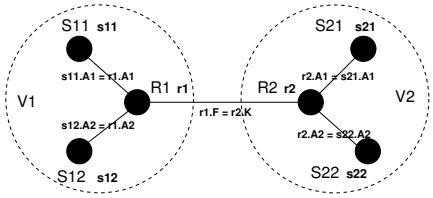

Figure 1: Query graph of Q

One can think of $\mathtt{R}_1$, $\mathtt{S}_{11}$ and $\mathtt{S}_{12}$ as storing together one large conceptual relation $\mathtt{U}_1$ that has been normalized for storage efficiency. Thus, the attributes $\mathtt{A}_1$ and $\mathtt{A}_2$ of $\mathtt{R}_1$ are foreign keys into $\mathtt{S}_{11}$ and, respectively, $\mathtt{S}_{12}$. The attribute K of $\mathtt{R}_1$ is the key of $\mathtt{U}_1$ and therefore of $\mathtt{R}_1$. Similarly, $\mathtt{R}_2$, $\mathtt{S}_{21}$ are $\mathtt{S}_{22}$ are the result of normalizing another large conceptual relation $\mathtt{U}_2$. For simplicity, we used the same name for attributes $\mathtt{A}_1$, $\mathtt{A}_2$ and K of $\mathtt{U}_1$ and $\mathtt{U}_2$ but they can store different kind of information. In addition, the conceptual relation $\mathtt{U}_1$ has a foreign key attribute F into $\mathtt{U}_2$ and this attribute is stored in $\mathtt{R}_1$. We want to perform the foreign key join of $\mathtt{U}_1$ and $\mathtt{U}_2$, which translates to a complex join across the entire database. The query returns the values of the attribute B from each of the "corner" relations $\mathtt{S}_{11}, \mathtt{S}_{12}, \mathtt{S}_{21}, \mathtt{S}_{22}$. (Again for simplicity we use the same name B here, but each relation may store different kind of information).

$$
\begin{array}{lll}
(Q) & \underline{\text{select}} & \underline{\text{struct}}(\mathtt{B}_{11} : s_{11}.\mathtt{B},\ \mathtt{B}_{12} : s_{12}.\mathtt{B},\\
 & & \quad \mathtt{B}_{21} : s_{21}.\mathtt{B},\ \mathtt{B}_{22} : s_{22}.\mathtt{B})\\
 & \underline{\text{from}} & \mathtt{R}_1\ r_1,\ \mathtt{S}_{11}\ s_{11},\ \mathtt{S}_{12}\ s_{12},\\
 & & \mathtt{R}_2\ r_2,\ \mathtt{S}_{21}\ s_{21},\ \mathtt{S}_{22}\ s_{22}\\
 & \underline{\text{where}} & r_1.\mathtt{F} = r_2.\mathtt{K}\ \underline{\text{and}}\\
 & & r_1.A_1 = s_{11}.A_1\ \underline{\text{and}}\ r_1.A_2 = s_{12}.A_2\ \underline{\text{and}}\\
 & & r_2.A_1 = s_{21}.A_1\ \underline{\text{and}}\ r_2.A_2 = s_{22}.A_2
\end{array}
$$

Suppose now that the attributes B of the "corner" relations have few distinct values, therefore the size of the result is relatively small compared to the size of the database. However, in the absence of any indexes on the attributes B of the "corner" relations, the execution time of the query is very long. Instead of indexes, we assume the existence of materialized *views* $\mathtt{V}_i(\mathtt{K}, \mathtt{B}_1, \mathtt{B}_2)$ $(1 \leq i \leq 2)$, where each $\mathtt{V}_i$ joins $\mathtt{R}_i$ with $\mathtt{S}_{i1}$ and $\mathtt{S}_{i2}$ and retrieves the B attributes from $\mathtt{S}_{i1}$ and $\mathtt{S}_{i2}$ together with the key K of $\mathtt{R}_i$:

$$
\begin{array}{lll}
(V_i) & \underline{\text{select}} & \underline{\text{struct}}(\mathtt{K} : r.\mathtt{K},\ \mathtt{B}_1 : s_1.\mathtt{B},\ \mathtt{B}_2 : s_2.\mathtt{B})\\
 & \underline{\text{from}} & \mathtt{R}_i\ r,\ \mathtt{S}_{i1}\ s_1,\ \mathtt{S}_{i2}\ s_2\\
 & \underline{\text{where}} & r.A_1 = s_l.A_1\ \underline{\text{and}}\ r.A_2 = s_2.A_2
\end{array}
$$

It is easy to see that the join of $\mathtt{R}_2$, $\mathtt{S}_{21}$, and $\mathtt{S}_{22}$ can now be replaced by a scan over $\mathtt{V}_2$:

$$
\begin{array}{lll}
(Q') & \underline{\text{select}} & \underline{\text{struct}}(\mathtt{B}_{11} : s_{11}.\mathtt{B},\ \mathtt{B}_{12} : s_{12}.\mathtt{B},\\
 & & \quad \mathtt{B}_{21} : v_2.\mathtt{B}_1,\ \mathtt{B}_{22} : v_2.\mathtt{B}_2)\\
 & \underline{\text{from}} & \mathtt{R}_1\ r_1,\ \mathtt{S}_{11}\ s_{11},\ \mathtt{S}_{12}\ s_{12},\ \mathtt{V}_2\ v_2\\
 & \underline{\text{where}} & r_1.\mathtt{F} = v_2.\mathtt{K}\ \underline{\text{and}}\\
 & & r_1.\mathtt{A}_1 = s_{11}.\mathtt{A}_1\ \underline{\text{and}}\ r_1.\mathtt{A}_2 = s_{12}.\mathtt{A}_2
\end{array}
$$

However, the join of $\mathtt{R}_1$, $\mathtt{S}_{11}$, and $\mathtt{S}_{12}$ cannot be replaced by a scan over $\mathtt{V}_1$. Q", the obvious candidate for a rewriting of Q using both $\mathtt{V}_1$ and $\mathtt{V}_2$ is *not* equivalent to Q in the absence of additional semantic information.

$$
\begin{array}{lll}
(Q'') & \underline{\text{select}} & \underline{\text{struct}}(\mathtt{B}_{11} : v_1.\mathtt{B}_1,\ \mathtt{B}_{12} : v_1.\mathtt{B}_2,\\
 & & \quad \mathtt{B}_{21} : v_2.\mathtt{B}_1,\ \mathtt{B}_{22} : v_2.\mathtt{B}_2)\\
 & \underline{\text{from}} & \mathtt{R}_1\ r_1,\ \mathtt{V}_1\ v_1,\ \mathtt{V}_2\ v_2\\
 & \underline{\text{where}} & r_1.\mathtt{K} = v_1.\mathtt{K}\ \underline{\text{and}}\ r_1.\mathtt{F} = v_2.\mathtt{K}
\end{array}
$$

The reason is that $\mathtt{V}_1$ does not contain the F attribute of $\mathtt{R}_1$, and there is no guarantee that joining the latter with $\mathtt{V}_1$ will recover the *correct* values of F. On the other hand, if we know that K is a key in $\mathtt{R}_1$ then Q" is guaranteed to be equivalent to Q, being therefore an additional (and likely better) plan.

The C&B technique covers and amply generalizes the two examples shown in this section.

3 Practical Solutions

In this section we describe the implementation techniques used to make C&B feasible and worthwhile and we point to some of the experiments that show that this goal can be achieved. In particular, we discuss:

Feasibility of the chase (section 3.1)
This is critical because the chase is heavily used: both to build the universal plan and in order to check the validity of a constraint used in a backchase step. In section 5.2 we measure for all our experimental configurations the time to obtain the universal plan as a function of the size of the query and the number of constraints. The results prove that the cost of the (efficiently implemented) chase is negligible.

Feasibility of the backchase (section 3.2)
A *full implementation of the backchase* (*FB*) consists of backchasing with all available constraints starting from the universal plan obtained by chasing also with all constraints. This implementation exposes the bottleneck of the approach: the exponential (in the size of the universal plan) number of subqueries explored in the back chase phase. A general analysis suggests using *stratification* heuristics: dividing the constraints in smaller groups and chasing/backchasing with each group successively.

We examine two approaches to this: (1) fragmenting the query and stratifying the constraints by relevance to each fragment (*On-line Query Fragmentation (OQF)*, section 3.2.1); and (2) splitting the constraints independently of the query (*Off-line Constraint Stratification (OCS)*, section 3.2.2). In the important case of materialized views [16], OQF can be used without losing any plan that might have been found by the full implementation (theorem 3.3). To evaluate and compare FB, OCS and OQF strategies, we measure in section 5.3: (1) number of plans generated, (2) the time spent per generated plan and (3) the effect of fragment granularity.

3.1 Chase Feasibility

Each chase step includes searching for homomorphisms mapping a constraint into the query. A **homomorphism** from a constraint $c = \forall(\vec{u} \in \vec{U})\ \ B_1(\vec{u}) \ \Rightarrow\ \exists(\vec{e} \in$

$\vec{E})\ B_2(\vec{u},\vec{e})$ into a query Q is a mapping from the universally quantified variables of c into the variables of Q such that, when extended in the natural way to paths, it obeys the following conditions:
1) any universal quantification $u \in U$ of c corresponds to a binding $P\ h(u)$ of Q such that either $h(U)$ and P are the same expression or $h(U) = P$ follows from the where clause of Q.
2) for every equality $P_1 = P_2$ that occurs in B_1 either $h(P_1)$ and $h(P_2)$ are the same expression or $h(P_1) = h(P_2)$ follows from the where clause of Q.

Finding a homomorphism is NP-complete, but only in the size of the constraint (always small in practice). However, the basis of the exponent is the size of the query being chased which can become large during the chase. Since our language is more complicated than a relational one because of dictionaries and set nesting, homomorphisms are more complicated than just simple mappings between goals of conjunctive queries, and checking that a mapping from a constraint into a query is indeed a homomorphism is not straightforward.

We list below some techniques that we use to avoid unnecessary checks for homomorphisms, and to speed up the chase:

- Use of congruence closure, a variation of [17], for fast checking if an equality is a consequence of the where clause of the query.
- Pruning variable mappings that cannot become homomorphisms by reasoning early about equality. Instead of building the entire mapping and checking in one big step whether it is a homomorphism, this is done incrementally. For example, if h is a mapping that is defined on x and y and $x.\mathtt{A} = y.\mathtt{A}$ occurs in the constraint then we check whether $h(x).\mathtt{A} = h(y).\mathtt{A}$ is implied by the where clause of the query. This works well in practice because the "good" homomorphisms are typically just a few among all possible mappings.
- Implementation of the chase as an inflationary procedure that evaluates the input constraints on the internal representation of the input query. The evaluation looks for homomorphisms from the universal part of constraints into the query, and "adds" to the internal query representation (if not there already[1]) the result of each homomorphism applied to the existential part of the constraint. The analogy with query evaluation on a small database is another explanation of why the chase is fast.

The experimental results about the chase shown in section 5.2 are very positive and show that even chasing queries consisting of more than 15 joins with more than 15 constraints is quite practical.

[1] This is translated as a check for trivial equivalence.

3.2 Backchase Feasibility

The following analysis of a simple but important case (just indexes) shows that a full implementation of the backchase can unnecessarily explore many subqueries.

Example 3.1 Assume a chain query that joins n relations $\mathtt{R}_1(\mathtt{A},\mathtt{B}),\ldots,\mathtt{R}_n(\mathtt{A},\mathtt{B})$:

(Q) select struct($\mathtt{A} = r_1.\mathtt{A},\ \mathtt{B} = r_n.\mathtt{B}$)
from $\mathtt{R}_1\ r_1,\ldots,\ \mathtt{R}_n\ r_n$
where $r_1.\mathtt{B} = r_2.\mathtt{A}$ and ... and $r_{n-1}.\mathtt{B} = r_n.\mathtt{A}$

and suppose that each of the relations has a primary index $\mathtt{I}_i$ on A. Let $D = \{d_1, d_1^-, \ldots, d_n, d_n^-\}$ be all the constraints defining the indexes (here d_i and d_i^- are the constraints for $\mathtt{I}_i$).

In principle, any of the 2^n plans obtained by either choosing the index $\mathtt{I}_i$ or scanning $\mathtt{R}_i$, for each i, is plausible. One direct way to obtain all of them is to chase Q with the entire set of constraints D, obtain the universal plan U (of size $2n$), and then backchase it with D. The backchase inspects top-down all subqueries of U, from size $2n-1$ to size n (any subquery with less than n loops cannot be equivalent to U), for a total of: $C_{2n}^{2n-1} + \ldots + C_{2n}^{n} = 2^{2n-1} + \frac{1}{2}C_{2n}^{n} - 1$.

The same 2^n plans can be obtained with a different strategy, much closer to the one implemented by standard optimizers. For each i, handle the ith loop of Q independently: chase then backchase the query fragment Q_i of Q that contains only $\mathtt{R}_i$ with $\{d_i, d_i^-\}$ to obtain two plans for Q_i, one using $\mathtt{R}_i$ the other using the index $\mathtt{I}_i$. At the end, assemble all plans generated for each fragment Q_i in all possible combinations to produce the 2^n plans for Q.

The number of plans inspected by this *"stratified"* approach can be computed as follows. For each stage i the universal plan for fragment Q_i has only 2 loops (over $\mathtt{R}_i$ and $\mathtt{I}_i$) and therefore the number of plans explored by the subsequent backchase is 2. Thus the work to produce all the plans for all fragments is $2n$. The total work, including assembling the plans, is then $2n + 2^n$. This analysis suggests that detecting classes of constraints that do not "interact", grouping them accordingly and then stratifying the chase/backchase algorithm, such that only one group is considered at a time, can *decrease exponentially* the size of the search space explored.

The crucial intuition that explains the difference in efficiencies of the two approaches is the following. In the first strategy, for a given i, the universal plan contains at the beginning of the backchase both $\mathtt{R}_i$ and $\mathtt{I}_i$. At some point during the backchase, since a plan containing both is not minimal, there will be a backchase step that eliminates R_i and another backchase step, at the same level, that eliminates I_i (see figure 2). The minimization work that follows is exactly the same in both cases because it operates only on the rest of the relations. This duplication of work is avoided in

the second strategy because each loop of Q is handled exactly once. A solution that naturally comes to mind to avoid such situations is to use dynamic programming. Unfortunately, there is no direct way to do this in general (we discuss this more in section 7). Instead, the next section gives a stratification algorithm that solves the problem for a restricted but common case.

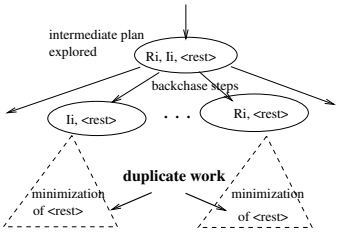

Figure 2: Duplication of work during minimization

3.2.1 On-line Query Fragmentation (OQF)

The main idea behind the OQF strategy is illustrated on the following example.

Example 3.2 Consider a slightly more complicated version of example 2.2 shown in figure 3. The query graph is shaped like a chain of 2 stars, star i having $\mathtt{R}_i$ for its hub and $\mathtt{S}_{ij}$ for its corners ($1 \leq i \leq 2$, $1 \leq j \leq 3$). The attributes selected in the output are the $\mathtt{B}$ attributes of all corners $\mathtt{S}_{ij}$. Assume the existence of materialized views $\mathtt{V}_{il}(\mathtt{K}, \mathtt{B}_1, \mathtt{B}_2)$ ($1 \leq i \leq 2, 1 \leq l \leq 2$), where each $\mathtt{V}_{il}$ joins the hub of star i ($\mathtt{R}_i$) with two of its corners ($\mathtt{S}_{il}$ and $\mathtt{S}_{i(l+1)}$). Each V_{il} selects the $\mathtt{B}$ attributes of the corner relations it joins, as well as the $\mathtt{K}$ attribute of $\mathtt{R}_i$.

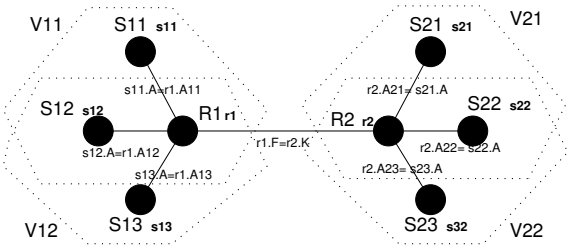

Figure 3: Chain-of-stars query Q with views

If we apply the FB algorithm with all the constraints describing the views we obtain all possible plans in which views replace some parts of the original query. However V_{11} or V_{12} can only replace relations from the first star, thus not affecting any of the relations in the second star. If a plan P using V_{11} and/or V_{12} is obtained for the first star, such that it "recovers" the $\mathtt{B}$ attributes needed in the result of Q, as well as the $\mathtt{F}$ attribute of $\mathtt{R}_1$ needed in the join with $\mathtt{R}_2$, then P can be joined back with the rest of the query to obtain a query equivalent to Q. We say that V_{11} overlaps with neither V_{21} nor V_{22}. On the other hand this does not apply to V_{11} and V_{12}, because the parts of the query that they cover overlap (and any further decomposition will lose the plan that uses both V_{11} and V_{12}). Q can thus be decomposed into precisely two query fragments, one for each star, that can be optimized independently.

Before we give the full details of the OQF algorithm, we need to formalize the ideas introduced in the previous example.

Query Fragments. We define the *closure* Q^* of query Q as a query with the same select and from clauses as Q while the where clause consists of all the equalities occuring in or implied by Q's where clause. Q^* is computable from Q in PTIME and is equivalent to Q ([18] shows a congruence closure algorithm for this construction).

Given a query Q and a subset S of its from clause bindings we define a *query fragment Q' of Q induced by S* as follows: 1) The from clause consists of exactly the bindings in S; 2) The where clause consists of all the conditions in the where clause of Q^* which mention only variables bound in S; 3) The select clause consists of all the paths P over S that occur in the select clause of Q or in an equality $P = P'$ of Q^*'s where clause where P' depends on at least one binding that is not in S. In the latter case, we call such P a *link path* of the fragment.

Skeletons. While in general the chase/backchase algorithm can mix semantic with physical cosntraints, in the remainder of this section we describe a stratification algorithm that can be applied to a particular class of constraints which we call *skeletons*. This class is sufficiently general to cover the usual physical access structures: indexes, materialized views, ASRs, GMAPs. Each of these can be described by a pair of complementary inclusion constraints. We define a *skeleton* as a pair of complementary constraints:

$d = \forall(\vec{x} \in \vec{\mathtt{R}})\ [\ B_1(\vec{x})\ \Rightarrow\ \exists(\vec{v} \in \vec{\mathtt{V}})\ B_2(\vec{x}, \vec{v})\ ]$

$d^- = \forall(\vec{v} \in \vec{\mathtt{V}})\ \exists(\vec{x} \in \vec{\mathtt{R}})\ B_1(\vec{x})$ and $B_2(\vec{x}, \vec{v})$

such that all schema names occuring among $\vec{\mathtt{V}}$ belong to the physical schema, while all schema names occuring among $\vec{\mathtt{R}}$ belong to the logical schema.

Algorithm 3.1 (Decomposition into Fragments.) Given a query Q and a set of skeletons $\mathcal{V}$:

1. Construct an *interaction graph* G as follows: 1) there is a node labeled (V, h) for every skeleton $V = (d, d^-)$ in $\mathcal{V}$ and homomorphism h from d to Q; 2) there is an edge between (V_1, h_1) and (V_2, h_2) iff the intersection between the bindings of $h(d_1)$ and $h(d_2)$ is nonempty.

2. Compute the connected components $\{C_1, \ldots, C_k\}$ of G.

3. For each $C_m = \{(V_1, h_1), \ldots, (V_n, h_n)\}$ $(1 \leq m \leq k)$ let S be the union of the sets of bindings in $h_i(d_i)$ for all $1 \leq i \leq n$ and compute F_m as the fragment of Q induced by S.

4. The decomposition of Q into fragments consists of $F_1, \ldots, F_k$ together with the fragment F_{k+1} induced by the set of bindings that are not covered by $F_1, \ldots, F_k$.

The resulting fragments are disjoint, and Q can be reconstructed by joining them on the link paths.

Now we are ready to define the on-line query fragmentation strategy:

Algorithm 3.2 (OQF) Given a query Q and a set $\mathcal{V}$ of skeletons:

1. Decompose Q into query fragments $\{F_1, \ldots, F_n\}$ based on $\mathcal{V}$ using Algorithm 3.1.

2. For each fragment F_i find the set of all minimal plans by using the chase/backchase algorithm

3. A plan for Q is the "cartesian product" of sets of plans for fragments (cost-based refinement: the best plan for Q is the join of the best plans for each individual fragment)

Theorem 3.3 *For a skeleton schema, OQF produces the same plans as the full backchase (FB) algorithm.*

In the limit case when the physical schema contains skeletons involving only one logical schema name (such as primary/secondary indexes), OQF degenerates smoothly into a backchase algorithm that operates individually on each loop of the query to find the access method for that loop. One of the purposes of the experimental configuration **EC1** is to demonstrate that OQF performs well in a typical relational setting. However, OQF can be used in more complex situations, such as rewriting queries with materialized views. While in the worst case when the views are strongly overlapping, the fragmentation algorithm may result in one fragment (the query itself), in practice we expect to achieve reasonably good decompositions in fragments. Scalability of OQF in a setting that exhibits a reasonable amount of non-interaction between views is demonstrated by using the experimental configuration **EC2**.

3.2.2 Off-line Constraint Stratification

One disadvantage of OQF is that it needs to find the fragments of a query Q. While this has about the same complexity as chasing Q [2] (and we have argued that chase itself is not a problem) in practice there may be situations in which interaction between constraints can be estimated in a pre-processing phase that examines only the constraints in the schema. The result of this phase is a partitioning of constraints into disjoint sets (*strata*) such that only the constraints in one set are used at one time by the algorithm.

As opposed to OQF this method tries to isolate the independent optimizations that may affect a query by stratifying the constraints without fragmenting the query. During the optimization the entire query is pipelined through stages in which the chase/backchase algorithm uses only the constraints in one set. At each stage different parts of the query are affected.

We first give the algorithm that computes the stratification of the constraints.

Algorithm 3.4 (Stratification of Constraints.) Given a schema with constraints, do:

1. Construct an *interaction graph* G as follows:
a) there is a node labeled c for every constraint c.
b) there is an edge between nodes c_1 and c_2 if there is a homomorphism[3] from the tableau of c_1 into that of c_2, or viceversa. The tableau $T(c)$ of a constraint $c = \forall(\vec{u} \in \vec{U})\ \ B_1(\vec{u}) \ \Rightarrow\ \exists(\vec{e} \in \vec{E})\ B_2(\vec{u}, \vec{e})$ is obtained by putting together both universally and existentially quantified variables and by taking the conjunction of all conditions: $T(c) = \forall(\vec{u} \in \vec{U})\ \forall(\vec{e} \in \vec{E})\ B_1(\vec{u}) \wedge B_2(\vec{u}, \vec{e})$.

2. Compute the connected components $\{C_1, \ldots, C_k\}$ of G. Each C_i is a stratum.

Using algorithm 3.4, we define the following refinement of the C&B strategy, the *off-line constraint stratification* (OCS) algorithm:

Algorithm 3.5 (OCS) Given a query Q and a set of constraints $\mathcal{C}$:

1. Partition $\mathcal{C}$ into disjoint sets of constraints $\{S_i\}_{1 \le i \le k}$ by using algorithm 3.4.

2. Let $P_0 = \{Q\}$. For every $1 \le i \le k$, let P_i be the union of the sets of queries obtained by chase/backchase each element of P_{i-1} with the constraints in S_i.

3. Output P_k as the set of plans.

Algorithm 3.4 makes optimistic assumptions about the non-interaction of constraints: even though there may not be any homomorphism between the constraints, depending on the query they might still interact by mapping to overlapping subqueries at run time. Therefore, the OCS strategy is subsumed by the on-line query fragmentation but it has the advantage of being done before query optimization.

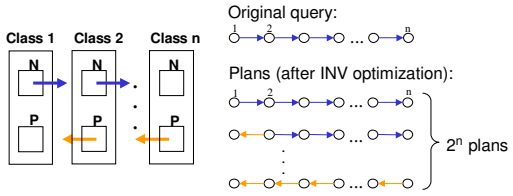

Figure 4: Inverse Relationships

Example 3.3 Consider 3 classes (see figure 4 with $n = 3$) described by dictionaries $\mathtt{M}_1, \mathtt{M}_2, \mathtt{M}_3$. Each $\mathtt{M}_i$ includes a set-valued attributed `N` ("next") and a set-valued attribute `P` ("previous"). For each $i = 1, 2$, there

[2] The chase also needs to find all homomorphisms between constraints and the query.

[3] Similar to those defined in section 3.1.

exists a many-many inverse relationship between $\mathtt{M}_i$ and $\mathtt{M}_{i+1}$ that goes from $\mathtt{M}_i$ into $\mathtt{M}_{i+1}$ by following the N references and comes back from $\mathtt{M}_{i+1}$ into $\mathtt{M}_i$ by following the P references. The inverse relationship is described by two constraints, INV_{iN} and INV_{iP}, of which we show below the first:

$\forall(k \in \underline{\mathsf{dom}}\,\mathtt{M}_i)\forall(o \in \mathtt{M}_i[k].\mathtt{N})$
$\exists(k' \in \underline{\mathsf{dom}}\,\mathtt{M}_{i+1})\exists(o' \in M_{i+1}[k'].\mathtt{P})\; k' = o \;\underline{\mathsf{and}}\; o' = k$

By running algorithm 3.4 we obtain the following stratification of constraints into two strata: $\{\mathrm{INV}_{1N}, \mathrm{INV}_{1P}\}$ and $\{\mathrm{INV}_{2N}, \mathrm{INV}_{2P}\}$. Suppose now that the incoming query Q is a typical navigation, following the N references from class M_1 to class M_2 and from there to $\mathtt{M}_3$:

$\underline{\mathsf{select}}$ $\underline{\mathsf{struct}}(\mathtt{F} = k_1,\ \mathtt{L} = o_2)$
$\underline{\mathsf{from}}$ $\underline{\mathsf{dom}}\,\mathtt{M}_1\ k_1,\ \mathtt{M}_1[k_1].\mathtt{N}\ o_1,\ \underline{\mathsf{dom}}\,\mathtt{M}_2\ k_2,\ \mathtt{M}_2[k_2].\mathtt{N}\ o_2$
$\underline{\mathsf{where}}$ $o_1 = k_2$

By chase/backchasing Q with the constraints of the first stratum, $\{\mathrm{INV}_{1N}, \mathrm{INV}_{1P}\}$, we obtain, in addition to Q, query Q_1 in which the sense of navigation from $\mathtt{M}_1$ to $\mathtt{M}_2$ following the N attribute is "flipped" to a navigation in the opposite sense: from $\mathtt{M}_2$ to $\mathtt{M}_1$ along the P attribute.

(Q_1) $\underline{\mathsf{select}}$ $\underline{\mathsf{struct}}(\mathtt{F} = o_1,\ \mathtt{L} = o_2)$
$\underline{\mathsf{from}}$ $\underline{\mathsf{dom}}\,\mathtt{M}_2\ k_2,\ \mathtt{M}_2[k_2].\mathtt{P}\ o_1,\ \mathtt{M}_2[k_2].\mathtt{N}\ o_2$

In the stage corresponding to stratum 2, we chase and backchase $\{Q,\ Q_1\}$ with $\{\mathrm{INV}_{2N}, \mathrm{INV}_{2P}\}$, this time flipping in each query the sense of navigation from $\mathtt{M}_2$ to $\mathtt{M}_3$ via N to a navigation from $\mathtt{M}_3$ to $\mathtt{M}_2$ via P. The result of this stage consists of four queries: the original Q and Q_1 (obtained by chasing and then backchasing with the same constraint), plus two additional queries. One of them, obtained from Q_1, is shown below:

$\underline{\mathsf{select}}$ $\underline{\mathsf{struct}}(\mathtt{F} = o_1,\ \mathtt{L} = k_3)$
$\underline{\mathsf{from}}$ $\underline{\mathsf{dom}}\,\mathtt{M}_3\ k_3,\ \mathtt{M}_3[k_3].\mathtt{P}\ o_3,\ \underline{\mathsf{dom}}\,\mathtt{M}_2\ k_2,\ \mathtt{M}_2[k_2].\mathtt{P}\ o_1$
$\underline{\mathsf{where}}$ $o_3 = k_2$

The OCS strategy does not miss any plans for this example (see also the experimental results for OCS with **EC2**), but in general it is just a heuristic. Our algorithm 3.4 makes optimistic assumptions about the non-interaction of constraints, which depending on the input query, may turn out to be false, therefore it is not complete. **EC2** is an example of such a case and we leave open the problem of finding a more general algorithm for stratification of constraints.

4 The Architecture of the Prototype

The architecture of the system that implements the C&B technique (about 25,000 lines of Java code), is shown in figure 5. The arrowed lines show the main flow of a query being optimized, constraints from the schema, and resulting plans. The thick lines show the interaction between modules. The main module is the *plan generator* which performs the two basic phases of the C&B : chase and backchase. The backchase is implemented top-down by removing one binding at a time and minimizing recursively the subqueries obtained (if they are equivalent). Checking for equivalence is performed by verifying that the dependency equivalent to one of the containments is implied by the input constraints[4]. The module that does the check, *dependency implication* shown in the figure as $D \Rightarrow d$, uses the chase. The most salient features of the implementation are summarized below:

- queries and constraints are compiled into a (same!) internal congruence closure based canonical database representation (shown in the figure as $DB(Q)$ for a query Q, respectively $DB(d)$ for a constraint D) that allows for fast reasoning about equality.
- compiling a query Q into the canonical database is implemented itself as a chase step on an empty canonical database with one constraint having no universal but one existential part isomorphic to Q's from and where clauses put together. Hence, the query compiler, constraint compiler and the chase modules are basically one module.
- a language for queries and constraints that is in the spirit of OQL.
- a script language that can control the constraints that are fed into the chase/backchase modules. This is how we implemented the off-line stratification strategy and various other heuristics.

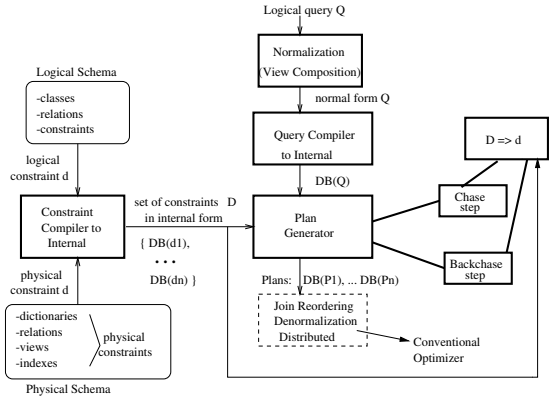

Figure 5: C&B Optimizer Architecture

5 Experiments

In this section we present our experimental configuration and report the results for the chase and the backchase. Finallly, we address in section 5.4 the question whether the time spent in optimization is gained back at execution time.

5.1 Experimental configurations

We consider for our experiments three different settings that exhibit the mix of physical structures and semantic

[4]The other containment is always true.

constraints that we want to take advantage of in our optimization approach. We believe that the scenarios that we consider are relevant for many practical situations.

Experimental Configuration EC1: The first setting is used to demonstrate the use of our optimizer in a relational setting with indexes. This is a simple but frequent practical case and therefore we consider it as a baseline.

The schema includes n relations, each relation $\mathtt{R}_i$ with a key attribute K on which there is a primary index $\mathtt{PI}_i$, a foreign key attribute N, and additional attributes. The first j of the relations have secondary indexes $\mathtt{SI}_i$ on N, thus the total number of indexes in the physical schema is $m = n + j$. As in Example 3.1 we consider chain queries, of size n, in which there is a foreign key join (equating attributes N and K) between each $\mathtt{R}_i$ and $\mathtt{R}_{i+1}$. The scaling parameters for **EC1** are n and m.

Experimental Configuration EC2: The second setting is designed to illustrate experimental results in the presence of materialized views and key constraints. We consider a generalization of the chain of stars query of examples 2.2 and 3.2 in which we have i stars with j corner relations, $\mathtt{S}_{i1}, \ldots, \mathtt{S}_{ij}$, that are joined with the hub of the star $\mathtt{R}_i$. The query returns all the B attributes of the corner relations. For each we assume $v \leq j - 1$ materialized views $\mathtt{V}_{i1}, \ldots, \mathtt{V}_{iv}$ each covering, as in the previous examples, three relations. We assume that the attribute K of each $\mathtt{R}_i$ is a primary key. The scaling parameters are i, j and v.

Experimental Configuration EC3: This is an object-oriented configuration with classes obeying many-to-many inverse relationship constraints. We use it to show how we can mix semantic optimization based on the inverse constraints to discover plans that use access support relations (ASRs). The query that we consider is not directly "mappable" into the existing ASRs, and the semantic optimization "component" of C&B enables rewriting the query into equivalent queries that *can* map into the ASRs.

We generalize here the scenario of example 3.3 by considering n classes with inverse relationships. The queries Q (see figure 4) that we consider are long navigation queries across the entire database following the N references from class M_1 to class M_n. In addition we have, as part of the physical schema, access support relations (ASRs) that are materialized navigation joins across three classes going in the backwards direction (i.e. following two P references). Each ASR is a binary table storing oids from the beginning and from the end of the navigation path. Plans obtained after the inverse optimization phase are rewritten in the second phase into plans that replace a navigation chain of size 2 with one navigation chain of size 1 that uses an ASR (thus being likely better plans). The parameter of the configuration is the number of classes n. There are $\frac{n-1}{2}$ non-overlapping ASRs that cover the entire navigation chain.

Experimental settings. All the experiments have been realized on a dedicated commodity workstation (Pentium III, Linux RH-6.0, 128MB of RAM). The optimization algorithm is run using IBM JRE-1.1.8. The database management system used to execute queries is IBM DB2 version 6.1.0 (out-of-the-box configuration). For **EC2**, materialized views have been produced by creating and populating tables. All times measured are *elapsed times*, obtained using the Unix shell `time` command. In all the graphs shown in this section, whenever values are missing, it means that the time to obtain them was longer than the timeout used (2 mins).

5.2 Chase Feasibility: Experiments

We measured the complexity of the chase in all our experimental configurations varying both the size of the input query and the number of constraints.

In **EC1** (figure 6, left) the constraints used in the chase are the ones describing the primary (2 constraints/index) and/or secondary (3 constraints/index) indexes. For example, chasing with 10 indexes, therefore 20+ constraints, takes under 1s. For **EC2** (figure 6, middle) the variable is the number of relations in the from clause, giving a measure of the query size. The number of constraints comes from the number of views (2 constraints/view) and the number of key constraints (1 constraint/star hub). For **EC3** (figure 6, right) the variable is the number of classes n (measuring both the size of the schema and that of the queries we use). The chase is done with the inverse relationship constraints and with the ASR constraints. Chasing with 8 classes (20 constraints) takes 3s. Overall, we conclude that the normalized chase time grows significantly with the size of the query and the number of constraints. In comparison, numbers for the chase time are much smaller than those of the backchase.

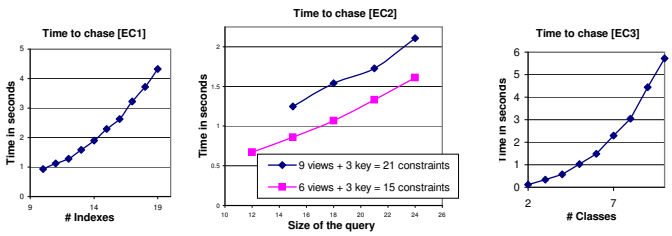

Figure 6: Chase time

5.3 Backchase Feasibility: Experiments

To evaluate and compare the two stratification strategies (OQF and OCS) and the full approach (FB) we measure the following:

- **The number of plans generated** measures the completeness with respect to FB. We found that OQF was complete for all experimental configurations considered, beyond what theorem 3.3 guarantees, while OCS is not complete for **EC2**.
- **The time spent per generated plan** allows for a fair comparison between all three strategies. We measured the time per plan as a function of the query size and number of constraints. Moreover, we studied the scale-up for each strategy by pushing the values of the parameters to the point at which the strategy became ineffective. We found that OQF performed much better than OCS which in turn outperformed FB.
- **The effect of fragment granularity on optimization time** is measured by keeping the query size constant and varying the number of strata in which the constraints are divided. This evaluates the benefits of finding a decomposition of the query into minimal fragments. The OQF strategy performs best by achieving the minimal decomposition that doesn't lose plans. The results also show that OCS is a trade-off giving up completeness for optimization time.

Number of generated plans. This experiment compares *for completeness* the full backchase algorithm with our two refinements: OQF (section 3.2.1) and OCS (section 3.2.2). We measured the number of generated plans, as a function of the size of the query and the number of constraints. The three strategies yielded the same number of generated plans in configurations **EC1** and **EC3**. The table below shows some results for configuration **EC2** in which OCS cannot produce all plans. However, the time spent for generating the plans differs spectacularly among the three techniques, as shown by the next experiment.

s	c	v	FB	OQF	OCS
1	5	1	2	2	2
1	5	2	4	4	3
1	5	3	7	7	5
1	5	4	13	13	8
2	5	1	4	4	4

Time per plan. This experiment compares the three backchase strategies by *optimization time*. Because not all strategies are complete and hence output different numbers of plans, we ensured fairness of the comparison by normalizing the optimization time which was divided by the number of generated plans. This normalized measure is called *time per plan* and was measured as a function of the size of the query and the number of constraints. The results are shown in figures 7 and 8.

By running the experiment in configuration **EC1** we showed that for the trivial yet common case of index introduction, our algorithm's performance is comparable to that of standard relational optimizers. Figure 7 shows the results obtained for three query sizes: 3, 4 and 5. By varying the number of secondary indexes for each query size, we observed an exponential behavior of the time per plan for the FB strategy, but a negligible time per plan for both OQF and OCS.

For configuration **EC3**, OQF degenerates into FB because the images of the inverse constraints overlap. We show a comparison of FB(=OQF) and OCS. OCS outperforms the other two strategies on this example because each pair of inverse constraints ends up in its own stratum. This stratification results in a *linear* time per plan (each stratum flips one join direction).

The most challenging configuration is **EC2**, dealing with large queries and numerous constraints: the point `[2,3,5]` of figure 8 corresponds to a query with 17 joins, 6 views (12 constraints), and 3 key constraints. Figure 8 divides the points into 3 groups, each corresponding to the same number of views per star. This value determines the size of the query fragments for OQF and is the most important factor influencing its time per plan[5]. While all strategies exhibit exponential time per plan, OCS is fastest, while FB cannot keep pace with the other two strategies [6].

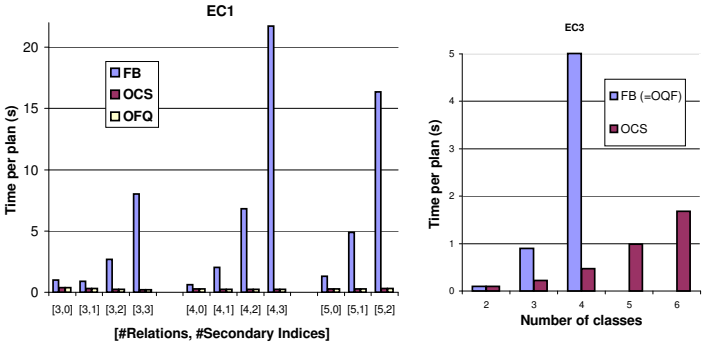

Figure 7: Comparison of FB, OQF, OCS for EC1, EC3

The effect of stratification. This experiment was run in configurations **EC2** and **EC3** by keeping the query size constant and varying the number of strata in which the constraints are divided[7]. For **EC3**, we considered two queries: one navigating over 5 classes and one over 6 classes, with 8, respectively 10 applicable constraints. The query considered in **EC2** joins three stars of 3 corners each, with one view applicable per star (for a total of 9 constraints). The results are shown in figure 9 and exhibit the exponential reduction inferred in example 3.1.

[5]OCS achieves a finer stratification than OQF, but misses the best plan, which uses all the views.

[6]We only measure time per plan here, not the quality of the plans. We compare the two in 5.4.

[7]Stratum size 1 corresponds for **EC3** to OCS.

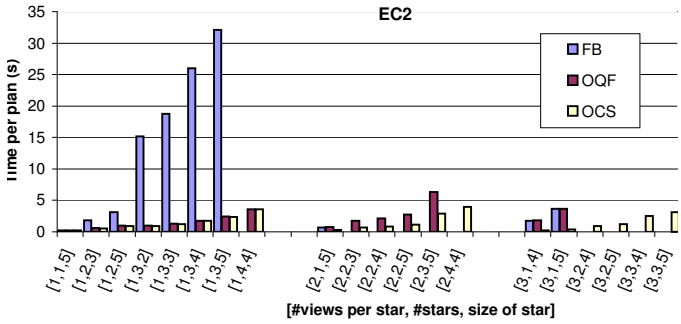

Figure 8: Comparison of FB, OQF, OCS for EC2

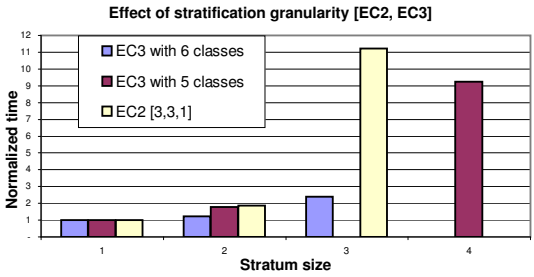

Figure 9: Stratification vs. optimization time

5.4 The Benefit of Optimization

Next we measure, in **EC2**, the real query processing time (optimization time plus execution time). Since we didn't implement our own query execution engine, we made use of DB2 as follows. Queries are optimized using the OQF strategy and resulting plans are fed into DB2 to compare their processing times.

Parameters measured We denote by OptT the time taken to generate all plans; by ExT the execution time of the query given to DB2 in its original form (no C&B optimizaton); and by ExTBest, the DB2 execution time of the best plan generated by the C&B. We assume that the cost of picking the best plan among those generated by the algorithm is negligible. Figure 10 gives the details of the plans generated and their ExT values for a setting with 3 stars, each with 2 corners and 1 view. OptT is 8s; plan 8 is the original query. For each plan, we present the views and corner relations used (in addition to the star hubs which appear in all plans).

Plan	ExT	Views	Corner relations
1	5.54s	V_{11},V_{21},V_{31}	
2	66.39s	V_{11},V_{21}	S_{31},S_{32}
3	33.13s	V_{11},V_{31}	S_{21},S_{22}
4	143.75s	V_{11}	S_{21},S_{22},S_{31},S_{32}
5	105.82s	V_{21},V_{31}	S_{11},S_{12}
6	61.45s	V_{21}	S_{11},S_{12},S_{31},S_{32}
7	43.54s	V_{31}	S_{11},S_{12},S_{31},S_{32}
8	132.90s		S_{11},S_{12},S_{21},S_{22},S_{31},S_{32}

Figure 10: Generated plans.

Performance indices We define and display in figure 11, for increasing complexity of the experimental parameters, the following performance indices:

$$\text{Redux} = \frac{\text{ExT} - (\text{ExTBest} + \text{OptT})}{\text{ExT}}$$

$$\text{ReduxFirst} = \frac{\text{ExT} - (\text{ExTBest} + (\text{OptT}/\#plans))}{\text{ExT}}$$

Redux represents the time reduction resulting from our optimization with respect to ExT assuming that no heuristic is used to stop the optimization as soon as reasonable. ReduxFirst represents the time reduction resulting from our optimization with respect to ExT assuming that a heuristic is used to return the best plan first and stop the optimization. Our current implementation of OQF (similar for OCS) is able to return the best plan first for all the experiments presented in this paper (see section 7 for a discussion).

Dataset used These performance indices correspond to experiments conducted on a small size database with the following characteristics[8]:

$\|R_i\|$	$\|S_{i,j}\|$	$\sigma(R_i \bowtie S_{i,j})$	$\sigma(R_i \bowtie R_{i+1})$
5000 tup.	5000 tup.	4%	2%

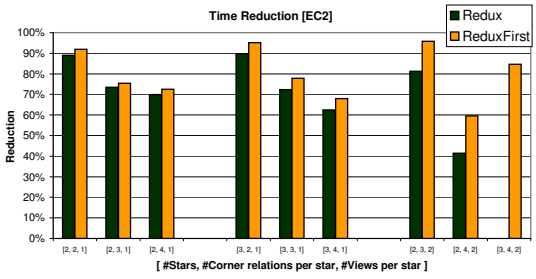

Figure 11: Time reduction (negative Redux not shown).

Our current implementation is not tuned for maximum performance, thus skewing the results against us. Using C or C++ and embedding the C&B as a built-in optimization (e.g. inside DB2) would lead to even better performance. We obtain excellent results nevertheless, proving that the time spent in optimization is well worth the gained execution time.

Even without the heuristic of stopping the optimization after the first plan, the C&B posts significant time reductions (40% to 90%), up to optimizing chain of stars queries with 9 joins, using 4 views (`[2,4,2]` in figure 11). The practicality range is extended even further when using the *"best plan first"* heuristic, with reductions of 60% to 95%, up to optimizing queries with 14 joins, using 6 views (`[3,4,2]` in figure 11).

6 Related work

There are many papers that discuss semantic query optimization for relational systems([6, 13, 4] and the

[8]On a larger database, the benefits of C&B should be even more important.

references therein). The techniques most frequently used are [6] *index introduction, join elimination, scan reduction, join introduction, predicate elimination* and detection of *empty answers*. Of these, scan reduction, predicate elimination and empty answers use boolean and numeric bounds reasoning of a kind that we have left out of our optimizer for now. We have shown examples of index and join introduction in section 2 and [13] contains a nice example of join introduction. The C&B technique covers index and join introduction and in fact extends them by trying to introduce any relevant physical access structure. The experiments with **EC2** and **EC3** are already more complex than the examples in section 2 and [13]. It also covers join elimination (at the same time as tableau-like minimization) as part of subquery minimization during the backchase. The work that comes closest to ours in its theoretical underpinnings is [14] where chasing with functional dependencies, tableau minimization and join elimination with referential integrity constraints are used. Surprisingly, very few experimental results are actually reported in these papers. [6] reports on join elimination in star queries that are less complex than our experiments with **EC2**. Examples of SQO for OO systems appear in [8, 2, 10, 13, 7]. A general framework for SQO using rewrite rules expressed using OQL appears in [12, 11].

Techniques for using materialized views in query optimization are discussed in [5, 11, 12, 20, 3]. A survey of the area appears in [16]. From our perspective, the work on join indexes [21] and precomputed access support relations [15] belongs here too. The general problem is forced by data independence: how to reformulate a query written against a "user"-level schema into a plan that also/only uses physical access structures and materialized views efficiently. The GMAP approach [20] works with a special case of conjunctive queries (PSJ queries). The core algorithm is exponential but the restriction to PSJ is used to provide polynomial algorithms for the steps of checking relevance of views and checking a restricted form of query equivalence. However, the results we report here on using the chase show that there is no measurable practical benefit from all these restrictions. In the end, the exponential behavior of the GMAP algorithm and the difficulties we had to resolve for the backchase phase are closely related.

Our experiments include schemas, views and queries of significantly bigger complexity than those reported in [22, 20, 5]. Their experiments show that using views can be done and in the case of [20] that it can produce faster plans. But [22] measures only optimization time and [20] does not separate the cost of the optimization itself, so they do not offer any numbers that we can compare with our time reduction figures (section 5.4). [5] shows a very good behavior of the optimization time as a function of plans produced, but cannot be compared with our figures because the bag semantics they use restricts variable mappings to isomorphisms thus greatly reducing the search space.

7 Discussion and Extensions

Dynamic programming and cost-based pruning. Dynamic programming can only be applied when a problem is decomposable into independent subproblems, where common subproblems are solved only once and the results reused. Unfortunately, the minimization problem lacks common subproblems of big enough granularity: one cannot minimize in general a subpart of a subquery independently of how the subpart interacts with the rest of the query. In general, each subset of the bindings of the original query explored by the backchase must be considered as a different subproblem.

The non-applicability of dynamic programming is in general a problem for rewriting queries using views. What [20, 5] mean by incorporate optimization with views/GMAPs into standard System R-style optimizer is actually the blending of the usual cost-based dynamic programming algorithm with a brute-force exponential search of all possible covers. The algorithms remain exponential but cost-based pruning can be done earlier in the process.

Our optimizer can be easily extended in the same way. We have not yet done this, nor have we added any cost-based pruning to our system/experiments because we considered valuable as a first step to measure the effect of the C&B-specific issues in isolation. On the other hand, OQF already incorporates the principle of dynamic programming in the sense that it identifies query fragments that can be minimized independently.

Top-down vs bottom-up. In the top-down, full approach, the backchase explores only *equivalent* subqueries (call them *candidates*), and tries to remove one from binding at a time until a candidate cannot be minimized anymore (all of its subqueries are not equivalent). The main advantage of this approach is that through depth-first search it finds a first plan fast while the main disadvantage is that the cost of a subquery explored cannot be used [9] for cost-based pruning because a backchase step further might improve the cost. In the bottom-up approach the backchase would explore *non-equivalent* candidates. It would assemble subqueries of the universal plan by considering first candidates of size 1 then of size 2 and so on, until an equivalent candidate is reached. Then cost-based pruning is possible because a step of the algorithm can only increase the cost. A best-first strategy can

[9] We are ignoring here heuristics that need preliminary cost estimates.

be easily implemented by sorting the fragments being explored based on cost. The main disadvantage of this strategy is that it involves breadth-first search and the time for finding the first plan can be long.

In practice one could combine the two approaches: start top-down, find the first plan, then switch to bottom-up (combined with cost-based pruning) using the cost of the first plan as the cost of the best plan. While our FB implementation is a top-down approach now, we plan to extend it to include both strategies.

Other extensions. The two stratification strategies (OQF and OCS) introduced here are a first promising step in the direction of a deeper understanding of how the interference of constraints affects the chase/backchase rewrites. This is an attractive theoretical problem which we believe to be more tractable than the study of interference of rules in arbitrary rule-based optimizers. We intend to explore backchase strategies that are complete for query reformulation with other commonly used physical structures and integrity constraints.

Conclusion. In this work, we report on the implementation and evaluation of the uniform approach to semantic optimization and physical independence proposed in [9]. We developed and evaluated two refinements of the full C&B algorithm: OQF, a strategy preserving completeness in restricted but common scenarios, and OCS, a heuristic which achieves the best running times. Our experiments show that the strategies are practical and that OQF scales reasonably well, while OCS scales even better.

Finally, we remark that our comprehensive approach to optimization tries to exploit more optimization opportunities than common systems, thus trading optimization time for quality of generated plans. The experiments clearly show the benefits of this trade-off, even though we used a prototype rather than an implementation tuned for performance.

References

[1] Bonnie Baker. Responsible SQL: Creative Solutions for Performance Problems in DB2 for OS/390. *DB2 Magazine*, 4(2):54–55, Summer 1999. Available at `http://www.db2mag.com/summer99/99sp_prog.shtml`.

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

[3] R. Bello and al. Materialized Views in Oracle. In *Proc. of 24th VLDB Conference*, pages 659–664, 1998.

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

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

[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 *Proc. of VLDB*, pages 687–698, September 1999.

[7] M. Cherniack and S. B. Zdonik. Inferring Function Semantics to Optimize Queries. In *Proc. of 24th VLDB Conference*, pages 239–250, 1998.

[8] 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.

[9] Alin Deutsch, Lucian Popa, and Val Tannen. Physical Data Independence, Constraints and Optimization with Universal Plans. In *VLDB*, September 1999.

[10] 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.

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

[12] 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.

[13] J. Grant, J. Gryz, J. Minker, and L. Raschid. Semantic query optimization for object databases. In *Proc. of ICDE*, April 1997.

[14] M. Jarke, J. Clifford, and Y. Vassiliou. An optimizing prolog front-end to a relational query system. In *Proceedings of ACM-SIGMOD*, pages 316–325, 1984.

[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] A. Levy. Answering Queries Using Views: A Survey. Forthcoming.

[17] Greg Nelson and Derek C. Oppen. Fast decision algorithms based on union and find. In *FOCS*, pages 114–119.

[18] Lucian Popa and Val Tannen. Chase and axioms for PC queries and dependencies. Technical Report MS-CIS-98-34, University of Pennsylvania, 1998. Available online at http://www.cis.upenn.edu/~ techreports/.

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

[20] O. Tsatalos, M. Solomon, and Y. Ioannidis. The GMAP: A Versatile Tool for Physical Data Independence. *VLDB Journal*, 5(2):101–118, 1996.

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

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