University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS) Department of Computer & Information Science

January 1999

A Chase Too Far

Lucian Popa
University of Pennsylvania

Alin Deutsch
University of Pennsylvania

Arnaud Sahuguet
University of Pennsylvania

Val Tannen
University of Pennsylvania, val@cis.upenn.edu

Follow this and additional works at: https://repository.upenn.edu/cis_reports

Recommended Citation

Lucian Popa, Alin Deutsch, Arnaud Sahuguet, and Val Tannen, "A Chase Too Far", . January 1999.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-99-28.

*Previous paper referred to: "Chase & Backchase: A Method for Query Optimization with Materialized Views and Integrity Constraints", MS-CIS-01-16.

This paper is posted at ScholarlyCommons. https://repository.upenn.edu/cis_reports/101
For more information, please contact repository@pobox.upenn.edu.

A Chase Too Far

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, e.g. 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 terms 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).

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-99-28.

*Previous paper referred to: "Chase & Backchase: A Method for Query Optimization with Materialized Views and Integrity Constraints", MS-CIS-01-16.

This technical report is available at ScholarlyCommons: https://repository.upenn.edu/cis_reports/101

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 [11] 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 [27]

*Contact: University of Pennsylvania, Department of Computer and Information Science, 200 South 33rd Street, Philadelphia, PA 19104, Tel: (215)898-8701, Fax: (215)898-0587, Email: lpopa@gradient.cis.upenn.edu

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 [11, 22], the C&B technique is in fact sound for a much larger class of queries, physical structures and constraints. We describe here the performance of a first prototype that uses path-conjunctive query graphs internally. Extensions are possible and planned. We believe that 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, with security and encapsulation, and with multiple layers of logical/physical separation.

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 make the backchase phase efficient and very worthwhile even for quite challenging queries. Moreover, one of these strategies is *complete* for the important case of path-conjunctive materialized views [11, 22] just like the general technique.
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 [11, 27].

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 questions summarized above. In choosing them, we took as a starting point the experiments of [7, 31, 33, 29]. We reconstructed those experiments and found that our optimizer can also find the desired plans for a set of chosen queries. However, we went further by repeating the experiments on families of queries and schemas of similar structure but of increasing complexity. This allows us to find out *how far* (as the title of the paper asks) the technique can take us [1] and to show that the applicability range of the implementation likely includes the range of practical queries. And, for one of the configurations where we can use a conventional execution engine, we have also measured the global benefit of the C&B technique by measuring the reduction in total processing (optimization + execution) time, as a function of the complexity of the queries and the schema. Here we have additionally slanted the experiments against our technique by running the queries on a relatively small database.

[1] No doubt such breaking points also exist for the implementations in the cited papers, but no information about them has been published.

The rest of the paper is organized as follows. In section 2 we describe two motivating examples that support the goals of the C&B technique. Section 3 is one of the two central sections of the paper. It describes the implementation techniques we have designed to make C&B feasible and worthwhile.

Section 4 describes the architecture of our prototype. Section 5 is the other central section of the paper. It contains a description of our experimental configurations in 5.1, a description of the experiments evaluating the chase phase in 5.2, a description of the experiments evaluating the backchase phase in 5.3, and finally a description of the experiments that evaluate the global (optimization + execution) benefits of the approach in 5.4.

We survey related work in section 6. Section 7 discusses some possible improvements and extensions, while section 8 summarizes the work and describes some plans for the future.

Appendix A is based on [11] and briefly surveys our earlier ideas. A reader unfamiliar with [11] might want to read it between sections 2 and 3. Some of the details related to the two special backchase strategies introduced in 3 have been relegated to appendices B and C.

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 [2], showing the benefits of exploiting referential integrity constraints.

Consider a relation `R(A,B,C,E)` and a query that asks for all tuples in `R` with given values for attributes `B` and `C`:

```
(Q)      select  struct (A = r.A, E = r.E)
         from    R r
         where   r.B = b  and  r.C = c
```

The relation is very large, but the number of tuples that meet the where clause criteria is very small. However, the SQL engine is taking a long time in returning an answer. Why isn't the system using an index on `R` ? Simply because there is no index on the attributes `B` and `C`. The only index on `R` that includes `B` and `C` is an index, call it `I`, on `ABC`. There is no index with `B` and/or `C` in the high-order position(s), and the SQL optimizer chooses to do a table scan over `R` to answer the query (it might have been better to choose an index scan over `I` instead of a scan over the whole relation `R`).
There are several solutions to force the SQL optimizer to use the index on `ABC`: for example, if all possible values of `A` are known to be in the set $\{'01','02','03','04'\}$, one can hard-code in the where clause the condition `A` *in* $\{'01','02','03','04'\}$ and the problem is solved. Of course, this is not a real solution because tomorrow the values for `A` might change! The reader can find several other solutions in [2] but none are satisfactory except one: rewrite Q into an equivalent query that does a join of `R` with a small table `S` on attribute `A` knowing that there is a foreign key constraint from `R` into `S` on `A`:

```
(Q')     select  struct (A = r.A, E = r.E)
         from    R r, S s
         where   r.B = b  and  r.C = c  and  r.A = s.A
```

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

Example 2.2 Here we show that integrity constraints also create opportunities for rewriting queries using materialized views. Note that the experimental configuration **EC3** (section 5.1) is a generalization of this example.

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 0 depicts Q's join graph, in which the nodes represent the bindings of the query variables and the edges represent equijoins between them. The join conditions are shown on the edge labels.

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).

<u>select</u> <u>struct</u>($\mathtt{B}_{11} : s_{11}.\mathtt{B}$, $\mathtt{B}_{12} : s_{12}.\mathtt{B}$,
$\mathtt{B}_{21} : s_{21}.\mathtt{B}$, $\mathtt{B}_{22} : s_{22}.\mathtt{B}$)
<u>from</u> $\mathtt{R}_1\ r_1$, $\mathtt{S}_{11}\ s_{11}$, $\mathtt{S}_{12}\ s_{12}$,
$\mathtt{R}_2\ r_2$, $\mathtt{S}_{21}\ s_{21}$, $\mathtt{S}_{22}\ s_{22}$
<u>where</u> $r_1.\mathtt{F} = r_2.\mathtt{K}$ <u>and</u>
$r_1.A_1 = s_{11}.A_1$ <u>and</u> $r_1.A_2 = s_{12}.A_2$ <u>and</u>
$r_2.A_1 = s_{21}.A_1$ <u>and</u> $r_2.A_2 = s_{22}.A_2$

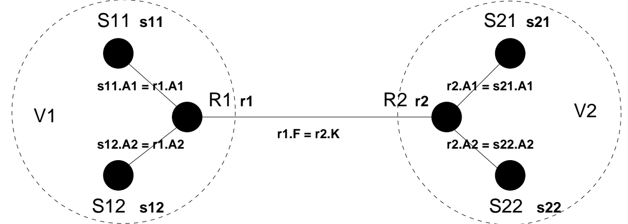


Figure 0: OQL definition and query graph for Q.

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* $\mathtt{V}_i(\mathtt{K},\mathtt{B}_1,\mathtt{B}_2)$ $(1 \le i \le 2)$, 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 $\mathtt{R}_i$:

(V_i) <u>select</u> <u>struct</u>($\mathtt{K} : r.\mathtt{K}$, $\mathtt{B}_1 : s_1.\mathtt{B}$, $\mathtt{B}_2 : s_2.\mathtt{B}$)
<u>from</u> $\mathtt{R}_i\ r$, $\mathtt{S}_{i1}\ s_1$, $\mathtt{S}_{i2}\ s_2$
<u>where</u> $r.A_1 = s_l.A_1$ <u>and</u> $r.A_2 = s_2.A_2$

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 $\mathtt{V}_{21}$:

(Q') <u>select</u> <u>struct</u>($\mathtt{B}_{11} : s_{11}.\mathtt{B}$, $\mathtt{B}_{12} : s_{12}.\mathtt{B}$, $\mathtt{B}_{21} : v_2.\mathtt{B}_1$, $\mathtt{B}_{22} : v_2.\mathtt{B}_2$)
<u>from</u> $\mathtt{R}_1\ r_1$, $\mathtt{S}_{11}\ s_{11}$, $\mathtt{S}_{12}\ s_{12}$, $\mathtt{V}_2\ v_2$
<u>where</u> $r_1.\mathtt{F} = v_2.\mathtt{K}$ <u>and</u>
$r_1.\mathtt{A}_1 = s_{11}.\mathtt{A}_1$ <u>and</u> $r_1.\mathtt{A}_2 = s_{12}.\mathtt{A}_2$

Less intuitively though, 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.

(Q'') <u>select</u> <u>struct</u>($\mathtt{B}_{11} : v_1.\mathtt{B}_1$, $\mathtt{B}_{12} : v_1.\mathtt{B}_2$, $\mathtt{B}_{21} : v_2.\mathtt{B}_1$, $\mathtt{B}_{22} : v_2.\mathtt{B}_2$)
<u>from</u> $\mathtt{R}_1\ r_1$, $\mathtt{V}_1\ v_1$, $\mathtt{V}_2\ v_2$
<u>where</u> $r_1.\mathtt{K} = v_1.\mathtt{K}$ <u>and</u> $r_1.\mathtt{F} = v_2.\mathtt{K}$

The reason is that V_1 does not contain the F attribute of R_1, and there is no guarantee that joining the latter with V_1 will recover the *correct* values of F. On the other hand, if we know that K is a key in 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.

At this point we suggest the following strategy for reading the rest of the paper. In appendix A we present a brief overview of the main ideas behind the C&B *technique, following [11]. The readers familiar with that paper could continue with the next section, while the other readers may want to read appendix A before continuing.*

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 the following:

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:

- fragmenting the query and stratifying the constraints by relevance to each fragment (the *On-line Query Fragmentation* aka *OQF* technique, section 3.2.1);
- splitting the constraints independently of the query (the *Off-line Constraint Stratification* aka *OCS* technique, section 3.2.2)

In the important case of materialized views [22], OQF can be used without losing any plan that might have been found by the full implementation (theorem 3.2). To evaluate and compare FB, OCS and OQF strategies, we measure in various experimental configurations (section 5.1) the: (1) number of plans generated (section 5.3.1), (2) time spent per generated plan (section 5.3.2) and the effect of fragment granularity (section 5.3.3). Finally, we address in section 5.4 the question whether the time spent in optimization is recovered by the gains in execution time.

3.1 Feasibility of the Chase

Each chase step of our algorithm includes searching for homomorphisms (see appendix A) mapping a constraint into the query. Finding a homomorphism is NP-complete, but only in the size of the universal part[2] 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. We mention here that our language is more complicated than a relational language because of dictionaries and nestings of sets. Therefore homomorphisms are more complicated (see appendix A for full definition) than just simple mappings between goals of conjunctive queries, and checking that a mapping from a constraint into a query is indeed a homomorphism is not cheap (even though polynomial).

~~Here are several techniques that we use to sp~~eed-up and/or avoid unnecessary checks for homomorphisms:

[2]See appendix A for the logical form in which we express constraints.

- Use of congruence closure, a variation of [25], for fast checking if an equality is a consequence of the where clause of the query.
- Ruling out (because of redundancies) homomorphisms previously used in the chase sequence[3]
- 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. The idea is the following: if h is a mapping that is defined on variables 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[4]) the result of each homomorphism applied to the existential part of the constraint. The similarity between chase and 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.

3.2 Feasibility of the Backchase

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

Example 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 $\mathtt{A}$. 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 a plausible plan. One direct way to obtain all of them is to chase Q with the entire set of constraints D, obtain the universal plan (of size $2n$), and then backchase it with D. If the backchase goes top-down from the universal plan, it inspects all possible subqueries of $2n-1$, ..., n loops (it stops at n because any subquery with less than n loops cannot be equivalent to Q, in this case), for a total of: $C_{2n}^{2n-1} + \ldots + C_{2n}^{n} = 2^{2n-1} + \frac{1}{2}C_{2n}^{n} - 1$.

Continuing the example, the same 2^n resulting plans can be obtained with the following different strategy, much closer to the one implemented by standard optimizers. For each i, handle the 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.

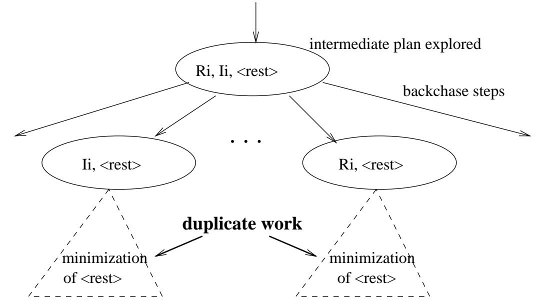
The number of plans inspected by this *"stratified"* approach can be computed as follows. For each stage i the universal plan for fragment Q_i has only 2 loops (over $\mathtt{R}_i$ and $\mathtt{I}_i$) and therefore the number of plans explored by the subsequent backchase is 2. Thus the work to produce all the plans for all fragments is $2n$. The total work, including assembling the plans, is then $2n + 2^n$.

This analysis suggests that 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*

[3] Without this, a check for non-redundancy must be done and this is also NP-complete [27]!
[4] this is translated as a check for trivial equivalence

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 on the right). The minimization work that follows is exactly the same in both cases because it operates only on the rest of the relations. This duplication of work is avoided in the second strategy because each loop of Q is handled exactly once. A solution that naturally comes to mind to avoid such situations is to use dynamic programming. Unfortunately, there is no straightforward way to do this and we leave the discussion of this issue in section 7. Instead, the next section gives a stratification algorithm that solves the problem for a restricted but common case.



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 1. 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 B attributes of all corners $\mathtt{S}_{ij}$.

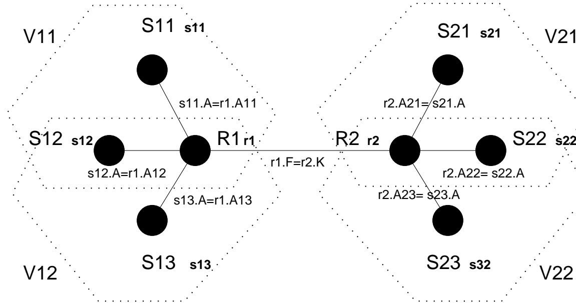


Figure 1: Chain-of-stars query Q with views

As suggested by the dotted polygonal lines, 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 B attributes of the corner relations it joins, as well as the K attribute of $\mathtt{R}_i$.

If we apply the FB algorithm with all the constraints describing the views we obtain all possible plans in which views replace some parts of the original query. However it should be clear that V_{11} or V_{12} can only replace relations from the first star, thus not affecting any of the relations in the second star. If a plan P using V_{11} and/or V_{12} is obtained for the first star, such that it "recovers" the B attributes needed in the result of Q, as well as the 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} does not overlap with neither V_{21} nor V_{22}. On the other hand this does not apply to V_{11} and V_{12}, because the parts of the query that they cover overlap (and any further

decomposition will in fact lose the plan that uses both V_{11} and V_{12}). Q can thus be decomposed into precisely two query fragments, one for each star, that can be optimized independently.

In appendix B we give the full description of the algorithm for *query decomposition into fragments*, Algorithm B.1. Here we only mention that it is based on computing the connected components of the *interaction graph* of constraints that map homomorphically into the query, and that it is restricted to a class of physical access structures that we call *skeletons*, class that includes indexes, materialized views, ASRs etc. (see full definition in appendix B). With this, we define the on-line query fragmentation strategy as follows:

Algorithm 3.1 (OQF) Given a query Q and a set $\mathcal{V}$ of skeletons:

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

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

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

Theorem 3.2 *For a skeleton schema, OQF produces the same plans as the full backchase (FB) algorithm.*

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

3.2.2 Off-line Constraint Stratification (OCS)

One disadvantage of OQF is that it needs to find the fragments of a query Q. While this has about the same complexity as chasing Q [5] (and we have argued that chase itself is not a problem) in practice there may be situations in which interaction between constraints can be estimated in a pre-processing phase that examines only the constraints in the schema. The result of this phase is a partitioning of constraints into disjoint sets such that only the constraints in one set are used at one time by the backchase algorithm. As opposed to query fragmentation this method tries to isolate the independent optimizations that may affect a query by stratifying the constraints without fragmenting the query. During the optimization process the entire query is pipelined through stages in which the chase/backchase algorithm uses only the constraints in one set. At each stage different parts of the query are affected.

Similarly to OQF, this algorithm finds first the connected components in a constraint interaction graph which however is constructed in a different, query-independent way. The result of this stage is a partitioning of the set of initial constraints into disjoint sets of constraints (*strata*). The full details of the algorithm for stratification of constraints, algorithm C.1, are left for the appendix C. Based on the above partitioning, the following refinement of the C&B strategy, the *off-line constraint stratification* backchase (OCS) uses only constraints from one stratum at a time.

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

Algorithm 3.3 (OCS) Given a query Q and a set of constraints $\mathcal{C}$:

Step 1. Partition $\mathcal{C}$ into disjoint sets of constraints $\{S_i\}_{1\leq i\leq k}$ by using algorithm C.1.
Step 2. Let $P_0 = \{Q\}$.
Step 3. For every $1 \leq i \leq k$, 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.
Step 4. Output P_k as the set of plans.

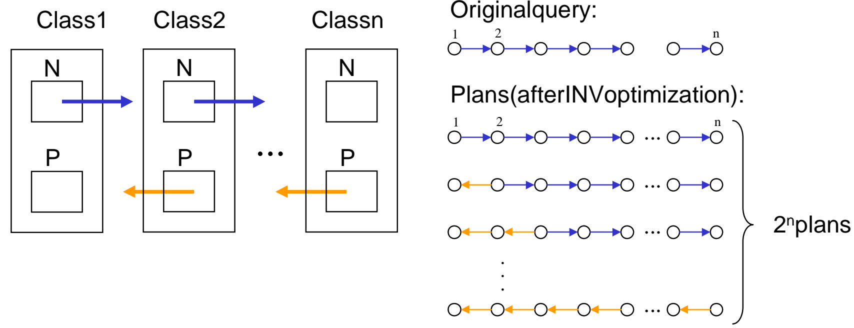


Figure 2: Inverse Relationships

Example 3.3 To illustrate the algorithm, we consider 3 classes (see figure 2 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 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 the following constraints:

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

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

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

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

By chase/backchasing Q with the constraints of the first stratum, $\{INV_{1N}, INV_{1P}\}$, 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.

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

In the stage corresponding to stratum 2, we chase/backchase $\{Q,\ Q_1\}$ with $\{INV_{2N}, INV_{2P}\}$, 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), and the additional Q_2 (obtained from Q) and Q_3 (obtained from Q_1 and shown below).

$$Q_3 \qquad \begin{array}{ll} \underline{\mathsf{select}} & \underline{\mathsf{struct}}(\mathtt{F} = o_1,\ \mathtt{L} = k_3) \\ \underline{\mathsf{from}} & \underline{\mathsf{dom}}\,\mathtt{M}_3\ k_3,\ \mathtt{M}_3[k_3].\mathtt{P}\ o_3,\ \underline{\mathsf{dom}}\,\mathtt{M}_2\ k_2,\ \mathtt{M}_2[k_2].\mathtt{P}\ o_1 \\ \underline{\mathsf{where}} & o_3 = k_2 \end{array}$$

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 C.1 makes optimistic assumptions about the non-interaction of constraints, which depending on the input query, may turn out to be false, therefore there is no completeness guarantee. **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

In this section we give a brief overview of the prototype that is used for our experimental results. The implementation of the prototype has been done in Java (25,000 lines of code).
The architecture of the system that implements the C&B based optimization is shown in figure 3. The arrowed lines show the main flow of a query being optimized, constraints from the schema, and resulting plans. The thick lines show the interaction between modules. The main module is the *plan generator* which, when given a query, 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[6]. The module that does the check, *dependency implication* shown in the figure as $D \Rightarrow d$, uses the chase (and therefore the chase module) and the *triviality check* module.

The most salient features of the implementation are summarized below:

- queries and constraints are compiled into a (same!) internal congruence closure based canonical database representation (shown in the figure as $DB(Q)$ for a query Q, respectively $DB(d)$ for a constraint D) that allows for fast reasoning about equality.
- compiling a query Q into the canonical database is implemented itself as a chase step on an empty canonical database with one constraint having no universal but one existential part isomorphic to Q's from and where clauses put together. Hence, the query compiler, constraint compiler and the chase modules are basically one module.
- in addition to internal, a language for describing queries and constraints that is as user friendly as 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.

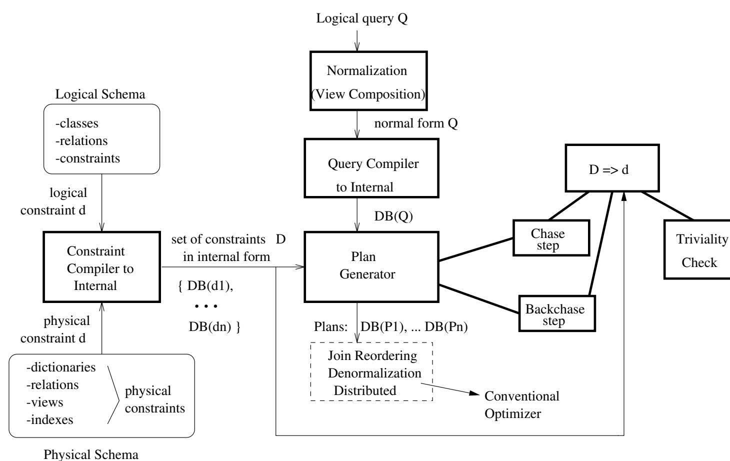


Figure 3: C&B Optimizer Architecture

[6]The other containment is always true.

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 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 for which we want to demonstrate that our optimizer performs quite well under various strategies.

The schema includes n relations, each relation $\mathtt{R}_i$ having a key attribute K on which there is a primary index $\mathtt{PI}_i$, a foreign key attribute N, and some 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 (see figure 4) that join $\mathtt{R}_i$ with $\mathtt{R}_{i+1}$ on attributes N and K, respectively. The attributes in the select clause are not very important here and we return all the key attributes of the relations involved. The two scaling parameters for our experiments are n and m.

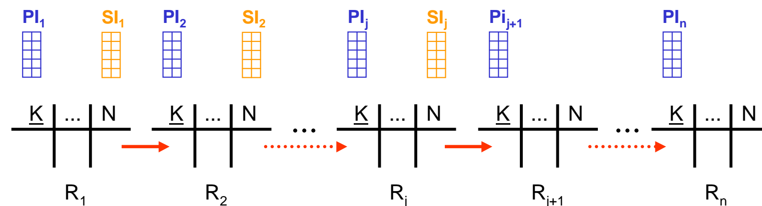


Figure 4: Chain query

Experimental Configuration EC2:
The second setting is designed to illustrate experimental results in the presence of materialized views and key constraints that the optimizer can take advantage of in finding good plans.

We consider a generalization of the chain of stars query of examples 2.2 and 3.2 (see figure 1) 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 that are i, j and v.

Experimental Configuration EC3:
The third experimental setting is an object-oriented configuration with classes obeying many-to-many inverse relationship constraints. We use it to show how we can mix semantic optimization based on the inverse constraints to discover plans that use access support relations (ASRs). The query that we consider is not directly "mappable" into the existing ASRs, and the first optimization phase of our experiments (semantic optimization) enables rewriting the query into equivalent queries that *can* map into the ASRs. The mapping into ASRs is done in the second phase (physical optimization).

We generalize here the scenario considered in example 3.3 by considering n classes with inverse relationships.

The queries Q (see figure 2) that we consider are long navigation queries across the entire database following the `N` references from class M_1 to class M_n. In addition we consider as part of the physical schema access support relations (ASRs) that are materialized navigation joins across three classes going in the backwards direction (i.e. following the `P` references). Each ASR is a binary table storing oids from the beginning of the navigation path and the corresponding oids from the end of the navigation path. Plans obtained after the inverse optimization phase are rewritten in the second phase into plans that replace a navigation chain of size 2 with one navigation chain of size 1 that uses an ASR (thus being likely better plans).

The parameters of the configuration are the number of classes, n, and the number of ASRs, m.

Experimental settings
All the experiments have been realized on a dedicated commodity workstation (Pentium III, Linux Red Hat 6.0, 128MB of RAM, 6.4GB of hard-drive). The optimization algorithm (chase, backchase) is fully implemented in Java and is run using IBM runtime environment for Linux (alpha version 1.1.8).
The database management system used to execute queries is IBM DB2 version 6.1.0 for Linux (out-of-the-box configuration). For **EC2**, materialized views have been produced by creating and populating tables.
All times measured are *elapsed times*, obtained using the Unix shell `time` command. 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.

5.2 Feasibility of the Chase: Experiments

We measured the complexity of the chase in all our experimental configurations varying both the size of the input query and the number of constraints in the schema. We did not consider any stratification of the query or constraints because the numbers for the *full* chase are fine.

In **EC1** (figure 5, left) the constraints used in the chase are the ones describing the primary (2 constraints/index) and/or secondary (3 constraints/index) indexes. For example, chasing with 10 indexes, therefore 20+ constraints, takes under 1s. For **EC2** (figure 5, middle) the variable is the number of relations in the from clause, giving a measure of the query size. The number of constraints comes from the number of views (2 constraints/view) and the number of key constraints (1 constraint/star hub). For **EC3** (figure 5, right) the variable is the number of classes C (measuring both the size of the schema and that of the queries we use). The chase is done with the inverse relationship constraints (2 constraints/relationship, $2 \times (C-1)$ total) and with the ASR constraints (2 constraints/ASR, $\lfloor (C-1)/2 \rceil$ total). For example, chasing with 8 classes, therefore 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.

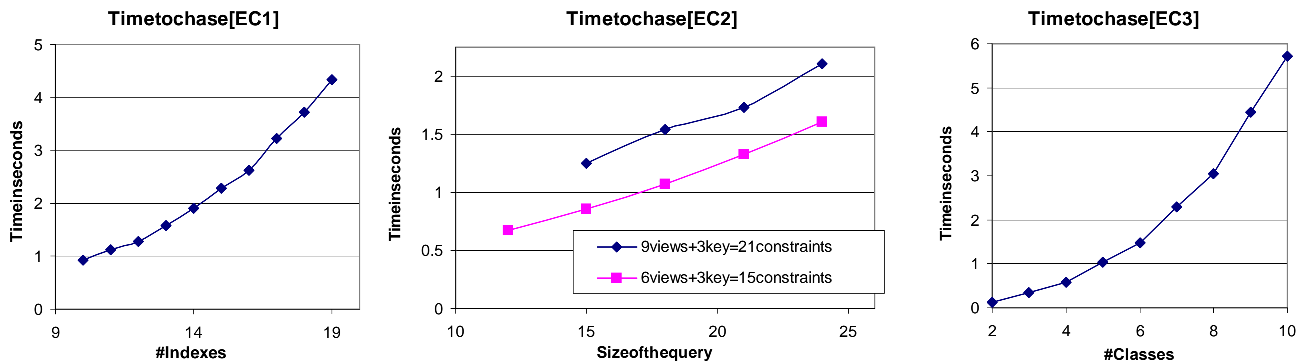


Figure 5: Effect on chase time of increasing schema and query parameters

5.3 Feasibility of the Backchase: Experiments

To evaluate and compare the two stratification strategies (OQF and OCS) and the full approach (FB) we measure, in each of the experimental configurations (section 5.3), the following:

- **The number of plans generated** (section 5.3.1) measures the completeness with respect to FB. We found that OQF was complete for all experimental configurations considered, beyond what theorem 3.2 guarantees. As expected, both OQF and FB outperformed OCS.
- **The time spent per generated plan** (section 5.3.2) allows for a fair comparison between all three strategies. We measured the time per plan as a function of the query size and number of constraints. Moreover, we studied the scale-up for each strategy by pushing the values of the parameters to the point at which the strategy became ineffective. We found that OQF performed much better than OCS which in turn outperformed FB.
  Remark. Another possible measure would be the efficiency of the search (the useful work performed during the backchase) measured as the ratio between the number of generated plans and the number of explored subqueries. We expect that OQF would greatly outperforms FB here but OCS would be difficult to compare because it does not generate the same number of plans. However, a pleasant experimental observation and an indicator of the robustness of the implementation is that the time per subquery explored stays relatively constant for all three strategies, in all experimental configurations, for various query sizes and various numbers of constraints. This means that the efficiency of the search can in fact be estimated as the inverse of the time per generated plan, mentioned above.
- **The effect of fragment granularity on optimization time** (section 5.3.3) 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.

5.3.1 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).

The number of generated plans, as a function of the size of the query and the number of constraints.

We ran the experiment for all three configurations. For **EC1**, we varied the number r of relations involved in the join (which equals the number of primary indexes) and the number j of secondary indexes at our disposal. For **EC2**, we varied the query size by increasing the number s of stars per query and the number c of corners per star. The number of key constraints was fixed to the number of stars (one constraint for every star hub). We varied the overall number of constraints by varying the number v of views applicable per star. The query size is given by $s(c+1)$, the number of constraints by $s(1+2v)$ (two constraints per view). For **EC3**, we varied the query size by increasing the number n of classes traversed during the navigation. The number of inverse constraints necessarily varied linearly with the size of the query.

The three strategies yielded the same number of generated plans in configurations **EC1** and **EC3**. The table below shows the results for configuration **EC2**:

Number of plans in EC2

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

As expected, the complete FB strategy outperforms CQF, which in turn performs much better than OCS. Note that in the common case of index introduction, all three strategies generate all the plans. The same holds for the less conventional **EC3** scenario. However, the time spent for generating the plans differs spectacularly among the three techniques, as shown by the next experiment.

5.3.2 Optimization time spent per generated plan

This experiment compares the three backchase strategies by *optimization time.*

Because not all strategies are complete and hence output different numbers of plans, we ensured fairness of the comparison by normalizing the optimization time which was divided by the number of generated plans. This normalized measure is called *time per plan (tpp)* and was measured as a function of the size of the query and the number of constraints.

We ran the experiment for all three configurations, varying the parameters as described in the previous experiment and the results are shown in figures 6 and 7.

The purpose of running the experiment in configuration **EC1** was to show that for the trivial yet common case of index introduction, our algorithm's performance is comparable to that of standard relational optimizers. Indeed, figure 6 shows the results obtained for three query sizes: 3, 4 and 5. By varying the number of secondary indexes for each query size, we observed an exponential behavior of the time per plan for the FB strategy, but a negligible time per plan for both OQF and OCS.

For configuration **EC3**, it turns out that OQF degenerates into FB because the images of the inverse constraints overlap[7]. We show a comparison of FB(=OQF) and OCS. The missing FB bars for a number of traversed classes larger than 4 indicate that the total optimization time needed by FB exceeded our timeout threshold of 2 minutes and the experiment was interrupted. OCS outperforms the other two strategies on this example because each pair of inverse constraints ends up in its own stratum. This stratification results in a *linear* time per plan (each stratum flips one join direction).

The most challenging configuration is **EC2**, dealing with large queries and numerous constraints. For example, the point corresponding to 4 stars of 4 corners and 2 views each corresponds to a query of 19 joins to which 20 constraints apply! Figure 7 divides the points into 4 groups, each group corresponding to the same number of views per star. This value determines the size of the query fragments and constraint strata for OQF, respectively OCS, and turns out to be the most important factor influencing the complexity. Again, missing data corresponds to timeout for our experiments.

While all strategies exhibit exponential time per plan, OCS is fastest, while FB cannot keep pace with the other two strategies [8].

5.3.3 The effect of stratification on the optimization time

This experiment was run in configurations **EC2** and **EC3** by keeping the query size constant and varying the number of strata in which the constraints are divided. For **EC3**, we considered two queries: one navigating

[7] The inverse between $\mathtt{M}_i$ and $\mathtt{M}_{i+1}$ with that between $\mathtt{M}_{i+1}$ and $\mathtt{M}_{i+2}$ overlap on a binding involving $\underline{\mathsf{dom}}\, M_{i+1}$ - see appendix B

[8] Note though that we only measure time per plan here, not the quality of the generated plans (OCS systematically misses the best plan, which uses all the views). For a comparison of the cost versus benefit in this configuration, see experiment 5.4

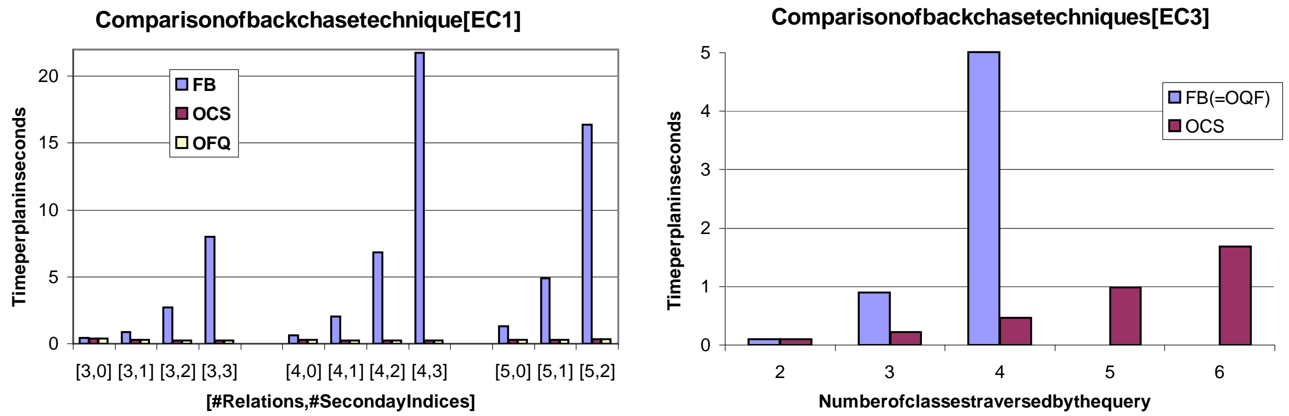


Figure 6: Comparison of FB, OQF, OCS for: EC3 (left) and EC1 (right)

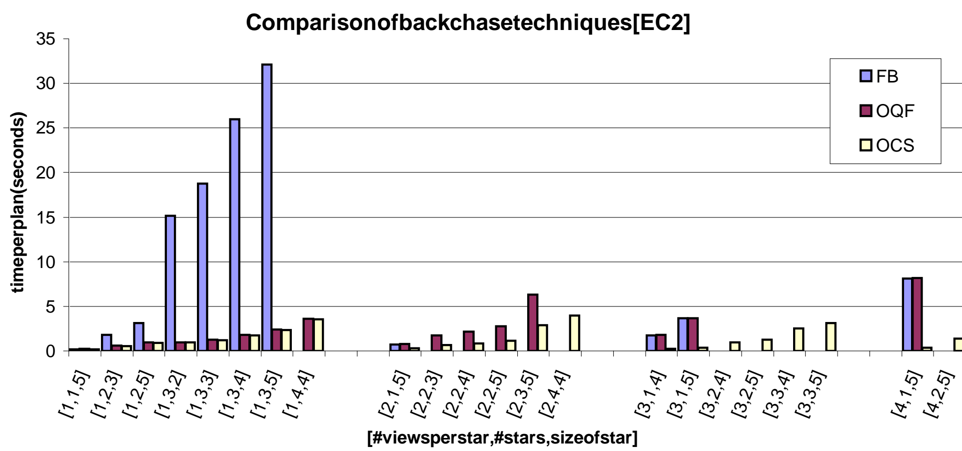


Figure 7: Comparison of FB, OQF, OCS for EC2

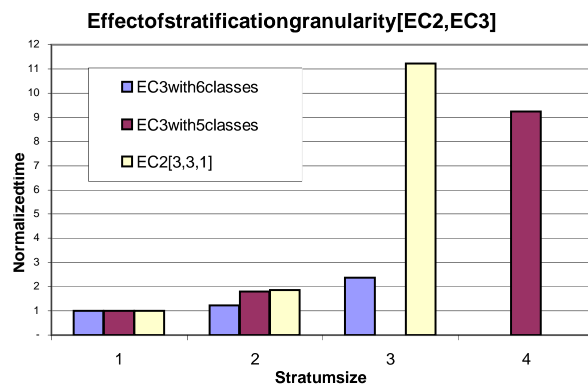


Figure 8: Effect of stratification on the optimization time

over 5 classes and one over 6 classes, with 8, respectively 10 applicable constraints. The query considered in configuration **EC2** joins three stars of 3 corners each, with one view applicable per star (for a total of 9 constraints).
The results are shown in figure 8. We observe an exponential reduction of the optimization time with the reduction in strata size. Note that the point of stratum size 1 corresponds for **EC3** to OCS. These results corroborate the analytical analysis of example 3.1: by decomposing a fixed query into fragments of decreasing size in a completeness-preserving way [9], we observe an exponential reduction of the optimization time. This result validates the OQF strategy which achieves the minimal decomposition that doesn't lose plans. Moreover, it suggests that by decomposing beyond the threshold of preserving completeness, heuristics such as OCS are trade-offs giving up completeness for optimization time.

5.4 The Benefit of Optimization

In this section, we measure 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. We use **EC2** with materialized views and key constraints, as presented at the beginning of section 5. Queries are optimized using the OQF strategy and fed into DB2 for comparing their processing times.

Plan #	Execution time (s)	Views used	Corner relations used
1	5.54	$V_{1,1}$, $V_{2,1}$, $V_{3,1}$	
2	66.39	$V_{1,1}$, $V_{2,1}$	$S_{3,1}$, $S_{3,2}$
3	33.13	$V_{1,1}$, $V_{3,1}$	$S_{2,1}$, $S_{2,2}$
4	143.75	$V_{1,1}$	$S_{2,1}$, $S_{2,2}$, $S_{3,1}$, $S_{3,2}$
5	105.82	$V_{2,1}$, $V_{3,1}$	$S_{1,1}$, $S_{1,2}$
6	61.45	$V_{2,1}$	$S_{1,1}$, $S_{1,2}$, $S_{3,1}$, $S_{3,2}$
7	43.54	$V_{3,1}$	$S_{1,1}$, $S_{1,2}$, $S_{3,1}$, $S_{3,2}$
8	132.90		$S_{1,1}$, $S_{1,2}$, $S_{2,1}$, $S_{2,2}$, $S_{3,1}$, $S_{3,2}$ (*) original query
# Stars:3, # Corner relations per star:2, # Views per star:1. 8 plans generated. Time to generate all plans: 8s			

Figure 9: A detail of the plans generated for one instance of **EC2**

Parameters measured We denote by OptT the time take by C&B to optimize the query; by ExT the execution time of the query given to DB2 in its original form (no C&B optimizaton); and by ExTBest, the DB2 execution time of the best plan generated by the C&B optimization.
We have $\mathsf{ExTBest} \leq \mathsf{ExT}$ since the original query is always part of the generated plans.
We assume that the cost of picking the best plan among those generated by the algorithm is negligible.

Performance indices We define and display in figure 10, for increasing complexity of the experimental parameters, the following performance indices:

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

$$\mathsf{Redux} = \frac{\mathsf{ExT} - (\mathsf{ExTBest} + \mathsf{OptT})}{\mathsf{ExT}} \quad \text{and} \quad \mathsf{ReduxFirst} = \frac{\mathsf{ExT} - (\mathsf{ExTBest} + \frac{\mathsf{OptT}}{\#plans})}{\mathsf{ExT}}.$$

Negative values of Redux are not displayed.

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

[9] Note that FB and OQF are obtained as the extremes of this spectrum of decompositions.

$\lvert R_i \rvert$	$\lvert S_{i,j} \rvert$	$\sigma(R_i \bowtie S_{i,j})$	$\sigma(R_i \bowtie R_{i+1})$
5,000 tuples	5,000 tuples	4%	2%

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

We also give the details of all the plans generated (8 plans in this case) and their ExTBest values for one instance of the configuration parameters in figure 9. For each generated plan, we present the views used and the star corner relations that these views and the star hub relations are joined with.

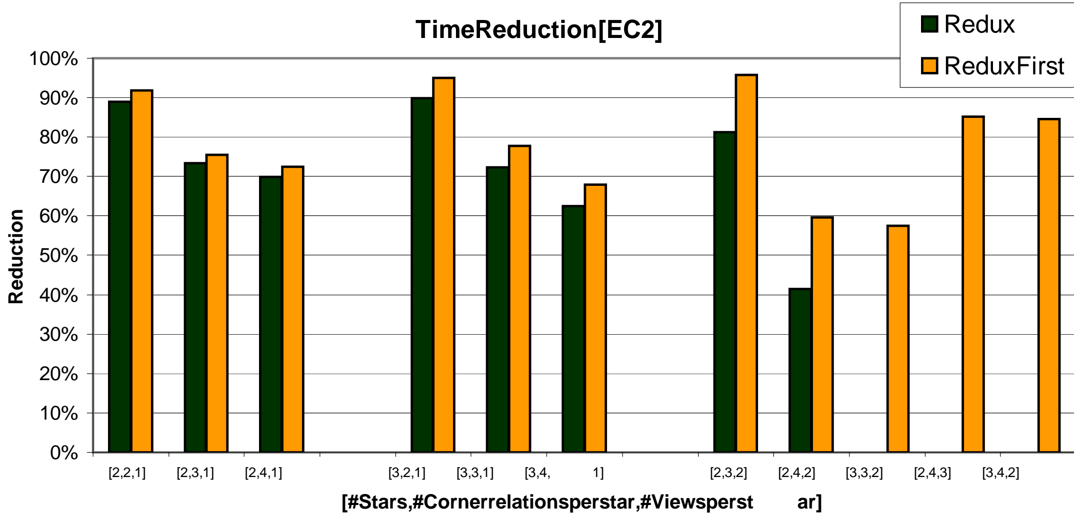


Figure 10: Time reduction

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

Even without the heuristic of stopping the optimization after the first plan, the C&B posts significant time reductions (40% to 90%), up to optimizing chain of stars queries as complex as having $2 \times (4+1) = 10$ relations with 9 joins, using $2 \times 2 = 4$ views and $2 \times 4 + 2 = 10$ constraints (parameter `[2,4,2]` in figure 10). The practicality range is extended even further when using the *"best plan first"* heuristic, with reductions of 60% to 95%, up to optimizing queries with $3 \times (4+1) = 15$ relations with 14 joins, using $2 \times 3 = 6$ views and $2 \times 6 + 3 = 15$ constraints (parameter `[3,4,2]` in figure 10).

Note that these numbers correspond to one run of the query. The benefit is much higher when the cost of optimization is amortized over multiple runs (as is often the case, e.g. OLAP environments).

6 Related work

There are many papers that discuss semantic query optimization for relational systems. An incomplete list includes [7, 17, 24, 5, 29] and the references therein. The techniques most frequently used are [29] *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 [17] 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 [17]. 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 [18] 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. [29] contains one experiment each for index introduction and join elimination, both with queries and schemas of lesser complexity than what we have considered. [7] reports on join elimination in star queries that are still less complex than our experiments with **EC2**.

Examples of SQO for OO systems appear in [28, 10, 9, 3, 14, 13, 17, 8] Use of referential integrity constraints to eliminate *dependent* joins is implicit in [19, 10, 20, 21]. A general framework for SQO using rewrite rules expressed using OQL appears in [16, 15].

Techniques for using materialized views in query optimization are discussed in [33, 31, 6, 15, 16, 30, 12]. A survey of the area appears in [22]. From our perspective, the work on join indexes [32] and precomputed access support relations [20, 21] 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 [31, 30] works with a special case of conjunctive queries (PSJ queries). In contrast to the query plans obtained by our rewriting process, the output of the GMAP rewriting is a family of plans represented by a PSJ query. The burden of choosing a specific plan is shifted on the next phase of the optimizer. The core algorithm is exponential but the restriction to PSJ is used to provide polynomial algorithms for the steps of checking relevance of views and checking a restricted form of query equivalence. Both checks are made more flexible by taking certain restricted integrity constraints into account. However, the results we report here on using the chase show that there is no measurable practical benefit from all these restrictions. In the end, the exponential behavior of the GMAP algorithm and the difficulties we had to resolve for the backchase phase are closely related.

Our experiments include schemas, views and queries of significantly bigger complexity than those reported in [33, 31, 30, 6]. These experiments show that using views can be done and in the case of [31, 30] that it can produce faster plans. But [33] measures only optimization time and [31, 30] does not separate the cost of the optimization itself, so they do not offer any numbers that we can compare with our figures time reduction (section 5.4). [6] 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 Possible Improvements and Extensions

Dynamic programming (?) and cost-based pruning. Dynamic programming can be applied when a problem is decomposable into subproblems such that the subproblems *share* some of *their* subproblems. In that case the common subproblems are solved only once and the results reused. However, the backchase minimization problem lacks common subproblems of big enough granularity. We illustrate this on a simple example:

Consider the query $Q = R \bowtie S \bowtie T \bowtie U$ and assume the existence of a materialized view $V = R \bowtie S \bowtie T$. We have to minimize the universal plan $R \bowtie S \bowtie T \bowtie U \bowtie V$, and let's say we need to solve the following two of the subproblems of size 3: $\mathcal{S}_1 = \{R, S, T\}$ and $\mathcal{S}_2 = \{R, S, V\}$. One would like now to identify $\{R, S\}$ as a common subproblem of $\mathcal{S}_1$ and $\mathcal{S}_2$, minimize it (once!) and use the result ($\{R, S\}$ in this case) in both $\mathcal{S}_1$ and $\mathcal{S}_2$. Then the solutions for $\mathcal{S}_1$ and $\mathcal{S}_2$ would be computed as $\{R, S, V\}$ and $\{R, S, T\}$, respectively. While the solution found for $\mathcal{S}_1$ is minimal the solution found for $\mathcal{S}_2$ is not!

The problem here is that $R \bowtie S$ was identified *falsely* as a common subproblem of $\mathcal{S}_1$ and $\mathcal{S}_2$. One cannot minimize in general a subpart of a subquery independently of how the subpart interacts (through redundancy) 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 if it appears in a different combination with the rest of the query.

The same argument shows that one cannot employ (in a straightforward way) dynamic programming for finding, for example, minimal covers with materialized views or GMAPs of a query. What [31, 30, 6] 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 an extension of the dynamic programming principle in the sense that it identifies query fragments that can be minimized independently.

Top-down vs bottom-up backchase. 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 (a minimal candidate) fast while the main disadvantage is that the cost of a subquery explored cannot be used [10] for cost-based pruning because a backchase step further might improve the cost. In the bottom-up approach the backchase would explore only *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 a candidate that is equivalent to the universal plan is reached. The main advantage of this approach is that cost-based pruning is possible because a step of the algorithm can only increase the cost. A best-first strategy can 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: for example, 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. One interesting question here is whether one can estimate (maybe through heuristics), given the universal plan and a set of constraints, what is the ratio between equivalent and non-equivalent candidates, and then choose the right approach: top-down or bottom-up.

While our FB implementation is a top-down approach now, we plan to extend it to include both strategies.

Best plan first with stratification For the stratified techniques (OQF and OCS) our experiments showed that using the simple heuristics of sorting the plans by giving priority to the ones that use more views or indexes usually yields the best plans. This could be done more systematically by using cost and we plan to extend our experiments into this direction for OQF, OCS and the future bottom-up FB.

8 Conclusion and Future Work

In this work, we report on the implementation and evaluation of the uniform approach to semantic optimizations and physical independence proposed in [11]. Our implementation went through two stages. The original stage only implemented the full backchase (FB) strategy. Only after running the first experiments did the necessity of more refined strategies for the backchase emerge (the chase turned out to be very fast).

We have developed and evaluated two refinements of the full backchase algorithm: OQF, a strategy which

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

preserves completeness in restricted but common scenarios, and OCS, a heuristic which achieves the best running times by giving up completeness. Our experiments show that both strategies are practical and that OQF scales reasonably well, while OCS scales even better.

The modular architecture of our optimizer was crucial in allowing us to easily add the query fragmentation and constraint stratification techniques described in this work. Moreover this made it easy to conduct additional experiments (not included in this paper) involving other heuristics, such as using indexes, ASRs and views whenever available. In all these cases the optimization times were negligible.

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.

Future Work. 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.

The chase technique handles only equality conditions, hence our algorithm does not perform any reasoning on the bounds of range selection predicates, which is a common technique in relational optimizers. We plan to extend our algorithm to incorporate this.

We also intend to explore backchase strategies that are complete (in the sense of theorem 3.2) for query reformulation with other commonly used physical structures and integrity constraints.

References

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

[2] 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`.

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

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

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

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

[7] 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 *International Conference on Very Large Databases (VLDB)*, pages 687–698, September 1999.

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

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

[10] 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.

[11] Alin Deutsch, Lucian Popa, and Val Tannen. Physical Data Independence, Constraints and Optimization with Universal Plans. In *International Conference on Very Large Databases (VLDB)*, pages 459–470, September 1999.

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

[13] 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.

[14] Leonidas Fegaras and David Maier. Towards an effective calculus for object query languages. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 47–58, San Jose, California, May 1995.

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

[16] 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.

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

[18] 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.

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

[20] 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.

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

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

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

[24] A. Levy and Y. Sagiv. Semantic query optimization in datalog programs. In *Proceedings of PODS*, 1995.

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

[26] 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/.

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

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

[29] S. Shenoy and M. Ozsoyoglu. Design and implementation of a semantic query optimizer. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):344–361, 1989.

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

[31] Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. The GMAP: A Versatile Tool for Physical Data Independence. In *Proc. of 20th VLDB Conference*, pages 367–378, Santiago, Chile, 1994.

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

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

A Earlier Ideas

In this section we present a brief overview of the main ideas behind the C&B technique, following [11]. The optimization algorithm introduced there starts with a query Q against a logical schema and produces a query plan Q' against the physical schema. Q' is equivalent to Q under all the constraints and is selected according to a cost model. In addition to optimization for physical data independence, the algorithm performs semantic optimizations allowed by the constraints of the logical schema and eliminates superfluous computations (as in tableau minimization [1]).

We use *dictionaries* (finite partial functions) to represent physical access structures with cheap random access, such as indexes (captured as dictionaries from the key attribute to the corresponding tuples), and the implementation of classes (represented as dictionaries from the oid to the tuple containing the attributes of the object).

The language for expressing the logical and physical schema as well as queries and plans against them is ODMG/ODL and ODMG/OQL ([4]), extended with a few constructs concerning dictionaries. We used ODL's $\mathsf{Dict}\langle T_1, T_2\rangle$ for the type of dictionaries with keys of type T_1 and entries of type T_2, and OQL's $M\,[\,k\,]$ the *lookup* operation that returns the entry corresponding to the key k in the dictionary M. To this we added the operation $\underline{\mathsf{dom}}\, M$ that returns the **domain** of the dictionary M, i.e., the set of keys for which M is defined.

Example. Recall relation R from example 2.1 and assume it has type $\mathsf{Set}\langle T\rangle$ where $T : \mathsf{Struct}\{\mathtt{A} : int, \mathtt{B} : int, \mathtt{C} : string, \mathtt{D} : int\}$). Then the composite key index I is modeled as a dictionary of type $\mathsf{Dict}\langle\mathsf{Struct}\{\mathtt{A} : int, \mathtt{B} : int, \mathtt{C} : string\}, T\rangle$ and the plan using I can be expressed in our extended OQL syntax as

$$(P) \qquad \underline{\mathsf{select}}\ \ \underline{\mathsf{struct}}(\mathtt{A} = s.\mathtt{A}, \mathtt{E} = \mathtt{I}\,[\,\underline{\mathsf{struct}}(\mathtt{A} = s.\mathtt{A},\ \mathtt{B} = b,\ \mathtt{C} = c)\,].\mathtt{E}) \\ \underline{\mathsf{from}}\ \ \mathtt{S}\ s$$

The unifying approach to semantic optimizations and physical independence is made possible by representing both constraints on the logical schema and physical access structures in the same way.

Continuing the above example, the referential integrity constraint (RIC) from R into S on attribute A is expressed as $\forall(r \in \mathtt{R})\ \exists(s \in \mathtt{S})\ r.\mathtt{A} = s.\mathtt{A}$, while the key constraint (KEY_1) on relation $\mathtt{R}_1$ in example 2.2 is $\forall(r \in \mathtt{R}_1)\forall(r' \in \mathtt{R}_1) r.\mathtt{K} = r'.\mathtt{K} \Rightarrow r = r'$.

The index I, though expressible as a query in our extended OQL syntax, is modeled as a set of constraints three constraints, of which the more interesting two describe the inclusion relationships between the data stored in the index and the data in the relation:

$$(IDX^f) \quad \forall(k \in \underline{\mathsf{dom}}\,\mathtt{I})\ \forall(t \in \mathtt{I}\,[\,k\,])\ \exists(r \in \mathtt{R}) r.\mathtt{A} = k.\mathtt{A}\ \underline{\mathsf{and}}\ r.\mathtt{B} = k.\mathtt{B}\ \underline{\mathsf{and}}\ r.\mathtt{C} = k.\mathtt{C}\ \underline{\mathsf{and}}\ r = t$$
$$(IDX^b) \quad \forall(r \in \mathtt{R})\ \exists(k \in \underline{\mathsf{dom}}\,\mathtt{I})\ \exists(t \in \mathtt{I}\,[\,k\,])\ r.\mathtt{A} = k.\mathtt{A}\ \underline{\mathsf{and}}\ r.\mathtt{B} = k.\mathtt{B}\ \underline{\mathsf{and}}\ r.\mathtt{C} = k.\mathtt{C}\ \underline{\mathsf{and}}\ r = t$$

We use a pair of inclusion constraints of exactly the same shape to represent the materialized view $\mathtt{V}_1$ from example 2.2:

$$(V_1^f) \quad \forall(r_1 \in \mathtt{R}_1) \quad \forall(s_{11} \in \mathtt{S}_{11})\ \forall(s_{12} \in \mathtt{S}_{12}) r.\mathtt{A}_1 = s_{11}.\mathtt{A}\ \underline{\mathsf{and}}\ r.\mathtt{A}_2 = s_{12}.\mathtt{A} \\ \Rightarrow\ \exists(v_1 \in \mathtt{V}_1) v_1.\mathtt{K} = r.\mathtt{K}\ \underline{\mathsf{and}}\ v_1.\mathtt{B}_1 = s_{11}.\mathtt{B}\ \underline{\mathsf{and}}\ v_1.\mathtt{B}_2 = s_{12}.\mathtt{B}$$

$$(V_1^b) \quad \forall(v_1 \in \mathtt{V}_1) \quad \exists(r \in \mathtt{R}_1)\ \exists(s_{11} \in \mathtt{S}_{11})\ \exists(s_{12} \in \mathtt{S}_{12}) \\ r.\mathtt{A}_1 = s_{11}.\mathtt{A}\ \underline{\mathsf{and}}\ r.\mathtt{A}_2 = s_{12}.\mathtt{A}\ \underline{\mathsf{and}} \\ v_1.\mathtt{K} = r.\mathtt{K}\ \underline{\mathsf{and}}\ v_1.\mathtt{B}_1 = s_{11}.\mathtt{B}\ \underline{\mathsf{and}}\ v_1.\mathtt{B}_2 = s_{12}.\mathtt{B}$$

Join indexes, access support relations and GMAPs are captured in a similar way ([11]).

The algorithm has two main phases: the first one, called the *chase*, introduces all physical structures in the

implementation that are relevant for Q together with all logical schema elements that are related via semantic constraints to Q. It does so by rewriting Q to a *universal plan* U that explicitly uses them. The second phase, that we call the *backchase* searches for a minimal plan for Q among the subqueries of U.

Phase 1: chase. Given a constraint of the form [11]

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

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

select	$O(\vec{r})$	$\mapsto$	select	$O(\vec{r})$
from	$\ldots, R_1\ r_1, \ldots, R_m\ r_m, \ldots$		from	$\ldots, R_1\ r_1, \ldots, R_m\ r_m,\ S_1\ s_1, \ldots, S_n\ s_n, \ldots$
where	$\cdots$ and B_1 and $\cdots$		where	$\cdots$ and B_1 and B_2 and $\cdots$

Example. By chasing query Q from example 2.2 with constraint (V_1^f), we obtain

(Q_c)
select struct($B_{11} : s_{11}.B,\ B_{12} : s_{12}.B,\ B_{21} : s_{21}.B,\ B_{22} : s_{22}.B$)
from $R_1\ r_1,\ S_{11}\ s_{11},\ S_{12}\ s_{12},\ R_2\ r_2,\ S_{21}\ s_{21},\ S_{22}\ s_{22},\ V_1\ v_1$
where $r_1.F = r_2.K$ and
$r_1.A_1 = s_{11}.A_1$ and $r_1.A_2 = s_{12}.A_2$ and
$r_2.A_1 = s_{21}.A_1$ and $r_2.A_2 = s_{22}.A_2$ and
$v_1.K = r_1.K$ and $v_1.B_1 = s_{11}.B$ and $v_1.B_2 = s_{12}.B$

Note that (V_1^b) does not apply to chasing Q and the only additionally applicable constraint is (V_2^f). Chasing with it, we reach the universal plan U obtained from Q_c by adding a new binding $V_2\ v_2$ in the from clause and the new conditions $v_2.K = r_2.K$ and $v_2.B_1 = s_{21}.B$ and $v_2.B_2 = s_{22}.B$ in the where clause.

A few remarks are in order with respect to the chase stage. First, the simplified chase step is defined when there is a one-to-one mapping from the universally quantified variables of the constraint into the variables in the from clause of the query, and when there is an exact match between the B_1 condition of the constraint and the one in the where clause. In general, we chase a query Q with a constraint c when there is a *homomorphism* from c into Q

In general a *homomorphism* from a query Q_1 into a query Q_2 is a mapping from the variables of Q_1 into the variables of Q_2 such that, when extended in the natural way to paths, it obeys the following conditions:

1) any binding $P\ x$ in Q_1 corresponds to a binding $P'\ h(x)$ in Q_2 such that either $h(P)$ and P' are the same expression[12] or the equality $h(P) = P'$ follows from the where clause of Q_2.

2) for every equality $P_1 = P_2$ that occurs in the where clause of Q_1 either $h(P_1)$ and $h(P_2)$ are the same expression or the equality $h(P_1) = h(P_2)$ follows from the where clause of Q_2.

The definition of homomorphism that we give here does not take into account the output paths in the select clauses of the queries, involving only the from and where clauses. Hence the same definition can be applied to *homomorphisms from constraints into queries.* In that case the universally quantified prefix $\vec{x} \in \vec{P}$ of the constraint plays the role of the from clause of a query while the condition $B_1(\vec{x})$ of the constraint plays the role of the where clause of the query.

The problem of finding a homomorphism is known to be NP-complete in the size (number of variables) of the constraint (which are very small in practice, no larger than 3 in the examples throughout this paper). Second, [27] shows that for the class of queries and constraints we consider , the size of the universal plan is polynomial (liniar in our example) in the size of the original query and the number of constraints.

[11] Note that all our constraints are of this form.

[12] In the relational conjunctive queries this translates to the fact that any of Q_1's goals must be mapped into one of Q_2's goals with the same relation name.

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

select $O(\vec{x}, y)$ from $R_1\ x_1, \ldots, R_m\ x_m,\ R\ y$ where $C(\vec{x}, y)$	$\mapsto$	select $O'(\vec{x})$ from $R_1\ x_1, \ldots, R_m\ x_m$ where $C'(\vec{x})$

provided that: (1) the conditions C' are implied by C, (2) the equality of O and O' is implied by C, and (3) $D \cup D'$ implies $(\delta)\quad \forall(x_1 \in R_1)\ldots\forall(x_m \in R_m)[\ C'(\vec{x})\ \Rightarrow\ \exists(y \in R)\ C(\vec{x}, y)\ ]$

The purpose of a backchase step is to eliminate (if possible) a *binding* $R\ y$ from the from clause of the query. For any two queries Q and Q' as above such that conditions (1) and (2) are satisfied, we say that Q' is a *subquery* of Q. [27] gives a procedure for computing O' and C'. While the first two conditions ensure that the backchase reduces a query to a subquery of it, condition (3) guarantees that it reduces it to an *equivalent* subquery. This is true because its reverse is just the chase step with constraint (δ) (hence the name "backchase")followed by a simplification given by (1) and a replacement of equals given by (2). Sometimes the backchase can apply just by virtue of constraints (δ) that hold in all instances (so-called *trivial* constraints). Relational tableau minimization [1] is precisely such a backchase.

(Backchase-)Minimal queries We call a subquery Q_1 of Q_2 a *strict* subquery if Q_1 has strictly fewer bindings than Q_2. We say that a query Q is **minimal** if there does not exist a strict subquery Q' of Q such that Q' is equivalent to Q. In other words, we cannot remove any bindings from Q without losing equivalence. (It turns out that this is a generalization of the minimality notion of [23].) In general, we can think of the backchase as minimization for a larger (than just relational tableaux) class of queries, and under constraints. Checkig that (δ) of condition (3) above is implied by the existing constraints is actually done using the chase presented above when constraints are viewed as boolean-valued queries [27].

B OQF - Formal Details

Query Fragments. Given a query Q as above, we define its *closure* as a query Q^* that has the same select and from clauses as Q while the where clause consists of all the equalities that occur in or are implied by the Q's where clause. Q^* is computable from Q in PTIME and is equivalent to Q ([26] shows a congruence/transitive closure based 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, and
3) The select clause consists of all the paths P over S that occur in the select clause of Q or in an equality $P = P'$ of Q^*'s where clause where P' depends on at least one binding that is not in S. In the latter case, we call such P a *link path* of the fragment.

Example B.1 Recalling example 2.2 the query fragment of Q induced by $S = \{\mathtt{R1}\ r_1,\ \mathtt{S}_{11}\ s_{11},\ \mathtt{S}_{12}\ s_{12}\}$ is the query:

select struct($\mathtt{B}_{11} = s_{11}.\mathtt{B},\ \mathtt{B}_{12} = s_{12}.\mathtt{B},\ L_{\{r_1.\mathtt{F}, r_2.\mathtt{K}\}} = r_1.\mathtt{F}$)
from $\mathtt{R}_1\ r_1,\ \mathtt{S}_{11}\ s_{11},\ \mathtt{S}_{12}\ s_{12}$
where $r_1.A_1 = s_{11}.A_1$ and $r_1.A_2 = s_{12}.A_2$

Notice that $r_1.\mathtt{F}$ must occur in the select clause because it appears in an equality condition in Q with a path ($r_2.\mathtt{K}$) outside of the fragment (condition 3) above). Also $s_{11}.\mathtt{B}$ and $s_{12}.\mathtt{B}$ must occur in the select clause by

condition 3 above. Essentially condition 3) will allow us to recover later a query from its query fragments by joining the fragments on the corresponding link paths and therefore we will be able to find a plan for the query by joining plans for the fragments. The label $L_{\{r_1.\mathtt{F},r_2.\mathtt{K}\}}$ for the link path $r_1.\mathtt{F}$ is generated so that it uniquely identifies the corresponding join condition.

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. As seen in section A, each of these can be described by a pair of complementary inclusion constraints.

We define a skeleton as a pair of complementary constraints:

$$d = \forall(\vec{x} \in \vec{\mathtt{R}})\ [\ B_1(\vec{x})\ \Rightarrow\ \exists(\vec{v} \in \vec{\mathtt{V}})\ B_2(\vec{x},\vec{v})\ ] \qquad\qquad d^- = \forall(\vec{v} \in \vec{\mathtt{V}})\ \exists(\vec{x} \in \vec{\mathtt{R}})\ B_1(\vec{x})\ \underline{\mathsf{and}}\ B_2(\vec{x},\vec{v})$$

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. Note that while materialized views and primary indexes are described precisely by skeletons, secondary indexes require an additional non-emptiness constraint (see [11]).

Algorithm B.1 (Decomposition into Fragments.) Given a query Q and a set of skeletons $\mathcal{V}$:

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

Step 2: Compute the connected components $\{C_1,\ldots,C_k\}$ of the interaction graph.

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

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

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

C Off-line Constraint Stratification - Formal Details

Algorithm C.1 (Stratification of Constraints.) Given a schema with constraints, do:

Step 1: Construct an *interaction graph* as follows:
1) there is a node labeled c for every constraint c in the schema.
2) there is an edge between nodes c_1 and c_2 whenever there is a homomorphism c_1 into the tableau of c_2, or 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: $T(c) = \forall(\vec{u} \in \vec{U})\ \forall(\vec{e} \in \vec{E})\ B_1(\vec{u}) \wedge B_2(\vec{u},\vec{e})$.

Step 2: Compute the connected components $\{C_1,\ldots,C_k\}$ of the interaction graph. Each C_i corresponds to a constraint stratum.

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