Query Evaluation using Overlapping Views: Completeness and Efficiency

Gang Gou, Maxim Kormilitsin, Rada Chirkova

Computer Science Department, North Carolina State University
Campus Box 8206, Raleigh, NC 27695-8206
email: {ggou,mvkormil,rychirko}@ncsu.edu

ABSTRACT

We study the problem of finding efficient equivalent view-based rewritings of relational queries, focusing on query optimization using materialized views under the assumption that base relations cannot contain duplicate tuples. To enable consideration of such rewritings in a cost-based query optimizer, we develop local conditions under which two views can be combined in an efficient query rewriting, under bag-set and set semantics for query evaluation. For each semantics and for SQL select-project-join queries and views, possibly involving grouping and aggregation, we propose efficient sound and complete algorithms for testing equivalence of a query to a rewriting (the algorithms are complete with respect to the language of rewritings). Our results apply not only to query optimization, but to all areas where the goal is to obtain efficient equivalent view-based query rewritings. Based on these results, for the problem of finding an efficient execution plan for a user query in terms of materialized views we propose sound algorithms that extend the cost-based query-optimization approach of System R [19]. We present a study of the completeness-efficiency tradeoff in the algorithms, and provide experimental results that show the viability of our approach and test the limits of query optimization using overlapping views.

1. INTRODUCTION

We study the problem of finding efficient equivalent rewritings of relational queries using views. The problem has received significant attention because of its applications in a number of data-management problems [15], such as query optimization [6, 17, 24], maintenance of physical data independence [22, 23], and data warehousing [14, 21]. Our paper focuses in particular on one application, query optimization in presence of materialized views. Given a user query, the task of a query optimizer is to search the space of all physical query plans for an optimal plan. Besides being correct, the process should be efficient, cost based, and as complete as possible. Traditional optimizers such as the System-R optimizer [19] search in the space of left-deep join trees of a logical plan for an optimal physical plan, which specifies execution details such as join ordering and implementation of a join (e.g., hash join or sort-merge join). As shown in [6], a System-R-style query optimizer can be extended to consider logical plans that stem from various rewritings of the query using materialized views; the optimizer of [6] can choose in a cost-based and syntax-independent way between a materialized view and a view expansion. The optimization algorithm of [6] produces only valid view-based query rewritings (i.e., is *sound*) for SQL select-project-join (SPJ) queries and views with inequality comparisons and without grouping or aggregation, and produces all such valid rewritings (i.e., is *complete*) for SPJ queries and views without inequality comparisons, under the assumption that both base relations and query answers can have multiple identical tuples — that is, under *bag semantics* for query evaluation [7].

We continue the line of work of [6] by studying query optimization using views in a System-R-style optimizer [19], for SPJ queries and views that may involve grouping and aggregation. We make an important assumption that base relations cannot contain duplicate tuples. In this setting there may exist efficient equivalent rewritings of a given query using given views such that these rewritings cannot be obtained using the approach of [6]. We distinguish between two possible scenarios:

- *bag-set semantics* for query evaluation [7], where duplicate tuples are retained in query answers, and
- *set semantics*, where duplicate tuples are eliminated from query answers.

Note that evaluation under bag-set semantics is essential for aggregate SQL queries involving aggregation functions `SUM` or `COUNT` [1, 8], whereas, for instance, queries with the `DISTINCT` keyword must be evaluated under set semantics.

To enable consideration of such additional rewritings in a cost-based query optimizer, we focus on local conditions under which two materialized views can be combined in an efficient query rewriting; our work builds on the contributions of [2, 18]. The challenge of the problem is that, even with the approach proposed in [2], there are cases where a straightforward algorithm to combine views may perform poorly. The theoretical results we present in this paper apply not only to query optimization, but to all areas where

the goal is to obtain equivalent view-based query rewritings. Consider an example that highlights the novelty and focus of our approach.

EXAMPLE 1. *Consider a star-schema [5] data warehouse with five base relations (key attributes are underlined):*

```
Sales (saleID, custID, prodID, saleMonth, price)
Customers (custID, custType, nationID)
Products (prodID, supID)
Suppliers (supID, supRegion)
Nations (nationID, nationName)
```

Here, `Sales` *is the fact table, and the remaining tables are dimension tables. We make the following realistic assumptions about the contents of these relations. First, each of the first three relations stores a large number of tuples (e.g., hundreds of millions for* `Sales`*, tens of thousands for* `Customers`*, etc). Second, customers make frequent repeat purchases of the same multiple types of products; as a result, many specific combinations of the values of* `cust ID`*,* `prodID`*, and* `saleMonth` *occur each in a large number of tuples in the* `Sales` *table. Third, on average* `Customers` *stores a large number of customer IDs per customer type, and* `Products` *— a large number of product IDs per supplier. Finally, customer types are nation specific.[1] While our approach is applicable in presence of indexes (cf. [6]), in this example we assume that there are no indexes on the base relations.*

Suppose a user is interested in the maximal product price per customer type per product-supplier region per month, starting in January 2005; the answer should include information about the customers' nations. This query `Q` *can be expressed in SQL as follows:*

```
Q: SELECT custType, supRegion, saleMonth, nationName,
MAX(price)
FROM Sales S, Customers C, Products P, Suppliers SP,
Nations N
WHERE S.custID = C.custID AND S.prodID = P.prodID
AND C.nationID = N.nationID AND P.supID = SP.supID
AND saleMonth >= '2005-01'
GROUP BY custType, supRegion, saleMonth, nationName;
```

Suppose the data warehouse maintains materialized aggregate views `V` *and* `W`*. The view* `V` *is defined on* `Sales, Customers,` *and* `Nations` *and returns maximal product price per customer type per product ID per month, including information about the customers' nations. The view* `W`*, defined on* `Sales, Products,` *and* `Suppliers,` *returns maximal product price per customer ID per product ID per supplier region per month. Under our assumptions on the base data, the size of the relation for each of* `V` *and* `W` *would be much smaller than the size of the* `Sales` *relation. (Note that the values of* `nationName` *and* `supRegion` *are functionally determined by the values of* `custType` *and* `prodID`*, respectively.)*

It is straightforward to find a plan for answering the user query `Q` *exactly (i.e., using an equivalent rewriting) by using either materialized view in combination with the "missing" dimension tables. That is, the view* `V` *can be combined with tables* `Products` *and* `Suppliers` *into a rewriting* `Q(V)`*, and the view* `W` *— with tables* `Customers` *and* `Nations` *into a rewriting* `Q(W)`*. In fact, some commercial query optimizers, as well as extensions to query optimizers proposed in the literature (see, e.g., [3, 12, 20, 24]) would find both rewritings. Each of* `Q(V)` *and* `Q(W)` *would be worth considering in a cost-based query optimizer, as either rewriting would reduce significantly the evaluation costs of* `Q` *on data warehouses that satisfy our assumptions.*

We now show that we can use the two views `V` *and* `W` together *to answer the user query* `Q`*, and argue that the reduction in the evaluation costs for* `Q` *in this case is significant under our assumptions — in particular, it is at least on par with the reduction in costs obtained in each of* `Q(V)` *and* `Q(W)`*. Consider this rewriting* `R` *of the query* `Q` *using the two views:*

```
R: SELECT custType, supRegion, V.saleMonth, nationName,
MAX(vMaxPrice) FROM V, W
WHERE V.prodID = W.prodID AND V.saleMonth = W.saleMonth
AND V.saleMonth >= '2005-01'
GROUP BY custType, supRegion, V.saleMonth, nationName;
```

Here, `vMaxPrice` *is the aggregated attribute of the view* `V`*. It can be shown [1, 8] that* `R` *is an equivalent rewriting of the query* `Q`*. At the same time, under our assumptions on the base data,* `R` *would result in a more efficient evaluation of* `Q` *than either* `Q(V)` *or* `Q(W)`*, provided that the size of* `Products` *(*`Customers`*, respectively) is comparable to the size of the view* `V` *(*`W`*, respectively). Thus, it would be desirable to consider the rewriting* `R` *in a cost-based optimizer.*

We use this example to make two observations. First, `R` would not be an equivalent rewriting of the query `Q` if the view `W` did not have `prodID` among its grouping attributes, even though each of `Q(V)` and `Q(W)` would still be an equivalent rewriting of `Q`. This observation leads us to consider *local conditions* for combining a pair of views in an equivalent rewriting of a given query. Second, if we modify each of `Q`, `V`, `W`, and `R` to be a `SUM`- rather than `MAX`-query (we give an extended example in the full version of the paper), the modified rewriting `R'` will not be an equivalent rewriting of the modified query `Q'`, as the unaggregated SPJ *core* of `R` is not equivalent to the core of `Q` under bag-set semantics for query evaluation [1, 8]. (The cores of `Q` and `Q'` are the same, as are those of `R` and `R'`.) This observation leads us to consider construction of efficient view-based query rewritings separately under set semantics (for, e.g., `MAX`- or `MIN`-queries or for queries with the `DISTINCT` keyword) and under bag-set semantics [7] (for, e.g., `SUM`- or `COUNT`-queries).

Our contributions: To enable consideration of all efficient view-based rewritings of a relational query when base relations contain no duplicate tuples, for each of bag-set and set semantics for query evaluation we develop efficient local conditions under which two views can be combined in a query rewriting, for SPJ queries and views with or without aggregation. For each of bag-set and set semantics we propose an efficient sound and complete algorithm for testing equivalence of queries and rewritings; the algorithm for the set-semantics case builds on results of [2, 18]. (Our algorithms are complete with respect to the language of rewritings — SPJ rewritings for SPJ queries, and central rewritings [1] for queries with aggregation.) Areas where our algorithms are applicable include query optimization, maintenance of physical data independence, and data warehousing.

We propose sound dynamic-programming algorithms based on our local conditions, for finding efficient execution plans

[1]None of these functional dependencies is needed for the correctness of the rewritings in the example.

for user queries using indexes and materialized views; queries and views may have grouping and aggregation. Our approach is a relatively simple generalization of the cost-based query-optimization algorithm of [6], which works in an expanded search space of rewritings. While exploring strictly more rewritings than the approach of [6], in general our algorithms for bag-set and set semantics are incomplete; we present a theoretical study of the completeness-efficiency tradeoff and describe enhancements that would make the algorithms complete. Finally, we provide experimental results that show good efficiency and scalability of one of our algorithms for bag-set and set semantics and test the limits of query optimization using overlapping views.

We start by giving the background and our problem statement in Section 2. We use Datalog for concise notation, but all our results hold for SPJ queries, views, and rewritings with equality comparisons and possibly with the `DISTINCT` keyword or with aggregation `SUM`, `COUNT`, `MAX`, or `MIN`. Sections 3 and 4 introduce (1) our efficient local conditions under which two views can be combined in a rewriting, and (2) our equivalence-testing algorithms. We describe our optimization algorithms in Section 5, and present experimental results in Section 6. Section 7 discusses related work; we conclude in Section 8. Due to space constraints, we are unable to provide full proofs of our theoretical results in this paper; we provide intuition for the results wherever possible.

2. PRELIMINARIES

In this section we review bag, bag-set, and set semantics for query evaluation, discuss some concepts in answering queries using views, give a formal statement of the problem considered in this paper, and discuss some relevant past work.

2.1 Equivalent rewritings and views

We consider SQL SPJ queries with equality comparisons, a.k.a. safe *conjunctive queries* (CQs), possibly with grouping and aggregation, posed on stored (base) relations in a relational database. A relation can be either a set or a *bag;* a bag can be thought of as a set of elements with multiplicities attached to each element. A database is *set-valued* if all its base relations are sets, and is *bag-valued* otherwise.

Two CQ queries Q_1 and Q_2 are *set-equivalent* (*bag-set-equivalent*, *bag-equivalent*, respectively), denoted by $Q_1 \equiv_s Q_2$ ($Q_1 \equiv_{bs} Q_2$, $Q_1 \equiv_b Q_2$, respectively), if they produce the same set (or bag, respectively) of answers on every database (every set-valued database for the first two cases, every bag-valued database for the third case). Chandra and Merlin [4] show that under set semantics, a CQ Q_1 is contained in a CQ Q_2, $Q_1 \sqsubseteq_s Q_2$, if and only if there exists a *containment mapping,* which maps the head and all the subgoals of Q_2 to Q_1; the mapping maps each variable to a single variable or constant, and each constant to the same constant. Q_1 and Q_2 are equivalent under set semantics if and only if $Q_1 \sqsubseteq_s Q_2$ and $Q_2 \sqsubseteq_s Q_1$. For bag and bag-set semantics, the following conditions hold for CQ query equivalence [7]:

THEOREM 1. *Let Q_1 and Q_2 be conjunctive queries. (1) Q_1 and Q_2 are equivalent under* bag *semantics if and only if Q_1 and Q_2 are isomorphic. (2) Q_1 and Q_2 are equivalent under* bag-set *semantics if and only if Q'_1 and Q'_2 are equivalent under* bag *semantics, where Q'_1 and Q'_2 are canonical representations of Q_1 and Q_2 respectively.*[2]

A *view* refers to a named query. A view V is said to be *materialized* in a database D if its answer $V(D)$ is stored in the database; $D_{\{V_1,\ldots,V_n\}}$ is a database D with added relations for the views $V_1, \ldots, V_n$. A rewriting R of a query Q on a set of views $\mathcal{V}$ is an *equivalent rewriting* under set semantics if for every set-valued database D, $R(D_{\mathcal{V}}) = Q(D)$, that is, $R(D_{\mathcal{V}})$ and $Q(D)$ are the same as sets. The definitions for the bag-set and bag semantics are analogous. In the rest of the paper, unless otherwise noted, the term "rewriting" means an "equivalent rewriting" of a query using views, under the semantics specified either explicitly or by the context. We also assume that the definitions of queries and views under consideration are minimized [4].

For a rewriting R of a query Q using views, we obtain the *expansion* R^{exp} of R by replacing all view atoms in the body of R by their definitions in terms of base relations. We now give necessary and sufficient conditions for CQ query-rewriting equivalence under each of the three semantics:

THEOREM 2. *For a CQ query Q and a set of CQ views $\mathcal{V}$, let R be a CQ rewriting of Q using $\mathcal{V}$. Then (1) $Q \equiv_s R$ iff $R^{exp} \equiv_s Q$ on all set-valued databases D; (2) $Q \equiv_b R$ iff $R^{exp} \equiv_b Q$ on all bag-valued databases D; and (3) $Q \equiv_{bs} R$ iff $R^{exp} \equiv_b Q$ on all set-valued databases D.*

We consider CQ queries and views with or without aggregation, and assume that all aggregate queries and views are unnested and use aggregation functions `COUNT`, `SUM`, `MAX`, or `MIN`. We denote such queries and views CQA; CQ(A) stands for CQ queries that may or may not involve aggregation. We consider only central rewritings [1] of CQA queries. Intuitively, these are rewritings where the unaggregated *core* is a CQ query, and only one view contributes to computing the aggregated query output. For instance, rewriting `R` in Example 1 is a central rewriting of the query Q. Equivalence tests for a CQA query Q and a central rewriting R are [1, 8] by reduction to (1) *bag-set* equivalence of the unaggregated cores of Q and R^{exp} for `SUM` and `COUNT` queries, and to (2) *set* equivalence of the cores for `MAX` and `MIN` queries. In our analysis, we chose central rewritings for their simplicity — compare these with rewritings in [8] where, in general, the aggregate term in the rewriting is a product of aggregate terms from all constituent views. However, as there is a natural relationship between some classes of rewritings in [8] and in [1], our results also apply to rewritings of [8].

We now show that given a CQ query and a set of CQ views, in general the search space of all equivalent rewritings of the query using the views depends on whether the base relations or query answer may contain duplicate tuples.

EXAMPLE 2. *Consider a database D with three base relations, $P(A, B)$, $S(B, C, F)$, and $T(F, G)$. Let a CQ query Q be a natural join of P, S, T that returns the values of all attributes of P and T. Let the database include three materialized views, U, V, and W; each view is a natural join of two of the base relations. For instance, U is a join of P and S that returns the values of attributes A, B, and F. Here are the definitions of the query and views in Datalog:*

[2] Q' is a *canonical representation* of a query Q if Q' is obtained by removing all duplicate literals from Q.

$$
\begin{array}{ll}
q(X_1, X_2, X_3, X_4) & \leftarrow p(X_1, X_2),\ s(X_2, X_5, X_3),\ t(X_3, X_4). \\
u(Y_1, Y_2, Y_3) & \leftarrow p(Y_1, Y_2),\ s(Y_2, Z, Y_3). \\
v(Y_1, Y_2, Y_3, Y_4) & \leftarrow p(Y_1, Y_2),\ s(Y_2, Y_4, Y_3). \\
w(Y_2, Y_3, Y_4, Y_5) & \leftarrow s(Y_2, Y_4, Y_3),\ t(Y_3, Y_5).
\end{array}
$$

We can rewrite the query Q as a query R_1 that joins the view U with the base relation T; R_1 is an equivalent rewriting of Q under bag semantics. In addition, $R_1 \equiv Q$ under bag-set and set semantics, because each semantics is a restriction on the bag-semantics setting. Under bag-set semantics, in addition to R_1, Q has another rewriting R_2 — a natural join of views V and W; $R_2 \equiv Q$ under bag-set but not under bag semantics. (R_2 is also equivalent to Q under set semantics, because it is a restriction on the bag-set-semantics setting.) Finally, under set semantics Q has an additional rewriting R_3, which is a natural join of views U and W. R_3 is not equivalent to Q under bag or bag-set semantics.

In the full version of the paper we spell out assumptions on the base relations and indexes in the database D, under which (1) evaluating R_1 is cheaper than evaluating Q without views, (2) under bag-set semantics, evaluating R_2 is cheaper than evaluating R_1, and (3) under set semantics, evaluating R_3 is cheaper than evaluating either R_1 or R_2.

Given that bag-set semantics is a restriction on bag semantics, to find an optimal view-based rewriting of a given query under bag-set semantics we need to consider at least all the equivalent rewritings for the bag-semantics case. A similar relationship holds between bag-set and set semantics. At the same time, as shown in Example 2, given a CQ query and a set of CQ views, the search space of equivalent rewritings of the query that use the views can be strictly larger (1) under bag-set semantics than under bag semantics, and (2) under set semantics than under bag-set semantics.

2.2 Our problem statement

We now introduce feasible, efficient, and optimal rewritings, give a formal statement of our problem, and discuss our assumptions. For a fixed database schema $\mathcal{S}$ and given a CQ(A) query Q on $\mathcal{S}$ and a set of CQ(A) views $\mathcal{V}$ defined on $\mathcal{S}$, a CQ(A) query R is a *feasible rewriting of* Q for set, bag-set, or bag semantics X if (1) R is defined in terms of $\mathcal{V}$, and (2) R is equivalent to Q under the semantics X, $R \equiv_X Q$. Efficient rewritings are defined as follows: Given a database D with schema $\mathcal{S}$, a CQ(A) query Q on D, and a set of CQ(A) views $\mathcal{V}$ defined on $\mathcal{S}$, a CQ(A) query R is an *efficient rewriting of* Q for set, bag-set, or bag semantics X and given a cost model M, if (1) R is a feasible rewriting of Q, and (2) the cost of evaluating R on D using the cost model M, $C_M(R, D)$, is lower than the cost $C_M(Q, D)$ of evaluating the query Q in its original formulation. A feasible rewriting R is *optimal* if $C_M(R, D)$ is minimal among all $C_M(R', D)$ where R' is a feasible rewriting of Q for $\mathcal{V}$, X. The cost $C_M(F, D)$ of evaluating a query Q on database D using some formulation F of Q (e.g., a rewriting using views) is the cost of computing the answer to Q using a lowest-cost query plan for F on D, for a fixed cost model M.

Our *problem statement* is as follows: Produce an optimal CQ(A) rewriting of a minimized [4] CQ(A) query Q on a database D using minimized CQ(A) views $\mathcal{V}$, for set, bag-set, or bag semantics X and given a cost model M.

All theoretical results in this paper hold under *monotonic cost models* [2]. Intuitively, a cost model is monotonic if replacing a relation by a smaller relation in — or removing a redundant subgoal from — a query plan never results in higher execution costs; many cost models proposed for and used in query optimization are monotonic.

Our results in this paper hold under the following additional assumptions. We consider CQ(A) queries, views, and rewritings of the form described in Section 2.1. Further, we allow rewritings that use base relations alongside views (see, e.g., rewritings `Q(V)` and `Q(W)` in Example 1 and rewriting R_1 in Example 2). As the database schema is not part of our problem input, in proving our complexity results we assume that the number of attributes in each base relation is a constant. For the cases of bag-set and set semantics, we assume that each rewriting R that is equivalent to a given query Q contains no filtering views [2, 17, 18], that is, does not contain views whose removal does not result in loss of equivalence of R to Q. Finally, we assume that a query optimizer considers only left-deep join trees for query plans [6, 11, 19].

2.3 View tuples and efficient rewritings

In this paper we use the results of [2] on restricting the search space of efficient rewritings under set semantics. We first give the definition of a view tuple [2, 17]. Given a query Q, a *canonical database* D_Q of Q is obtained by turning each subgoal into a tuple by replacing each variable in the body of Q by a distinct constant, and treating the resulting subgoals as the only tuples in D_Q. Let $\mathcal{V}(D_Q)$ be the result of applying the view definitions $\mathcal{V}$ on D_Q. For each tuple in $\mathcal{V}(D_Q)$, we restore each new constant back to the original variable of Q, and obtain a *view tuple* of each view with respect to the query. By definition, each variable in each view tuple of Q occurs in the definition of Q. Let $\mathcal{T}(D_Q)$ denote the set of all view tuples after the replacement.

LEMMA 3.2 of [2]. *Under set semantics for any equivalent CQ rewriting $R : r(\bar{X}) \leftarrow p_1(\bar{Y}_1), \ldots, p_k(\bar{Y}_k)$ of a CQ query Q using CQ views $\mathcal{V}$, there is a CQ rewriting $R' \equiv_s Q$, such that R' is in the form $r'(\bar{X}) \leftarrow p_1(\bar{Y}'_1), \ldots, p_k(\bar{Y}'_k)$. In addition, each $p_i(\bar{Y}'_i)$ is a view tuple in $\mathcal{T}(D_Q)$, and $R' \sqsubseteq R$.*

[2] gives examples, for the set-semantics case, of equivalent rewritings whose subgoals are not view tuples.

THEOREM 3. *Given a database D, a CQ(A) query Q, and a set of CQ(A) views $\mathcal{V}$, under set, bag-set, or bag semantics X and using a monotonic cost model M, consider the set $\mathcal{R}$ of all feasible CQ(A) rewritings R of Q in terms of $\mathcal{V}$. Let $\mathcal{R}_{opt} \subseteq \mathcal{R}$ be the set of optimal CQ(A) rewritings of Q on D using the cost model M. Then for at least one rewriting $R_{opt} \in \mathcal{R}_{opt}$, each subgoal of R_{opt} is a view tuple in $\mathcal{T}(D_Q)$.*

Theorem 3 holds for CQ queries, views, and rewritings both with and without aggregation. The proof of Theorem 3 is immediate from the definition of monotonic cost models (and, for set semantics, from Theorem 5.1 in [2]), after we make the following observation for the case of bag or bag-set semantics [1, 7, 8]. Suppose that R is a CQ(A) rewriting of a query Q using views $\mathcal{V}$ under bag or bag-set semantics, such that at least one subgoal of R is not a view tuple in $\mathcal{T}(D_Q)$. Then R is not equivalent to Q under the given semantics. In

the remainder of the paper we consider only rewritings that consist entirely of view tuples (in a broader sense than that of [2], i.e., we also consider view tuples for base relations), for a given query, views, and base relations.

2.4 Some additional terminology

We now describe some terminology that we will need to present our approach. First, we extend the notion of tuple core of [2] by defining "tuple coverage." Intuitively, given a query and a view tuple, a tuple coverage of the view tuple is some set of query subgoals that can be "covered" by the view tuple in some rewriting of the query. Formally, given a CQ query Q, a CQ rewriting $R \equiv_s Q$, and a CQ view V in R, let $t_V \in \mathcal{T}(D_Q)$ be V's view tuple for Q. Then a *tuple coverage* $s(t_V, Q)$ of t_V is a nonempty set G of subgoals of Q, such that (1) G is isomorphic to some set of subgoals in the expansion t_V^{exp} of t_V, and (2) each variable Y of G is a head variable of t_V whenever either Y is a head variable of the query Q or Y is used in a subgoal $p_i(\bar{Y}_i)$ of Q such that $p_i(\bar{Y}_i)$ is not in G. [2] introduced the notion of *tuple core* $s_{max}(t_V, Q)$ of a view tuple t_V for query Q: $s_{max}(t_V, Q)$ is a tuple coverage that uses a maximal set G of subgoals of Q. A tuple core is unique for a given pair (Q, t_V) [2].

The notions of tuple core and tuple coverage extend naturally to CQA queries and views, provided that central rewritings [1] are used. We will see in Section 4 how our approach decides on whether it is possible to combine two view tuples t_V and t_W in a rewriting that is equivalent to the given query Q under set semantics; the idea of the approach is to examine tuple coverages $s(t_V, Q)$ of t_V and $s(t_W, Q)$ of t_W, where $s(t_V, Q)$ and $s(t_W, Q)$ are not necessarily full tuple cores of the respective view tuples. At the same time, under bag-set and bag semantics the only tuple coverages that matter in equivalent rewritings are full tuple cores.

We now restate the bag-semantics part of Theorem 2 using the notion of tuple core, and extend the theorem to aggregate queries, views, and rewritings under bag semantics:

PROPOSITION 1. *Given a CQ(A) query Q on database with schema $\mathcal{S}$, and given a CQ(A) rewriting R in terms of CQ(A) views $V_1, \ldots, V_m$ defined on $\mathcal{S}$. Let $s_1, \ldots, s_m$ be the tuple cores of all the view tuples in the rewriting R. Then Q and R are equivalent under bag semantics if and only if (1) all the s_i's are pairwise disjoint, and (2) the union of all the s_i's is the set of all subgoals of the query Q.*

2.5 Query optimization under bag semantics

The cost-based query-optimization algorithm of [19] uses dynamic programming to find optimal query plans. The approach of [6] (we call it "the CKPS approach") extends the algorithm of [19] by including view-based query plans into the search space of the optimizer. The CKPS approach uses a preprocessing stage, which determines for each view or base relation V whether V is usable in answering the query and, if so, which subgoals of the query V "covers." That is, for each view (or base relation) V which is usable for the query, the preprocessing stage returns all view tuples h_j of V and the respective *nonempty* tuple cores D_j. The output of this stage is a mapping table called *MapTable*. The main stage of the CKPS approach is given as Algorithm 1. Intuitively, the algorithm considers each set of query subgoals as a subproblem (we call it "cell") in dynamic programming; while the algorithm of [19] finds for each cell an optimal plan in terms of base relations only, the CKPS algorithm also checks whether each cell is "exactly covered" by (i.e., whether each cell is the tuple core of) any view tuple returned by the preprocessing stage and, if so, estimates the cost of evaluating the cell using each relevant view tuple. Then the algorithm chooses for the cell one optimal plan from all available (including view-based) plans.

Algorithm 1: Query-optimization algorithm CKPS of [6]

```
Input  : query Q, table MapTable, cost model M
Output: optimal execution plan for Q w.r.t. M
begin
    for each cell D_i of PlanTable in increasing size do
        for each (D_j, h_j) in MapTable do
            if cells D_i and D_j are disjoint then
                AddPlan(D_i ∪ D_j, P = PlanTable[D_i] ⋈ h_j)
                //set P as plan for cell D_i ∪ D_j
                //if cost(P) is the best so far for that cell
    return the plan for the cell for all subgoals of Q
end
```

Note that the CKPS algorithm considers only those query rewritings whose all possible pairs of view tuples have non-overlapping tuple cores. (In Example 2 the algorithm would produce the rewriting R_1 but not R_2 or R_3.) It is shown in [6] that the CKPS algorithm is sound and complete for CQ queries and views under bag semantics. We use the CKPS approach as the basis of our query-optimization algorithms, see Section 5.

3. REWRITINGS: BAG-SET SEMANTICS

In this section and in Section 4 we characterize equivalence of queries and rewritings using tuple coverages. Under bag-set semantics we need only tuple cores for the characterization, whereas under set semantics (Section 4) tuple coverages that are not tuple cores play a very important role.

Recall rewritings R_1, R_2 in Example 2; unlike R_1, the tuple cores of two view tuples in R_2 overlap on subgoal s of the query Q, and each of R_1 and R_2 is bag-set equivalent to Q. The following theorem gives conditions for possible overlaps of CQ views in such equivalent rewritings:

THEOREM 4. *Given a CQ(A) query Q and two CQ(A) views V and W, with view tuples t_V and t_W, respectively, such that the tuple cores of t_V and of t_W in Q have in common a nonempty set G of subgoals of Q. Then t_V and t_W can be combined in an equivalent rewriting of Q under bag-set semantics if and only if all the variables in G are head variables of both view tuples t_V and t_W.*

(Recall that under *bag* semantics, tuple cores of two view tuples in an equivalent rewriting cannot have a nonempty overlap.) This result follows from Theorem 1; an extension to the CQA case is not hard due to use of central rewritings.

Theorem 4 gives rise to a straightforward cubic-time algorithm[3] for checking equivalence of a rewriting to a query

[3]We measure the complexity of all our algorithms in the number of subgoals in the input query and views.

under bag-set semantics; we give a complexity proof in the full version of the paper. For instance, in Example 2 rewriting R_2 is equivalent to query Q because all the attributes of the subgoal s of Q are head attributes of both views, V and W, in R_2. This condition is not satisfied in rewriting R_3, which is not equivalent to Q under bag-set semantics.

4. REWRITINGS: SET SEMANTICS

We now characterize set-equivalence of CQ(A) queries and rewritings. We start with a motivating example.

EXAMPLE 3. *Consider a CQ query Q and three CQ views:*

$$
\begin{array}{ll}
q(X_1, X_5) & \leftarrow p_1(X_1, X_2),\ p_2(X_2, X_3),\ p_3(X_3, X_4, X_5). \\
v_1(Y_1, Y_3) & \leftarrow p_1(Y_1, Z_1),\ p_2(Z_1, Y_3). \\
v_2(Y_2, Y_5) & \leftarrow p_2(Y_2, Z_2),\ p_3(Z_2, Z_3, Y_5). \\
v_3(Y_2, Y_3, Y_5) & \leftarrow p_2(Y_2, Y_3),\ p_3(Y_3, Z_4, Y_5).
\end{array}
$$

The definition of each view has the subgoal p_2 of the query Q; the only difference between V_2 and V_3 is an extra head variable in V_3. We look for efficient rewritings of Q using only these views (i.e., no base relations) under set semantics. As $V_1(X_1, X_3)$, $V_2(X_2, X_5)$, and $V_3(X_2, X_3, X_5)$ are all the view tuples of the three views for the query Q, any such rewriting would use these view tuples and nothing else. Using containment mappings [4] we can show that a product R_1 of $V_1(X_1, X_3)$ with $V_2(X_2, X_5)$ is not an equivalent rewriting of the query Q, even though the union of the tuple cores of the view tuples for V_1 and V_2 is the set of all subgoals of Q. At the same time, a natural join R_2 of $V_1(X_1, X_3)$ with $V_3(X_2, X_3, X_5)$ is equivalent to the query Q.

We proceed to define a local condition for checking whether two view tuples can be combined in a rewriting that is equivalent to the given query under set semantics. This local condition will help us disqualify combinations such as V_1 with V_2 in Example 3, and will help us pursue combinations such as V_1 with V_3 in the same example.

4.1 Containment: partition condition

In this subsection we formulate a necessary and sufficient *partition condition* for combining two view tuples in a rewriting of a CQ(A) query using CQ(A) views under set semantics; this condition gives us a procedure for constructing efficient equivalent rewritings of queries using views.

Intuitively, a query Q is equivalent to a rewriting R if and only if there is a partition of the set of subgoals of Q into subsets, such that each subset G_i is "covered" by a view tuple for view V_i in R. We now state this result formally, as Theorems 5 (extended from [2] to the CQA case) and 6.

THEOREM 5. *Given a CQ(A) query Q on a database with schema $\mathcal{S}$ and a set of CQ(A) views $V_1, \ldots, V_n$ defined on $\mathcal{S}$, with view tuples $t_{V_1}, \ldots, t_{V_n}$, respectively. Then a rewriting R using $t_{V_1}, \ldots, t_{V_n}$ contains Q under set semantics, $Q \sqsubseteq_s R$, if and only if the union of the tuple cores of $t_{V_1}, \ldots, t_{V_n}$ is the set of all subgoals of the query Q.*

THEOREM 6. *Given a CQ(A) query Q on a database with schema $\mathcal{S}$ and a CQ(A) rewriting R in terms of CQ(A) views $V_1, \ldots, V_n$ defined on $\mathcal{S}$, with view tuples $t_{V_1}, \ldots, t_{V_n}$, respectively. Let R set-contain Q, $Q \sqsubseteq_s R$. Then $R \sqsubseteq_s Q$ (and therefore $R \equiv_s Q$) if and only if the set $\{t_{V_j}\}$ of view tuples in R has a subset $\{t^*_{V_i}\}$, $i \in \{1, \ldots, m\}$, with tuple coverages $s(t^*_{V_i}, Q)$, such that the following* partition condition *holds: (1) $s(t^*_{V_k}, Q) \bigcap s(t^*_{V_l}, Q)$ is an empty set for each $k, l \in \{1, \ldots, m\}$ $(k \neq l)$, and (2) $\bigcup_{i=1}^{m} s(t^*_{V_i}, Q) = G_Q$, where G_Q is the set of subgoals of the query Q.*

We illustrate this result using Example 3. It is easy to see that each of R_1 and R_2 contains the query Q. The rewriting $R_2 = t_{V_1} \bowtie t_{V_3}$ is contained in the query Q because the tuple coverages $s(t_{V_1}, Q) = \{p_1, p_2\}$ and $s(t_{V_3}, Q) = \{p_3\}$ of the two view tuples partition the set $\{p_1, p_2, p_3\}$ of all subgoals of Q. At the same time, for the rewriting $R_1 = t_{V_1} \bowtie t_{V_2}$ in Example 3, the partition condition does not hold for any combination of tuple coverages of the view tuples V_1 and V_2. As a result, R_1 is *not* contained in the query Q. Intuitively, we cannot find a containment mapping from Q to R_1^{exp}, such that the mapping could map the attributes of Q's subgoal $p_2(X_2, X_3)$ consistently into R_1^{exp}.

4.2 Efficient algorithm for partition checking

In this subsection we study the complexity of checking the partition condition of Theorem 6. Consider a CQ query Q and a CQ rewriting R that contains Q, such that R comprises two view tuples t_V and t_W. Let the union of the tuple cores of t_V and t_W be the set of subgoals of Q; thus, $Q \sqsubseteq_s R$. Does Q also contain R? Suppose the intersection S^I of the tuple cores of t_V and t_W has "nearly all" subgoals of Q. Not surprisingly, if we try to check the partition condition of Theorem 6 by exploring subsets of subgoals of Q in this "overlap area" S^I, we can use an adversarial argument to show that the complexity of the checking can be exponential in the number of subgoals of Q and of the two views.

We now present a polynomial-time algorithm E-MCD-Rewriting for checking the partition condition of Theorem 6 under set semantics, for CQ(A) queries, views, and rewritings. The algorithm has been inspired by MiniCon Descriptions (MCDs), developed in [18] in the context of computing maximally contained rewritings. We adapt MCDs to our context of equivalent rewritings, by defining an *E-MCD* for a given view tuple t_V of a CQ(A) view V on CQ(A) query Q as a *minimal* tuple coverage of t_V. That is, an E-MCD is any tuple coverage $s(t_V, Q)$ of t_V on Q, such that no proper subset of subgoals of $s(t_V, Q)$ is also a tuple coverage of t_V on Q. (In Example 3, $\{p_2, p_3\}$ is the only E-MCD of t_{V_2}, whereas t_{V_3} has two E-MCDs, $\{p_2\}$ and $\{p_3\}$.) It is easy to see that, given a CQ(A) query Q and a view tuple t_V for a CQ(A) view V, the set of all E-MCDs of t_V on Q provides a partition of the tuple core of t_V on Q. There exists a quadratic-time procedure for finding all E-MCDs of a view tuple for a query. (Quadratic is also an upper bound on the cost of finding the tuple core of a view tuple for a query.)

Our algorithm E-MCD-Rewriting is based on an equivalent rephrasing of Theorem 6, which replaces "tuple coverages" by "sets of E-MCDs." For ease of exposition, we present here a version of E-MCD-Rewriting called E-MCD-Partition; E-MCD-Partition operates on exactly two view tuples, instead of an arbitrary number in E-MCD-Rewriting. Consider an input $\mathcal{I}$ to E-MCD-Partition; $\mathcal{I}$ comprises: (1) a set S^U of all subgoals of a query Q, (2) two views V and W and the view tuples for the views on Q, t_V and t_W respectively, such that (a) the union of the tuple cores of t_V and t_W is exactly

S^U, and (b) the intersection S^I of the tuple cores of t_V and t_W is not an empty set. We require additionally that neither view tuple be redundant in the rewriting. That is, the tuple core of each of t_V and t_W includes at least one subgoal in $S^U - S^I$. Given such an input $\mathcal{I}$, E-MCD-Partition returns *true* if there is a partition P of some E-MCDs of t_V and t_W, such that P "covers" exactly the set S^U.

Algorithm 2: Procedure E-MCD-Partition

```
begin
  S_1^V ← t_V's E-MCDs "crossing border" (S^I, S^U − S^I)
  S_1^W ← s_max(t_W, Q); i ← 2; if S_1^V = φ then
    return true; //one partition uses t_W's tuple core
  while true do
    S_i^W ← all subgoals in S_{i−1}^W covered by those
        E-MCDs in t_W that do not overlap with S_{i−1}^V;
    if S_i^W misses some subgoal of s_max(t_W, Q) − S^I
    then
      return false; //no partition exists
    S_i^V ← all t_V's E-MCDs that cover S^I − S_i^W;
    if S_i^V = S_{i−1}^V then
      return true; //one partition uses all E-MCDs
      //of t_W that cover S_i^W
    i ← i + 1;
end
```

We give a pseudocode for E-MCD-Partition as Algorithm 2. In each iteration i the algorithm maintains a partial set S_i^V of E-MCDs in t_V that must be in each partition P; S_1^V is initialized to the set $\{s\}$ of E-MCDs in t_V where each s covers subgoals in both S^I and $S^U - S^I$, that is, "crosses the border" between the two sets. (If S_1^V is empty, E-MCD-Partition returns *true*, because a partition P exists that has all E-MCDs of t_W.) S_1^W is initialized to the tuple core $s_{max}(t_W, Q)$ of t_W. Each set S_i^W contains all subgoals of S^U that are covered by those E-MCDs in t_W that do not overlap with S_i^V. If no partition P exists, then S_i^W for some i must miss at least one subgoal s in the tuple core of t_W, such that s is *not* in the "overlap area" S^I; the algorithm returns *false* when it first discovers any such subgoal s. Whenever at least one partition P exists, S_i^V for some i is the same as S_{i-1}^V; the algorithm returns *true* once $S_i^V = S_{i-1}^V$. Otherwise S_i^V grows in each iteration, by all E-MCDs of t_V that cover the subgoals of S^I that are neither in S_i^W nor in S_{i-1}^V.

Our algorithm E-MCD-Rewriting is an easy generalization of E-MCD-Partition to an arbitrary number of views in the problem input. We have the following correctness result:

THEOREM 7. *Algorithm E-MCD-Rewriting is sound and complete for CQ(A) queries, views, and rewritings.*

In the full version of the paper we prove that the complexity of E-MCD-Partition is quadratic in the number n of subgoals of the query and views in the input $\mathcal{I}$, and thus the complexity of E-MCD-Rewriting is $O(n^3)$ for *locally minimal* [17] rewritings, that is, for rewritings that never contain more than n view tuples.

5. COST-BASED QUERY OPTIMIZERS FOR BAG-SET AND SET SEMANTICS

When looking for an efficient view-based execution plan for a given CQ query, the dynamic-programming optimization algorithm CKPS of [6] (Section 2.5) considers all combinations of view tuples with each other and with base relations, such that the respective tuple cores "do not overlap" — that is, all pairwise intersections of the tuple cores are empty; see, for instance, rewriting R_1 in Example 2. To be able to produce additional view-based query plans under bag-set and set semantics, such as rewritings R_2 and R_3 in Example 2, we look for simple efficient local conditions on combining, in a partial rewriting, view tuples whose tuple cores do overlap. For the set-semantics case we present efficient local conditions for special cases, as the "overlap test" for the general case could lead the algorithm to revisit some dynamic-programming (DP) subproblems once it has visited unrelated DP subproblems, which would defeat the purpose of dynamic programming. Section 6 reports our experimental validation of the results of this section.

5.1 Basic DP algorithm BDPV

Our basic DP algorithm with views (BDPV) is the CKPS algorithm of [6] enhanced by processing view tuples with overlapping tuple cores. The preprocessing step of BDPV is exactly the same as that of CKPS; the pseudocode for the main stage of BDPV is shown as Algorithm 3. Just like the main stage of the CKPS algorithm, BDPV considers each set of query subgoals as a subproblem, or "cell," in dynamic programming, and checks whether each cell is "exactly covered" by (i.e., whether each cell is the tuple core of) any view tuple returned by the preprocessing stage. Then BDPV chooses for each cell one optimal plan from all the available — including view-based — plans.

Algorithm 3: Query-optimization algorithm BDPV

```
Input  : query Q, table MapTable, cost model M
Output: optimal execution plan for Q w.r.t. M
begin
  for each cell D_i of PlanTable in increasing size do
    for each (D_j, h_j) in MapTable do
      if cells D_i and D_j are disjoint then
        AddPlan(D_i ∪ D_j, P = PlanTable[D_i] ⋈ h_j)
        //set P as plan for cell D_i ∪ D_j
        //if cost(P) is the best so far for that cell
      else
        if GoodNontrivialOverlap(PlanTable[D_i], D_j)
        == true then
          AddPlan(D_i ∪ D_j, PlanTable[D_i] ⋈ h_j)
  return the plan for the cell for all subgoals of Q
end
```

The only difference between CKPS and BDPV is the `GoodNontrivialOverlap` segment in Algorithm 3. Intuitively, at each DP cell we try to ensure that the partial rewriting we are building is contained in some part of the query Q. In the `GoodNontrivialOverlap` segment, after deciding on an optimal plan $P_i = PlanTable(D_i)$ for a cell D_i of a query Q, BDPV additionally checks whether the set S_{D_i} of base relations and views in the plan P_i[4] can be combined with each view tuple t_V whose tuple core $s_{max}(t_V, Q)$ "nontrivially overlaps" with the set of query subgoals D_i. Then, for each view tuple t_V that can be successfully combined with a

[4]S_{D_i} can have either no base relations or no views.

cell plan P_i (with tuple core D_i), BDPV adds the resulting combination to the set of candidate plans for the DP cell whose set of query subgoals is exactly the union of the tuple cores of t_V and P_i. For instance, in Example 2 BDPV would send to the cell for all subgoals of the query Q (1) a combination of view tuples for V and W (i.e., rewriting R_2) under bag-set semantics, and (2) two combinations of view tuples under set semantics, (U, W) and (V, W), where (U, W) — that is, the rewriting R_3 — could be chosen as the more efficient plan for Q.

We now give an outline of this additional processing in BDPV, by first defining the nontrivial-overlap condition and then describing the combination test, that is, necessary and sufficient conditions for combining view tuples with partial query plans. We say that two sets s_1 and s_2 of subgoals of query Q *nontrivially overlap* if (1) the intersection of s_1 and s_2 is nonempty, as is (2) each of the sets $s_1 - s_2$ and $s_2 - s_1$. Intuitively, condition (1) provides the "overlap," whereas condition (2) ensures that each of s_1 and s_2 contributes to the combination some query subgoals that the other set does not. (Recall that our cost model discourages rewritings with redundant base-relation or view subgoals.)

We now describe the combination test that BDPV does. Consider a DP cell D_i and the combination S_{D_i} of base relations and views in an optimal plan P_i for D_i; suppose t_V is a view tuple whose tuple core $s_{max}(t_V, Q)$ nontrivially overlaps with the set of query subgoals in D_i. The view tuple t_V *can be combined* with the plan P_i if an *equivalent-overlap condition* holds. The equivalent-overlap condition is not the same for bag-set and set semantics, as the equivalence conditions between a query and rewriting are different in these two cases. Either equivalent-overlap condition we give here ensures generation of equivalent view-based rewritings of a given query under the respective semantics, as the conditions are based on Theorem 4 in Section 3 (for bag-set semantics) and on Theorem 6 in Section 4 (for set semantics). We will see shortly that using the DP-algorithm template results in loss of completeness, and will discuss in Section 5.2 what should be done to restore completeness in the context of a cost-based query-optimization approach of [19].

Under bag-set semantics, view tuple t_V and optimal plan P_i for DP cell D_i can be combined if for each subgoal s in the intersection of the tuple core $s_{max}(t_V, Q)$ of t_V with the expansion of the set S_{D_i} of subgoals of P_i, it holds that s does not have nonhead variables in either t_V or S_{D_i}. Under set semantics, t_V and P_i can in general be combined if the partition condition of Theorem 6 (Section 4) holds for the union of some tuple coverage $s(t_V, Q)$ of t_V with some tuple coverage of P_i. In our BDPV algorithm under set semantics we use efficient equivalent-overlap conditions for the special cases of chain and star queries. The following example illustrates the reasons for this restriction.

EXAMPLE 4. *We look for equivalent CQ rewritings of a CQ query Q in terms of CQ views V, W, and U:*

$$\begin{array}{ll} q(X,Y,U) & \leftarrow p_1(X,Y),\ p_2(Y,Z),\ p_3(Z,S),\ p_4(S,T),\ p_5(T,U). \\ v(A,B,D) & \leftarrow p_1(A,B),\ p_2(B,H),\ p_3(H,D). \\ w(C,F,G) & \leftarrow p_3(C,J),\ p_4(J,F),\ p_5(F,G). \\ u(B,F) & \leftarrow p_2(B,C),\ p_3(C,D),\ p_4(D,F). \end{array}$$

Let the view tuple t_V for view V be an optimal plan P_{123} for the DP cell $D_{123} = \{p_1, p_2, p_3\}$; we now combine P_{123} with the view tuple t_W for view W. Mapping the query Q into the cross product $t_V \times t_W$ [4] shows us that $t_V \times t_W$ has two *tuple cores, $S_1 = \{p_1, p_2, p_3, p_5\}$ and $S_2 = \{p_1, p_3, p_4, p_5\}$. Moreover, S_2 is* not *a superset of the DP cell D_{123}. At the same time, the combination of t_V with t_W should be considered when looking for equivalent rewritings of the query Q, as $(t_V \times t_W) \bowtie t_U$ is a set-equivalent rewriting of Q.*

As shown in Example 4, the first complication of dynamic-programming query optimization under set semantics is that, unlike the bag-set-semantics case, an optimization algorithm may produce in a DP cell D_i two or more partial query plans $P_i^1, \ldots, P_i^k$ for other DP cells, where each P_i^j is a combination of an optimal plan P_i for D_i with some view tuple t_{V_j}, *and* not all the combinations P_i^j use the same tuple coverage of P_i. (In Example 4, we use tuple coverage $\{p_1, p_2, p_3\}$ of plan P_{123} to obtain a combination (P_{123}, t_W) whose tuple core is S_1, and use tuple coverage $\{p_1\}$ of P_{123} to obtain a combination (P_{123}, t_W) whose tuple core is S_2.) While this problem could be solved by considering all tuple coverages of P_i when producing partial plans P_i^j, the second point illustrated by Example 4 is that a combination of two view tuples might have to be sent from a DP cell D_i (cell D_{123} in Example 4) to another DP cell D_j (cell S_2 in Example 4) that may miss some subgoals of D_i — which would defeat the purpose of dynamic programming. While this behavior can also be demonstrated for queries whose shape is not chain or star, in developing our equivalent-overlap conditions we succeeded in ruling out this undesirable flow behavior for BDPV for chain and star queries. For the set-semantics case we require that BDPV do not consider cross products of partial plans P_i (for DP cells D_i) with view tuples that satisfy the nontrivial-overlap condition. This choice has been made in other query-optimization approaches as well (e.g., [19]); note that this no-cross-products requirement is satisfied naturally in BDPV in the bag-set semantics case.

For the special cases of CQ chain and star queries under set semantics, BDPV uses the following necessary and sufficient equivalent-overlap conditions:

- for chain queries, a view tuple and a partial query plan must have a query (i.e., head) variable in common; note that this requirement is exactly the no-cross-products assumption;
- for star queries, we distinguish the *hub variable* of the query, which is the "center of the star"; if a view tuple t_V and a partial query plan P overlap on a query subgoal that has the hub variable, we require that the hub variable be the head variable of both t_V and P.

The following result holds by construction of BDPV:

THEOREM 8. *The cost-based DP algorithm BDPV is sound (1) for all CQ queries, views, and rewritings under bag-set semantics, and (2) for chain or star CQ queries, views, and rewritings under set semantics.*

The complexity of the algorithm is $O(l \times 2^n)$, where n is the number of query subgoals and l is the number of entries in the mapping table MapTable. By construction of MapTable, l is an upper bound on the number of view tuples applicable to the query. We obtain this bound from the bound $O(l \times 2^n)$ [6] of the CKPS algorithm, after observing

that for each dynamic-programming cell BDPV additionally considers overlaps of its optimal plan with at most l views.

5.2 Complete algorithm CDPV

In this subsection we justify the need for and give a description of our complete cost-based algorithm with views. It has been shown [6] that the CKPS algorithm is complete for CQ queries and views under bag semantics; in Proposition 1 we extended this result to aggregate queries and views under bag semantics. In contrast, while our algorithm BDPV is sound under bag-set or set semantics (Theorem 8), it is in general incomplete for CQ(A) queries and views. This result is not suprising, because it is possible for a query to have an optimal equivalent rewriting that uses views, such that for some cells in DP search, partial plans for this particular rewriting are not the most efficient plans for the cells. In the full version of this paper we (1) give examples that prove incompleteness of BDPV for CQ queries and views for each of bag-set and set semantics, and (2) show completeness of BDPV for the following special cases:

- all variables in all applicable views are head variables, under bag-set or set semantics;
- for each pair of view tuples applicable to a query, either the two view tuples have no query subgoals in common, or in all the common subgoals all the variables in both view tuples are head variables, under bag-set semantics;
- each pair of views has the same head variables on each base relation they have in common, under set semantics (e.g., in Example 2 views V and W satisfy this condition, while view U violates it on base relation S).

For combinations of CQ queries and views that do not satisfy the above requirements, we can augment BDPV to make it complete, by trying at each DP cell to combine with overlapping views *all* possible partial plans for the cell, rather than just one optimal partial plan; in this sense, these partial plans can be compared to "interesting orders" in [19]. All successful combinations are then "sent up" to the relevant cells for consideration. We call this complete cost-based algorithm with views CDPV. Note that while it uses BDPV's DP search structure, CDPV is *not* a DP algorithm.

THEOREM 9. *The cost-based algorithm CDPV is sound and complete (1) for all CQ queries, views, and rewritings under bag-set semantics, and (2) for chain or star CQ queries, views, and rewritings under set semantics.*

While having a complete algorithm for these cases may sound like good news, there are two problems: First, CDPV breaks the flow of dynamic programming by potentially using two or more partial query plans for some DP cells. Second, the complexity of CDPV is $O(r \times l \times 2^n)$, where r is the number of equivalent rewritings of the query (r may be exponential in the number of applicable views), and $O(l \times 2^n)$ is the complexity of BDPV. This complexity makes CDPV impractical in general. At the same time, using the complete algorithm CDPV might be time efficient either for queries with just a small number of subgoals, or for precompilation of time-critical queries, when it is important to find an optimal query plan under the assumption that the base relations do not change significantly over long periods of time.

5.3 Optimizing aggregate queries

In this subsection we discuss how to use views to find efficient central rewritings [1] of queries with aggregation, such as rewriting `R` in Example 1. It is rather straightforward to extend our algorithm BDPV for CQ queries, views, and rewritings to queries and views with aggregation using central rewritings. (Similarly, it is not hard to extend the optimization algorithm of [6] to queries, views, and rewritings with aggregation under bag semantics using Proposition 1.) The extension is straightforward because using central rewritings allows us to first find an equivalent rewriting of the unaggregated core of the query, by applying BDPV to unaggregated cores [8] of the views, and to then take care of the needed grouping and aggregation on top of that core rewriting. It is shown in [1] that (1) some or even all views in a central rewriting of an aggregate query may have no aggregation, and that (2) depending on the available aggregate views, an equivalent rewriting of an aggregate query does not even have to have aggregation in the head.

We still need to address two issues in our extension of BDPV to aggregate queries and views. First, BDPV needs to decide which grouping and aggregation operators, if any, to select as operators on top of the winning (possibly view-based) query plan for the unaggregated core of the query. This issue is addressed by using the algorithm of [1] for choosing the central view in a central rewriting: Intuitively, the central view determines the grouping/aggregation operators in the rewriting. (A view is *central* in a central rewriting if the view's head argument is used to obtain the aggregated head argument of the rewriting; for instance, view `V` is the central view in rewriting `R` in Example 1.)

Second, we need to modify the view-overlap conditions that BDPV uses for CQ queries, views, and rewritings without aggregation. We have to consider separately the cases of bag-set and set semantics: We use *bag-set* equivalence to find efficient equivalent rewritings of `SUM` or `COUNT` queries, and *set* equivalence for `MAX` or `MIN` queries. Under set semantics, our approach applies to queries on star-schema data warehouses [5]; it generates rewritings that join aggregate views on one base relation with (possibly views on) other relations, and additionally covers the case of combining several aggregate views, such as in rewriting `R` in Example 1.

For `MAX` or `MIN` queries on a star-schema database, a necessary and sufficient view-overlap condition is as follows. We use the same no-cross-products condition as in the CQ case, and require additionally that the head of the rewriting aggregate further (if needed) only the aggregated output argument of the view whose tuple coverage *in the rewriting* includes the table whose attribute is aggregated in the query. (This covers aggregation on either fact or dimension tables.) This condition is satisfied in rewriting `R` in Example 1.

In case of bag-set semantics, we do not restrict ourselves to star-schema queries. Notably, in rewritings of `SUM` or `COUNT` queries two view tuples cannot overlap on a subgoal where at least one of the view tuples has aggregation [1, 8]. On the other hand, BDPV can produce rewritings where two view tuples overlap on query subgoals on which neither view has aggregation. To the best of our knowledge, such rewritings cannot be produced by algorithms in the literature.

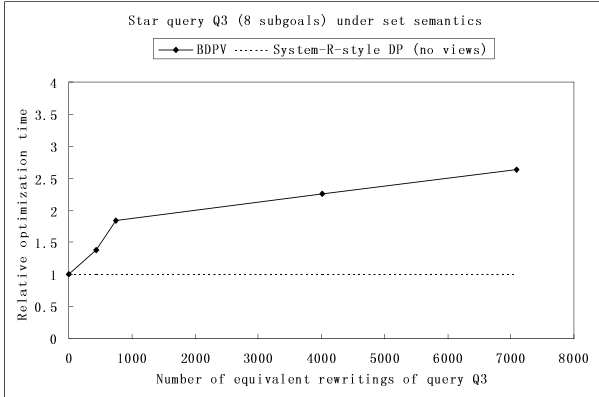

Figure 1: BDPV: Relative optimization time

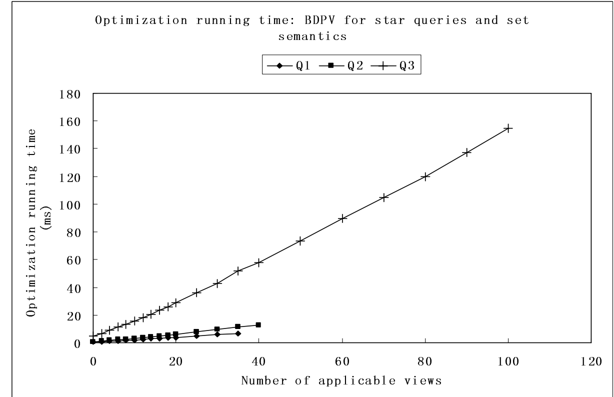

Figure 2: BDPV: Absolute optimization time

It is straightforward to extend CDPV to queries and views with aggregation. In the full version of the paper we prove for the extensions of BDPV and CDPV to aggregate queries that (1) our extension of BDPV is sound, and (2) our extension of CDPV is complete with respect to central rewritings.

6. EXPERIMENTAL RESULTS

In our experiments we measured the relative increase in optimization time (cf. [6]) for our view-based algorithms BDPV and CDPV; in addition, we studied the scalability of both algorithms in the number of views and of potential rewritings. The results of our tests seem to point to the computational efficiency of our algorithm BDPV for reasonable numbers of applicable views and rewritings per query.

In this section we give the details of our experimental setup and results. We implemented all the algorithms in C++. All the experiments were run on a machine with a 1 GHz Pentium3 processor, 1GB RAM, and a 18 GB hard drive, running RedHat Linux 3.2.3 and its built-in GCC compiler.

For the experiments we implemented a generator for CQ queries and views with only binary subgoals and without self-joins; the generator takes as inputs (1) the number of base relations, (2) the number of views, (3) the number of subgoals in a query/view, and (4) the shape of the query and views. The views were generated based on the respective queries; for instance, each chain view was generated by randomly selecting the "start point" and "end point" relations in the chain for the respective query, and by then randomly selecting the head variables of the view. In our experiments we used twelve chain and star queries consisting of between five and eight relations. For each query, between around 100 and 300 views were generated; the head variables of the views were chosen in such a way that between 35 and 100 views could be used in rewritings of the respective queries. (That is, for each query, only between 35 and 100 of the views ended up in the table MapTable, see Section 2.5.) For each query, the number of applicable views determined the number of rewritings per query. We have explored queries with up to 8000 possible equivalent rewritings, which corresponded to only 100 applicable views. (Recall that our rewritings can use both base relations and views.)

In all the experiments, we tested each query and its respective views using the bag-set- and set-semantics versions of our DP algorithm BDPV and of its complete version CDPV. We used an experimental framework similar to that of [6]. Our optimizer cost model estimated query-plan costs using the sizes of base and intermediate relations, in bytes, and the costs of joins as described in [11]; we did not take indexes into account. In the optimizer cost estimates, we used several sets of database statistics for each version of the query-shape/semantics setting (e.g., for chain queries under set semantics); in the statistics, the ratio of the size S of the base relations to the value of the parameter M for memory size was in the interval $0 < S < M^4$. When measuring running times for query optimization, we did not take into account the preprocessing time (i.e., building the mapping table MapTable), as it is linear in the number of applicable views [6] and, in our experiments, in nearly all cases it used less than 10% of the total optimization time. (This distinghuishes our setting from that of, e.g., [12], where view matching takes up a significant portion of the total optimization time.) For each query-shape/semantics setting, each result we recorded was the average of the elapsed time for ten runs for each query. Due to lack of space, in this section we present mainly our BDPV results for star queries under set semantics. The full version of the paper has all the details for BDPV and CDPV for all four settings: bag-set or set semantics, for chain or star queries and views.

In our first set of experiments we measured relative optimization time for BDPV and CDPV, calculated (similarly to [6]) as the ratio of the running time of each of BDPV and CDPV to the running time of the System-R-style query optimizer without views, run on the same query and base relations. Not surprisingly, our results for relative optimization time for CDPV were not reassuring: Even for the "easiest" case of chain queries under bag-set semantics, the ratio was over ten for as few as 100 rewritings. The experimental results for BDPV for one of the star queries under set semantics are shown in Figure 1; we can see that the relative optimization time for the query $Q3$ is under three even for over 7000 rewritings. In our results for all four query-shape/semantics settings, in each setting the relative optimization time of BDPV was under 6 in the low range of the number of equivalent rewritings (e.g., for up to 1000 rewrit-

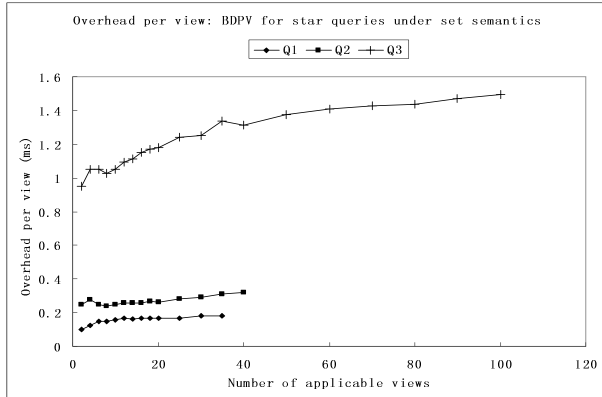

Figure 3: BDPV: Overhead per view

ings for chain queries under set semantics), and decreased[5] with the increase in the number of query subgoals. As the rewritings that are equivalent to a given query under set semantics include all rewritings that are equivalent to the same query under bag-set semantics (when using the same applicable views), the low range of the number of equivalent rewritings in our experiments was much higher under set than under bag-set semantics — for instance, 2000 vs 200 for star query $Q3$.

We saw that our algorithm BDPV can handle over 7000 equivalent rewritings with somewhat reasonable scalability. For the case of up to 20 equivalent rewritings — that is, for the experimental setting of [6] — for both semantics (bag-set or set) and both query shapes (chain or star), the relative optimization time of BDPV was comparable to that of the CKPS approach of [6]. This effect is easy to explain using the theoretical complexity of BDPV, because for each combination of a partial plan P_i for a DP cell D_i with an overlapping view tuple h_j, the additional work in BDPV (as compared to CKPS on which BDPV is based) is done using (1) efficient view-overlap tests, and (2) cost estimates for just one plan $P_i \bowtie h_j$.

We now discuss our results on scalability of BDPV and CDPV in the number of applicable views. Figure 2 shows absolute running times for BDPV for three star queries under set semantics; queries $Q1$, $Q2$, and $Q3$ have five, six, and eight subgoals respectively, and had up to 35, 40, and 100 applicable views respectively. (The expectably disappointing execution time of CDPV is not shown.) In Figure 2, we show on the X-axis only applicable views, as the preprocessing step takes care of filtering out all other views. Figure 2 shows that, for instance, when we provided 20 applicable views to each query, the running time of BDPV was under 10 ms for queries $Q1$ and $Q2$ and under 40 ms for $Q3$. Overall, the running time of BDPV grows linearly in the number of views; it was under 160 ms for star queries under set semantics (Figure 2) and under 35 ms for chain queries under set semantics, and was correspondingly lower for star and chain queries under bag-set semantics. These results show good scalability of BDPV in the number of applicable views, in all four query-shape/semantics settings.

[5]Recall that we report running times that are *relative* to the running times of the optimizer that does not use views.

Finally, we considered overhead per applicable view for both BDPV and CDPV. We measured overhead as the difference between the running time with views and the running time without views (i.e., using a System-R-style optimizer), divided by the number of views. As expected, the overhead numbers for CDPV were not acceptable. Figure 3 shows the overhead values for BDPV for the same queries and experimental setting as those for Figure 2. We can see in Figure 3 that overhead per view is modest for each query for under 40 applicable views; even at 100 applicable views, the overhead per view for query $Q3$ does not exceed 1.5 ms. The overhead values for up to 100 applicable views were under .3 ms for chain queries under set semantics; under bag-set semantics, the respective values were lower as expected.

The reason BDPV has good efficiency and scalability for a realistic number of views per query is that we use efficient local view-overlap tests. Note that under set semantics, equivalent rewritings of a query include at least all those rewritings that are equivalent to the query under bag-set semantics, which explains the relatively higher efficiency of the bag-set-semantics version of both algorithms. In summary, our experiments show two points. (1) While the running times of CDPV were acceptable in our experiments in the "easiest case" of chain queries under bag-set semantics, in general CDPV did not show promising performance or scalability. It seems that to obtain efficient rewritings of a query using overlapping views under bag-set or set semantics, we may need to settle for incomplete query optimizers, such as BDPV. (2) The BDPV optimization algorithm has good efficiency and scalability for CQ queries and views whose shape is chain or star, under both bag-set and set semantics.

7. RELATED WORK

Algorithms for rewriting queries using views are surveyed in [15]. Algorithms have been developed for CQ queries with comparison predicates [23], aggregate queries and views [9, 13, 20], and for queries and views in presence of integrity constraints [12, 22]. Cohen et al. [9] present algorithms for rewriting aggregate queries using views, under bag-set and set semantics. The novelty of the approach is in extending to aggregate queries previously known algorithms for obtaining view-based rewritings of CQ queries. In contrast, we propose a systematic unified approach to finding efficient view-based rewritings for queries with or without aggregation.

Query optimization using materialized views has been explored extensively [3, 10, 12, 16, 22, 24]. Zaharioudakis and colleagues [24] describe an efficient bottom-up algorithm for semantic matching of complex aggregate queries with views. The work does not address integration with a cost-based optimizer. The syntactic-matching approach of [22] enhances a System-R-style query optimizer to integrate views and indexes into plans for queries without aggregation. Our work goes beyond query optimization and covers evaluation of queries with or without aggregation. In addition, in our optimization algorithms we guarantee completeness for some practical cases.

Goldstein et al. [12] present an approach to expanding a set of query plans in a transformation-based query optimizer, by matching queries and views with aggregation `SUM` and `COUNT` in presence of integrity constraints; using a novel in-

dex structure, the algorithm generates all rewritings where a query is computable from a single materialized view. In our work we propose algorithms for constructing rewritings using multiple views, for queries with or without aggregation, while extending a System-R-style query optimizer. DeHaan and colleagues [10] present rules for a transformation-based query optimizer; the rules match queries with indexed aggregate views that may be defined on other views. The efficiency of the approach comes from using signatures in matching views with queries. We are currently exploring the use of such signatures in our work.

Our approach builds on the work of [1, 2, 6, 18]. The contributions of [2] were made to the problem of finding efficient rewritings for CQ queries without aggregation under set semantics. In our work we focus on finding efficient rewritings of CQ(A) queries using views under bag-set and set semantics, and develop criteria for combining view tuples efficiently into a partial rewriting of a query. [18] proposed an efficient scalable algorithm for finding maximally contained rewritings of queries in the context of data integration. We adapt some contributions of [18] to applications where equivalent query rewritings are required. The query-optimization approach of [6] for queries and views without aggregation is the basis of our optimization algorithms BDPV and CDPV, which also apply to queries and views with aggregation and give completeness guarantees (CDPV) under bag-set and set semantics. Afrati and colleagues [1] introduced central rewritings for aggregate queries using views; the idea is to build a rewriting that aggregates over an aggregated argument of just one view, while possibly grouping over a different set of grouping attributes. Using central rewritings allows us to extend our rewriting techniques for CQ queries and views to CQA queries and views.

8. CONCLUSION

We considered the problem of finding efficient equivalent view-based rewritings of relational queries, focusing on query optimization using materialized views, under the assumption that base relations cannot contain duplicate tuples. To enable consideration of all efficient rewritings of a query we developed efficient local conditions under which two views can be combined in a rewriting, for SPJ queries and views with or without aggregation, under bag-set and set semantics for query evaluation. For each semantics we described an efficient sound and complete algorithm for testing query-rewriting equivalence. Our algorithms are complete with respect to SPJ rewritings for SPJ queries and with respect to central rewritings [1] for aggregate queries.

We proposed sound dynamic-programming algorithms based on our local conditions, for finding efficient execution plans for user queries using indexes and materialized views; queries and views may have grouping and aggregation. Our approach is a relatively simple generalization of the query-optimization algorithm of [6]; similarly to [6], our approach can be used to exploit cached results. While exploring strictly more rewritings than the approach of [6], in general our algorithms for bag-set and set semantics are incomplete; we presented a theoretical study of the completeness-efficiency tradeoff and described enhancements that would make the algorithms complete. Finally, we provided experimental results that show good efficiency and scalability of one of our algorithms, BDPV, for bag-set and set semantics.

In our current work we study how to extend BDPV and its complete version CDPV to more general classes of CQ(A) queries and views under set semantics, by using the E-MCD-based overlap test of Section 4.2. We also plan to extend our approach to more general query, view, and rewriting languages, in particular to those involving inequality comparisons and integrity constraints.

9. REFERENCES

[1] F. Afrati and R. Chirkova. Selecting and using views to compute aggregate queries. In *Proc. ICDT*, 2005.

[2] F. Afrati, C. Li, and J.D. Ullman. Generating efficient plans for queries using views. In *Proc. SIGMOD*, 2001.

[3] R.G. Bello, K. Dias, A. Downing, J.J. Feenan Jr., J.L. Finnerty, W.D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized views in Oracle. In *VLDB*, 1998.

[4] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proc. 9th ACM STOC*, pages 77–90, 1977.

[5] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, 1997.

[6] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proc. ICDE*, pages 190–200, 1995.

[7] S. Chaudhuri and M. Y. Vardi. Optimization of real conjunctive queries. In *Proc. PODS*, pages 59–70, 1993.

[8] S. Cohen, W. Nutt, and A. Serebrenik. Rewriting aggregate queries using views. In *Proc. PODS*, pages 155–166, 1999.

[9] S. Cohen, W. Nutt, and A. Serebrenik. Algorithms for rewriting aggregate queries using views. In *Proc. ADBIS-DASFAA*, pages 65–78, 2000.

[10] D. DeHaan, P.-Å. Larson, and J. Zhou. Stacked indexed views in Microsoft SQL server. In *Proc. SIGMOD*, 2005.

[11] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database Systems: The Complete Book.* Prentice Hall, 2002.

[12] J. Goldstein and P.-Å. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *Proc. SIGMOD*, 2001.

[13] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proc. VLDB*, pages 358–369, 1995.

[14] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for OLAP. In *Proc. ICDE*, 1997.

[15] Alon Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001.

[16] P.-Å. Larson and H.Z. Yang. Computing queries from derived relations. In *Proc. VLDB*, pages 259–269, 1985.

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

[18] R. Pottinger and A.Y. Halevy. MiniCon: A scalable algorithm for answering queries using views. *VLDB Journal*, 10(2-3):182–198, 2001.

[19] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. SIGMOD*, 1979.

[20] D. Srivastava, S. Dar, H.V. Jagadish, and A.Y. Levy. Answering queries with aggregation using views. In *Proc. VLDB*, pages 318–329, 1996.

[21] D. Theodoratos and T. Sellis. Data warehouse configuration. In *Proc. VLDB*, pages 126–135, 1997.

[22] O.G. Tsatalos, M.H. Solomon, and Y.E. Ioannidis. The GMAP: A versatile tool for physical data independence. *VLDB Journal*, 5(2):101–118, 1996.

[23] H.Z. Yang and P.-Å. Larson. Query transformation for PSJ-queries. In *Proc. VLDB*, pages 245–254, 1987.

[24] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex SQL queries using automatic summary tables. In *Proc. SIGMOD*, pages 105–116, 2000.