Computing Closest Common Subexpressions for View Selection Problems

Wugang Xu
New Jersey Institute of Technology
wx2@njit.edu

Dimitri Theodoratos
New Jersey Institute of Technology
dth@cs.njit.edu

Calisto Zuzarte
IBM Canada Ltd.
calisto@ca.ibm.com

ABSTRACT

Selecting a set of views for materialization is a required task in many current database and data warehousing applications including the design of a data warehouse, and the maintenance of multiple materialized views. The selected views can be materialized permanently or transiently depending on the specific view selection problem. The view selection algorithms are expensive due to the size of the search space of the problem.

In this paper we propose an approach for generating candidate views for materialization for view selection problems based on the definition of the input queries. We also provide rewritings of the input queries using the generated candidate views. In generating candidate views, we do not apply cost-based techniques but we try to maximize the operations in the views. Subsequently, view selection algorithms can exploit problem dependent cost functions to choose among the generated candidate views. Our approach is not restricted to a specific view selection problem. Compared to a previous one, it generates views that involve more relation occurrences (or operations) and can reduce the size of the search space which can be very large. We implement our approach and we report some experimental evaluation with comparison to previous works.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages—*Query languages*; H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms

Management, Algorithms, Performance

Keywords

query graph, common subexpression, data warehouse, view selection

DOLAP'06, November 10, 2006, Arlington, Virginia, USA.

1. INTRODUCTION

Current databases and warehouses are getting bigger in size. Also, queries are getting more complex (e.g. queries involving roll-up and drill down operations and top-k queries) [21] while the user's requirements on query performance are getting stricter. Traditional query optimizers based on relations and indexes can not satisfy these new needs. Materialized views have been found to be a very efficient way to speed up query evaluation. They are increasingly supported by commercial database management systems [4, 25, 2]. A lot of research work has focused on the problem of view selection, which can be abstractly modeled as follows: given a set of queries and a number of cost-determining parameters (e.g. query frequency/importance and source relation update propagation frequencies), output a set of view definitions that minimizes a cost function and satisfies a number of constraints. The cost function can be, for instance, the query evaluation cost, the view maintenance cost, or a combination of them. A constraint can be a storage space constraint, or an upper bound on the materialized view maintenance cost [22]. Depending on the type of problem considered, the selected views can be materialized permanently (e.g. in the case of the data warehouse design problem) or transiently materialized (e.g. in the case of intermediate results during the concurrent evaluation of multiple queries) or both (e.g. in the case of the maintenance of multiple materialized views where some views are computed because they are permanently stored while other views are stored transiently as auxiliary views during the maintenance process, in order to assist the maintenance of multiple other views [15, 24]).

Most the work on view selection problem focuses on data cube operations on multidimensional datasets [10, 3, 20, 12, 14, 6]. The reason is not only that this kind of operations is particularly important in On Line Analytical Processing(OLAP) queries, but also because the search space for the problem with this simplified class of queries can be easily modeled and constructed as a multidimensional lattice. There are also some works on view selection problem on general database systems. However these approaches assume, explicitly or implicitly, the existence of a search space in a form of an AND/OR graph [17, 8, 9]. However, these AND/OR graphs require the construction of alternative common plans for multiple general queries and the common subexpressions among input queries need to be detected and exploited. This is an intricate problem. Further, the original queries need to be rewritten using the common subexpressions.

In order to determine a search space for a view selection problem we need to identify common subexpressions among workload queries. These common subexpression represent part of the work needed to compute a query. When identified they can be computed only once and the result be used by all queries that share it. This is expected to importantly save computation time. Depending on the problem, a common subexpression might also be a good candidate view for storage (e.g. in the case of the data warehouse design problem). In this paper, we focus on the computation of common subexpressions for view selection problems.

1.1 Related Work

The concept of common subexpression initially referred to identical or equivalent expressions [7]. Later on, the term included subsumption [11]. Then the term included overlapping selection conditions [5]. More generally. the term common subexpression between two queries refers to a view that can be used in the rewritings of both queries, either completely or partially [13]. One approach to exploit common subexpressions is on the query evaluation plan level [19, 18, 15]. This method can give the global evaluation plan in addition to selecting a set of views to materialize. It requires the enumeration of all possible evaluation plans for all queries in the workload. Although some heuristics can be applied to reduce the number of evaluation plans considered, this method is still too expensive when the number of queries is big or when the queries in the workload are complex. The resulting search space is also too big for most view selection problems. Another approach to exploit common subexpressions is on the query definition level[5, 13, 21]. In [5], all queries in a workload are represented as a global multigraph. Using heuristic transformation rules, this multigraph is transformed to a one where no more transformations can be applied. This approach can give the common subexpressions as well as the corresponding rewritings. All the queries in the workload can be considered together. However, this paper assumes there are no self-joins in the queries and this assumption is too restrictive for the queries of current applications. In [13], a general concept of common subexpression is introduced and some constraints (e.g. key/foreign key constraints) are considered. An artificial common subexpression (common subsumer) between a pair of queries is constructed if there is no subsumption relationship between them. Then rewritings (called compensations) of the original queries using the common subexpressions are given. This approach considers query pairs that follow some specific patterns. In [21], common subexpressions between different parts of one single query are exploited and used to apply multi query optimization on a single query. The algorithm to exploit commonalities in this paper is more on a topology similarity level than on a predicate level. This is because it is not known in advance which parts of the query need to considered for comparison. In general, the problem of answering queries using views is a NP-hard problem [1]. Implementation aspects of this problem, for restricted classes of queries and views have been addressed in [16, 25].

1.2 Our Approach and Contributions

In order to compute common subexpressions between queries, a "naive" approach would employ cost-based techniques on all possible common subexpressions of two queries. This is practically impossible because of the huge number of possible common subexpressions. Our approach is based on the observation that, usually, including more operations in the common subexpressions of two queries is beneficial for their overall query evaluation cost. Therefore, we use the concept of closest common derivator (CCD) of two queries to represent the maximum commonality between two queries. Intuitively, we are trying to include as many operations (select, join, projection) as possible in a CCD. The operations have to be as restrictive as possible. For instance, join and selection conditions are as strict as possible while project operations have as few attributes as possible. A CCD of two queries should be such that both queries can be rewritten using it. Our approach is general in that we do not require these rewritings to be complete (they can be partial as well). The rewritings of the queries are also produced. Our approach uses syntactic rules for the computation of CCDs. Subsequent view selection algorithms can operate on the search space defined by the CCDs and the queries by applying cost-based techniques relevant to the view selection problem at hand. In a previous work [23], we defined CCDs by matching a relation occurrence in one query to at most one relation occurrence in the other query. In this paper we relax this restriction. We show that this relaxation brings more operations in the CCDs. Additionally, it reduces the size of the search space (by reducing the number of CCDs). Both improvements are expected to be beneficial to the subsequent application of view selection algorithms.

1.3 Outline

The paper is organized as follows: in Section 2, we define the problem and show with intuitive examples how the relaxation of the definition of CCD is beneficial to some query workloads. In Section 3, we formally define the new concept of CCD and provide rules for computing CCDs between query pairs along with their rewritings using CCDs. In Section 4, we provide experimental result to compare the new definition of CCDs with the old definition in [23]. Section 5 summarizes and suggests some future work.

2. MAXIMUM COMMONALITY BETWEEN TWO QUERIES

In a typical view selection problem, the search space is constructed using all common subexpressions among queries in the workload. This search space is usually too big for a view selection algorithm to examine it exhaustively. In this paper, we define a subset of common subexpressions that represents possible commonalities among queries.

2.1 Queries and Rewritings

We consider select-project-join (SPJ) queries, possibly with self-joins. This class of queries constitutes the basis for more complex queries that involve grouping aggregation operations. We use the relational algebra expression $\pi_P(\sigma_C(O))$ to represent a query , O denotes a set of occurrences representing their cartesian product , C denotes a set of atomic conditions (we assume conjunctive queries), and P denotes the set of projected attributes. We assume set-theoretic semantics. Since we allow self-joins, the same relation may occur more than once in O. Therefore, relation occurrence renaming may be needed. For simplicity, we give each occurrence a unique name in addition to the relation name, and use the expression $N[R]$ to represent an occurrence N

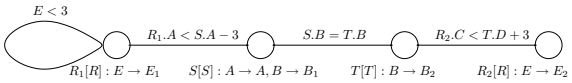


Figure 1: Query Graph for Q1

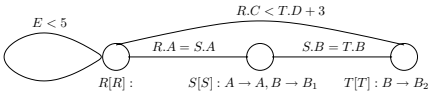


Figure 2: Query Graph for Q2

of relation R. For example, $O = \{R_1[R], S[S], R_2[R]\}$ is a possible occurrence set. Note that the occurrence name may be the same as the relation name, but must be unique among the names in the occurrence set. C represents the predicates to be applied to O. It consists of a set of atomic conditions of the form $A\ \theta\ B + c$ or $A\ \theta\ c$ where θ is an operator from $\{<, \leq, =, \geq, >\}$, A and B are attributes and c is a constant. The latter atomic condition is a selection condition. The former one is a selection condition when A and B are from the same occurrence, and it is a join condition when A and B are from different occurrences. For example, $C = \{R_1.A < S.B + 3,\ R_1.C = 5,\ S.C = R_2.C,\ R_2.D \leq R_2.E\}$ are possible atomic conditions for the previous occurrence set. Note that we disallow "pure" cartesian products in queries. This means that every occurrence in the occurrence set is involved in at least one atomic join condition. P represents all projected attributes. We allow attribute renaming in the projected attribute set. The expression $R_1.A \rightarrow A$ renames attribute $R_1.A$ as A. For example, set P can be $\{R_2.A \rightarrow A, R_1.B \rightarrow B_1, R_2.B \rightarrow B_2\}$. Note that we also allow the renaming of one attribute to more than one attribute in the projected attribute set. For example, one possible projected attribute set may be $\{R_1.A \rightarrow A_1, R_1.A \rightarrow A_2\}$. Similarly to relation occurrence names, the new names of the attributes must be unique among all attribute names in the projected attribute set. An example query follows:

EXAMPLE 1. *Query Q_1:*

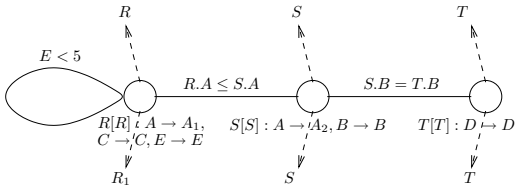
- *SQL*

```
select R1.E as E1, S.A as A, S.B as B1,
       T.B as B2, R2.E as E2
from R as R1, S as S, T as T, R as R2
where R1.E<3 and R1.A=S.A and S.B=T.B and
      R2.C<T.D+3
```

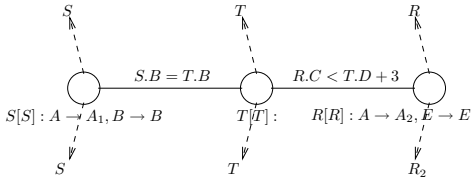
- *Relational Algebra expression:* $\pi_P(\sigma_C(O))$
  - $P = \{R_1.E \rightarrow E_1, S.A \rightarrow A, S.B \rightarrow B_1, T.B \rightarrow B_2, R_2.E \rightarrow E\}$
  - $O = \{R_1[R], S[S], T[T], R_2[R]\}$
  - $C = \{R_1.E < 3, R_1.A \leq S.A-3, S.B = T.B,\ R_2.C < T.D + 3\}$

Note that the order of the projected attributes, atomic conditions and relation occurrences are of no importance. Query *rewritings* are queries of the same form that involve also at least one view.

We use *query graphs* to graphically represent queries. Relation occurrences are represented by nodes. Relation occurrence and relation names along with new and old attribute names are shown by the corresponding nodes. The query graph of query Q_1 of the previous example is shown in Figure 1. Figure 2 shows the query graph of another query Q_2.

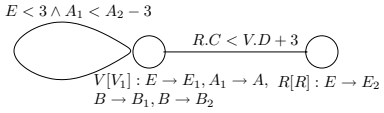


(a) V_1

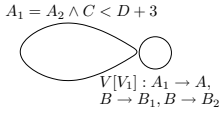


(b) V_2

Figure 3: CCDs of query Q_1 and Q_2

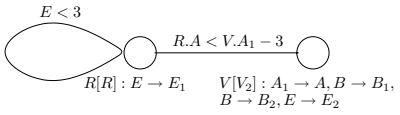


(a) Q'_1

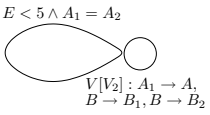


(b) Q'_2

Figure 4: Rewriting of queries Q_1 and Q_2 using V_1



(a) Q'_1



(b) Q'_2

Figure 5: Rewriting of queries Q_1 and Q_2 using V_2

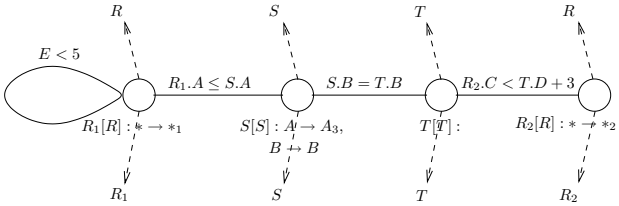
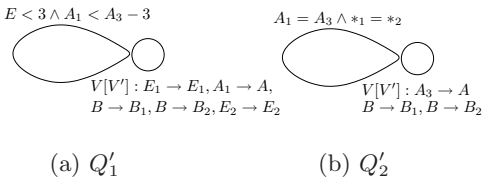


Figure 6: Common subexpression V' of Q_1 and Q_2



(a) Q'_1 (b) Q'_2

Figure 7: Rewriting of queries Q_1 and Q_2 using V'

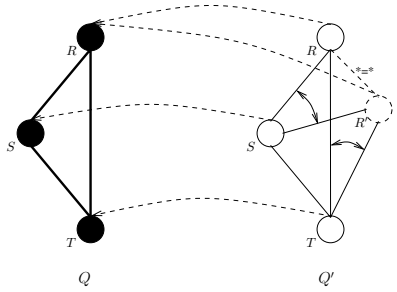


Figure 8: Node splitting

2.2 Query Commonalities

In [23], we define CCDs of two queries by requiring minimal rewritings of both queries using each CCD. A rewriting is *minimal* if, for each relation, the sum of the number of its occurrences in the rewriting and the views of the rewriting equals the number of its occurrences in the query. Although this requirement simplifies the definition and computation of a CCD, it also prevents the detection of some commonalities between queries. Consider the queries Q_1 and Q_2 shown in Figures 1 and 2. Based on the old definition of a CCD, these queries have two CCDs V_1 and V_2 shown in Figure 3. In general, we call an attribute A *redundant* in a query if a condition of the form $A = B + c$, where B is another attribute and c is a constant, can be implied from the condition of the query.

Note that in both V_1 and V_2, we do not project attribute B from T although it is projected in both Q_1 and Q_2. This is because the condition in each of V_1 and V_2 imply that $S.B = T.B$. Therefore, we can project only one attribute among $S.B$ and $T.B$ (e.g. in V_1 and V_2, we project $S.B$) and then use attribute renaming twice in the rewriting of the queries using the CCDs. This is shown in both Figure 4 and Figure 5. In Figure 4, the two queries Q_1 and Q_2 are rewritten using V_1. In Figure 5, they are rewritten using V_2.

However, for the previous two queries Q_1 and Q_2, we can also find the common subexpression V' shown in Figure 6. Symbol * is an abbreviation for all the attributes of a relation. Observe that V' has 4 relation occurrences in stead of 3, V' can be used for rewrite Q_1 minimally, but cannot be used for rewriting Q_2 minimally. This is because V' contains two occurrences of R while Q_2 contains only one occurrence of R. Notice though that V' can be used for rewriting both queries. Figure 7 shows rewritings of Q_1 and Q_2 using V'.

It is possible that V' is smaller than CCD_1 and CCD_2. In this case, if the goal is to reduce the materialization space, it is better to use V' as a materialized view for rewriting Q_1 and Q_2. Materializing V' will bring more benefit per space unit. Even if V' is larger than CCD_1 and CCD_2, it might be beneficial to consider V' instead of CCD_1 and CCD_2 because V' comprises one more join operation. This increases the possibility to find more useful common subexpressions between V' and other queries in the workload. Next we relax the definition of [23] so that a minimal rewriting is not a necessary requirement for a CCD.

3. DEFINITION AND COMPUTATION OF CCDS

In this section, we provide a definition for a CCD. Then we give an algorithm to compute CCDs. We also show how queries can be rewritten using their CCDs, which is part of the search space construction.

3.1 CCD Definition

If a query Q can be rewritten using a view V, we call view V a *subexpression* of Q.

DEFINITION 1. *A view V is a CCD of queries Q_1 and Q_2 if and only if:*

1. *V contains no redundant attribute*
2. *V is a subexpression of Q_1 and Q_2*
3. *There exists not another subexpression of Q_1 and Q_2, V' such that V is a subexpression of V while V' is not a subexpression of V.*

3.2 CCD Computation

To compute the CCDs of two queries, we map nodes of one query to those of the other so that the induced edge mapping associates join edges of the two queries whose conditions are mergeable [23]. Two join conditions are mergeable if there is another join condition that is implied by each of them. Clearly this is possible only if a node of one query is mapped to a node of the other query labeled by the same relation. The mapped nodes and edges in each one of the two queries should form a connected component. CCDs correspond to mappings that cannot associate more edges of the two queries.To compute CCDs according to the definition in [23], the computation of one-to-one mappings between the nodes of the two queries is sufficient.

With the definition of CCD in this paper, one node in a query can be mapped to more than one node in the other query. This is based on the observation that a query can be rewritten equivalently by splitting a relation occurrence. For example, R can be rewritten as $R \rhd \lhd_{*=*} R$. An example of occurrence splitting is shown in Figure 8 with query graph. Note that in this example, we have a new edge between the original occurrence node and the splitting node with edge condition $* = *$. This condition denotes a conjunction of equalities between all attributes with the same names from the two relation occurrences. We refer to this edge as an all-attribute-equal edge. We depict it by dashed line. If there is an edge (S, R) between a node S and the split node R, then we may add an edge (S, R') between S and the new node R' with the same condition as that of edge (S, R).

Before introducing the process of finding CCDs of two queries, we show how a common subexpression of them can be extended to contain one more edge. If this common subexpression can not be extended any more, then it is a CCD.

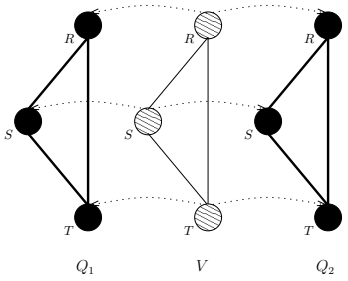


Figure 9: One to one node and edge mapping

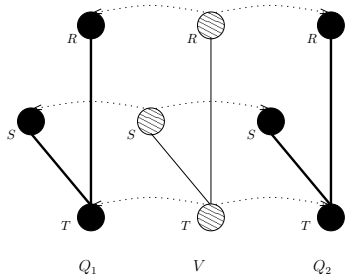


Figure 10: One to one node and edge mapping

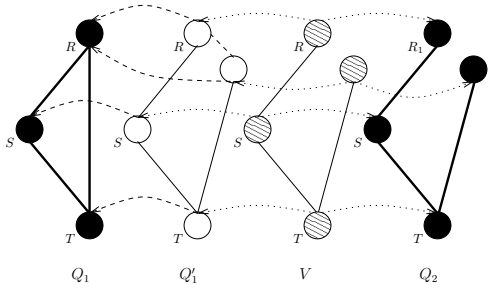


Figure 11: One node splitting

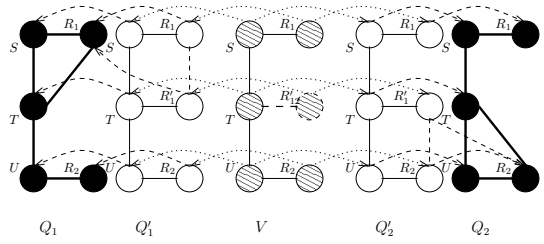


Figure 12: Two nodes splitting

Assume we have a common subexpression V of queries Q_1 and Q_2. Initial V contains only one node. For a node N in V, find its corresponding nodes N_1 in Q_1 and N_2 in Q_2 according to the mapping. For an edge $e_1 = (N_1, N_1')$ in Q_1, find an edge $e_2 = (N_2, N_2')$ from Q_2 that is mergable to e_1. Let C_m be the merge condition. Based on whether e_1 and e_2 have been mapped and whether N_1' and N_2' has been mapped, we have the following extension rules:

1. Edges e_1 and e_2 have not been mapped in V:

   (a) Rule 1: If N_1' and N_2' have been mapped to the same node N' in V, add the merging edge (N, N') with condition C_m to V. An example of an application of this extension rule is shown in Figure 9. Initially edges (S, T) and (R, T) have been mapped and Rule 1 adds edge (S, R) to V.

   (b) Rule 2: If neither N_1' nor N_2' has been mapped in V, add a new node N' and map N_1' in Q_1 and N_2' in Q_2 to N'. Add a new edge (N, N') with condition C_m to V. An example of an application of this extension rule for edge (T, R) is shown in Figure 10.

   (c) Rule 3: If N_1' has been mapped to N' in V while N_2' has not been mapped, rewrite Q_1 as follows: split node N_1' to get a new node N_1'', add a virtual edge (N_1, N_1'') and an all-attribute-equal edge (N_1', N_1''), remove edge (N_1, N_1'). After the rewriting, map edge (N_1, N_1'') in the rewriting Q_1' of Q_1 to (N_2, N_2') in Q_2 with extension Rule 2. An example of an application of this extension rule is shown in Figure 11.

   (d) Rule 4: If N_1' in Q_1 maps to N_1' in V and N_2' in Q_2 maps to N_2' in V, rewrite Q_2 as follows: split node N_2' to get a new node N_2'', add the virtual edge (N_2, N_2'') and the all-attribute-equal edge (N_2', N_2''), remove edge (N_2, N_2'). After the rewriting, map edge (N_1, N_1') in Q_1 to (N_2, N_2'') in the rewriting Q_2' of Q_2 using extension Rule 3. This will generate a node splitting in Q_1 similar to that in Q_2. An example of an application of this extension rule is show in Figure 12.

2. Edge e_1 in Q_1 has been mapped to $e = (N, N')$ in V while e_2 in Q_2 has not been mapped:

   (a) Rule 5: If N_2' in Q_2 has not been mapped, rewrite Q_1 to Q_1' as follows: split node N_1' to get a new node N_1'', and add to Q_1 an all-attribute-equal edge (N_1', N_1'') and a virtual edge (N_1, N_1''). After this rewriting, map edge (N_1, N_1'') Q_1' to (N_2, N_2') in Q_2 with extension Rule 2. An example of an application of this extension rule is show in Figure 13.

   (b) Rule 6: If N_1' in Q_1 has been mapped to N_1' in V, while N_2' in Q_2 has been mapped to N_2' in V, rewrite Q_2 to Q_2' as follows: split node N_2' to get a new node N_2'', add to Q_2 an all-attribute-

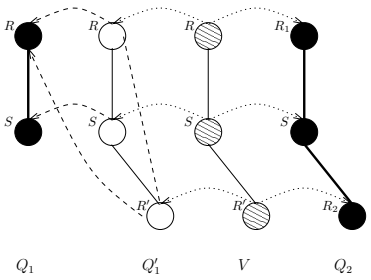


Figure 13: One node splitting

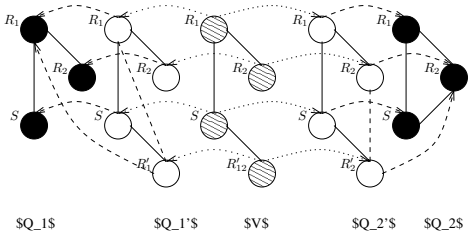


Figure 14: Two nodes splitting

equal edge (N_2', N_2'') and a virtual edge (N_2, N_2''), and remove edge (N_2, N_2') from Q_2. After the rewriting, map edge (N_1, N_1') in Q_1 to (N_2, N_2'') in Q_2' using extension Rule 3. An example of an application of this extending rule is show in Figure 14.

All the above 6 extension rules map regular edges and not virtual or all-attribute-equal edge. There may be cases where mapping those edges with regular edges generates some useful CCDs. For simplicity here, we ignore this kind of mappings.

Note that extension Rules 1 and 2 apply one to one node mapping; extension Rules 3 and 4 apply one to many node mapping, but only one to one edge mapping; extension Rule 5 and 6 apply one to many edge mapping (and consequently one to many node mapping). Using the previous extension rules, we can specify our CCD computation process as follows:

Algorithm 1 CCD Computation

1: Put both Q_1 and Q_2 into full form [23]
2: Find all possible single node mappings between Q_1 and Q_2 to create initial common subexpressions V that contains only one node
3: **for** each V **do**
4: extend V in all possible ways using the previous extension rules until it can not be extended anymore. Return the resulting V
5: **end for**

Using extension rules 1 and 2, we get CCDs exactly the same as they are defined in [23]. Using additionally Rules 3-6 makes the process more expensive since more possible extensions are considered. To reduce the execution time, one possible heuristic disallows extending rules 3-6 if one common subexpression can be extended using Rule 1 or 2.

3.3 Rewriting the Queries Using the CCDs

The general problem of rewriting queries using views is NP-hard. This is even true for simple queries and views such as SPJ queries with self-joins, the case we are considering in this paper. However, here we consider only simple rewritings, which means that queries are rewritten using only one occurrence of a view. Further, since we have a mapping function from all occurrence nodes of the view to occurrence nodes of the query, (generated during the computation of the CCDs), the rewriting is straightforward. The process is outlined below:

1. Construct the relation occurrence set: Put an occurrence of the CCD to the occurrent set of the rewritten query. Put also to the occurrence set of the rewritten query all the relation occurrences in the query which have not been mapped to a node in the CCD. Note that all occurrence names must be unique in the constructed occurrence set.

2. Construct the projected attribute list: For each relation occurrence in the query which has been mapped to a node in the CCD, list its projected attributes after the CCD occurrence. List all other projected attributes in the query after their original relation occurrences. all projected attributes on occurrences which has a mapped node in CCD, list the attribute under the occurrence of CCD. Note that in the former case, one attribute may be derived from another attribute listed in the CCD. In this case, we need attribute renaming to recover the projected attribute of the query in the rewriting.

3. Construct the predicate set: For edges between occurrences (join edges) or on one occurrence (loop edges) in the query that have not been mapped to occurrences in the CCD, add edges with the same condition between the corresponding occurrences in the rewriting of the query. For edges between an unmapped occurrence and a mapped occurrence, add an edge with the same condition between the corresponding occurrence and the CCD occurrence in the query rewriting. For edges between mapped occurrences, add a loop edge to the CCD occurrence in the query rewriting with the same condition. If the edge already exists, just add the condition to the edge.

Figures 4, 5 and 7, show some examples of rewriting queries using views.

4. EXPERIMENTAL RESULTS

We ran experiments to show the effectiveness of the new definition of CCDs. We refer to the definition of CCD in [23] as one-to-one CCD because it is constructed by allowing each occurrence node from one query to be mapped to at most one occurrence node from the other query. We refer to the new definition of CCD as many-to-many because it allows many occurrence nodes from one query to be mapped to many occurrence nodes of the other query. In the experiments, we initially generate a schema which includes a set of base relations. Based on this schema, we generate a series of workload queries (20 queries in each workload). For a given workload, we compute all possible CCDs between each pair of queries. Then, for each one of the two CCD definitions, we measure the average time for computing the CCDs of a pair, the average number of CCDs generated for a pair, and the average number of occurrences in each CCD. Each workload follows the following rules:

- Each query contains m base relations.

- Each query contains n relation occurrences, $n \geq m$. In general, the larger the number of occurrences for a fixed number of relations, the higher the number of node mappings between the two queries.

- Approximately, $n/3$ of the queries have selection conditions in every workload.
- In order to avoid having query graphs that are too dense or too sparse, each one of them has in it slightly over 20% of all possible edges in the query graph.

In the first experiment, we fix n to 8 and vary m from 7 down to 5. The result is shown in Figure15. In the second experiment, we fix the difference of n and m to 3 and vary n from 6 to 10. The result is shown in Figure 16. One can see that for the same values of m and n, the number of many-to-many CCDs is less than the number of one-to-one CCDs, and the number of occurrences per many-to-many CCD is larger than the number of occurrences per one-to-one CCD. These remarks show that the new CCD incorporates more operations. This is expected to improve the overall cost of evaluating all the input queries together. Further, the many-to-many CCD gives less options to view selection algorithms since less CCDs are generated. Therefore, the many-to-many CCD is expected to increase the speed and accuracy of view selection algorithms in determining the optimal solution.

When the ratio of relations to occurrences of relation is high (75% or higher), the computation time of the CCDs for the two approaches is similar (e.g. column sets 1 and 2 in Figure 15(a), and column set 3 in Figure 16(a)). When this ratio drops bellow 75% the computation time of the many-to-many CCD is much higher (e.g. column set 3 in Figure 15(a), and column sets 1 and 2 in Figure 16(a)). However, this loss in CCD computation time might be compensated by the important gains in time of the view selection algorithm (due to the reduction in the number of many-to-many CCDs - Figure 15(b) column set 3 and Figure 16(b) column sets 1 and 2). Therefore, when the ratio drops bellow 75%, the type (e.g. greedy, heuristic) and the speed of the employed view selection algorithm will determine whether the one-to-one or the many-to-many CCD definition is preferable.

5. SUMMARY AND FUTURE WORK

We have defined CCDs of queries. CCDs are uesd for constructing search spaces for view selection problems that involve multiple queries. Our definition extends a previous one to allow many to many mappings of relation occurrences in queries. It is particularly useful for query workloads that involve also self-joins. An experimental evaluation shows that our approach produces CCDs that involve more relational occurrences. Therefore, it increases the chances for these CCDs to be exploited in computing other input queries. It also reduces the size of the search space to be exploited by cost based view selection algorithms.

Although it is not difficult to construct a search space for queries that involve exclusively grouping/aggregation operations (data cube), it is a hard problem to construct a search space for generic queries which include both SPJ and grouping/aggregation operations. In this paper, we considered SPJ queries only. In the future, we will extend the concept of CCD to queries that include both SPJ and grouping/aggregation operations.

Database schemas include not only definitions of relations, but also integrity constraints such as keys, foreign keys or general check constraints. It is obvious that these constraints can play a role in detecting common subexpressions between queries. In the future, we will also consider taking into account existing constraints on the schema in computing the CCDs of a query workload.

6. REFERENCES

[1] Serge Abiteboul and Oliver M. Duschka. Complexity of answering queries using materialized views. In *Proc. of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1998.*

[2] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *Proc. of the 26th International Conference on Very Large Data Bases, 2000.*

[3] Dimitri Theodoratos and Timos Sellis. Data Warehouse Configuration. In *Proc. of the 23rd International Conference on Very Large Data Bases, 1997.*

[4] Randall G. Bello, Karl Dias, Alan Downing, James J. Feenan Jr., James L. Finnerty, William D. Norcott, Harry Sun, Andrew Witkowski, and Mohamed Ziauddin. Materialized views in oracle. In *Proc. of the 24rd International Conference on Very Large Data Bases, 1998.*

[5] Fa-Chung Fred Chen and Margaret H. Dunham. Common Subexpression Processing in Multiple-Query Processing. *IEEE Trans. Knowl. Data Eng.*, 1998.

[6] Zhimin Chen and Vivek R. Narasayya. Efficient computation of multiple group by queries. In *Proc. of the ACM SIGMOD International Conference on Management of Data, 2005.*

[7] Sheldon J. Finkelstein. Common subexpression analysis in database applications. In *Proc. of the ACM SIGMOD International Conference on Management of Data, 1982.*

[8] Himanshu Gupta. Selection of views to materialize in a data warehouse. In *Proc. of the 6th International Conference on Database Theory, 1997.*

[9] Himanshu Gupta and Inderpal Singh Mumick. Selection of views to materialize under a maintenance cost constraint. In *Proc. of the 6th International Conference on Database Theory, 1999.*

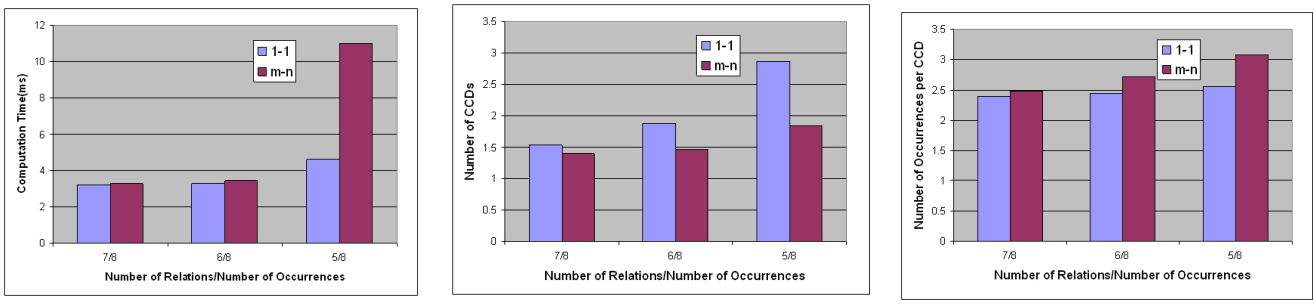
[10] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *Proc. of the ACM SIGMOD International Conference on Management of Data, 1996.*

[11] Matthias Jarke. Common subexpression isolation in multiple query optimization. In *Query Processing in Database Systems.* Springer, 1985.

[12] Panos Kalnis and Dimitris Papadias. Optimization algorithms for simultaneous multidimensional queries in olap environments. In *Proc. of the Third International Conference on Data Warehousing and Knowledge Discovery, 2001.*

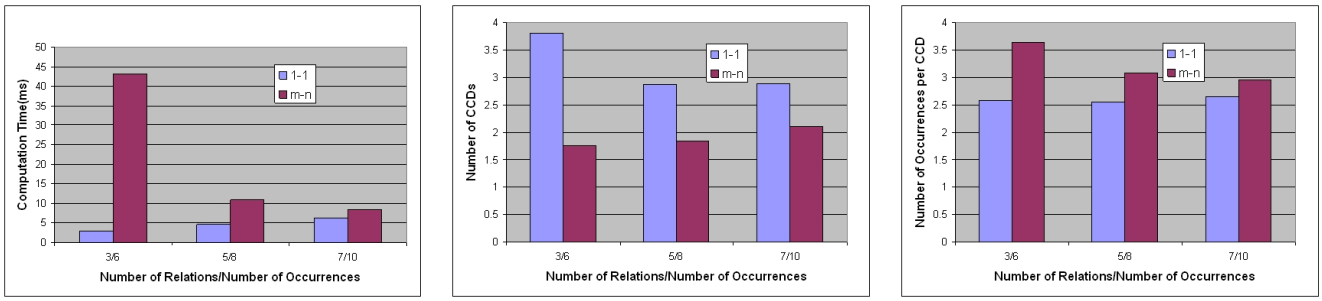
[13] Wolfgang Lehner, Roberta Cochrane, Hamid Pirahesh, and Markos Zaharioudakis. fAST Refresh Using Mass Query Optimization. In *Proc. of the 17th International Conference on Data Engineering, 2001.*

[14] Jingni Li, Zohreh Asgharzadeh Talebi, Rada Chirkova, and Yahya Fathi. A formal model for the problem of view selection for aggregate queries. In *Proc. of the 9th East European Conference on Advances in Databases and Information Systems, 2005.*



(a) (b) (c)

Figure 15: Effectiveness of CCD definitions



(a) (b) (c)

Figure 16: Effectiveness of CCD definitions

[15] Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *Proc. of the ACM SIGMOD International Conference on Management of Data, 2001.*

[16] Rachel Pottinger and Alon Y. Levy. A scalable algorithm for answering queries using views. In *Proc. of 26th International Conference on Very Large Data Bases, 2000.*

[17] Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proc. of the 1996 ACM SIGMOD International Conference on Management of Data, 1996.*

[18] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proc. of the ACM SIGMOD International Conference on Management of Data, 2000.*

[19] Timos K. Sellis. Multiple-Query Optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[20] Amit Shukla, Prasad Deshpande, and Jeffrey F. Naughton. Materialized view selection for multidimensional datasets. In *Proc. of 24rd International Conference on Very Large Data Bases, 1998.*

[21] Yingying Tao, Qiang Zhu, and Calisto Zuzarte. Exploiting common subqueries for complex query optimization. In *Proc. of the conference of the Centre for Advanced Studies on Collaborative Research, IBM, 2002.*

[22] Dimitri Theodoratos and Mokrane Bouzeghoub. A General Framework for the View Selection Problem for Data Warehouse Design and Evolution. In *Proc. of the ACM International Workshop on Data Warehousing and OLAP, 2000.*

[23] Dimitri Theodoratos and Wugang Xu. Constructing Search Spaces for Materialized View Selection. In *Proc. of the ACM International Workshop on Data Warehousing and OLAP, 2004.*

[24] Wugang Xu, Calisto Zuzarte, and Dimitri Theodoratos. Preprocessing for fast refreshing materialized views in DB2. In *Proc. of the 8th International Conference on Data Warehousing and Knowledge Discovery, 2006.*

[25] Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, and Monica Urata. Answering complex SQL queries using automatic summary tables. In *Proc. of the ACM SIGMOD International Conference on Management of Data, 2000.*