Using Patterns for Faster and Scalable Rewriting of Conjunctive Queries

Ali Kiani and Nematollaah Shiri

Dept. of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada
Email:{ali_kian,shiri}@cse.concordia.ca

Abstract. Rewriting of conjunctive queries using views has many applications in database and data integration. We investigate ways to improve performance of rewriting and propose a new algorithm which has two phases. In the first phase, similar to Minicon, we find mapping information, which we call *coverages*, from subgoals in the query to subgoals in view, and assign positive integers ($< 2^n$) as identifiers to these coverages, where n is the number of subgoals in the query. In the rewriting phase, based on the available identifiers and partitions of the set $\{1, ..., 2^{n-1}\}$, we define *patterns* and use them to encode the buckets and the coverages they contain. This breaks the cartesian product of a set of large buckets into several cartesian products on sets of smaller buckets. In other words, an expensive cartesian product could be broken into a maximum of $B(n)$ small cartesian products, where B is the Bell number. Our numerous experiments using different query types and sizes indicate significant time and space improvement for computing the cartesian products of the buckets and generating the output.

1 Introduction

Query rewriting (QR) is a well-known problem in databases and data integration and has been the subject of numerous studies [5, 9, 1, 8, 6, 2, 4, 7]. Given a conjunctive query Q and a set of views V, a query rewriting algorithm uses the views to generate a rewriting R which is a union of queries each of which is contained in Q. Due to restrictions in the mappings (view definitions), it is not always possible to generate a complete rewriting (i.e., a rewriting based on views only) that is equivalent to the original query Q. The goal of rewriting is to return maximally contained rewritings (MCR). That is, on every database D, the result $R(D)$ is contained in $Q(D)$, denoted as $R \sqsubseteq Q$. Further, R contains every other contained rewriting of Q.

In this paper, we investigate ways to improve performance of query rewriting and propose a pattern-based algorithm which exploits *Bell* (*Stirling*) numbers to generate rewriting more efficiently. We evaluate the performance of our algorithm in terms of speedup and space requirement and compare it with Minicon [8] and Treewise [7] algorithms.

After finding mappings from the subgoals in the query to view, a major part of a rewriting process is spent in performing the cartesian products of a set of large buckets that contain these mappings, called Minicon Descriptions (MCDs) in Minicon. The idea in our algorithm is to break this set of large buckets into several smaller buckets so that the cartesian products are in smaller scale. Also important in practice, this helps finding the number of rules in a rewriting before generating them.

The rest of this paper is organized as follows. We next provide some background and review related work. In Section 3, we introduce our rewriting algorithm and discuss its complexity. Section 4 presents the experiments and results. We provide concluding remarks in Section 5.

2 Background and Related Work

A conjunctive query Q is a statement of the form: $Q: \ h(\bar{X}) \ :-g_1(\bar{X_1}), \ldots,$ $g_k(\bar{X_k})$, where $g_i(\bar{X_i})$ are ordinary predicates (also called subgoals), $\bar{X_i}$ is a sequence of variables, and the head predicate h does not appear in the body [3, 10]. A variable X in the rule body is distinguished if it also appears in the head.

Example 1. Let $r(A,B)$ and $s(C,D,E)$ be relations. Consider the following query and views:

$Q: \quad h(X,Y)$:- $r(X,Y),\ s(Y,Z,W).$
$V_1: \quad v_1(A,B)$:- $r(A,B).$
$V_2: \quad v_2(B,D)$:- $s(B,D,D).$

Rule R below is a contained rewriting for Q. The reason is that if we unfold views V_1 and V_2, i.e., replacing in the body of Q the views by their definitions, we get a query which satisfies the containment $R \sqsubseteq Q$.

$R: \quad h(X,Y)$:- $v_1(X,Y),\ v_2(Y,Z).$

In order to generate rewriting, an algorithm should in general consider different combinations of view heads and ensure to produce MCR.

Corresponding to Minicon descriptions (MCDs), we define *Coverage* which is a data structure of the form $C{=}{<}S,\phi,h,\delta{>}$ where S is a subset of the subgoals in query Q, ϕ is a mapping from S to a subset T of subgoals in view v_i, h is the head of v_i, i.e., $v_i(\bar{X_i})$, and δ is a variable substitution that unifies every group of variables in ϕ that are mapped to the same variable. Intuitively, when generating rewriting, we can remove subgoals of S from Q, put a specialization of v_i head in the query, and apply δ on Q so that the resulting query becomes contained in Q. The specialization of v_i is generated by applying the inverse of ϕ on $\delta(v_i(\bar{X_i}))$.
In example 1, there are coverages $C_1{=}{<}\{r(X,Y)\},\{X/A,Y/B\},v_1(A,B),\{\}{>}$, and $C_2{=}{<}\{s(Y,Z,W)\},\{Y/B,Z/D,W/D\},v_2(B,D),\{W/Z\}{>}$, where C_1 and C_2 generate specializations $v_1(X,Y)$ and $v_2(Y,Z)$, respectively.

Based on this, we can say that there are two phases in query rewriting, (1) finding coverages and (2) considering all possible/valid combinations of coverages to form contained rules and getting the union of all such rules to generate the maximally contained rewriting. Different algorithms use different terms to refer to the result of phase 1, e.g., *Bucket* in Bucket algorithm, *MCD* in Minicon algorithm, Quadruple in [4], Tuple in Treewise, to all of which we refer as coverage. The Bucket algorithm [8] tests containment for every combination. As shown in [8], this can be avoided by choosing proper coverages and combining them efficiently. In fact, the criteria for choosing coverages should be such that they guarantee containment. To see how, we define the notion of *Joint variables* as follows.

Let S be a subset of subgoals in the body of Q and S' be the rest of subgoals in Q including the head. We call the variables appearing in both S and S' as *joint variables* in S. Intuitively, if there is a view V_i covering S, then for V_i to contribute in a rewriting (i.e., yield a coverage) the joint variables in S must be distinguished in V_i. We will also consider the notion of partial containment mapping from query Q_1 to query Q_2 where not every subgoal of Q_1 is required to be mapped. A coverage C, defined based on V_i, is a useful coverage if the joint variables of subgoals S of C are distinguished in V_i.

After forming the coverages, we combine coverages in the second phase to generate a rewriting. It it shown in [8] that combinations of coverages that do not have overlap are useful only. The challenge here is how to find such combinations efficiently?

Example 2. Let $r(A, B)$, $s(C, D)$, and $t(E, F)$ be relations. Consider the following query and views:

$$\begin{array}{ll} Q: & h(A) \text{ :- } r(A,B),\ s(B,D),\ t(D,E). \\ V_1: & v_1(A,D) \text{ :- } r(A,B), s(B,D). \\ V_2: & v_2(B) \text{ :- } s(B,D),\ t(D,E). \\ V_3: & v_3(A,B,D) \text{ :- } r(A,B),\ t(D,E). \\ V_4: & v_4(A,B,D) \text{ :- } r(A,B), s(B,D),\ t(D,E). \\ V_5: & v_5(A) \text{ :- } r(A,B), s(B,D),\ t(D,E). \end{array}$$

The coverages we can create for this example are as follows: C_4 covering $\{r\}$, C_2 covering $\{s\}$, and C_1 covering $\{t\}$ based on V_4, C_6 covering $\{r, s\}$ based on V_1, C_3 covering $\{s, t\}$ based on V_2, C_5 covering $\{r, t\}$ based on V_3, C_7 covering $\{r, s, t\}$ based on V_5. As shown in Fig. 1, there are three buckets in this example, one for each subgoal in Q, and the coverages in the buckets have overlaps. Note that the numbering of these coverages is based on the buckets they appear in. For example, since coverage C_6 appears in the bucket of r (position 2 from right to left in query), and also in bucket of s (position 1), we have assigned number 6 $(= 2^2 + 2^1)$ to C_6.

Consider coverages C_6 and C_3. It is easy to see that we cannot combine C_6 and C_3 to generate a rule in the rewriting, since there is no way to match subgoal s from C_6 with subgoal s from C_3. In fact we can see that the binary representation of 6 (110) and 3 (011) have a common 1 in the second bit. This is the basic idea in our algorithm that assigns numbers to coverages in such a way that their bitwise comparison indicate whether or not their combination is useful.

3 Pattern-Based Query Rewriting

Consider a conjunctive query Q with n subgoals and a set of views. In order to generate rewriting for Q, we first find all coverages, and place each in corresponding buckets. To find all coverages, we consider a set S with a single subgoal sg_i (from Q) and its join variables J_S in Q. For every view V_j that includes sg_i, we check if all the variables in J_S are distinguished in V_j. If this is the case, we create a new coverage C_{ji} based on V_j and assign S to C_{ji}. Otherwise, we add more subgoals to S, update J_S accordingly, and check if V_j can be useful in forming a coverage. In order to add subgoals to S we use the join variables that are not distinguished. If $A \in J_S$ is not distinguished, we include all the subgoals in Q in which A appears. After adding the new subgoals to S, we recompute J_S and repeat the test. This process continues until we find a useful coverage for sg_i (and possibly some other subgoals), or we fail. This is very similar to Minicon when creating MCDs. Our algorithm mainly differs from other rewriting algorithms in the combining step and hence we focus more on this step.

A significant amount of time in most query rewriting algorithms including Minicon and Treewise is spent to discard combinations with overlaps. The reason is that these algorithm need to consider n^m possible combinations where m is the number of coverages in each bucket. In our algorithm, we introduce a new technique to perform this step more efficiently, described as follows.

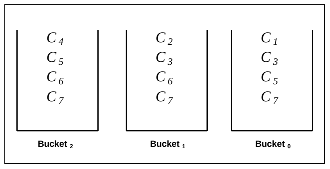

Fig. 1. Occurrences of coverages for a query with 3 subgoals; Existing algorithms need 4^3 cartesian product operations to find all rules in the rewriting.

Let us consider all possible combinations. We consider every bucket as a bit in a binary representation of n bits, and hence every coverage consists of n bits. If a coverage C_j is placed in a bucket B_i, the ith bit in the sequence denoting coverage C_j is 1; otherwise it is 0. The sequence corresponding to a coverage indicates its presence in different buckets, which we consider as an identifier for that purpose. The maximum number of sequences is $2^n - 1$, where n is the number of subgoals in the query; the sequence with all zeros is excluded.

We define *occurrence classes* based on occurrence identifier. All coverages with the same identifier are members of the same occurrence class. In other words, an occurrence identifier defines an equivalence class on coverages. Assuming that there is only one coverage from every occurrence class, Fig. 1 shows all possible occurrences of coverages for a query with 3 subgoals. This figure also shows the bucket structure for the query and views in Example 2. Note that to find the rewriting in this example, existing algorithms would perform 4^3 cartesian products. We show that our algorithm breaks this relatively expensive operation to 5 smaller cartesian products.

Considering these coverages, we need to identify all sets of coverages that do not have overlap and cover all the subgoals in the query.

3.1 Finding All Occurrence Classes: A Partitioning Problem

Assume that there are n subgoals in a query, and subgoals are indexed based on their position in the query, with the rightmost subgoal as 0 and the leftmost as $n-1$. For every position i, we consider number 2^i and include it in a set P. We then find all partitions of P. The following example lists all possible partitions for a query with 3 subgoals.

Example 3. Let Q be a query with three subgoals. So, there will be positions 0, 1, and 2 and hence, $P = \{2^2, 2^1, 2^0\} = \{4, 2, 1\}$. There are 5 partitions of P: $P_1 = \{\{4, 2, 1\}\}$, $P_2 = \{\{2, 1\}, \{\{4\}\}$, $P_3 = \{\{4, 2\}, \{1\}\}$, $P_4 = \{\{4, 1\}, \{2\}\}$, and $P_5 = \{\{4\}, \{2\}, \{1\}\}$.

It is easy to see that the partitions do not have overlap and their union forms the original set P. Moreover, the sum of the numbers in a set in a partition gives an occurrence identifier. In this example, we have occurrence identifiers 1, 2, 3, 4, 5, 6, and 7 from $\{1\}$, $\{2\}$, $\{2, 1\}$, $\{4\}$, $\{4, 1\}$, $\{4, 2\}$, and $\{4, 2, 1\}$, respectively. That is, the maximum number of occurrence identifiers for a query with n subgoal is $2^n - 1$, and finding the proper combinations is a partitioning problem.

We know that the number of partitions of a set of size n is the Bell number, denoted $B(n)$. We can also find the maximum number of rules in a rewriting of Q. If there is only one coverage from every occurrence class, then the number of rules in the rewriting is $B(n)$. In general, the number of coverages for occurrence class could be any number. To define the maximum number of rules, we use Stirling numbers as follows. Stirling numbers of the second type $S(n, k)$ indicate the number of ways to partition a set of n elements into k nonempty subsets. Using this, the Bell number $B(n)$ can be defined as $B(n) = \sum_{k=1}^{n} S(n, k)$

We can use Stirling numbers to define maximum number of rules in the rewriting of a query Q with n subgoals. Suppose for every occurrence class, there are m coverages. Then, the maximum number of rules in the rewriting of Q would be $\sum_{k=1}^{n} m^k.S(n, k)$. In fact, $S(n, k)$ gives the number of cases in which k different occurrence classes together cover the body of Q without overlap. This is important as it can be used to perform the combining phase more efficiently. Also it gives the maximum number of rules in a rewriting which helps provide a more precise upper bound for the complexity of query rewriting.

We next define the notion of *Combination Pattern* based on the occurrence identifiers in each partition. Intuitively every pattern shows how many buckets we need and what coverages (based on occurrence identifier) they should contain. The maximum number of patterns for a query with n subgoals is $B(n)$.

Definition 1. *[Combination Pattern] For a query with* n *subgoals, a set of positive integers* $\mathbb{P}$ *is a Combination Pattern, if it satisfies the following properties.*
1. $\mathbb{P} \subseteq \{1, \ldots, 2^n - 1\}$, *2.* $\sum_{i \in \mathbb{P}} i = 2^n - 1$,
3. $\forall\ i, j\ \in \mathbb{P},\ i \wedge j = 0$, *where* $\wedge$ *is the extended bitwise operation on integers.*

The complexity of finding combination patterns is exponential, as it is based on computing $B(n)$, which is exponential. For example, the number of combination patterns for queries with 5, 10, 15 subgoals are 52, 115975 and 1382958545, respectively. However, when not all of the identifiers are present, the number of patterns reduces significantly. For lack of space, we do not provide details of finding combination patterns for a set of identifiers; it could be done using a partitioning algorithm. In our implementation, we sort the list of identifiers and use recursion which was quite fast for our purpose. In order to combine the coverages using the idea of *Combination Pattern*, we assign to query a number $2^n - 1$, and assign *occurrence identifier* to every coverage C_i which is based on the subgoals it contains and their position in the query body. For instance, in Example 2, occurrence identifiers assigned to C_1 and C_2 would be 6 $(= 2^2 + 2^1)$ and 3 $(= 2^1 + 2^0)$, respectively. Let N be the sorted list of all occurrence identifiers. We compute the list of patterns based on available coverages. For each combination pattern, we create a set of buckets, assign coverages to them accordingly, and perform a simple cartesian product. Since a combination pattern does not have overlap, no further checking is required. Also, we know that the sum of the identifiers in every pattern is $2^n - 1$. For example, consider the coverages listed in Fig. 1, for which based on the partitions in Example 3, we break the original bucket into smaller buckets and perform 5 different cartesian products since there are 5 different combination patterns, listed as follows.

1. $\mathbb{P}_1 = \{\{7\}\}$: We have a single bucket $[C_7]$ so no need for cartesian product
2. $\mathbb{P}_2 = \{\{3\}, \{4\}\}$: Cartesian product of two buckets: $[C_3] \times [C_4]$
3. $\mathbb{P}_3 = \{\{6\}, \{1\}\}$:Cartesian product of two buckets: $[C_6] \times [C_1]$
4. $\mathbb{P}_4 = \{\{5\}, \{2\}\}$: Cartesian product of two buckets: $[C_5] \times [C_2]$
5. $\mathbb{P}_5 = \{\{4\}, \{2\}, \{1\}\}$: Cartesian product of three buckets: $[C_4] \times [C_2] \times [C_1]$

In Example 2, there are 5 sets of buckets, one of size 1, three of size 1×1, and one of size $1 \times 1 \times 1$. To better understand our algorithm, we add the following view (which is similar to V_4) to the list of views in Example 2 .

V_4' : $v_4'(A, B, D)$:- $r(A, B), s(B, D),\ t(D, E)$.

Since we consider open world assumption, we need to consider V_4' in the rewriting even though it has the same definition as V_4. When finding coverages, V_4' would create C_4' covering $\{r\}$, C_2' covering $\{s\}$, and C_1' covering $\{t\}$, i.e., we get a new coverage in every bucket. In this case, other algorithms would consider a cartesian product of size 5^3, while our algorithm would consider the cartesian product based on the pattern $\mathbb{P}_5$ for which it would perform a product of size $2 \times 2 \times 2$. To summarize, next, we briefly explain our pattern-based query rewriting algorithm.

Consider Q, the user query and V, the set of given views as the input. The goal is to generate R, the maximally contained rewriting of Q using the views, as the output. The algorithm has two phases:

Phase 1: Finding coverages: For every view v_i, find the coverages that v_i can generate for subgoals in Q. Assign the occurrence identifier to each coverage and maintain a set of available occurrence identifiers.

Phase 2: Combining Coverages: Based on the set of available occurrence identifiers, find the patterns. For every pattern, form the buckets, place the related coverages in these buckets, and perform the cartesian product. Every combination in the result of the cartesian product generates a rule. Assign the union of all these rules to R and return R as the output.

3.2 Complexity of Query Rewriting

Based on the above algorithm, we provide a more accurate upper bound for the complexity of rewriting of conjunctive queries under the OWA and set semantics.

The complexity of query rewriting includes two parts. First, when finding coverages, we need to find mappings from subgoals in the query to those in views, which is NP-complete in the number of subgoals in the query. If the number of different predicates in the query and view V_i is a, and the number of subgoals from each predicate in the query and view V_i are b and c, respectively, then the maximum number of partial mappings (hence maximum number of coverages) from the query to V_i is $a.b.c$. The reason is that every subgoal in the query can be mapped to a maximum of c subgoals in the view. Since for every predicate name in query, there are b subgoals with that name, there will be $b.c$ partial mappings for each group of subgoals. Since there are a groups of subgoals, there will be a maximum of $a.b.c$ partial mappings each of which could generate one coverage.

The second part of the complexity is in combining coverages where we need to perform cartesian products over the buckets. For that we use combination patterns. Assuming that we have n subgoals in the query, in the worst case, the number of different set of buckets we form is $Bell(n)$. Since the cost of cartesian product operations on these sets of buckets can be expressed in terms of the number of rules they generate, the complexity of the second phase is $\sum_{k=1}^{n} m^k S(n,k)$, assuming the number of coverages in each bucket is m. Since the complexity of the second part is larger than the first part, we may ignore the first part. The reason is that the factor m in the formula for the second part is proportional to the number of mappings from the first part. Moreover, the growth of Stirling numbers is much faster than $a.b.c$. As a result, the upper bound for query rewriting is the complexity of the second phase, which is $\sum_{k=1}^{n} m^k S(n,k)$. Even though the complexity of this problem is known to be NP-complete, this result is important as it shows a more accurate upper bound and also it provides a way of finding the maximum number of rules in a rewriting before performing the cartesian product which is very useful in real life applications.

4 Experiments and Results

In this section, we report our experiments and results on query rewriting using our proposed algorithm and compare its performance with Minicon and Treewise algorithms. The reason for not considering other algorithms such as Bucket algorithm and Inverse rules in our experiments is that it has been shown that Minicon outperforms these algorithms [8]. In our experiments, we have used two sets of input queries, (1) “real” queries collected from papers and articles related to query rewriting and (2) synthetic queries. As the number of queries in the first category was not much, we mainly used them to check the output of our algorithm with human generated rewritings, as appeared in the papers. As synthetic data, we created different types and sizes of conjunctive queries including *Chain queries*, *Star queries*, *Duplicate queries*, and *Random queries*. We also introduced a new class of queries which we call as *All-Range queries*, which can generate all possible occurrence identifiers. This would create the worst case scenario for our algorithm. Example 2 is an instance of such a query and views.

We developed a running prototype of our Pattern-based query rewriting algorithm in Java. For the experiments, we used a regular desktop computer with Pentium 4, 1.73 GHz and 1GB RAM. This prototype is made available to the reviewers at http://users.encs.concordia.ca/˜ ali_kian/qr/.

Most of the test data were created by query generator which was made available to us for the experiments. For each type, we considered parameters such that we could compare our results with those reported in [8] and [7]. In order for the comparison to be fair and meaningful, we used the implementation of the Treewise and Minicon algorithms developed and used in [7]. Moreover, we also developed our version of Minicon and confirmed its identical performance with Minicon and Treewise [7]. For synthetic queries, we measure memory utilization, performance, and scalability of our pattern-based algorithm and compare them with Minicon and Treewise.

Fig. 2(a) compares these algorithms based on memory utilization. To push all these algorithms to their limits, we also used All-range queries and conducted experiments using different number of views. As shown in the figure, Pattern-based algorithm used the least amount of memory. At each step, the same input was used for all these algorithms. For 63 and 127 views (with 203 and 877 rules in the rewriting generated), all the algorithms completed the process using 16 MB of memory. For 255 views (4140 rules in the rewriting), Minicon used 64 MB, Treewise used 32 MB, and Pattern-based still using 16 MB. For 511 views (21147 rules in the rewriting), Minicon could not complete the task due to memory exception. For this case, Treewise and Pattern-based finished using 128 MB and 32 MB, respectively. We checked the buckets and found out for 511 views (and a query with 9 subgoals), Minicon formed 9 buckets each containing 256 coverages, i.e., it needed to perform 256^9 cartesian products. Treewise could finish this case because of it uses a tree structure to organize its run-time which helps prune away many unnecessary combinations. The case was easier for Pattern-based because it basically broke this cartesian product into 21147 of much smaller cartesian products. For 1023 views (115975 rules in the rewriting), only Pattern-based algorithm could finish the task for which it used 128 MB of memory. We continued with 2047 views (678570 rules in rewriting) for which Pattern-based completed the task using 448 MB of memory. For 4095 views where there were 4213597 rules in the resulting rewriting, Pattern-based could not finish the task.

To evaluate and compare the performance of the algorithms, we used different classes of queries including Chain, Star, Duplicate, Random, All-Range queries.

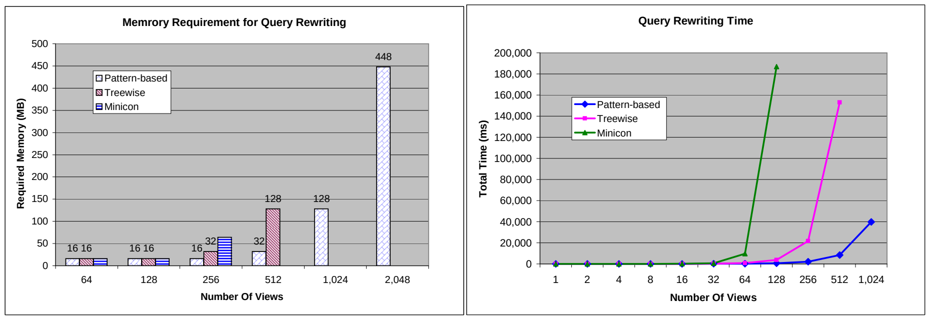

Fig. 2. (a) Memory requirement for Minicon, Treewise and Pattern-based Algorithms. (b)Performance on All-range queries

For each class, we used the same input to the algorithms and compared them in terms of rewriting time. For this, we performed a similar set of tests reported in [8] so that we can better compare the algorithms. In all these experiments, Pattern-based algorithm outperformed others significantly. Here we only report the results for Chain and All-Range queries, as performance of other queries we observed was similar to Chain queries. We also noted that when the number of partitions are more, pattern-based performs better than others. In no case the performance of our algorithm was inferior to the others.

Fig. 2(b) shows the rewriting time for All-Range queries. As we can see, the rewriting time for up to 32 views are almost the same for all these algorithms, however, for larger inputs, our Pattern-based algorithm outperforms others significantly. For instance, for 127 views, Pattern-based, Treewise, and Minicon took 625 ms, 3875 ms, and 134344 ms, respectively. For 255 views, Pattern-based completed in 2031 ms – almost 10 times faster than Treewise (21641 ms). Minicon could not even finish the task. As the input size increased the difference between Pattern-based and Treewise increased. For example, for 511 views, Pattern-based finished in about 8 seconds, whereas Treewise finished in 153 second.

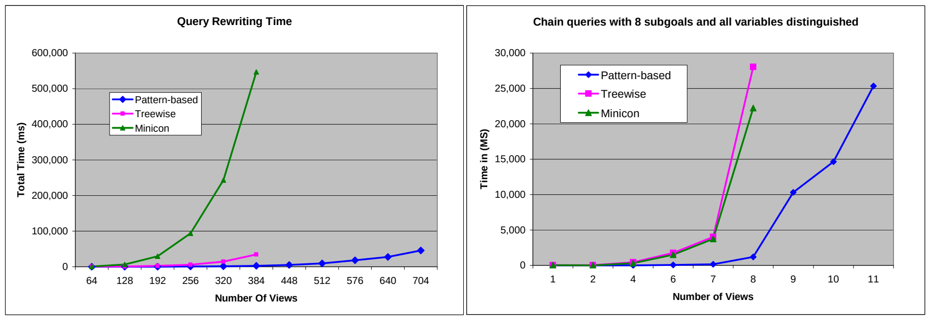

Fig. 3. (a) Scalability for All-Range queries with 6 subgoals and up to 20 repetition. (b) Scalability for queries with 8 subgoals and all variables distinguished

Fig. 3 (a and b) illustrates scalability of our algorithm for All-Range and Chain queries with 8 subgoals and all variables distinguished. As shown in the figure, only Pattern-based algorithm could process more than 256 views under 50 seconds for All-range queries. In fact, we continued increasing the number of views to more than 700 where our algorithm was still capable of generating rewritings in less than 50 seconds. Also, as shown in Fig. 3(b) for Chain queries, only Pattern-based could process more than 8 views.

5 Conclusion and Future Work

We studied rewriting of conjunctive queries using views and proposed a novel algorithm based on Bell numbers which outperforms current algorithms. We provided a more precise upper bound for the number of rules in a maximally contained rewriting. We are currently investigating incorporation of query minimization on both input and output of the query rewriting and study the impact of query minimization on performance and scalability of query rewriting, as well as the quality of the rules generated.

Acknowledgments: This work was partially supported by Natural Sciences and Engineering Research Council (NSERC) of Canada and by Concordia University. We thank Dr. Pottinger for providing us the query generator program, which we extended and used in our work.

References

1. Abiteboul, Serge and Duschka, Oliver. Complexity of answering queries using materialized views. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Seattle, WA, 1998.
2. Afrati, Foto; Li, Chen; and Mitra, Prasenjit. Answering queries using views with arithmetic comparisons. In *Proc. of the 21st ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM Press, 2002.
3. Chandra A.K. and Merlin P.M. Optimal implementation of conjunctive queries in relational databases. In *Proc. 9th Annual ACM Symp. on the Theory of Computing*, pages 77–90, 1977.
4. Kiani, Ali and Shiri, Nematollaah. Answering queries in heterogenuous information systems. In *Proc. of ACM Workshop on Interoperability of Heterogeneous Information Systems*, Bremen, Germany, Nov. 4, 2005.
5. Levy, Alon; Mendelzon, Alberto; Sagiv, Yehoshua and Srivastava, Divesh. Answering queries using views. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, San Jose, CA, 1995.
6. Levy, Alon Y. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.
7. Shiri, Nematollaah and Mohajerin, Nima. A top-down approach to rewriting conjunctive queries using views. In *SDKB '08: Workshop on Semantics in Data and Knowledge Bases*, 2008.
8. Pottinger, Rachel and Levy, Alon Y. A scalable algorithm for answering queries using views. *The VLDB Journal*, pages 484–495, 2000.
9. Rajaraman, Anand; Sagiv, Yehoshua and Ullman, Jeffrey D. Answering queries using templates with binding patterns. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, San Jose, CA, 1995.
10. Ullman, Jeffrey D. Information integration using logical views. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.