Join Minimization in XML-to-SQL Translation: An Algebraic Approach

Murali Mani Song Wang Dan Dougherty Elke A. Rundensteiner
Computer Science Dept, WPI
{mmani,songwang,dd,rundenst}@cs.wpi.edu

Abstract

Consider an XML view defined over a relational database, and a user query specified over this view. This user XML query is typically processed using the following steps: (a) our translator maps the XML query to one or more SQL queries, (b) the relational engine translates an SQL query to a relational algebra plan, (c) the relational engine executes the algebra plan and returns SQL results, and (d) our translator translates the SQL results back to XML. However, a straightforward approach produces a relational algebra plan after step (b) that is *inefficient* and has redundant joins. In this paper, we report on our preliminary observations with respect to how joins in such a relational algebra plan can be minimized. Our approach works on the relational algebra plan and optimizes it using novel rewrite rules that consider pairs of joins in the plan and determine whether one of them is redundant and hence can be removed. Our study shows that algebraic techniques achieve effective join minimization, and such techniques are useful and can be integrated into mainstream SQL engines.

1 Introduction

Queries, and their corresponding algebra plans, generated automatically by translating queries specified over virtual views tend to have unnecessary joins [16]. Such algebra plans take much longer time to execute when compared to an equivalent algebra plan without the unnecessary joins. In this paper, we study the problem of how to remove unnecessary joins from a relational algebra plan.

As it sounds, this problem has been extensively studied in the more than thirty years of SQL and relational history [2, 1, 8, 15, 4]. In spite of the large amount of research, current SQL engines do very minimal join-minimization; the only kind of join minimization done is that of removing a join such as $A \overset{\bowtie}{c} B$, where c is a condition of the form $A.key = B.fk$, and $B.fk$ is foreign key attribute(s) of B that reference A. The reason for this minimal adoption is because existing solutions in research assume a set semantics, which give incorrect results when we assume bag semantics required by SQL. As a simple example, consider the algebra plan $\pi_{att_A}(A \times B)$, where A, B are relations, and att_A is the set of attributes of A. This plan returns the attributes of A after doing a cartesian product of A and B. The above plan is equivalent to the plan A, under set semantics. However, under bag semantics the above two plans give different results.

Motivating Example: Let us consider an example application scenario from the medical domain to illustrate the practicality of this problem. Consider two relations in the database of a primary clinic: one that describes doctors, and their speciality, and another that describes patients, who their primary doctor is, and what their primary health issue is. The two relations and their sample data are shown in Table 1.

docID	name	speciality
ID1	Mike	ENT
ID2	Mary	General
ID3	Cathy	General

(a) Doctor Relation with Sample Data

patID	name	primaryHealthIssue	doctor
SSN1	Matt	Arthritis	ID1
SSN2	Joe	Polio	ID1
SSN3	Mark	Cancer	ID2
SSN4	Emily	Arthritis	ID2
SSN5	Greg	Cancer	ID2
SSN6	Terry	Cancer	ID3
SSN7	Tina	Cancer	ID3

(b) Patient Relation with Sample Data

Table 1: Example Relational Database

Now consider that the primary clinic needs to export an XML view of its data to a certain class

of users. The view must specify the patients who have been diagnosed with cancer, and their primary health care physicians, grouped by the physicians. Such view definitions have been studied in several systems such as SilkRoute [5], XPERANTO [14], and CoT [10]. Fig 1 shows this view defined using XQuery [17] (this query is slightly modified from the one in SilkRoute [5] for ease of explanation).

```
<root> {
for $d in //Doctor
where exists (//Patient[@doctor=$d/@docID
          and @primaryHealthIssue='Cancer'])
return <doctor DoctorID={$d/@docID}>
  for $p in //Patient[@doctor=$d/@docID
          and @primaryHealthIssue='Cancer'])
  return <patient PatientID={$p/@patID}/>
</doctor> }
</root>
```

Figure 1: An XML view of the relational database from Table 1 defined using an XQuery

```
<root>
  <doctor DoctorID='ID2'>
    <patient PatientID='SSN3'/>
    <patient PatientID='SSN5'/>
  </doctor>
  <doctor DoctorID='ID3'>
    <patient PatientID='SSN6'/>
    <patient PatientID='SSN7'/>
  </doctor>
</root>
```

Figure 2: The result from a user query `/root` against the view defined in Figure 1

Such a view is typically virtual, and not materialized. Once such a view is defined, one needs to support arbitrary queries to be specified over this view. For instance, the result of the query `/root` is shown in Figure 2. Consider a user query U_1 that retrieves all the patient IDs in the view, which could be specified as `//patient/@PatientID`. Such a query could be answered using the following steps: (a) our translator translates the above XML query into SQL queries, (b) the relational engine translates an SQL query into a relational algebra plan, (c) the relational engine executes the algebra plan to get SQL results, and (d) our translator translates the SQL results back to XML to conform to the view. After these steps, the user will get the answer {SSN3, SSN5, SSN6, SSN7}[1].

[1]Note that we are assuming an unordered semantics. Considering order constraints such as SSN3 and SSN5 must appear next to each other are outside the scope of this work. Such unordered semantics as we assume might be appropriate, if the user knows that the underlying data source is relational.

For step (a), our translator uses a mapping as shown in Figure 3. This mapping says that one root node always exists in the view; the set of doctor children of this root node is the doctors that have a patient with cancer; given a doctor, her patients are those who have cancer. Such mappings are derived from the view query definition [5, 14].

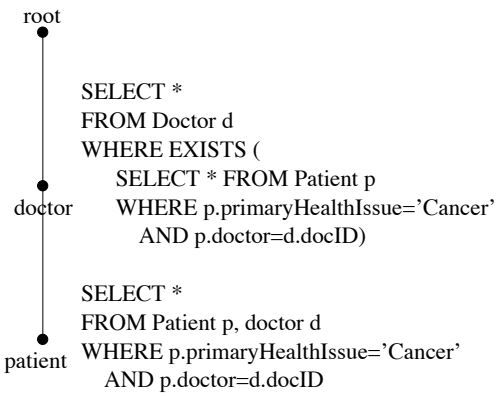


Figure 3: Mapping obtained from the view query (Figure 1) used to answer queries.

Let us see how the translator translates the user XML query U_1 into SQL using the above mapping. The translator can determine that the set of patients can be obtained from the SQL query corresponding to the patient node in the mapping. This query in turn uses the doctor node in the mapping, which in turn can be substituted by the SQL query corresponding to the doctor node in the mapping. After such substitutions, and some minor syntactic rewriting, we get the SQL query Q_1 that answers the user query as:

```
SELECT p.patientID
FROM Patient p,
     (SELECT * FROM Doctor d1
      WHERE EXISTS (
        SELECT * FROM Patient p1
        WHERE p1.primaryHealthIssue='Cancer'
              AND p1.doctor=d1.docID)) d
WHERE p.primaryHealthIssue='Cancer'
      AND p.doctor=d.docID
```

The above query specifies two joins: first there is a join between Doctor $d1$ and Patient $p1$ to produce d, that is the set of doctors who have cancer patients. This d is then joined with Patient p to get the final result. However, from the application semantics, we know that every patient who has cancer will appear in the view. Therefore a simpler SQL query Q_2 for answering U_1 would be:[2]

[2]Q_2 answers U_1 if we assume that every patient has one doctor. However even without this assumption, Q_1 can be optimized to a query which has only one join, as we will see later. In other words, Q_1 always has redundant joins.

```
SELECT p.patientID
FROM Patient p
WHERE p.primaryHealthIssue='Cancer'
```

Even if the query such as Q_1 specifies multiple joins, it might not be inefficient, if the relational engine can optimize the query. A relational engine first translates an SQL query into a relational algebra plan and tries to optimize this plan. This optimized plan is what is executed. However, when we feed Q_1 into a relational engine (we use IBM DB2 V8), we get a final plan that looks like the one shown in Figure 4. Observe that the plan still has the two joins.

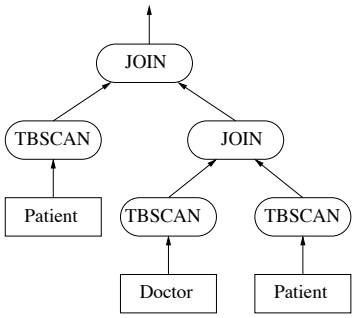


Figure 4: Algebra Plan corresponding to Q_1 generated by an SQL engine.

In this paper, we come up with a novel set of rules for minimizing joins in a relational algebra plan. Our rules determine whether a join in a algebra plan can be removed by examining other joins in the plan. Using our rules, as well as previously studied rules that minimize joins by examining semantic constraints in the schema, we are able to minimize the query plan in Figure 4 to an equivalent query plan without any joins.

Outline of the paper

The rest of the paper is organized as follows. Section 2 describes some of the related work in minimizing joins. Our rules for minimizing joins, along with an example illustration, are described in Section 3. We report on our preliminary experimental studies that show the performance gain possible by such join minimization in Section 4. Our conclusions and future directions are given in Section 5.

2 Related Work

Dan Suciu reported in [16] that the translator (step (a) in our process) in SilkRoute can produce SQL queries with unnecessary joins, and gave some insights as to why this problem might be more critical in the world of XML views, as opposed to plain SQL views. In XML views, there is a query associated with each "type" whereas in SQL views, there is only one "type" and a query associated with that type. Hence in XML views, queries that join multiple view queries are very frequent.

In [9], the authors study the problem of join minimization for XML views. Here the authors try to optimize step (a) (as opposed to step (b) in our approach). They do this by identifying which nodes in the view mapping such as Figure 3 form "bijective" mappings. A node in the view mapping is said to be a bijective mapping with respect to a relation in the SQL database, if there is an element of this node type in the view instance corresponding to every row in the relation. In our example view mapping shown in Figure 3, every row in the Doctor relation does not appear in the view; every row in the Patient relation also does not appear in the view. Therefore both the doctor node and the patient node in Figure 3 do not form bijective mappings. This means that the techniques studied in [9] will end up with an inefficient query plan such as the one in Figure 4.

In [10], the authors study a class of views where every node in the mapping is necessarily bijective. In other words, they disallow a view definition such as the one in Figure 1. By making this assumption, the authors are able to optimize step (a), and come up with minimal SQL queries easily: every XPath expression (or subexpression) that selects every element in the instance corresponding to a node can be obtained by a select query from the corresponding relation (and no joins are needed).

In the previous section, we mentioned the rich body of work that study join minimization assuming set semantics. In [2], Chandra and Merlin showed that there is a unique minimal query for any given conjunctive query, and that such minimization is NP-hard. In [1], the authors considered additional constraints such as functional dependencies specified on the relations, and came up with a *tableau* (matrix) based approach for decreasing joins. Minimization of joins in the presence of functional dependencies was also shown to be NP-complete in the size of the query. In [8], the authors considered functional and inclusion dependencies and showed that minimization of joins is still NP-complete. Here the authors came up with a *chase* technique that, given a query, expands the query by adding conjuncts based on the functional and inclusion dependencies. This expanded query can then be minimized. A graph based approach, consisting of *expansion* and *reduction* steps, for join minimization is studied in [15]. Recently, in [4], the authors consider physical structures such as primary and secondary indexes, extent-based representation of OO classes, join indexes, path indexes, access support relations, gmaps etc. The authors study how to

translate a logical query into a minimal query against the physical schema, using a chase step that expands the logical query to a universal query, and then a backchase step that minimizes the universal query.

The above approaches [2, 1, 8, 15, 4] do provide a good understanding of the problem; however, these techniques cannot be used in SQL engines, because SQL is based on bag semantics. The complexity of join minimization of conjunctive queries under bag semantics as in SQL is studied in [7, 3], and they report that query containment is Π_2^p-hard. Further, in [3], the authors consider `select-from-where` queries with bag semantics, and remark that such queries cannot be minimized without additional semantic constraints. In our work, we consider queries that produce semi-joins in the plans (such as queries with `exists`), and show that these joins can infact be reduced without any additional semantic constraints.

The approach that we propose for join minimization is an algebraic rewriting technique. Algebraic rewriting rules for SQL queries have been studied extensively, for example in [11, 12, 6, 13]. Some of the rules include removal of DISTINCT if one of the returned columns is known to be unique, techniques for decorrelation etc. However, none of the techniques study join minimization that can optimize the query plan shown in Figure 4. We expect that our techniques described in this paper will complement existing algebraic optimization techniques.

3 Rules for Minimizing Joins

In this section, we will describe our rules for minimizing joins in an algebra plan. We will state each rule informally, rather than using a formal notation, for ease of explanation. Further, we assume that some preliminary analysis of the algebra plan has already been done to identify characteristics such as for every operator, *what* columns are needed in the rest of the algebra plan (refer to any commercial optimizer like IBM DB2). We will use the following common notations for our relational algebra operators: select is denoted by σ; project is denoted by π; $\bowtie$ denotes join; $\ltimes$ denotes semi-join; $\overset{o}{\bowtie}_L$ denotes left-outer join; δ removes duplicates; γ denotes grouping.

Before we define the rules, we would like to introduce the notion of *logical entailment*. For instance, we say that the condition $(a = b) \wedge (c = d)$ logically entails the condition $(a = b)$. Given two conditions (boolean expressions) c_1 and c_2, c_2 logically entails c_1 if $c_2 \rightarrow c_1$ is always true. In other words, whenever c_2 evaluates to true c_1 will necessarily be true. A naive method for checking logical entailment is: identify common "terms" in c_1 and c_2 using syntactic analysis, and then check for all combinations of truth values of every term, whether $c_2 \rightarrow c_1$ is true.

Our first two rules are already studied and implemented in most commercial systems. They utilize semantic constraints (key-foreign key constraints) in the schema to remove joins.

Rule 1 $A \overset{\bowtie}{c} B$ can be reduced to $\sigma_{c'}(B)$ if c is a condition that logically entails the condition $A.key = B.fk$, where $B.fk$ is foreign key referencing A, no column in $B.fk$ can be NULL, and no column of A is needed in the rest of the algebra plan. c' is obtained from c by removing the condition $A.key = B.fk$. □

Rule 2 $A \overset{\bowtie}{c} B$ can be reduced to $\sigma_{c'}(B)$ if c is a condition that logically entails the condition $A.key = B.fk$, where $B.fk$ is foreign key of B that references A, and no column of A is needed in the rest of the algebra plan. c' is obtained from c by removing the condition $A.key = B.fk$, and by adding condition of the form $B.fk$ IS NOT NULL. □

Our third and fourth rules are more complex, and form the crux of our approach. They try to remove unnecessary semi-joins that may appear in the algebra plan. Semi-joins may appear in an algebra plan when we decorrelate a correlated SQL query. For example, consider the SQL query corresponding to the doctor node in Figure 3. It specifies a correlated query, which is translated into an algebra plan such as: $Doctor \overset{\ltimes}{c} Patient$, where c = (doctor = docID AND primaryHealthIssue = 'Cancer') is the join condition. The result of this semi-join is the set of rows in the *Doctor* relation, that satisfy the condition.

Now in Q_1, the above result is then joined with the *Patient* relation. The algebra plan corresponding to this is $(Doctor \overset{\ltimes}{c_1} Patient) \overset{\bowtie}{c_2} Patient$. Further, in this query the two conditions c_1 and c_2 are identical. In other words, the doctors who have patients are then joined with patients. We see that the first semi-join can be removed. We now get the query plan $Doctor \overset{\bowtie}{c_2} Patient$.

Rule 3 $(A \overset{\ltimes}{c_1} B) \overset{\bowtie}{c_2} B$ can be reduced to $A \overset{\bowtie}{c_2} B$ if the condition c_2 logically entails the condition c_1. □

The above rule can be implemented by doing a bottom-up traversal of the algebra plan. For any semi-join such as $(A \overset{\ltimes}{c_1} B)$, check if this operator has an "ancestor" operator in the plan that is a join with B, and has a join condition c_2 where c_2 logically entails c_1. This rule can be extended to an ancestor semi-join also, and the correctness holds.

Rule 4 $(A \overset{\ltimes}{c_1} B) \overset{\ltimes}{c_2} B$ can be reduced to $A \overset{\ltimes}{c_2} B$ if the conditions c_2 logically entails the condition c_1. □

Using the above rules, we can come up with an efficient relational algebra plan for Q_1, as shown in Figure 5. First we start with a plan that includes a semi-join and a join. Using Rule 3, we first remove the semi-join. We then use Rule 1 to remove the remaining join. The result is an efficient algebra plan with no unnecessary joins. In our experimental section, we show this efficient plan executes orders of magnitude faster; we achieved improvement of a factor of about 26 for simple queries[3].

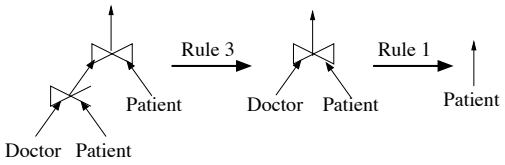


Figure 5: Using our rules to minimize relational algebra plan for query Q_1.

4 Experimental Evaluation

We performed some preliminary experiments to illustrate the effectiveness of our proposed approach. Our experiments were done on IBM DB2 V8 Database Server, which is installed on an 1.4 GHz Pentium machine with 512 MB RAM, running Windows XP. We used the TPC-H[4] benchmark data, loading data of different amounts from 500 MB to 4 GB.

We performed three sets of experiments. The first set of experiments illustrate that joins can be expensive. For this, we executed the following two queries:

Q_4:
```
SELECT COUNT (*) FROM LINEITEM l, PART p
  WHERE l.L_PARTKEY=p.P_PARTKEY
```

Q_5:
```
SELECT COUNT (*) FROM LINEITEM l
```

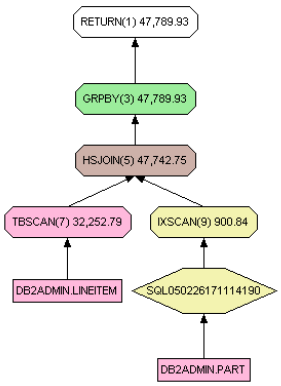
The plans for these two queries are shown in Figure 6. The execution times for these two queries against TPC-H data are shown in Figure 8. Note that this join can actually be very expensive, as it is not a key-foreign key join.

The second set of experiments is similar to our motivating example, and show the effectiveness of Rule 3. For this we executed the queries Q_6, and the equivalent query Q_5. Our rules are able to reduce Q_6 to Q_5. The plan for Q_6 is shown in Figure 7. The execution times for these two queries against TPC-H data are shown in Figure 8. Note that we get considerable performance gain using our rules.

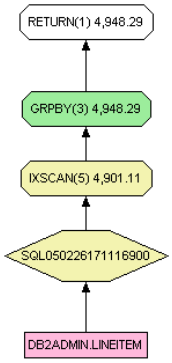
Q_6:
```
SELECT COUNT (*) FROM LINEITEM l,
    (SELECT * FROM ORDERS o1
```

[3]To clarify, for more complex queries, where the percentage of unnecessary joins is smaller, we expect to get lower factors of improvement, but larger absolute values of improvement.

[4]http://www.tpc.org



(a) Query Plan from Q_4



(b) Query Plan from Q_5

Figure 6: Illustrating that joins can be expensive. The execution times are shown in Figure 8.

```
WHERE EXISTS (
(SELECT * FROM LINEITEM l1
WHERE l1.L_ORDERKEY=o1.O_ORDERKEY)) o
WHERE l.L_ORDERKEY=o.O_ORDERKEY
```

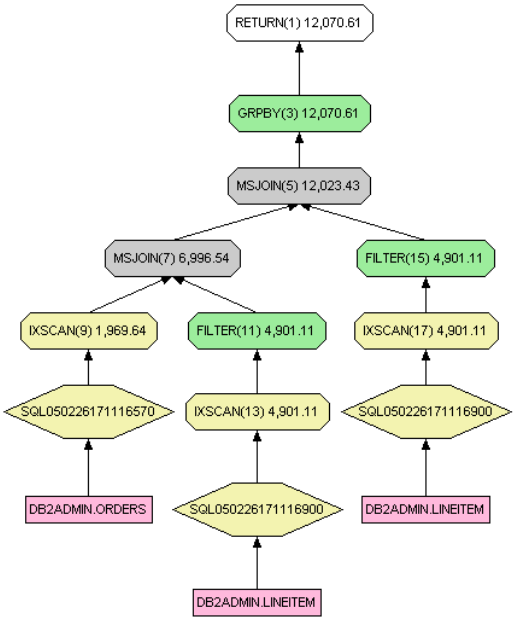


Figure 7: Query plan corresponding to Q_6. Our rules reduce this plan to the plan in Figure 6(b).

The third set of experiments illustrate the effectiveness of Rule 4. For this consider query Q_7 below:

```
SELECT COUNT (*) FROM LINEITEM l
WHERE EXISTS (SELECT * FROM ORDERS o1
        WHERE o1.O_ORDERKEY=l.L_ORDERKEY)
  AND EXISTS (SELECT * FROM ORDERS o1
        WHERE o1.O_ORDERKEY=l.L_ORDERKEY)
```

Using our Rule 4, we can remove one of the joins. We then get an algebra plan that is equivalent to the query Q_8 given below: (Execution times of Q_7 and Q_8 are shown in Figure 8.)

```
SELECT COUNT (*) FROM LINEITEM l
WHERE EXISTS (SELECT * FROM ORDERS o1
        WHERE o1.O_ORDERKEY=l.L_ORDERKEY)
```

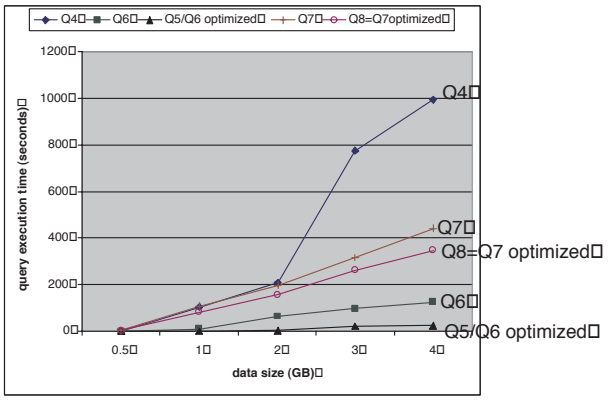


Figure 8: Execution times for the different queries

5 Conclusions and Future Work

In this paper, we have shown that significant performance gain can be achieved by performing join minimization, and that research so far has not solved the join minimization in a satisfactory manner. We have come up with a solution for join minimization that is based on the commercially used algebraic rewriting techniques and preserves SQL bag semantics. We expect that our work will open up renewed interest in this problem, and that the solutions will get adopted into commercial SQL engines. As part of future work, we need to integrate our solutions into commercial optimizers in order to study the query compilation time, as well as demonstrate the feasibility of our techniques.

References

[1] A. V. Aho, Y. Sagiv, and J. D. Ullman. "Efficient Optimization of a Class of Relational Expressions". *ACM Trans. on Database Systems (TODS)*, 4(4):435–454, 1979.

[2] A. K. Chandra and P. M. Merlin. "Optimal Implementation of Conjunctive Queries in Relational Data Bases". *ACM Symposium on Theory of Computing (STOC)*, pages 77–90, 1977.

[3] S. Chaudhuri and M. Y. Vardi. "Optimization of *Real* Conjunctive Queries". In *ACM PODS*, Washington, DC, May 1993.

[4] A. Deutsch, L. Popa, and V. Tannen. "Physical Data Independence, Constraints and Optimization with Universal Plans". In *VLDB*, Edinburgh, Scotland, Sep. 1999.

[5] M. F. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, and W. C. Tan. "SilkRoute: A Framework for Publishing Relational Data in XML". *ACM Trans. on Database Systems (TODS)*, 27(4):438–493, Dec. 2002.

[6] P. Gassner, G. M. Lohman, K. B. Schiefer, and Y. Wang. "Query Optimization in the IBM DB2 Family". *IEEE Data Eng. Bulletin*, 16(4):4–18, 1993.

[7] Y. E. Ioannidis and R. Ramakrishnan. "Containment of Conjunctive Queries: Beyond Relations as Sets". *ACM Trans. on Database Systems (TODS)*, 20(3):288–324, Sep. 1995.

[8] D. Johnson and A. Klug. "Testing Containment of Conjunctive Queries under Functional and Inclusion Dependencies". In *ACM PODS*, Los Angeles, CA, Mar. 1982.

[9] R. Krishnamurthy, R. Kaushik, and J. F. Naughton. "Efficient XML-to-SQL Query Translation: Where to Add the Intelligence". In *VLDB*, Toronto, Canada, Sep. 2004.

[10] D. Lee, M. Mani, F. Chiu, and W. W. Chu. "NeT & CoT: Translating Relational Schemas to XML Schemas". In *ACM CIKM*, McLean, Virginia, Nov. 2002.

[11] H. Pirahesh, J. M. Hellerstein, and W. Hasan. "Extensible/Rule Based Query Rewrite Optimization in Starburst". In *ACM SIGMOD*, San Diego, CA, June. 1992.

[12] H. Pirahesh, T. Y. C. Leung, and W. Hasan. "A Rule Engine for Query Transformation in Starburst and IBM DB2 C/S DBMS". In *IEEE ICDE*, Birmingham, UK, Apr. 1997.

[13] P. Seshadri, H. Pirahesh, and T. Y. C. Leung. "Complex Query Decorrelation". In *IEEE ICDE*, New Orleans, LA, Feb. 1996.

[14] J. Shanmughasundaram, J. Kiernan, E. Shekita, C. Fan, and J. Funderburk. "Querying XML Views of Relational Data". In *VLDB*, Roma, Italy, Sep. 2001.

[15] S. T. Shenoy and Z. M. Ozsoyoglu. "A System for Semantic Query Optimization". In *ACM SIGMOD*, San Francisco, CA, May. 1987.

[16] D. Suciu. "On Database Theory and XML". *ACM SIGMOD Record*, 30(3):39–45, Sep. 2001.

[17] W3C. XQuery Working Group. http://www.w3c.org/XML/Query.html.