Optimizing Large Star-Schema Queries with Snowflakes via Heuristic-Based Query Rewriting*

Yingying Tao Qiang Zhu
Department of Computer and Information Science
The University of Michigan, Dearborn, MI 48128, USA
{yytao,qzhu}@umich.edu

Calisto Zuzarte Wing Lau
IBM Toronto Laboratory
Markham, Ontario, Canada L6G 1C7
{calisto,lauwing}@ca.ibm.com

Abstract

User queries have been becoming increasingly complex (e.g., involving a large number of joins) as database technology is applied to some application domains such as data warehouses and life sciences. Query optimizers in existing database management systems often suffer from intolerably long optimization time and/or poor optimization results when optimizing large join queries. One possible solution to tackle these problems is to rewrite a user-specified complex query into another form that can better utilize the capability of the underlying query optimizer, based on some heuristic rules, before sending the query to the next query optimization stage. We focus on studying a special type of complex query possessing a star-schema structure with snowflakes, simply called the snow-schema query. The key idea is to split a given snow-schema query into several levels of small query blocks at the query rewriting stage. The query optimizer then optimizes the query blocks and integrates their results into the final query result. A set of heuristic rules on how to divide the query is introduced. A query rewriting framework adopting these heuristics is presented. Experimental results demonstrate that this heuristic-based query rewriting technique is quite promising in optimizing large snow-schema queries.

Keywords: Database management system, query optimization, query rewrite, complex query, query graph

1 Introduction

Query optimization is vital to the performance of a database management system (DBMS). The main task of a query optimizer in a DBMS is to seek an efficient query execution plan for a given user query. People have conducted extensive studies on query optimization in the last three decades. Many optimization techniques have been proposed. Comprehensive surveys of this area can be found in [3],[4],[5].

However, the database field is a rapidly growing one and queries are becoming increasingly complex[1]. The query optimization techniques adopted in commercial DBMSs often cannot cope effectively with large join queries.

The query optimization problem has been proven to be NP-complete [6]. That is, we cannot find an optimization algorithm with a polynomial time complexity to generate an optimal execution plan for every query. The dynamic programming technique, which is adopted in many commercial DBMSs, is the most popular technique for finding an optimal plan for a query. However, its worst-case time complexity is exponential, i.e., $O(2^n)$. Therefore, when the input query is too large (e.g., having more than 50 joins), the dynamic programming technique may take months or years to optimize a query (assuming that unlimited resources are avail-

*Research supported by the IBM Toronto Laboratory and The University of Michigan.

Copyright: ©2003 IBM Canada Ltd. Permission to copy is hereby granted provided the original copyright notice is reproduced in copies made.

[1] Although there are several dimensions of query complexity[9], we mainly consider the number of joins in a query as a complexity factor contributing to optimization time.

able to store alternate feasible execution plans). When a DBMS runs out of resources, the optimizer might choose to switch from dynamic programming to some form of greedy join enumeration. In this case, the execution plan is usually suboptimal.

To solve the problem of optimizing large join queries, several optimization techniques have been proposed to find a good plan for such a query within a polynomial time. Examples of such techniques include the iterative improvement (II), the simulated annealing (SA) [1], the AB algorithm (AB) [8], the genetic algorithm (GA) [2], etc. These techniques are mainly based on the randomization approach. They represent a compromise between the time the optimizer takes for optimizing a query and the quality of the resulting optimization plan.

We adopt an alternative approach to optimizing a large join query. The key idea is to rewrite the query into an equivalent one by dividing it into several query blocks in such a way that each block can fully utilize the dynamic programming capability of the query optimizer to generate an optimal plan for the subquery itself. The execution plan for the entire original query consists of these subplans. In this way the query optimizer is expected to generate a good plan within a shorter period of time for the query, compared to the approach of attempting a full dynamic programming technique and then dropping to a greedy join enumeration.

We focus on studying a class of queries whose query structure is of a special type, called the snow-schema structure. Such queries are gaining popularity in data warehousing and web based applications. In the appendix, we give an example of a large snow-schema query from a real-world user application. Some optimization strategies for such queries including limited Cartesian products, snowflake materialization and index key feedback have been suggested [7]. Although these strategies are effective for optimizing relatively smaller snow-schema queries, the challenge for optimizing larger snow-schema queries needs to be addressed. We develop a technique that applies a set of heuristics to divide a large snow-schema query into a number of small query blocks that can be effectively and efficiently optimized by the query optimizer within the given resource constraints.

The rest of this paper is organized as follows: Section 2 introduces an optimization degradation problem that occurs when a query optimizer optimizes a large join query and a technique to predict the occurrence of this problem in the context of a snow-schema queries. Section 3 presents a technique to split a large snow-schema query into small blocks based on a set of heuristic rules. Section 4 shows some experimental results to indicate the efficiency and effectiveness of this technique. Section 5 describes further improvements of the technique. Section 6 summarizes the conclusions.

2 Optimization Degradation and Its Prediction

In this section, we describe a query optimization degradation problem, introduce a technique to predict its occurrence and discuss the type of queries for which we solve this problem.

2.1 Optimization Degradation Problem

As mentioned in Section 1, for a large join query, to get an optimal execution plan within a short optimization time appears impossible using the current database techniques. A commercial DBMS usually supports several optimization levels that may be controlled by the user or by the optimizer itself. Optimization at the highest level in a system is usually based on the dynamic programming technique, while optimization at the lowest level is based on a greedy join enumeration. There may be intermediate levels that use mixed (dynamic programming and greedy) strategies. The higher the optimization level, the better the resulting query plan, but the greater the optimization time needed. If the query is too large and the system cannot optimize it at a higher level with the given resources, the optimization level may automatically drop down to a lower level. In this case, we say that optimization degradation has occurred. For simplicity, we only consider two optimization levels in this paper: one based

on dynamic programming and the other based on greedy strategies.

It is possible that the system has already spent some time at an optimization level for a query before the level drops down to a lower level. Such time is obviously wasted, and meanwhile, more often than not, a poor plan generated at the lower optimization level is still adopted for the query. Our goal is to avoid this situation and maximize the use of dynamic programming.

When a user query is received, we apply a technique to determine if the query will possibly drop down to a lower level. If the answer is yes, we then try to divide this query into several subqueries (query blocks) using some heuristic rules. Each subquery is optimized separately and replaced by its result table in the query. The revised query is then optimized. The query blocks are smaller and the optimizer uses dynamic programming for each of these separately and glues the subplans together. The challenge is to come up with a set of good query splitting rules.

Splitting a query into subqueries limits the search space of execution plans by the query optimizer. Tables in different subqueries do not have a chance to directly join with each other. In order to find a better execution plan for a query, the system should not split a query unless it is predicted that the optimization degradation problem will occur for the given query. Therefore, a method to determine if optimization degradation will occur for a given query is required.

2.2 Predicting Optimization Degradation Based-on Decision Tree

There are many factors that may cause optimization degradation for a query. These include the number of tables, the number of (2-way) joins, the types of join conditions, the size of the optimization heap[2], the cardinalities of tables, the structures of queries, etc. Among them, we have identified that the number of tables, the number of joins and the optimization heap size are the three most important factors. The more tables and joins a given query has, the more complex the query. On the other hand, the larger the optimization heap size, the more complex a query the optimizer can handle without having to drop down to a lower level of optimization.

We employ a decision-tree-based technique to use these factors to determine if optimization degradation will occur for a query in a given system environment. The key idea is as follows. We invoke the underlying query optimizer to optimize a set of sample queries with different numbers of tables and joins in various environments using different optimization heap sizes, and check whether optimization degradation occurs for each sample query. Using this set of experimental data as the training set, we apply the well-known classification algorithm C4.5 to generate a decision tree (i.e., a classification model). In general, the C4.5 algorithm takes a set of records with the same structure consisting of a number of attribute/value pairs as the training set. One or more of these attributes represent the category of the record. The algorithm determines a decision tree that correctly predicts the value of the category attribute on the basis of answers to questions about the non-category attributes. In our case, the non-category attributes are the three factors we mentioned above, and the category attribute takes the value “yes” or “no”, depending on whether optimization degradation occurs or not for the given query.

After the decision tree is generated for the query optimizer in a given environment, we can use it to guide us to make the decision on whether a user query needs to be split or not. A sample decision tree that we generated for a query optimizer in a real system is shown in Figure 1. The experiments were conducted on a real-world database with more than thirty tables varying in size from 2 to 662647 tuples. All features such as indexes, hash files and views were included in the database. 276 queries were used as the training set to generate the decision tree.

It should be pointed out that the decision trees for different DBMSs with different configurations are not the same. Hence, for each DBMS environment, we should generate its

[2]The optimization heap is the memory used for storing subplans during optimization.

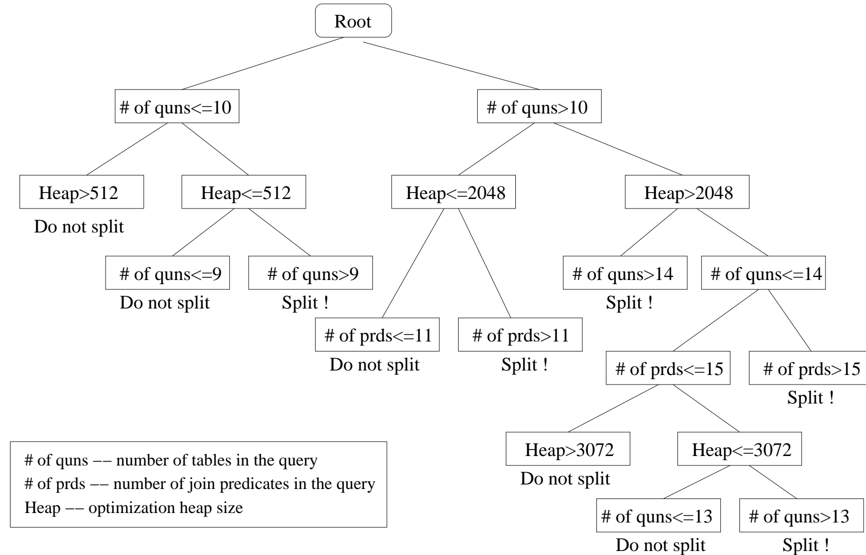

Figure 1: A decision tree generated using C4.5

own decision tree rather than using an existing one compiled for a different environment.

2.3 Snow-Schema Query

After deciding to split a query, we need to determine how to split it. It is difficult to find effective splitting rules for general queries. We focus on a special type of large join query that has a star-schema structure with snowflakes, i.e., the snow-schema query.

The star schema is a database/warehouse design in which the bulk of the raw data is kept in a fact table, and a number of normalized tables (known as dimension tables) surround it. A snow-schema structure consists of several star-schema substructures, known as snowflakes (or dimension table sets), surrounding a central table, i.e., the fact table of the snow schema. In fact, a snowflake itself can also be another snow schema. The query graph of a query on the tables of a snow schema typically also demonstrates a structure similar to that of the snow schema. We call a query with a query graph of such a structure as a star-schema query with snowflakes, or simply a snow-schema query.

Figure 2 shows a typical snow-schema query graph. In the figure, the snowflakes marked as “S4 (D4)” and “S5 (D5)” contain only one table. We call the snowflakes with only one table as trivial snowflakes (i.e., dimension tables). Those snowflakes with more than one table are non-trivial snowflakes. The fact table and its dimension tables (i.e., trivial snowflakes) together are regarded as the main (central) part of the query.

A typical way to process a snow-schema query in a DBMS is to put all its tables into one query block. For example, the following is a query block for a snow-schema query:

```
SELECT  T1.a1
FROM    T1, T2, T3, T4, T5, T6, T7, T8,
        T9, T10, T11, T12, T13, T14
WHERE   T1.a1=T2.a1 AND T1.a2=T3.a1 AND
        T1.a3=T4.a3 AND T1.a4=T5.a4 AND
        T1.a5=T6.a5 AND T1.a6=T7.a6 AND
        T1.a7=T8.a7 AND T2.a2=T9.a2 AND
        T2.a3=T10.a3 AND T2.a4=T11.a4
        AND T2.a5=T12.a5 AND
        T3.a2=T13.a2 AND T3.a3=T14.a3
```

where T_i ($1 \leq i \leq 14$) are tables and a_j ($1 \leq j \leq 7$) are the columns in the tables. As mentioned before, when the query is large, the optimization degradation problem will occur. Hence, our strategy is to split the query into several smaller query blocks (subqueries). Because of the special characteristic of the snow-schema structure, that is, each snowflake is an “island” with only one join link connected to

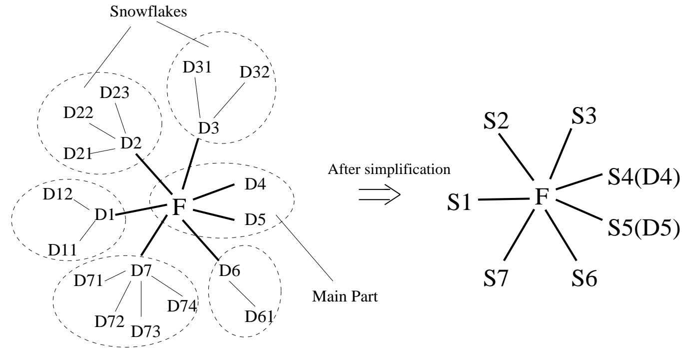

Figure 2: Example of a snow-schema query structure

the fact table in the main query block, moving some of the snowflakes to subqueries may not affect the remaining part of the query too much. For example, the above example query block can be divided into three query blocks as follows:

```
SELECT T1.a1
FROM T1, (SELECT T2.a1
          FROM T2, T9, T10, T11, T12
          WHERE  T2.a2=T9.a2 AND
            T2.a3 =T10.a3 AND
            T2.a4=T11.a4 AND
            T2.a5=T12.a5) AS V2,
         (SELECT T3.a1
          FROM T3, T13, T14
          WHERE T3.a2=T13.a2 AND
            T3.a3 =T14.a3) AS V3,
     T4, T5, T6, T7, T8
WHERE T1.a1=V2.a1 AND T1.a2=V3.a1 AND
      T1.a3=T4.a3 AND T1.a4=T5.a4 AND
      T1.a5=T6.a5 AND T1.a6=T7.a6 AND
      T1.a7=T8.a7 .
```

Note that the embedded subqueries may be implemented as views in a real system but are typically merged into one big query block to provide the optimizer with a maximum latitude to optimize the query. We will introduce a set of heuristic rules for splitting this type of query in Section 3. Note that if a snowflake has its own snowflakes, the subquery itself can be further split into subsubqueries. Figure 3 shows an example of a query with multiple levels of subqueries (blocks).

We first need to identify the snow-schema structure and in particular, the main part and non-trivial snowflakes. For example, the query shown in Figure 4 has a structure which contains only two stars connected to each other with a join predicate between their fact tables. For this query, either star can be treated as the main part, and the other as the snowflake, according to the concept of a snow-schema query structure described above. To resolve the ambiguity in this case, we regard the star with the larger number of tables (i.e., the "Star 1" in Figure 4) as the main part (i.e., the main query

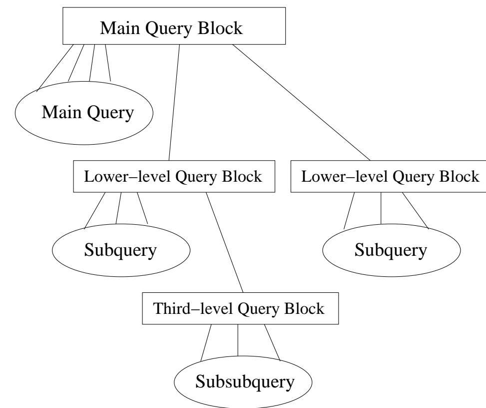

Figure 3: Example of a 3-level query block

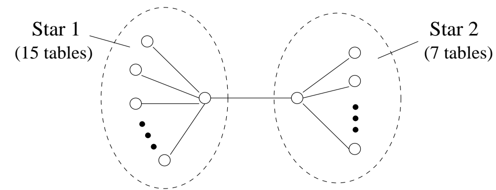

Figure 4: A query with two stars connected to each other

block), and the other star as the snowflake.

If a query has a structure shown in Figure 5, the situation is different. Although it does not violate the concept of the snow-schema query if we regard the part in circle marked with “Part 2” as the main part, and the part in circle “Part 1” as the snowflake, to follow the normal sense of a snow schema, we still regard the part in circle “Part 3” as the main part, and all other stars as snowflakes.

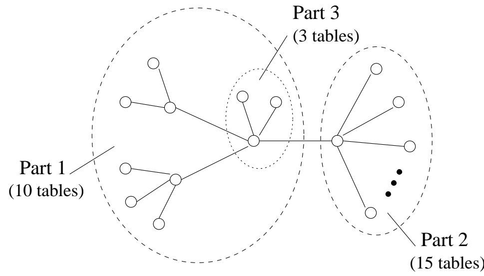

Figure 5: A query with one very large star

Considering a more complex query structure as shown in Figure 6, we choose the part that connects more snowflakes as the main part (i.e., Part 1). Part 2 which also possesses a snow-schema structure is regarded as a snowflake of Part 1. If both parts had the same number of snowflakes, either one could be regarded as the main part.

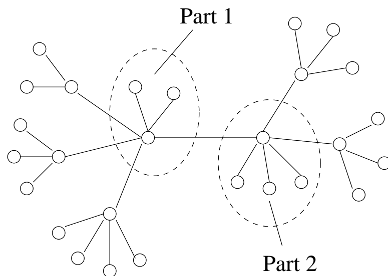

Figure 6: A more complex snow-schema query

3 Splitting Snow-Schema Queries Based on Heuristics

In this section, we introduce a technique to split a snow-schema query based on a set of heuristic rules to solve the optimization degradation problem.

3.1 Query Processing Workflow

When a user issues a query on a DBMS, it is first parsed. If it contains no syntactic or semantic errors, the system rewrites the query into an equivalent but more efficient one if possible. The system then generates an efficient execution plan for the transformed query and performs the query according to the plan to obtain the query result.

Our query splitting task is done based on a set of heuristic rules. It basically transforms a given query with the optimization degradation problem into another one without the problem. Therefore it should be part of the query rewriting job in the system. To minimize the effect of such splitting work on other existing components in the system, we perform query splitting at the end of the query rewriting stage. In this way, all previously valid query rewriting rules can still be applied to the query. The rewritten query is sent to the next stage of query optimization to choose an efficient execution plan for it. The workflow of query processing in a DBMS is shown in Figure 7.

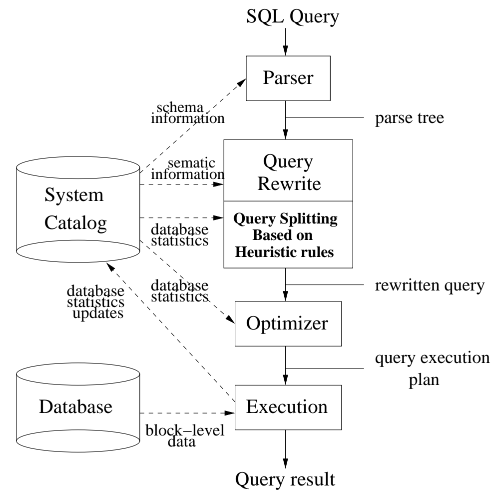

Figure 7: The workflow of query processing

3.2 Moving Snowflakes Considering Indexes

After the query has gone through the normal query rewriting process, we examine if the transformed query has a snow-schema structure. If this is the case, we apply the decision tree described before to check whether the query will have the optimization degradation problem or not. If this is a possibility, we go through the process of splitting the query.

The first splitting heuristic we apply is the following:

H1: *Move any snowflake in which many dimensions can join with its fact table via indexes into a subquery (block).*

In other words, for each snowflake, we check if the number of its dimensions (either a table or another snowflake) that can join with its fact (main hub) table via an index is greater than a threshold value. If so, we create a subquery for the snowflake. The subquery is linked to the main query block via the join condition between the snowflake and the fact table in the main part of the query.

This heuristic rule has the highest priority. The reason is as follows. As we know, indexes can usually improve the performance of query processing significantly. If a snowflake has many dimensions that could join with its fact table via indexes, it is better to make use of these indexes to perform the joins within the snowflake before we join the snowflake result with the fact table in the main part. If the fact table T_s of a snowflake joins with the fact table T_m of the main part first, then all indexes on T_s are no longer usable for joining with the other tables in the snowflake.

3.3 Moving Snowflakes Considering Number of Tables

After applying heuristic **H1**, some snowflakes of the given snow-schema query have been moved into subqueries from the original main query block. In other words, each of such snowflakes (subqueries) can be considered as if it were replaced by its result table in the original main query block from an optimizer point of view. If the revised query still suffers the optimization degradation problem, the splitting process should continue.

Assuming that the revised query still has other snowflakes in the main query block, we attempt to move some of these into separate query blocks as well. The decision on which snowflake to move is based on the number of dimensions (tables or nested snowflakes) in a snowflake. Our goal is to keep as just as many tables in the main query block as possible so that we are under the threshold to avoid the optimization degradation problem.

Let q be a snow-schema query. Let S_q be the set of all snowflakes (at the main level) in q. For a snowflake $s \in S_q$, we use $q - s$ to denote the remaining part in q after s is removed from q. We use $||q||$ to represent the total number of tables in q. Hence, $||q|| = ||q - s|| + ||s||$. Let k be the maximum number of tables in the original main query block for q that the optimizer can process without the optimization degradation problem. Let function φ_k be defined as follows

$$\varphi_k(x) = \begin{cases} TRUE & if\ x\ does\ not\ have\ the \\ & \quad degradation\ problem, \\ FALSE & otherwise, \end{cases}$$

where x is a query. Let V_q^k be the following set,

$$V_q^k = \{s|\ s \in S_q\ and\ \varphi_{k-1}(q - s) = TRUE\}.$$

Note that if a snowflake s is moved into a subquery, it is replaced by its result table in the original query block for q. If k is the maximum number of tables in the original query block that the query optimizer can process without the optimization degradation problem, the maximum number of tables for the remaining query block $q - s$ after removing s that can be handled by the optimizer would be $k-1$ (1 is reserved for the result table of s). If the original query block has already been modified because of the snowflake(s) following heuristic **H1**, k is assumed to have been properly adjusted.

For a given snow-schema query q, if $V_q^k \neq \Phi$, we only need to move one snowflake in V_q^k into a subquery to solve the optimization degradation problem for the original query block for q. The question is which snowflake we should choose. To fulfill the goal of keeping the main

query as large as possible, we split the smallest snowflake in set V_q^k to a subquery block. In other words, we apply the following heuristic:

H2: *Move the snowflake s with $||s|| =^{min}_{x \in V_q^k} \{ ||x|| \}$ (i.e., the smallest) in V_q^k into a subquery.*

In case $V_q^k = \Phi$, there does not exist a snowflake that would eliminate the optimization degradation problem in the remaining query block for q if it were moved into a subquery. In this case, we split the query by the following procedure:

1. let $q' = q$;
2. **while** ($S_q \neq \Phi$ and $V_{q'}^k = \Phi$) **do**
3. find a snowflake $s \in S_q$ such that $||s|| =^{max}_{x \in S_q} \{ ||x|| \}$;
4. move snowflake s into a subquery block;
5. $S_q = S_q - \{s\}$;
6. $q' = q' - s$;
7. $k = k - 1$;
8. **end while**;
9. **if** $V_{q'}^k \neq \Phi$ **then** apply heuristic **H2** to split q';
10. stop.

This procedure repeatedly applies the following heuristic to pick up a snowflake for the next subquery until the remaining query block has a snowflake applicable to heuristic **H2**, or until no more snowflake exists in the main query block.

H3: *Move the snowflake s with $||s|| =^{max}_{x \in S_q} \{ ||x|| \}$ (i.e., the largest) in S_q into a subquery block.*

This heuristic rule tells us to move the largest snowflake into a subquery each time when $V_q^k = \Phi$. The purpose of this heuristic is to make the size of the remaining (main) query block fall, as quick as possible, into the range that the optimizer can handle using dynamic programming and also keep the number of separated query blocks as small as possible.

Let us consider an example. Given a query whose structure is shown in Figure 8 (a), the final query blocks resulting from this procedure look like Figure 8 (b). We assume that a query block with more than 12 tables will need to be split in this example. In the first **while** loop, snowflake 4 is moved. Snowflake 3 is pushed to a subquery block in the second **while** loop. Finally, snowflake 1 is split following heuristic **H2** at step 9.

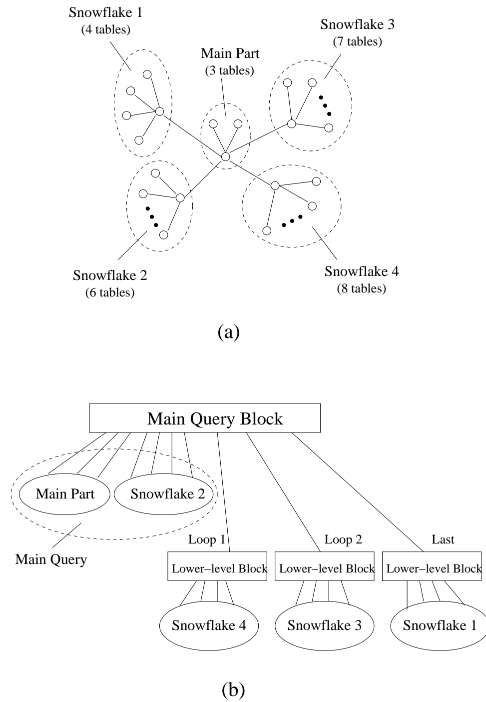

Figure 8: Example of splitting a snow-schema query

3.4 Handling Special Cases

Following the previous heuristics does not guarantee that the query blocks obtained can all be processed by the query optimizer without the optimization degradation problem. There are several cases that need to be further considered.

Case 1: *The main query block contains no snowflake but is still too large.*

If after all non-trivial snowflakes are moved to subqueries and the main query block is still too large to be processed by the optimizer without the optimization degradation problem, a possible solution is to push some of the dimension tables from the main query block to a higher-level query block.

An example of a query with one higher-level query block and two lower-level query blocks

is shown in Figure 9. As the query blocks are processed in a bottom-up fashion, the fact table will join with those dimension tables in the same main block first, and the result table is sent to the higher level query block to join with other dimension tables.

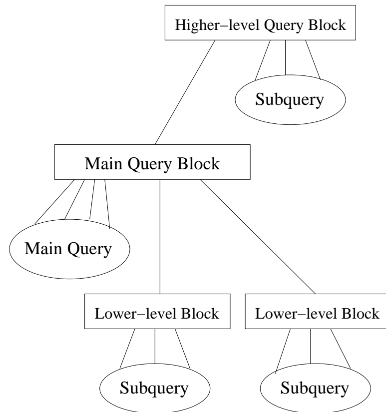

Figure 9: Example of higher-level query block

In some cases, there are so many tables in the main query block that even one higher-level query block is not enough. That is, the number of remaining tables still exceeds the limit. In such a case, we can further push some tables to yet another higher-level query block on top of the current one. In other words, query blocks at multiple higher levels are allowed.

Let q' be the revised query of query q. That is, all snowflakes have been processed and their result tables have been put back into the main query block. Then we further split q' into $m = 1 + \lceil \frac{||q'||-k}{k-1} \rceil$ query blocks. The block at the lowest level is the main query block, and others are query blocks at multiple higher levels. We keep k tables in the main block, and in each higher-level query block (except the highest one) there will be $k-1$ tables. The remaining tables will be put in the highest-level query block.

To determine which dimension tables are kept in the main query block, we apply the following the heuristic:

H4: *Keep the filtering dimension tables that can join with the fact table via indexes in the main query block.*

If there are more than $k-1$ dimension tables satisfying the usable index condition, the following heuristic is applied to decide which ones to put in the main query block:

H5: *Put the dimension tables that have lower selectivities (closer to 0) for their join conditions with the fact table in a lower-level query block.*

This heuristic is also applied to decide the other dimension tables between a lower-level block and a higher-level block. In other words, the higher the selectivity a dimension table has, the higher level the query block it belongs to. The reason behind this heuristic is to make intermediate result sizes as small as possible to reduce I/O cost.

Case 2: *A subquery itself is too large to be handled by the optimizer without the optimization degradation problem.*

It is possible that a subquery obtained previously itself is too large to be optimized without the optimization degradation problem. The snowflake corresponding to the subquery can have a simple star-schema structure or another snow-schema structure. In any case, all previously-discussed rules can be recursively applied to this subquery to solve the optimization degradation problem, which may yield multiple levels of subqueries.

Case 3: *Many small subqueries are obtained.*

The previous splitting process may yield many small subqueries. Although all the subqueries can be optimized by the optimizer without the optimization degradation problem, the capability of the optimizer for searching a good plan in a larger space is not exploited. Restricting the optimizer to optimizing many small query blocks may not yield a good integrated plan for the entire query.

We use a threshold value c to determine if a query block is too small or not. If a subquery is too small, we then estimate the result table size of this subquery using the database statistics available in the system catalog. If the result table size is very small (e.g., only a few tuples), we then consider reorganizing it. We sort all query blocks (at the same level) that need to be reorganized in the ascending order of the number of tables in them into a (sorted) list. We merge one query block with the next

in the list into a larger query block, until performing the next merge makes the total number of tables in the larger block exceed the maximum number k of tables allowed by the query optimizer. This procedure is repeated until all subqueries in the list are considered.

Note that the tables from two different query blocks (at the same level) can only be operated together by a Cartesian product. Reference [11] indicates that the Cartesian product may sometimes help reducing the cost of the execution plan. Hence, it is possible that the query optimizer generates a plan containing Cartesian products for a query. We have to guard against a large intermediate result due to a Cartesian product of two or more snowflakes.

3.5 Other Secondary Rules

Sometimes we may have multiple candidates (snowflakes) satisfying the condition of a heuristic rule (e.g., **H2** or **H3**). To break the tie, we adopt a secondary rule as follows:

H6: *Give the snowflake with a smaller estimated result table size a higher priority.*

The reason behind this rule is to keep the sizes of intermediate results as small as possible, which is similar to the goal of heuristic **H5**.

For a snow-schema query (block), if its fact table can be joined with the fact table of one of its non-trivial snowflakes directly via an index, it is better not to move the snowflake into a subquery unless there is no other choice. The reason for this is that an index usually improves the performance significantly for a join involving a large table. Moving the snowflake into a subquery may cause the query to lose the chance of using the index for execution. However, not splitting such a snowflake sometimes may also lead to a bad situation. For example, consider a snow-schema query (with 31 tables) has one large snowflake (with 20 tables) whose fact table can join with the fact table of the main block via an index and three other small snowflakes (with 3 tables in each) that cannot use any index when joining with the fact table of the main block. Assume $k = 12$. If we avoid splitting the largest snowflake and move the small snowflakes into three subqueries, we eventually will find that the largest snowflake still has to be moved into a subquery in order to solve the optimization degradation problem. On the other hand, if we move the largest snowflake into a subquery first, we solve the problem without moving any small snowflake into a subquery, which is a better solution to the problem. To moderate this problem, when we have a chance to hold a snowflake due to the index benefit, we look ahead the next m steps to see if the optimization degradation problem can be solved without moving the held snowflake into a subquery. If so, we hold the snowflake and move other snowflakes. Otherwise, we still move the snowflake under consideration into a subquery. Therefore, we have the following heuristic:

H7: *Keep a snowflake in the main query block if its fact (main hub) table can join with the fact table of the main query block via an index, and if the optimization degradation problem for the main query block can be solved by splitting other snowflakes within next m steps.*

The threshold value m can be specified by the user. The higher the m value is, the better the splitting (but with a higher overhead).

4 Experiments

To examine the effectiveness of our heuristic-based splitting technique for solving the optimization degradation problem, we conducted some experiments on the DB2 system running on an IBM AIX machine (4 $\times$ PowerPC 604 and 3064 MB memory) under OS AIX 4.3.3.75. The optimization heap size was set to 2048 $\times$ 4KB pages (not changing for the experiments), and the initial query optimization level was set to use the dynamic programming technique.

The test database used for the experiments consisted of 50 tables varying in size from 1000 to 10,000 tuples and having 5 to 20 attributes. The data in all the tables was randomly generated.

We first applied the C4.5 classification algorithm to the execution data of a set of sample queries (as the training data) to generate a decision tree for predicting if a query will have the optimization degradation problem in the given

experimental environment. The decision tree is shown in Figure 10.

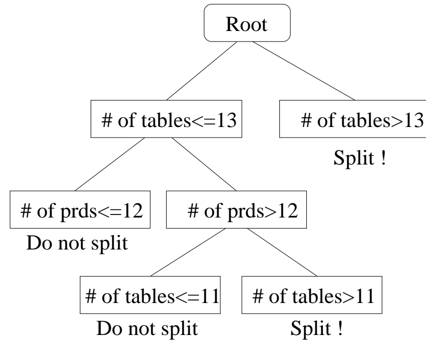

Figure 10: The decision tree for predicting the optimization degradation problem in the experiments.

A set of snow-schema queries were then tested in our experiments, which involved 18 to 50 tables and 3 to 8 snowflakes. A snowflake may have snowflakes of its own. Each test query was executed on the system twice: with and without our heuristic-based splitting technique. Every test query in the set suffered the optimization degradation problem when our technique was not applied. In other words, the system switched to using some greedy strategies to choose an execution plan for the query. After our technique was applied, the split queries no longer had the optimization degradation problem. In other words, each query block in a split query was optimized based on dynamic programming. We then compared the optimization time for each query in both ways and the execution time for each query following the corresponding plans from both ways. The comparison results are shown in Figures 11 and 12. Note that the actual time measuring unit in the experiments are not revealed here to avoid any potential license violation of the software.

Figure 11 indicates that our technique can reduce the optimization time, with an average improvement of 20.0%. Note that if the system were forced to use the dynamic programming technique without dropping down to the greedy strategies, the optimization time would be much higher than that when our technique is applied. Figure 12 indicates that our technique can help to generate better execution plans for snow-schema queries. Using the plans with our technique, the execution time for the queries can be improved by 21.4% on average.

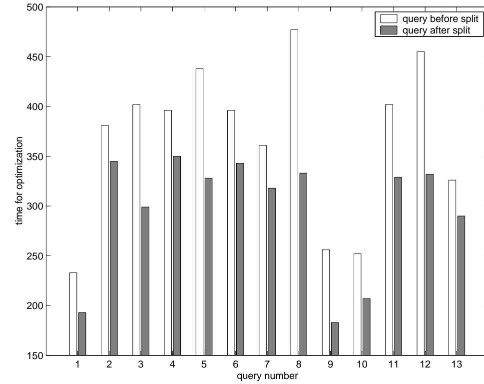

Figure 11: Experimental results on optimization time

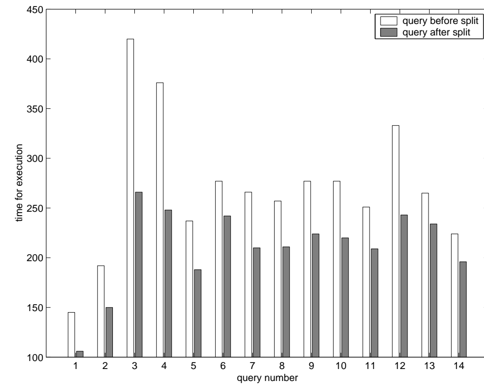

Figure 12: Experimental results on execution time

In summary, with our technique, a query optimizer can spend less time on optimization and generate better execution plans for snow-schema queries.

To study the performance of the dynamic programming technique for optimizing large snow-schema queries, we tested a snow-schema query with 14 tables. The query was compiled using the dynamic programming method with the optimization heap size set to 60,000 $\times$ 4KB pages. The optimization time for this query is 26 time units, while using our approach with the optimization heap size set to 2048 $\times$ 4KB pages, the optimization time for a snow-schema query with 43 tables is only 25 time units. As

we know, the optimization time for the dynamic programming approach increases exponentially with the number of tables joined in a single query block. Applying our technique it is possible to compile a very large query in a reasonably short time compared to the traditional "single query block" dynamic programming method, although the generated plan may be sub-optimal.

5 Further Improvements

The set of heuristics presented in Section 3 are sufficient to divide a large snow-schema query with the optimization degradation problem into a number of small query blocks without the problem. However, the technique can be further improved. In this section, we describe several possible suggestions for improvements that need to be further investigated.

Partial Splitting

The heuristic rules in Section 3 assume that we always move an entire snowflake into a subquery. Sometimes it may be beneficial to move only part of a snowflake into a subquery and keep the remainder in the main query block. For example, consider a snow-schema query having 15 tables with a unique non-trivial snowflake having 8 tables, assuming $k = 12$. Applying heuristic **H2**, the snowflake is moved into a subquery, resulting in a main query block having 7 tables and a subquery having 8 tables. Alternatively, if we only move 4 tables of the snowflake into a subquery and keep 11 tables in the main block, the full capability of the query optimizer can be utilized to generate a good plan for the larger main block, insteading of optimizing two smaller query blocks, which may lead to a better overall plan for the entire query. The relevant issues that need to be further studied include when to perform a partial splitting and which part of a snowflake should be split into a subquery.

Exploiting Common/Similar Subqueries

A large snow-schema query may contain some common/similar snowflakes. In [9],[10], we introduced techniques to optimize complex queries by exploiting common/similar subqueries. The basic idea is to identify groups of common/similar subqueries, optimize a representative subquery for each group and share the optimization result with the other members in the corresponding group. In this way, the optimization time for optimizing other common/similar subqueries is saved. These techniques can be extended to identify common/similar snowflakes and share the optimization result within each group. Since snowflakes have a special structure, it is possible to develop a more efficient method to examine their common/similar structures.

Handling More General Queries

The idea of splitting a query can be applied to more general cases. The reason why we choose to consider queries with snow-schema structures is that for this special type of query there are more clear rules that we can follow. Another reason is that, since each snowflake has only one link connecting it to the main part or fact table, replacing it by its result table would not affect the rest of the query too much. There are several ways to extend the splitting technique for snow-schema queries to more general cases. For example, it is not necessary for a snowflake to have a star-schema or snow-schema structure. As long as there is only one join condition between one part of a query and the remaining part, the former can be treated as a "snowflake" in the technique. Another extension may allow one part (cluster) of a query to have more than one connection to the remaining part. However, these cases are more complex. Further studies are required.

6 Conclusions

Large join queries are becoming increasingly common. Due to the limitation of system resources, a query optimizer often drops its optimization level down to a lower level when optimizing such queries. This problem may cause the query optimizer to waste much optimization time and meanwhile still obtain a poor execution plan for the given query.

We have presented a heuristic-based splitting technique to solve the optimization degradation problem for a special type of complex query, called the snow-schema query. Based on the natural structure of such a query, a set of useful heuristics are proposed to divide a large query into a number of smaller query blocks so that each query block can be optimized by the query optimizer using dynamic programming. Our analysis and experimental results demonstrate that the proposed technique can help the query optimizer improve the quality of the execution plan generated for a large snow-schema query with less optimization time. To determine if a query will suffer the optimization degradation problem in a given environment, a decision-tree-based predication method is suggested.

Our work is an alternate and practical approach to solve the query optimization issues for large complex queries that have a snow-schema structure using the existing infrastructure available in typical commercial database systems that use dynamic programming. Further work needs to be done to extend this approach to other large complex queries.

Acknowledgments

The authors would like to thank Berni Schiefer, Qi Cheng, John Hornibrook, Kelly Lyons, and Joe Wigglesworth for their useful discussions, suggestions and support for the work reported here.

About the Author

Yingying Tao is a graduate student in the Department of Computer and Information Science at The University of Michigan, Dearborn, MI, USA. She is also a graduate research assistant with an IBM CAS fellowship. She received a B.Sc. in Computer Science from Tsinghua University (China) in 2000. Her research interests include query processing and optimization in database systems.

Qiang Zhu is an Associate Professor in the Department of Computer and Information Science at The University of Michigan, Dearborn, MI, USA. He received his Ph.D. in Computer Science from the University of Waterloo in 1995. Dr. Zhu is a principal investigator for a number of database research projects funded by sources including the NSF and IBM at The University of Michigan. He has over 40 research publications in various refereed journals and conference proceedings. Some of his research results have been included in several well-known database research/text books. Dr. Zhu has served as a program/organizing committee member and session/workshop chair for a number of international conferences. His current research interests include query optimization for advanced database systems, multidatabases, self-managing databases, Web-based database technology, and data mining.

Calisto Zuzarte is a senior technical manager of the SQL Query Rewrite development team at the IBM Toronto Laboratory. He has been involved in several projects leading and implementing many features related to the DB2 SQL compiler. His main expertise is in the area of query optimization including cost based optimizer technology and automatic query rewriting for performance. Calisto is also a research staff member at the Center for Advanced Studies (CAS).

Wing Lau is a software developer in the IBM Toronto Laboratory, working in the Query Rewrite group of DB2 Universal Database. She received her B. Sc. and M. Sc. in Computer Science from The University of Michigan, Dearborn, MI, USA, in 1997 and 2000, respectively. She was a software engineer at Marquip Inc. (Madision, WI) from 1997 - 1998. Her research interests include query processing and optimization in database systems.

Appendix: A Real-World Large Snow-Schema Query

In this appendix, we show an example of the large snow-schema query from a real-world user application. To protect the user's information, we have masked the names of the tables and attributes in the example. Figure 13 shows the structure of this real-world query.

```
SELECT
SUM(F.R_CMS) AS CMS,
SUM(F.R_MPY) AS MPY,
SUM(F.R_NCD) AS NCD,
```

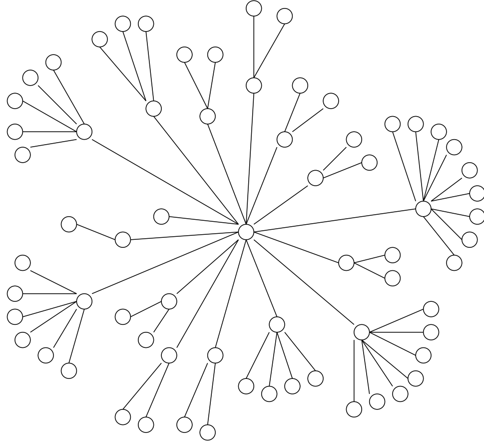

Figure 13: The structure of the large join snow-schema query

```
SUM(F.R_SPI) AS SPI
FROM
FR_7 F, DR_7 D7, S_CLA S1, S_CLA S2,
S_CLA S3, S_CLA S4, S_CLA S5, DR_2 D2,
SCUSR S6, DR_5 D5, SCUSR S7, SCUSR S8,
SCUSR S9, SCUSR S10, SCUSR S11,
SCUSR S12, SCUSR S13, DR_U DU,
SUNIT S14, DR_T DT, SDATE S15,
SCMON S16, SCQUA S17, SCWEK S18,
SCYER S19, DR_B DB, SCSMR S20,
DR_8 D8, SCTYP S21, SCRCY S22,
DR_1 D1, SCUPC S23, SMATR S24,
SPLT S25, DR_A DA, SR_BAC S26,
SR_BKY S27, DR_C DC, SR_CDF S28,
SR_CSR S29, SR_CSN S30, SR_CNB S31,
SR_CTP S32, SR_COD S33, SR_CON S34,
SR_COT S35, DR_6 D6, SR_CTR S36,
SR_CDF S37, SR_PDR S38, SR_PME S39,
SR_PON S40, DR_9 D9, SR_PPR S41,
DR_D DD, SR_PMO S42, DR_4 D4,
SR_RCN S43, SR_SGR S44, SR_SNO S45,
DR_3 D3, SR_SHR S46, SR_STP S47,
SR_SER S48, SUNIT S49, STIME S50,
SWDY1 S51, DR_P DP
WHERE
F.KC_1=D1.DID AND D1.ENP=S23.SID AND
D1.MAT=S24.SID AND F.KC_2=D2.DID AND
D2.CUS1=S6.SID AND D2.PLT=S25.SID AND
F.KC_3=D3.DID AND D3.SHR=S46.SID AND
D3.TIME=S50.SID AND F.KC_4=D4.DID AND
D4.RCN=S43.SID AND F.KC_5=D5.DID AND
D5.CUS2=S7.SID AND D5.CUS3=S8.SID AND
D5.CUS4=S9.SID AND D5.CUS5=S10.SID AND
D5.CUS6=S11.SID AND D5.CUS7=S12.SID AND
D5.CUS8=S13.SID AND D5.SGR=S44.SID AND
D5.SNO=S45.SID AND  F.KC_6=D6.DID AND
D6.CTR=S36.SID AND D6.PON=S40.SID AND
F.KC_7=D7.DID AND D7.CLA1=S1.SID AND
D7.CLA2=S2.SID AND D7.CLA3=S3.SID AND
D7.CLA4=S4.SID AND D7.CLA5=S5.SID AND
D7.CSR=S29.SID AND D7.CSN=S30.SID AND
F.KC_8=D8.DID AND D8.C_TYP=S21.SID AND
D8.COD=S33.SID AND D8.CON=S34.SID AND
D8.COT=S35.SID AND F.KC_9=D9.DID AND
D9.PPR=S41.SID AND D9.STP=S47.SID AND
F.KC_U=DU.DID AND DU.B_UOM=S14.SID AND
DU.D_CRY=S22.SID AND DU.S_UNI=S49.SID AND
F.KC_T=DT.DID AND DT.CDAY=S15.SID AND
DT.CMON=S16.SID AND DT.CQUA=S17.SID AND
DT.CWEK=S18.SID AND DT.CYER=S19.SID AND
DT.WDY1=S51.SID AND F.KC_A=DA.DID AND
DA.BAC=S26.SID AND DA.BKY=S27.SID AND
F.KC_B=DB.DID AND DB.CSM=S20.SID AND
DB.CDF=S37.SID AND F.KC_C=DC.DID AND
DC.CDF=S28.SID AND DC.CNB=S31.SID AND
DC.CTP=S32.SID AND DC.PDR=S38.SID AND
DC.PME=S39.SID AND DC.SER=S48.SID AND
F.KC_D=DD.DID AND DD.PMO=S42.SID AND
F.KC_P=DP.DID  AND ((((DP.RQD<=81)) AND
((S1.C_CLA BETWEEN 'K0' AND 'K6'))))
GROUP BY
S1.C_CLA, S2.C_CLA, S3.C_CLA, S4.C_CLA,
S5.C_CLA, S6.CUSR, S7.CUSR, S8.CUSR,
S9.CUSR, S10.CUSR, S11.CUSR, S12.CUSR,
S13.CUSR, S14.UNIT, S15.DAT, S16.CMON,
S17.CQUA, S18.CWEK, S19.CYER, S20.CSMR,
S21.CTYP, S22.CRCY, S23.CUPC, S24.MATR,
S25.PLT, S26.R_BAC, S27.R_BKY,
S28.R_CDF, S29.R_CSR, S30.R_CSN,
S31.R_CNB, S32.R_CTP, S33.R_COD,
S34.R_CON, S35.R_COT, S36.R_CTR,
S37.R_CDF, S38.R_PDR, S39.R_PME,
S40.R_PON, S41.R_PPR, S42.R_PMO,
S43.R_RCN, S44.R_SGR, S45.R_SNO,
S46.R_SHR, S47.R_STP, S48.R_SER,
S49.UNIT, S50.TIME, S51.WDY1   .
```

References

[1] Y. E. Ioannidis and E. Wong: Query Optimization by Simulated Annealing. In *Proceedings of ACM-SIGMOD International*

Conference on management of Data, pages 9-22, 1987.

[2] K. Bennett, M. C. Ferris, and Y. Ioannidis: A genetic algorithm for database query optimization. In *Proceeding of 4th International Conference on Genetic Algorithms*, pages 400-407, 1991.

[3] S. Chaudhuri: An overview of query optimization in relational systems. In *Proceeding of ACM PODS 1998*, pages 34-43, 1998.

[4] G. Graefe: Query Evaluation Techniques for Large Databases. In *ACM Computing Surveys*, 25(2) 111-152, 1993.

[5] M. Jarke and J. Koch: Query Optimization in Database Systems. In *ACM Computing Surveys*, 16(2) 111-152, 1984.

[6] T. Ibarake and T. Kameda: On the optimal nesting order for computing N-relational joins. In *ACM Transactions on Database Systems*, 9(3) 482-502, 1984.

[7] T. Purcell: Star join optimization: DB2 UDB for z/OS and OS/390. In *The International DB2 User Group (IDUG) Solutions Journal*, 9(1) 17-19, 2002.

[8] A. Swami and B. R. Iyer: A polynomial time algorithm for optimizing join queries. In *Proceeding of the 9th IEEE Conference on Data Engineering*, pages 345-354, 1993.

[9] Yingying Tao, Qiang Zhu, and Calisto Zuzarte: Exploiting Common Subqueries for Complex Query Optimization. In *Proceedings of the 2002 CASCON*, pages 21-34, 2002.

[10] Yingying Tao, Qiang Zhu, and Calisto Zuzarte: Exploiting Similarity of Subqueries for Complex Query Optimization. In *Proceedings of the 14th International Conference on Database and Expert Systems Applications (DEXA'2003)*, Sept. 2003.

[11] Y.C. Tay: On the optimality of strategies for multiple joins. In *Journal of the ACM*, 40(5) 1067-1086, 1993.