International Journal of Information & Computation Technology.
ISSN 0974-2239 Volume 4, Number 8 (2014), pp. 841-852
© International Research Publications House
http://www. irphouse.com

Performance Analysis of Optimization Techniques for SQL Multi Query Expressions Over Text Databases in RDBMS

Swati Jain and Paras Nath Barwal

Project Engineer(IT), E-Governance CDAC, Noida
Joint Director CDAC, Noida

Abstract

This paper begins by introducing the concept of Optimization in SQL Queries. The introductory section gives brief information on the strategies, an optimizer should follow. In order to examine the role of query optimization process in RDBMS, this paper will look at both static and dynamic process of optimization as well as all the general aspects of query optimization. The paper will then explore major principles of query optimization process with volcano query optimization. An enhancement to volcano query optimizer is proposed by adding some set of transformation rules and operators. The search space principle will elaborate various types of query tree. The search strategy principle will elaborate various search strategy techniques including dynamic and greedy algorithms that will play a big role in improving the overall efficiency of relational database systems. Finally, this paper ends with the conclusion of highlighted issues and solutions.

Index Terms— query optimization, text database, volcano query optimizer, static & dynamic optimization, QEP

1.0 INTRODUCTION TO Query Optimization

At present, most of the relational database application programs are written in high-level languages and integrating a relational language like SQL for query processing with the databases. Query processing is the process of translating a query expressed in a high-level language such as SQL into low-level data manipulation operations. While query optimization refers to the process by which the best execution strategy for a given query is found from a set of alternatives. There are mainly three steps involved in query processing: decomposition, optimization and execution. First step decomposes a relational query (a SQL query) using logical schema into an algebraic query. During this step syntactic, semantic and authorization are done. Second step is responsible for generating an efficient query execution plan (QEP-also called operator tree)[1] for the given SQL query from the considered search space. Third step

implements QEP. Query optimization in relational database systems has been a traditional research problem. A number of algorithms for optimizing queries have been proposed [2, 3]. They are based on a variety of paradigm [4], and work well for the traditional relational model. However, a number of recent proposals which enhance Codd's [5] model require the modification of the existing algorithms to optimize the new set of queries that were not possible before.

There are two types of query optimization approaches [6]: static, and dynamic. Static approach consists of generating an optimal (or close to the optimal) execution plan, then executing it until the termination. All the methods, using this approach, suppose that the values of the parameters used (e.g. sizes of temporary relations, selectivity factors, availability of resources) to generate the execution plan are always valid during its execution. However, this hypothesis is often unwarranted. Indeed, the values of these parameters can become invalid during the execution due to several causes [7].

The execution plans generated by a static optimizer can be sub-optimal. As far as the dynamic optimization approach, it consists in modifying the suboptimal execution plans at run-time. The main motivations to introduce 'dynamicity' into query optimization [9], particularly during the resource allocation process, are based on: (i) willing to use information concerning the availability of resources, (ii) the exploitation of the relative quasi-exactness of parameter values, and (iii) the relaxation of certain too drastic and not realistic hypotheses in a dynamic context (e.g infinite memory).

Selecting the optimal execution strategy for a query is NP-hard in the number of relations [10]. For complex queries with many relations, this incurs a prohibitive optimization cost. Therefore, the actual objective of the optimizer is to find a strategy close to optimal solution. The selection of the optimal strategy generally requires the prediction of execution cost of the alternative candidate ordering prior to actually executing the query. The execution cost is expressed as a weighted combination of I/ O, CPU, and Communication costs [11].

The task of an optimizer is nontrivial since for a given SQL Query, there can be large number of possible operator trees:

- The algebraic representation of the given query can be transformed into many other logically equivalent algebraic representations: e.g.,
  Join(Join(A, B), C)= Join(Join(B, C), A)
- For a given algebraic representation, there may be many operator trees that implement the algebraic expression, e.g., typically there are several join algorithms supported in a database system.

Query optimization can be viewed as a difficult search problem. In order to solve the problem, it is needed to provide:

- A space of plans (search space)
- A cost estimation technique so that a cost may be assigned to each plan in the search space. Intuitively, this is an estimation of the resources needed for the execution of the plan
- An enumeration algorithm that can search through the execution space.

A desirable optimizer is one where (1) the search space includes plans that have low cost (2) the costing technique is accurate (3) the enumeration algorithm is efficient. Each of these three tasks is nontrivial and that is why building a good optimizer is an enormous undertaking. Representative examples of extensible query optimizers include Starburst [12], Volcano [13], and OPT++ [14]. This paper reports a study to enhance the Volcano extensible query optimizer to support a relational algebra with temporal operators such as temporal join and aggregation.

New algorithms can be added to a DBMS via, e.g., user-defined routines in Informix [15] or PL/SQL procedures in Oracle, but these methods currently do not allow to define functions that take tables as arguments and return tables [16]; nor do they allow to specify transformation rules, cost formulas, and selectivity-estimation formulas for the new functions. Because of these limitations, a middleware component with query processing capabilities was introduced, which divides the query processing between itself and the underlying DBMS [17]. Intermediate relations can be moved between the middleware and the DBMS by the help of transfer operators.

To adequately divide the processing, the middleware has to take optimization decisions—for this purpose, we employ the Volcano extensible middleware optimizer. Use of a separate middleware optimizer allows us to take advantage of transformation rules and cost and selectivity-estimation formulas specific to the temporal operators. There are two main advantages of using Volcano Optimizer. The first is that Volcano has gained widespread acceptance in the industry as a state-of-the-art optimizer; the optimizers of Microsoft SQL Server [18] and Tandem Server-Ware SQL Product [19] are based on Volcano. Secondly, the Volcano optimization framework is not dependent on the data model or on the execution model. This makes Volcano extensible to new data models. DBMS has its own optimizer; therefore, the middleware optimizer does not have to focus on optimizing query parts to be passed to the DBMS for evaluation.

This paper is outlined as follows. Section I provides the introduction to query optimization and also discuss briefly about the strategies, an optimizer should follow. Section 2, describe Volcano's architecture, including its search space generation and plan-search algorithms. Section 3 describes the enhancements to Volcano. The algebraic framework is described first, with a focus on the parts that posed challenges to Volcano. Then the modification to the search-space generation and plan-search algorithms of Volcano is described, then, the new set of transformation rules and operators has been proposed to work with volcano query optimizer. Various search strategies has also been introduced in this section. Section 4 evaluate Cost Model. Section 5 summarizes the paper with conclusion.

Study of decomposition & execution steps of query processing, is outside the scope of this paper.

2.0 Description of Volcano Optimizer

Volcano optimizer is extensively used to optimize SQL queries. It optimizes queries in two stages. First, the optimizer generates the entire search space consisting of

logical expressions generated using the initial query plan (to which the query is mapped to) and the set of transformation rules. The search space is represented by a number of equivalence classes. An equivalence class may contain one or more logically equivalent expressions, also called elements; each of these includes an operator, its parameter (for example, predicates for the selection), and pointers to its inputs (which are also equivalence classes). Consider a simple example query, which performs a join on the EmpId attribute of POSITION and SALARY relations. It's one possible initial plan is shown in Figure 1(a) and its search space is shown in Figure 1(b).

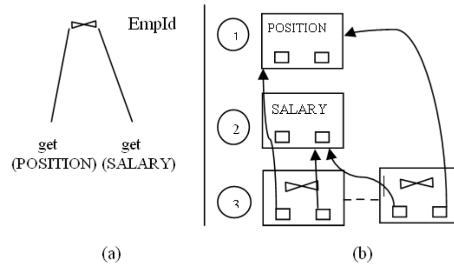

Figure 1: Initial Query Plan

The elements of classes 1 and 2 represent logical expressions returning partial results of the query, i.e., the operators retrieving, respectively, the POSITION and SALARY relations. The elements of class 3 represent logical expressions returning the result of the complete query; either the first or the second element may be used. Essentially, the given search space represents only two plans which differ in the order of the join arguments.

During the second stage of Volcano's optimization process, the actual search for the best plan is performed. Here, the implementation rules are used to replace operators by algorithms, and the costs of diverse sub-plans are estimated. For the given query, the number of plans to be considered is greater than two, because the relations may be retrieved by using either full scan or index scan, and the join has several implementations, such as nested-loop, sort-merge, or index join. One possible evaluation plan is to scan both relations and perform a nested-loop join.

The following two sections describe the search-space generation and the plan-search algorithms in more detail.

2.1 Stage One: Search-Space Generation

The two main functions for the search-space generation are Generate and MatchRule. In these functions initially one element is created for each operator in the original query expression, and then Generate function is invoked on the top element. The Generate function repeatedly invokes the MatchRule function, which applies a transformation rule to the given element, choosing from the list of applicable rules that have not so far been applied to the element (as found by the FindMatchRule function in MatchRule). The application of a transformation rule in MatchRule may trigger the creation of new elements and classes; for each newly generated element,

the Generate function is invoked.

For the query in Figure 1(a), the search space is generated as follows. Initially, three elements representing the three query-tree operators are created (first elements of equivalence classes 1–3 in Figure 1(b)). Then, the Generate function is invoked for the first element of class 3, which, in turn, invokes Generate for the first elements of classes 1 and 2. The latter two Generate calls do not do anything because no rules apply to the elements of class 1 and 2. For the first element of class 3, however, the join commutativity rule is applied, and a second element pointing to switched join arguments is added to class 3. Then, the MatchRule function is invoked on the new element of class 3, but no new elements are generated: the join commutativity rule is applied again, but its resulting right-hand element already exists in the search space.

2.2 Stage Two: Plan Search

When searching for a plan, the Volcano optimizer employs dynamic programming in a top-down manner, and it uses two mutually recursive functions, FindBestPlan and Optimize.

First, the optimizer invokes the FindBestPlan function for the first element of the top equivalence class—e.g., class 3 in Figure 1(b)—and the cost limit infinity (the cost limit can be lower in subsequent calls to the function). If all elements of the class containing the argument element have already been optimized, no further optimization for the element is necessary: if the plan has been found and its cost is lower than the cost limit, it is returned, otherwise NULL is returned. However, if the optimization for the class has not been completed, the function is invoked.

The Optimize function for each algorithm implementing the top operator (in our case, join) recursively invokes the FindBestPlan function for the inputs of the algorithm. The RemainingCost argument is used to prune the search when it is clear that the search would not come up with a more efficient plan than the current one. If optimization of the inputs is successful, and if the found plan is the first for the equivalence class containing the argument element or it beats the cost of the existing best plan, it is saved along with its cost.

Once all algorithms are considered for the operator, the Optimize function invokes the FindBestPlan function for each equivalent logical expression (in our case, for the second element in equivalence class 3) and looks if it can find a better plan. In case a better plan is found, it is saved in memory as the best one. Once all elements of the input-element class are considered, the algorithm marks that the optimization of the class is completed.

3.0 Enhancement of Volcano Optimizer

The implementation of the algebra and its accompanying transformation rules introduces several concepts that did not exist previously in Volcano. We describe these new concepts in Section 3.1. Sections 3.2 and 3.3 concern the actual implementation and describe the modifications of the Volcano search-space generation and plan-search algorithms.

3.1 Algebra and Transformation Rules

First, we overview the architecture for which the algebra has been designed. Next, we describe the actual algebra, the accompanying transformation rules, and their applicability, focusing on the new concepts.

Architecture The temporally extended relational algebra has been designed for an architecture consisting of a middleware component and an underlying DBMS. Expensive temporal operations such as temporal aggregation do not have efficient algorithms in the DBMS, but can be evaluated efficiently by the middleware, which uses a cursor to access DBMS relations. Consequently, query processing is divided between the middleware and the DBMS; the main processing medium is still the DBMS, but the middleware is used when this can yield better performance.

3.1.1 ALGEBRA

This algebra is different from the conventional relational algebra in several aspects. First, it includes temporal operators such as temporal join and temporal aggregation. Next, it contains two transfer operators, T^M and T^D, that allow to partition the query processing between the middleware and the underlying DBMS. The T^M operator transfers a relation from the DBMS to the middleware, and the T^D operator performs the opposite. Finally, the algebra provides a consistent handling of duplicates and order at logical level, by treating duplicate elimination and sorting as other logical operators and by introducing six types of equivalences between relations.

Figure 4 shows two temporal relations (relations having two attributes that encode a time period), POSITION and SALARY. We assume a closed-open representation for time periods and assume the time values for T1 and T2 denote months during some year. For example, Tom was occupying position Pos1 from February to August (not including the latter).

POSITION

Pos Id	EmpId	EmpName	T1	T2
Pos1	1	Tom	2	8
Pos2	2	Jane	3	8

SALARY

Emp Id	Amount	T1	T2
1	100K	2	6
1	120K	6	9
2	110K	3	8

RESULT-SET

Emp Id	Emp Name	Pos Id	Amount (K)	T1	T2
1	Tom	Pos1	100	2	6
1	Tom	Pos1	120	6	8
2	Jane	Pos2	110	3	8

Figure 4: Relations POSITION, SALARY and the Result Set of Temporal Join.

A temporal join is a regular join, but with a selection on the time attributes, ensuring that the joined tuples have overlapping time periods; Figure 4 shows the result of temporal join on the EmpId attribute of the POSITION and the SALARY relations.

3.1.2 Transformation Rules :

Six types of equivalences lead to six types of transformation rules, since a transformation rule may satisfy several of the six equivalences. Let us consider two rules for temporal join, T (regular join, but with a selection on the time attributes, ensuring that the joined tuples have overlapping time periods). For a given rule, we always specify the strongest equivalence type that holds; the ordering of equivalence types is given in Figure 5. The another rule exploits the fact that all temporal join algorithms in the middleware retain the sorting of their left arguments.

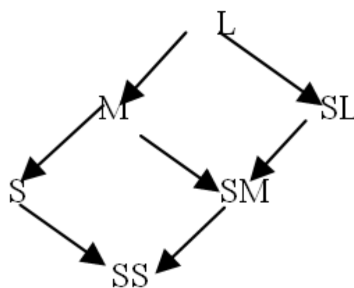

Figure 5: Ordering of Equivalence Types

3.2 The Search Strategy

One central component of a query optimizer is its search strategy or enumeration algorithm. The enumeration algorithm of the optimizer determines which plans to enumerate, and classically is based on dynamic programming.

There are basically two approaches to solve this problem. The first approach is the deterministic strategies that are proceeded by building plans, starting form base relations, joining one more relation at each step till the complete plans are obtained. When constructing QEPs through dynamic programming, equivalent partial plans are constructed and compared on some cost model. To reduce the optimization cost, partial plans that are not likely to lead to the optimal plan are pruned (discarded) as soon as possible, and the cheaper partial plans are retained and used to construct the full plan. A greedy strategy builds only one such plan using depth-first search, while dynamic programming builds all possible plans breadth-first. The other approach is the randomized strategies that concentrate on searching the optimal solution around some particular points. They do not guarantee that the optimal plan is obtained, but avoid the high cost of optimization, in terms of memory and time consumption [20].

3.2.1 Deterministic Strategies

In this section, the basic deterministic strategies, i. e. dynamic programming and greedy algorithm will be discussed as well as a brief discussion of a combination approach of both algorithms.

3.2.1.1 Dynamic Programming

This algorithm is pioneered in IBM's System R project [21] and it is used in almost all commercial database products [22]. The basic dynamic programming for query optimization as presented in [8]. It works in bottom-up way by building more complex sub-plans from simpler sub-plans until the complete plan is constructed. In the first phase, the algorithm builds access plan for every table in the query. Typically, there are several different access plans for a relation (table). If relation A, for instance, is replicated at sites S1 and S2, the algorithm would enumerate table-scan (A, S1) and table-scan (A, S2) as alternative access plans for table A. In the second phase, the algorithm enumerates all two-way join plans using the access plans as building blocks. Again, the algorithm would enumerate alternative join plans for all relevant sites; i. e. consider carrying out joins with A at S1 and S2. Next, the algorithm builds three-way join plans, using access-plans and two-way join plans as building blocks. The algorithm continues in this way until it has enumerated all n-way join plans. In the third phase, the n-way join plans are massaged by the finalizePlans function so that they become complete plans for the query;

In DBMS, neither table-scan (A, S1) nor table-scan (A, S2) may be immediately pruned in order to guarantee that the optimizer finds a good plan. Both plans do the same work, but they produce their result at different sites. Even if table-scan (A, S1) is cheaper than table-scan (A, S2), it must be kept because it might be a building block of the overall optimal plan if, for instance, the query results are to be presented at S2. Only if the cost of table-scan (A, S1) plus the cost of shipping A from S1 to S2 is lower than the cost of table-scan (A, S2), table-scan (A, S2) is pruned.

Lanzelotte et al. [20] addressed an important issue, i.e. which plans are equivalent in order to prune the expensive ones? At first glance, equivalent partial plans are those that produce the same result (tuples). In fact, the order of resulting tuples is important equivalence criterion. The reason is that in the presence of sort-merge join; a partial with a high cost could lead to a better plan, if a sort operation could be avoided. The researchers made some experiments showing that dynamic programming performs better than a randomize strategy for queries with small number of relations, but this situation is inverted when the query has 7 relations or more.

3.2.1.2 Greedy Algorithm

As an alternative to dynamic programming, greedy algorithms have been proposed. These greedy algorithms run much faster than dynamic programming, but they typically produce worse plans [23]. Just like dynamic programming, this greedy algorithm has three phases and constructs plans in a bottom-up way. It makes use of the same accessPlans, joinPlans, and finalizePlans functions in order to generate plans. However, in the second phase this greedy algorithm carries out a very simple and rigorous selection of the join order. With every iteration of the greedy loop, this algorithm applies a plan evaluation function, in order to select the next best join. Obviously, the quality of the plans produced by this algorithm strongly depends on the plan evaluation function [9].

4.0 Cost Model

Lanzelotte, Valduriez, and Zait [12] introduced a cost model that captures all aspects of parallelism and scheduling. They define the cost estimate of a QEP containing only join nodes. All formulas given below compute response time and they simply refer to it by cost. In addition to the traditional assumptions, uniform distribution of values and independence of attributes, they also assume that tuples of a relation are uniformly partitioned among nodes of different homes, and there is no overlap between nodes of different homes, although several relations may share the same home.

In the following, R refers to a base relation of the physical schema, and N to the operation captured by QEP node. P denotes, in the same time, a QEP and the transient relation produced by that QEP. The parameters, database schema or system parameters used in the cost model are shown in Table 1

An optimal execution of the join operation requires each operand to be partitioned the same way. For example, if p and q are both partitioned on n nodes using the same function on the join attribute, the operation join(p, q) is equivalent to the union of n parallel operations join(pi, qi), with I = 1, …, n. If the above mentioned condition is not satisfied, parallel join algorithm attempt to make such condition available by recognizing the relations, i. e. dynamically repartitioning the tuples of the operand relations on the nodes using the same function on the join attribute.

Table 1: Cost model parameters.

card(R)	Number of tuples in relation R
width(R)	Size of one tuple of relation R
cpu	CPU speed
network	Network speed
packet	The size of packet
Send	The time for a send operation
receive	The time for a receive operation

First, estimate the cost of partitioning an operand relation R. Obviously, if the relation is appropriately partitioned, this cost is 0. Let # source be the number of nodes over which R is partitioned, and # dest be the number of nodes of the destination home. Each source node contains card(R)/ # source tuples. Thus it will send card(R) * width(R)/ (n * packet) packets. If we assume that tuples will be uniformly distributed on destination nodes, then each node will receive card(R)/ # dest tuples, and thus will process card (R) * width(R)/ (m * packet) incoming packets. Since a destination node starts processing only when first packet arrives, the cost of repartitioning R on # dest nodes is:

cost (part(R)) = max((card(R) * width(R)/ (# source * packet)) * send, (card(R)* width(R)/ (# dest * packet)) * receive + send + packet/ network)

The cost of joining tuples of p and q, where p and q are, respectively, the pipelined and stored operands of the join operation, is:

cost (join(p, q)) = max(costalg (join(p, q)), cost(part(p))) + cost(part(q)).
where costalg(join(p, q)) is the cost to process the join at one node. It depends on the join algorithm used. The partitioning of p is performed simultaneously to the join processing, after the repartitioning of q has completed.

5.0 Conclusions

We have studied the problem of distributed query optimization. We focus on the major optimization issues being addressed in distributed databases. We have seen that a query optimizer is mainly consists of three components: The search space, the search strategy, and the cost model. Different kinds of search spaces are discussed with different schedules. Search strategies, the central part of the optimizer, can be seen as two classes. We have shown that all published algorithms of the first class, i. e. deterministic strategies, have exponential time and space complexity and are guaranteed to find the optimal plan. Whereas, the big advantage of the algorithms of the second class, i. e. randomized algorithms, is that they have constant space overhead. Typically, randomized algorithms are slower than heuristics and dynamic programming for simpler queries but this is inverted for large queries. Randomized strategies do not guarantee to find the optimal plan. Some cost models are discussed and the basic parameters for parallel environment are shown.

REFERENCES

[1] Graefe, G.: Query Evaluation Techniques for Large Databases. ACM Computing Survey 25(2), 73–170 (1993)

[2] Selinger, P. et al., “Access Path Selection in a Relational Database Management System, ” Proc. ACM-SIGMOD Conference on Management of Data, 1979.

[3] Wong, E., and Youssefi, K., “Decomposition-A strategy for query processing, ” ACM Trans. on Database Systems, Sept. 1976

[4] Jarke, M., and Koch, J., “Query Optimization in Database Systems, ” ACM Computing Surveys, June 1984.

[5] Codd, E.F., “A Relational Model of Data for Large Shared Data Banks, ” Comm. Of ACM, June 1970.

[6] Cole, R.L., Graefe, G.: Optimization of dynamic query evaluation plans. In: Proc. of the 1994 ACM SIGMOD, vol. 24, pp. 150–160. ACM Press, New York (1994)

[7] Morvan, F., Hameurlain, A.: Dynamic Query Optimization: Towards Decentralized Methods. Intl. Jour. of Intelligent Information and Database Systems (to appear, 2009) Morvan, F., Hameurlain, A.: Dynamic Query Optimization: Towards Decentralized Methods. Intl. Jour. of Intelligent Information and Database Systems

[8] Ioannidis, Y.E., Christodoulakis, S.: On the Propagation of Errors in the Size of Join Results. In: Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, pp. 268–277. ACM Press, New York (1991)
[9] Cole, R.L., Graefe, G.: Optimization of dynamic query evaluation plans. In: Proc. of the 1994 ACM SIGMOD, vol. 24, pp. 150–160. ACM Press, New York (1994)
[10] Ibraraki T. and Kameda T. "Optimal Nesting for Computing N-Relational Joins, " ACM Transactions on Database Systems, vol. 9, no. 3, pp. 482-502, 1984.
[11] Oszu M. T. and Valduriez P., Principles of Distributed Database Systems, Prentice Hall International, NJ, 1999
[12] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. G. Lindsay, H. Pirahesh, M. J. Carey, and E. J. Shekita. Starburst Mid-Flight: As the Dust Clears. IEEE TKDE, 2(1):143–160 (1990).
[13] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In Proceedings of IEEE ICDE, Vienna, Austria, pp. 209–218 (1993).
[14] N. Kabra and D. J. DeWitt. OPT++: An Object-Oriented Implementation for Extensible Database Query Optimization. VLDB Journal, 8(1):55–78 (1999).
[15] R. Bliujute, S. Saltenis, G. Slivinskas, and C. S. Jensen. Developing a DataBlade for a New Index In Proceedings of IEEE ICDE, Sydney, Australia, pp. 314–323 (1999).
[16] M. Jaedicke and B. Mitschang. User-Defined Table Operators: Enhancing Extensibility for ORDBMS. In Proceedings of VLDB, Edinburgh, Scotland, pp. 494-505 (1999)
[17] G. Slivinskas, C. S. Jensen, and R. T. Snodgrass. Adaptable Query Optimization and Evaluation in Temporal Middleware. In Proceedings of ACM SIGMOD, pp. 127–138 (2001)
[18] GRAEFE, G. The Cascades Framework for Query Optimization. Data Engineering Bulletin 18, 3 (1995).
[19] CELIS, P. The Query Optimizer in Tandem's new Server-Ware SQL Product. In Intl. Conf. Very Large Databases (1996).
[20] Lanzelotte R. S. G., Valduriez P., Zait M., and Ziane M., "Industrial-Strength Parallel Query Optimization: Issues and Lessons, " Information Systems, vol. 19, no. 4, pp. 311-330, 1994.
[21] GRAEFE, G. Query Evaluation Techniques for Large Databases. ACM Computing Surveys 25, 2 (1993).
[22] GRIFFIN, T., AND LIBKIN, L. Incremental maintenance of views with duplicates. In ACM SIGMOD Intl. Conf. on Management of Data (1995).
[23] GRAEFE, G., AND MCKENNA, W. J. The Volcano Optimizer Generator: Extensibility and Efficient Search. In Intl. Conf. on Data Engineering (1993)