PaperID: 507

Title: **Dynamic Re-grouping of Continuous Queries**

Authors: **Jianjun Chen** **(contact author)**
David J. DeWitt

Address: **Computer Sciences Department**
University of Wisconsin-Madison
1210 West Dayton Street,
Madison, WI 53705
USA

Email: **jchen@cs.wisc.edu**

Phone: **(608)2626625**

Topic Area: **Core Database Technology**

Category: **Research**

Topics: **Internet and WWW**

Dynamic Re-grouping of Continuous Queries

Jianjun Chen David J. DeWitt

Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton Street, Madison, WI 53705
{jchen, dewitt}@cs.wisc.edu

Abstract

In this paper, we design and evaluate an efficient and dynamic regrouping approach to optimize a large continuous query workload. The key idea of our regrouping algorithm is to find a best solution by removing redundant groups within the existing solution from incremental group optimization. Since the existing query groups usually constitute a very small portion of the entire search space, such a heuristic-based approach is key to achieving an efficient regrouping algorithm. In addition, timing of regrouping is also critical to the efficiency of regrouping. Overall, we believe that constant incremental grouping in conjunction with occasional dynamic regrouping can achieve a high-quality grouping at a fairly low cost for optimizing a large, dynamic continuous query workload.

1. Introduction

Continuous queries have attracted considerable attention recently and have been studied in [TGNO92] [LPBZ96] [CDTW00][CDN02]. In [CDTW00], an incremental grouping approach is proposed to efficiently group new continuous queries without having to regroup existing queries. Even though such an approach is efficient and scalable, it may result in a sub-optimal global plan, since existing queries are not regrouped to exploit new grouping opportunities introduced by subsequent queries. Since queries are continuously being added and removed from groups, over time the overall quality of the groups is very likely to deteriorate, leading to a reduction in the overall performance of the system. In this case, existing groups may require "dynamic re-grouping" to re-establish their effectiveness. We use a simple example for motivating purposes.

Example 1.1 Let Q1 (A⋈B⋈C) and Q2 (B⋈C) be two continuous queries that are submitted to the system sequentially. Assume that the incremental grouping algorithm chooses a plan ((A⋈B)⋈C) for the first query, and creates two join groups with join signatures (A⋈B) and (A⋈B⋈C), respectively. When Q2 is submitted to the system, neither of these groups can be used for Q2. Thus, a separate join group (B⋈C) must be created. However, a possible better global plan would have Q1 share the (B⋈C) group with Q2, removing the need for group (A⋈B).

In addition, an existing continuous query can be removed from the system, either because the query has expired or has been explicitly deleted. Simply removing existing groups used by the query can also reduce the overall quality. Thus, it may be beneficial to regroup existing queries periodically, as continuous queries are added and removed continuously.

This paper focuses on an efficient and dynamic query regrouping strategy for large, dynamic continuous query workloads. Our regrouping method, when applied in conjunction with the incremental grouping, can obtain a reasonable improvement over the incremental grouping method at a low extra overhead in regrouping time. Since continuous queries can be executed very frequently and they can also be added to and removed from the system at any time, the regrouping must be efficient enough so as not to impose a significant burden on the system.

A naive regrouping-algorithm would periodically perform a traditional global query optimization [RC88] [Sel86] over all existing queries. Such a process would be prohibitively expensive and no known algorithm can handle a continuous query workload with potentially hundreds of thousands of continuous queries. Furthermore, such a strategy does a significant amount of redundant work that has already been done by incremental group optimizations when queries were initially installed.

**Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002**

The key idea of our regrouping algorithm is to find a good solution by removing redundant groups within the existing solution from incremental group optimization. Since existing query groups usually constitute a very small portion of the entire search space, this heuristic is crucial to achieving an efficient regrouping algorithm. In addition, our algorithm can also reuse previous regroupings in order to avoid repetitive computations. Furthermore, the timing of regrouping is also very important to the regrouping efficiency. In this paper, we explore ways of determining a good regrouping interval using both analytical and experimental methods. We also propose an *active regrouping* approach in which regrouping can be dynamically invoked.

The rest of the paper is organized as follows. In Section 2 we illustrate several related grouping problems using a graph representation and present a formal definition of those problems. Our incremental grouping and regrouping algorithms are presented in Section 3. Section 4 examines the performance of the regrouping algorithm. Related work is described in Section 5. We conclude our paper and suggest some important future research directions in Section 6.

2. Graph Representations and Problem Definitions

This section presents the graph representation used by our algorithms. Even though the model itself can be extended to incorporate other operators, in this paper, we consider queries composed of only join operators. Since joins are among the most expensive operators in database systems, dynamically regrouping joins can significantly improve the entire system's performance. In addition, one potentially useful heuristic for optimizing a large number of continuous queries with similar join operators is to pull up selections over joins, allowing queries to share materialized join results [CDN01]. Thus, efficiently regrouping joins is important.

We then use the model to define our problems in a formal manner. Since the search space for all these problems is exponential in the size of queries in the workload, our focus is to find good heuristic-based solutions in a practical setting.

2.1 Graph Representations

Figure 2.1 shows a query graph of the grouped query plan after the example query workload {A⋈B⋈C, B⋈C} has been incrementally grouped. The left diagram in Figure 2.1 shows the plan before the regrouping, and the right one shows it after regrouping. Each query expression is represented by a node in the query graph, e.g. node AB represents the join A⋈B. In the following discussions we use nodes and queries interchangeably to express the same meaning, as long as doing so creates no confusion. A node is termed a **final node** if it represents a user-defined query, e.g. node ABC and BC in the above example (shown shaded in Figure 2.1). Final nodes correspond to user queries and must always be retained. Non-final nodes (e.g. node AB, shown non-shaded), however, are just intermediate computations for computing the final nodes and they can be removed when better alternatives arise.

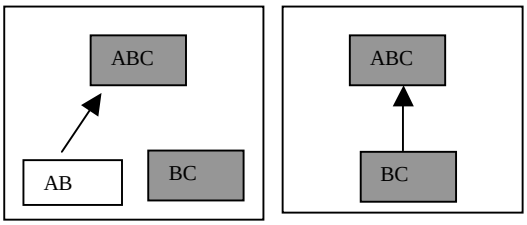

Figure 2.1: The query graphs after query workload {A⋈B⋈C, B⋈C} have been incrementally grouped. The left graph shows the group plan before regrouping. The right graph shows it after regrouping.

We now provide a brief description of the terminology used in this paper. A query x is termed a *sub-query* of another query y if x is defined over a subset of input files of y. We also say y *contains* x. In this discussion, we ignore the join predicates and thus, each join can be uniquely determined by its input files. Our model can be extended without difficulty to handle join predicates by creating multiple nodes for joins with the same input files but different join predicates. For the purpose of simplicity, we omit the join symbols in a query, e.g. AB for A⋈B. When referring to a query graph, we use terminology that is commonly used in graph theory. If query x is a sub-query of query y, node x is termed a *child* of node y and node y is termed a *parent* of node x. All the children under the same parent are called *siblings*. An edge from node AB to node ABC represents a possible access method to compute A⋈B⋈C from A⋈B. In our discussions, we only consider left-deep plans for each query. With this assumption, children of a node with n joins can only be nodes with n-1 joins or input files. Hence, our graph ignores input files, and each edge now uniquely identifies a join. The weight of an edge is the cost of corresponding operations, i.e. the cost of join between A⋈B and C.

One important point to observe from the left diagram in Figure 2.1 is that there is no edge between node BC and node ABC before regrouping occurs. This is because node BC is created after the first query ABC is installed. Thus, a group optimizer that only optimizes incrementally will miss potential grouping opportunities with groups that get generated for subsequent queries. However, our regrouping algorithms are able to solve this problem, and the expected group plan after regrouping is presented in the right graph of Figure 2.1.

2.2 Execution Costs of Nodes

In our system [CDTW00][CDN02], queries are triggered for execution by data changes on the input files. Intermediate query results are materialized and stored in intermediate files. Queries are evaluated against data changes and intermediate files are incrementally maintained. Thus, the total execution cost of each node includes the cost of query evaluation and the cost of maintaining the persistent intermediate files.

In this paper, for the simplicity purpose, we use the update frequency of a node as the metric of the cost for computing that node. Let $U_f(AB)$ represent the total update frequency over node AB, the cost of computing node AB will be proportional to $U_f(AB)$. In our approach, simultaneous updates to both join inputs will cause the join to be evaluated twice, each time with changes to one input only. Thus, a reasonable approximation to the total possible update frequency of a node is the sum of update frequencies of each child node. In the above example, $U_f(AB)= U_f(A) + U_f(B)$.

2.3 Problem Definitions

Using the graph model presented in Section 2.1, we provide a formal definition of three problems related to our study, namely *global optimization*, *incremental group optimization*, and *dynamic regrouping*. Such a formal definition can provide useful insights into these problems.

Definition 2.1 (A) Given a set of "input files" $S=\{s_1, \ldots, s_n\}$, and given a set of "queries" $Q=\{Q_1, \ldots, Q_k\}$, where $Q_i \subseteq S$, a ***global plan*** A is a set $\{A_1, A_2, \ldots, A_m\}$ s.t.

(1) for $1 \le i \le K$, $Q_i \in A$; for $1 \le j \le n$, $\{s_j\} \in A$

(2) $\forall A_i$, either $A_i \in S$, or $\exists A_j \in A$ and $\exists A_k \in S$,

$$A_i = A_j \cup A_k \, (A_j \cap A_k = \varnothing).$$

(B) The ***global optimization*** problem is the problem of finding an *optimal global plan* A s.t. A is a *global plan* and |A| is minimal, where |A| represents the sum of the cost of each node in A.

Definition 2.2 Given $<S, Q, A^k, Q_{k+1}>$, where S is a set of "input files" $\{s_1, \ldots, s_n\}$, Q is a set of "queries" $\{Q_1, \ldots, Q_k\}$, where $Q_i \subseteq S$, A^k is a *global plan* of Q, and Q_{k+1} is another query, the ***incremental group optimization*** problem is the problem of finding a set $B= \{B_1, B_2, \ldots, B_p\}$ s.t.

(1) $Q_{k+1} \in B$,

(2) $\forall B_i$, either $B_i \in S$, or $\exists B_j \in A^k \cup B$ and $\exists B_k \in S$,

$$B_i = B_j \cup B_k \, (B_j \cap B_k = \varnothing),$$

(3) |B| is minimal

Due to the dynamic nature of this problem definition, let $A^{k+1} = A^k \cup B$. It is not hard to prove that A^{k+1} is a *global plan* of $\{Q_1, \ldots, Q_{k+1}\}$.

The definition shows that incremental group optimization aims to find an optimal solution (i.e. B) for a new query (i.e. Q_{k+1}) with respect to a current *global plan* A^k for existing queries. Solution A^{k+1} for the K+1 queries, however, may not be an *optimal global plan*.

Definition 2.3 Given a set of "input files" $S=\{s_1, \ldots, s_n\}$, a set of "queries" $Q=\{Q_1, \ldots, Q_k\}$, where $Q_i \subseteq S$, and given a *global plan* A of Q, the ***dynamic regrouping*** problem is a problem of finding an *optimal global plan* B s.t. $\| B - A\|$ minimal, where $\| B - A\|$ represents a measure of differences between these two *global plans*.

Dynamic regrouping seeks to find an *optimal global plan* (B) that is "closest" to a given *global plan* (A). If B can be any *optimal global plan*, *dynamic regrouping* becomes the same problem of *global optimization*. The intuition behind the "closest" constraint is an attempt to reuse part of an existing global plan. This approach has two potential advantages. First, it may reduce time spent regrouping. This is especially desirable when we are confident of the quality of existing plans and don't want to re-optimize all queries from scratch. Second, it can reuse existing computation results. In our system, since join results of nodes are materialized, reusing existing plans allows materialized results to be used thereafter. While there can be many possible measures of the differences between the two solutions before and after regrouping, one reasonable difference measure would be the total cost of nodes in the set of nodes that are in B but not in A.

Finding a *global optimal plan* has an exponential search space over entire sub-trees of all queries in the query workload. To make things worse, the extra constraint of seeking a specific optimal solution that is "closest" to the existing solution makes the problem even more difficult. We hence modify the *dynamic regrouping* definition to finding a plan B that is a subset (generally, a proper subset) of A with a minimum weight. Such a definition can be thought of as an approximation of two very difficult goals: that B is an *optimal global plan* and that B is "closest" to A, in the Definition 2.3. Thus, B can be computed by removing redundant nodes in A. A problem definition of this new approach is given as follows.

Definition 2.4 Given a set of "input files" $S=\{s_1, \ldots, s_n\}$, and given a set of "queries" $Q=\{Q_1, \ldots, Q_k\}$, where $Q_i \subseteq S$, and given a *global plan* A of Q, the ***dynamic regrouping*** problem is a problem of finding a *global plan* B s.t. $B \subseteq A$ and |B| is minimum.

Since query nodes in the existing *global plan* usually constitute a very small portion of the entire search space required to find a globally optimal solution, the heuristic used in Definition 2.4 is critical to achieving an efficient regrouping algorithm. However, the *global plan* B in Definition 2.4 is not necessary an *optimal global plan*.

3. Algorithms

This section presents details of our dynamic regrouping algorithms.

3.1 Important Data Structures

In our algorithms, three main data structures are used.

(1) A query graph is a directed acyclic graph, with each node representing an existing join expression in the group plan. The structure of a node is shown in Figure 3.1. Each node contains a list of parent edges and a list of child edges. The weight of a child edge represents the cost of a join using that child to compute the parent. A weighted child edge corresponds to an AND node in an AND-OR DAG representation [Rou82][GM93]. Each node contains the ASCII representation of the query and a hash-based signature computed from the query string. In addition, each node also contains data structures specific for regrouping. The usage of query signatures and data structures for regrouping is discussed when we describe the algorithms.

```
Node {
 char* query;                 //ASCII query plan
 SIG_TYPE sig;                //signature of the query string
 int final_node_count;        //number of users that require
                               this query. 0: non-final node;
                               >0: final node
 list<Child*> children;       //children of this node, where
                               Child ={ Node*, weight }
 list<Node*> parents;         //parents of this node
 float updateFreq;            //update frequency of this node
 float cost;                  //the cost for computing this node

 //Following data structures used only for dynamic regrouping
 int reference_count;         //reference count
 bool visited;                //a flag that records whether
                              purgeSibling has performed on this node
}
```

Figure 3.1: Data structure of a query node.

(2) A group table is an array of hash-tables. In the following discussion, we refer to the number of joins in a query as the length of the query. All queries with the query length i are stored in the i-th hash table. Each hash table entry consists of a mapping from a query string to a pointer to the corresponding query node in the query graph.

(3) A query log is an array of vectors that are used to store new nodes that have been added to the query graph since the last regrouping. Similar to a group table, new queries with the same length are stored in the same vector in the query log. After each regrouping, the content of a query log is cleared.

3.2 Incremental Grouping Algorithms

Given a new query, incremental group optimization [CDTW00] attempts to find the optimal solution to this new query from all possible solutions, with or without using existing nodes. Since existing nodes must be computed regardless of this new node, the overall cost for the new query is the sum of the costs of all new nodes added.

We designed an algorithm, termed *top-down local exhaustive search*, to find an optimal incremental plan for a new query. This algorithm exhaustively enumerates all possible sub-query in a top-down manner and probes the group table to check whether a sub-query node exists. For each existing sub-query node, we compute the minimal cost of using this sub-query node to compute the new query. The minimal cost of a path without using any existing nodes can be computed similarly. The least-costly plan will be chosen ultimately.

This algorithm is very simple and has a fixed cost for searching entire space of a new query. Since the length of most queries is usually not long (e.g. less than 10 joins), the cost is not generally high. One special, but not uncommon case is that the new query itself exists in the query graph. In this case, we just increase the final node count (Figure 3.1) by 1 to indicate that this query is required by one more user.

3.3 Dynamic Regrouping Algorithms

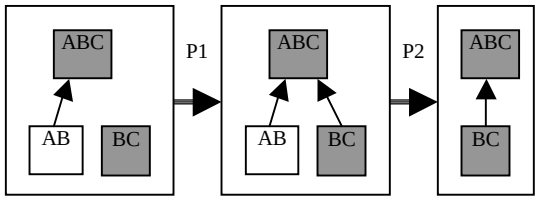

Figure 3.2: An illustration of our two-phrase regrouping algorithm using the query workload in Figure 2.1.

Our regrouping consists of two phases (Figure 3.2). The first phase (P1 in Figure 3.2) is to construct missing edges among nodes that have a sub-query relationship. For example, in Figure 2.1, there is no link between node ABC and node BC before regrouping, because node BC is created later than node ABC. Thus node ABC is not aware of the existence of node BC as a potential grouping opportunity. Thus, the first phase of our regrouping algorithm is to construct links between existing nodes and nodes that were added since the last regrouping was performed (node ABC and BC in Figure 3.2).

The goal of the second phase of regrouping is to find a minimal-weighted solution from the current solution by removing redundant nodes (P2 in Figure 3.2).

In the following discussion, we provide a description and a simple cost analysis of our regrouping algorithm.

3.3.1 Phase 1: constructing links among existing nodes and new nodes

The main idea of this algorithm is that for any pair of nodes in the graph, if one node is a sub-query of another node, it creates a link between them if it did not exist before.

Two important optimizations can significantly reduce phase-1 computation. First, since all possible edges have been created during previous regroupings, it is not necessary to re-compute sub-query relationships among the nodes that existed prior to the last regrouping. (*Attentions in phase-2 regrouping are required for this assertion to be hold and will be discussed in next section.*) Such relationships are, thus, only evaluated between existing nodes and nodes added since the last regrouping, which are stored in the query log. Since the number of existing nodes can be very large, this optimization can significantly reduce the computation costs associated with phase-1 regrouping. Second, since we don't generate a bushy tree plan, the difference of levels between a parent and a child is always 1. Thus, we need not construct links between nodes whose lengths differ more than 1. This is why nodes are separately stored according to their length in both group tables and query logs.

Based on the above optimizations, our algorithm works as follows: from bottom-up, it checks whether each node in the level i query log has any parents in the level i+1 group table. If so, the node is connected to its parent nodes. A similar process is done for nodes in the level i query log and nodes in the level i-1 group table. These steps effectively compute a nested-loop join among nodes in the level i query log and nodes in the level i+1 and the level i-1 group table. The join condition is a subset relationship if we treat each query as a set of data input file names. To evaluate the set-containment operation efficiently, the signature of each query is used. Given signature sig_C and sig_P for node C and P, respectively, if sig_C & ~sig_P is not true, we know that C is definitely not a sub_query of P. The advantage of using a signature is that it can quickly eliminate cases for which set containment does not hold, and it never generates "false drops". However, as in any hash-based algorithms, false matches can occur. When a match of signatures occurs, validation using query strings is required. Signature-based algorithms are fairly efficient for regrouping continuous queries, since most nodes do not have many edges. Thus, most set-containment check would be false and can be recognized immediately using signatures.

3.3.2 Phase 2: A greedy algorithm for level-wise graph minimization (*Level*)

Finding a set of nodes with minimum weight from an existing solution still has an exponential searching space. Thus, the motivation of *Level* algorithm is to use a greedy algorithm to find a good solution at each level to reduce the search space. At any time, the *Level* algorithm considers only nodes of two adjacent levels.

```
MinimizeGraph() {
    for each level L in group-tables {
    // L ranging from the maximum number of join-1 to 1
        for each node N in the level-L group table {
            InitializeSet(N)
        }
        for each node N in finalSet
            PurgeSiblings(N);
        while (remain set is not empty){
            scan each node R in the remain set {
                if (R's reference count == 0) {
                    remove R from the remain set
                    deleteNode(R)
                }
                else if (R.cost/R.reference_count
                        < Current_minimum) {
                    M=R
                    Current_minimum
                        =R.cost/R.reference_count;
                }
            }
            remove M from the remain set
            PurgeSiblings(M)
        }
    }
}

InitializeSet(Node N) {
    if N is a final node
        Add N into final_set
    else {
        add N into the remain_set
        N.reference_count = number of parents of N
    }
    N.visited = false
}

purgeSiblings(Node N) {
    For each parent P of N {
        if (!P.visited) {
            Decrease the reference count of N's
                siblings of same parent P by 1
            P.visisted = ture
    }
}
```

Figure 3.3 Important modules of the *Level* algorithm.

The main idea of the *Level* algorithm (Figure 3.3) is to traverse the query graph level-by-level and attempt to remove any redundant nodes at one level a time. This traversal can be done either top-down or bottom-up. Our current implementation uses a top-down approach. Since all top-level nodes in the group plan are final-nodes and hence must be kept, the level-wise minimization procedure starts from the second level from the top. Assuming that level i is being processed, a subset of level i nodes, which satisfy the following two conditions is

chosen to retain. First, all nodes at level i +1, that have been processed already, must have at least one child in this set. This constraint guarantees that each node at level i+1 has at least one access path to lower levels. Second, nodes in this set must have a minimum total cost. Nodes that are not selected at a level are removed permanently. Such a process stops after all levels of the query graph have been processed. The main module of *Level* is function MinimizeGraph() shown in Figure 3.3.

Next, we provide a few additional details about how the *Level* algorithm works at each level. Since all final nodes are user-defined continuous queries, they must be retained after regrouping. Thus, at each level, the *Level* algorithm first chooses all final nodes and invokes a function called purgeSiblings() (Figure 3.3) on each of the final nodes. The motivation of purgeSiblings is that since all parents of a chosen node x can use x for computation, x's siblings are not required for them any more. Thus, invoking purgeSiblings on node x removes edges between parents of x and all x's sibling nodes. However, removing edges in the graph can cause problems for level-1 regrouping. As we mentioned in the previous section, one desired property of phase-1 regrouping is to keep all edges between nodes that are chosen to keep after regrouping, so that phase-1 does not have to re-compute links among existing nodes each time regrouping is performed. Our solution to this problem is to have a reference count variable in each node. The initial value of the reference count of each node is set to the number of its parents at the beginning of each regrouping. Instead of actually removing an edge from a node's parent to a sibling node in purgeSiblings, the sibling node's reference count is decreased by 1. Thus, no edges are physically removed in purgeSiblings. One complexity of using a reference count mechanism is that to avoid repetitively invoking the purgeSiblings function on the same parent node, each node contains a variable *visited* with an initial value *false*. A parent node's *visited* variable is set to *true* after the first purgeSiblings operation in order to prevent subsequent invocations of purgeSiblings on this node.

The main complexity of this algorithm is to choose a set of non-final nodes that satisfy the following two constraints after all final nodes have been selected. On the one hand, join groups that are shared by many queries should be kept. On the other hand, join groups whose inputs change very frequently are not good choices to retain, since these groups must be evaluated very frequently and the associated cost can be high. These two considerations can conflict with each other in some situations and a good compromise must consider both. This problem is similar to the classic *set cover* problem [CLR90], which is a NP hard problem. To solve this problem, *Level* uses another greedy algorithm to choose a node with a minimum value of the ratio of its cost versus its reference count. The intuition behind this is that since the cost of a node is shared by all of its parent nodes, a minimal cost per parent seems a reasonable criterion for choosing a node to keep. Such a criterion is a compromise of the cost and the usefulness of a node. After a non-final node is chosen, purgeSiblings is also invoked on this node. Such a process will be done repeatedly until there is no non-final node left with a reference count greater than 0. One important optimization performed at each round is to remove nodes with a reference count of 0, since all of its parents have already found a sub-query node. Since each node usually does not have too many edges, many nodes can be removed within a few rounds. Thus, this optimization is very important in reducing the cost of regrouping.

3.4 Cost Analysis of Regrouping Algorithms

Assume that the total number of queries is N and that at each moment the total number of nodes is proportional to the number of queries already installed in the system. For example, if there are T queries installed, we assume the number of nodes in the system is C*T, where C is a positive constant. Furthermore, assume that each level has the same number of nodes and that each query contains no more than 10 joins in our analysis. Thus, each level will contain about C*N/10 nodes.

Phase-1 Assume that regrouping frequencies are either every R queries or K*R queries, where K is a positive integer. The number of regrouping is N/R, if regrouping occurs every R queries. Using nested loop join algorithms, at each regrouping time, each newly added C*R nodes will be joined with existing nodes at adjacent levels. Assuming that m-1 regroupings have been performed already, the number of nodes in the system thus is m*C*R. The number of comparisons for the m-th regrouping is $m*(C*R)^2$, ignoring a constant reduction factor. The total number of comparisons required for the regrouping is given in formula (1).

$$(C*R)^2+2*(C*R)^2+\ldots+N/R*(C*R)^2 = N(N+R)C^2/2 \quad (1)$$

Let n, the number of regrouping for every K*R queries, equal N/(K*R). The formula for the total number of comparisons for regrouping every K*R queries can be derived similarly and is given in (2).

$$(K*C*R)^2+2*(K*C*R)^2+\ldots+n*(K*C*R)^2$$
$$=N*(N+K*R)*C^2/2 \quad (2)$$

Thus, the ratio of number of comparisons for regrouping by every K*R queries versus by every R queries is as follows.

$$[N(N+KR)C^2/2]/[N(N+R)C^2/2]=(N+KR)/(N+R) \quad (3)$$

Since K is a positive integer, from (3) we can see that **more frequent** regroupings can reduce the total cost of phase-1 regrouping. This conclusion is interesting because it is counter-intuitive. From (1), we know that the accumulated costs of phase-1 of regrouping are bounded by $O(N^2)$. The computational complexity of a single phase-1 regrouping is O(N).

Phase-2 (**Level**) Assume that there are N queries in total, and that the total number of nodes in the system is C*N, where C is a positive constant.

The main cost of *Level* consists of two parts. The first part is to select nodes to keep using a greedy algorithm at each level. The algorithm repeatedly scans the nodes that have not been chosen so as to choose the one with a minimum value of the ratio of its cost versus its reference count. At each pass, one node will be chosen and any nodes with a reference count of 0 will be removed. In the worst case, each pass can only remove one node. Thus, in the worst case, at each level, the cost will be (C*N/10)+(C*N/10-1)+…+1=C*N*(C*N+10)/200. Since *Level* begins at the second level from the top, the total cost is the above cost multiplies 9. In most cases, each level contains quite a few final nodes, which can be selected in a single pass. In addition, as we mentioned in the previous section, each node usually does not contain many edges, and most nodes can be removed during a few initial passes. Thus the cost, in practice, is usually much smaller than the theoretical worst case.

The second part is to purge siblings for each chosen node. Notice that during regrouping, each node will be accessed at most once for each of its parents, because each access removes at least one parent edge by decreasing the reference count by one. Thus, the computational complexity of this step is O(E), where E is the number of edges in the group plan. We give a rough estimation of E as follows. Each node may contain no more parents than the entire number of nodes of its immediate parent level, which is bounded by C*N/10. Thus, the total cost of this part is bounded by (C*N/10*C*N/10)*9=0.09*$(C*N)^2$. In reality, since each node contains far fewer edges than the above estimation, this process is also very fast.

Thus, the total cost of a single phase-2 regrouping is bounded by O(N^2).

Timing of Regrouping Even though the cost of phase-1 regrouping favors more frequent regrouping, the cost of phase-2 is proportional to the number of regroupings performed, since minimization is done over the entire graph. Thus, very frequent regrouping will increase the cost of phase-2 regrouping. Thus, the cost of regrouping is minimized only when the regrouping frequency falls in the middle. An appropriate regrouping interval can be obtained either by performing a more detailed cost analysis than what we have shown above or by conducting experiments (shown in Section 5.4).

3.5 Continuous Query Delete

An existing continuous query can be removed from the system when it has expired or has been explicitly removed by a user. Grouping makes deletion a much more complex problem, because every query becomes a portion of the entire group plan. In particular, first, removing existing nodes cannot affect the correctness of other queries. Second, removing some nodes can degrade the quality of the remaining query plan. This problem has not been considered by traditional multiple query optimization algorithms [CM86] [Sel86].

Since regrouping inherently performs global query optimization, it is natural to let regrouping handle re-optimization after deletes occur. Our method handles a query delete very efficiently by simply reducing the final node count of the to-be-removed final node by 1. The rest of the work is automatically handled when the next regrouping is performed. If desired, regrouping can be triggered after a number of deletes have been performed.

4. Experimental Results

We conducted performance evaluations of our regrouping algorithms using synthetic query workloads in this section. The following experiments were conducted on a Sun Ultra 6000 with 1GB of RAM, running Solaris 2.6.

4.1 Query Workload

The default query workload used in the experiments consists of 10000 distinct queries. Each query contains a set of input file names from a predefined domain. The length of a query is defined by the number of joins in the query, which equals the number of input files of the query minus 1. The length of queries is distributed between 1 to 10 with a distribution similar to a normal distribution with a mean value at 5. The order of queries in the workload is generated randomly. In addition, queries are generated by following a Zipfian distribution with a skew factor Z_{input} over the input files. By increasing the skew factor Z_{input}, more queries are defined over a smaller number of input files. When Z_{input} equals 0, the queries are uniformly distributed across the input files in the domain.

The update frequency of input files also follows a Zipfian distribution, with a skew factor $Z_{update.}$ Varying the skew factor Z_{update} will generate different update frequency distributions. In order to investigate the correlation between the query and update frequency distribution, the above two frequencies can be allocated to input files following three particular orders in our experiments: *Random*, *Descending* and *Ascending*. For example, assume that the total number of input files is 20 and that the input files are named file_1 through file_20. *Descending*, *Ascending* and *Random* allocate frequencies to file_1 through file_20 in a descending, ascending and random order, respectively. Update frequencies and query frequencies are assigned independently. *Random* is the default order used for assigning both frequencies over the input files.

4.2 Experimental Methodology

In this paper, we consider dynamic regrouping of a large, dynamic continuous query workload (e.g. tens of

thousands of queries), while queries can be triggered for execution very frequently by data changes. An experiment in such a setting could take days, even weeks to finish. Instead of measuring the execution time over real data changes, in our experiments, we use the update frequency of a node as the metric of the cost for computing that node, as described in Section 2.2. With more accurate cost information, we expect that our regrouping algorithms will be more effective in choosing "good" nodes to retain.

Our goal is to improve the effectiveness of groups by periodically regrouping existing queries. In the pure incremental grouping method [CDTW00], each query in the query workload is sequentially added to the system using the incremental grouping algorithm. Assume that queries arrive at a fixed arrival rate with an inter-arrival time T. Assume the first query arrives at time T_0. Then the i-th query arrives at T_i, where $T_i = T_0 + i*T$. In addition, assume that updates over the input files follow the predefined update frequency. We measure the cost of computing all the joins in the system during each interval T, which equals the sum of the cost of nodes in the group plan during that interval. Note that the group plan is only changed at the beginning of each interval when a new query is being installed. Assume that there are N queries in the workload. We use the average of above costs over the N intervals from T_0 to T_N as a measure of evaluating the N queries using the incremental group optimization.

In our experiments, regrouping is performed immediately after a pre-defined number of new queries have been installed (e.g. every 10 new queries) while incremental grouping is performed every time a new query arrives. At each regrouping time, regrouping is immediately performed over the group plan after the new query has been incrementally group optimized. The group plan after each regrouping is used thereafter. The average cost for the entire query workload is computed in a manner analogous to the previous case when only incremental group optimization was used. The relative improvement between this cost and the cost measured with only incremental group optimization is used as the metric for evaluating the benefits obtained by regrouping.

Since we are only interested in a relative improvement, without losing generality, we let T equal 1 in our computation.

An illustration of this measurement strategy is shown in Figure 4.1 and 4.2 using an example query workload {ABC, BC}. Assume that ABC comes to the system before BC and update frequency over A, B and C are 1, 2, and 3, respectively. Figure 4.1 shows the group plans with the cost in the pure incremental group optimization approach. Figure 4.2 is for the incremental group optimization along with regrouping approach. The cost of each node equals the sum of the update frequencies of the input files and is shown in each node in Figure 4.1 and 4.2. The average cost for the incremental group optimization (Figure 4.1) is thus (9+14)/2=11.5. Assume that regrouping is performed after 2 new queries are added. Figure 4.2 shows that the group plan after the first query ABC is installed at T_0 is the same as that in the incremental group optimization. However, at time T_1 the group plan after regrouping is used. Thus the average cost in the regrouping approach is (9+11)/2=10. In the above example, the relative improvement of the regrouping approach over the pure incremental group optimization approach is thus (11.5-10)/11.5=13%.

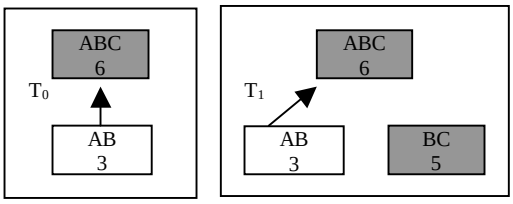

Figure 4.1: The group plans with the cost in the pure incremental group optimization approach.

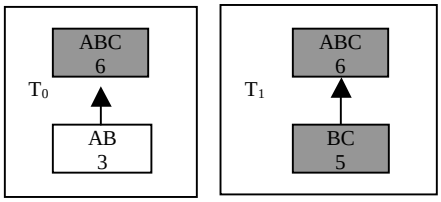

Figure 4.2: The group plans with the cost in the incremental group optimization approach along with the regrouping approach.

The overhead for incremental grouping and regrouping is measured by the average time spent performing these operations.

4.3 Performance Results

Symbol	Explanation	Values
K	Maximum number of joins in each query	10
L	Number of input files	20
$\mathbf{Z_{input}}$	Skew factor of distribution of queries over input files	1
$\mathbf{O_{input}}$	Order of allocating appearance frequencies over input files	Random
T	Frequency of regrouping in terms of number of new queries installed	10
D	Duplication ratio of queries in the query workload	1
N	Number of queries in the workload	10000
$\mathbf{Z_{update}}$	Skew factor of distribution of update frequency over input files	1
$\mathbf{O_{update}}$	Order of allocating update frequencies over input files	Random

Table 4.1: Experimental variables and their default values.

The parameters used in the experiments are listed in Table 4.1. Unless specified explicitly otherwise, the parameter values in Table 4.1 are used.

4.3.1 Regrouping v.s. Incremental Grouping

In the following experiments, we investigate how our regrouping algorithm behaves under different situations by varying several important experimental parameters.

(1) Varying Z_{input} (skew factor of the Zipfian distribution of queries over input files)

Z_{input}	Avg. Cost with RG	Avg. Cost w/o RG	% Cost Improvement	Total RG Time (s)	Total IG Time (s)	Avg. No of Nodes with RG
0	2640	2972	11.2%	5600	179	9772
1	1547	1832	15.6%	1040	130	6426
2	1195	1513	21.0%	410	71	4743
3	1106	1477	25.1%	370	42	4212

Table 4.2: Experimental results when varying Z_{input}.

In Table 4.2, as we increase Z_{input}, the skew factor of the Zipfian distribution of queries over input files, the computation costs of both incremental grouping and regrouping decrease. The reason is that as Z_{input} increases, more queries are defined over a small subset of the input files, which makes incremental grouping and regrouping more effective. More interestingly, the relative performance improvement of regrouping over incremental grouping increases from 11% to 25% as Z increases from 0 to 3. This indicates that when there are good grouping opportunities among queries, pure incremental grouping is not able to capture all of them. Regrouping, in such cases, can obtain a large improvement.

The total incremental grouping time stays very low, between 179 seconds and 42 seconds. The time drops because, when Z_{input} becomes large, a new query has an increased chance of finding a node with the same query existing in the group plan already. In such cases, incremental grouping simply increases the final node count of the existing node by 1, which is much less costly than the normal case when the entire sub-tree of the new query is traversed. As Z_{input} increases, the average number of nodes in the group plan drops from 9772 to 4212, causing the total time spent for regrouping drops sharply from 5600 to 370 seconds. Each regrouping takes less than a half second, when Z_{input} equals 2 or 3.

(2) Varying Z_{update} (skew factor of the Zipfian distribution of update frequencies over input files)

Table 4.3 shows that both regrouping and incremental grouping become more effective as Z_{update}, the skew factor of the Zipfian distribution of update frequencies over input files, increases from 0 to 3. The relative improvement of regrouping over incremental grouping decreases slightly from about 17% to 14%. This occurs because when Z_{update} is 0, each input is updated at the same frequency. In this case, the regrouping algorithm only needs to select nodes to keep by the number of parents of a node. As mentioned earlier, when selecting nodes to retain while regrouping two, potentially conflicting, properties are considered, the number of parents and the update frequency of a node.

Z_{update}	Avg. Cost with RG	Avg. Cost w/o RG	% Cost Improvement	Total RG Time (s)	Total IG Time (s)	Avg. No of Nodes with RG
0	2242	2695	16.8%	1080	134	6516
1	1547	1832	15.6%	1040	130	6426
2	1023	1198	14.6%	1130	133	6519
3	858	1001	14.3%	1170	129	6586

Table 4.3: Experimental results when varying Z_{update}.

The time spent on both incremental grouping and regrouping is almost constant when Z_{update} increases from 0 to 3. This is because in this experiment, queries follow the same distribution over the input files. The average number of nodes in the group plan remains almost constant.

(3) Varying K (maximum number of joins in a query)

K	Avg. Cost with RG	Avg. Cost w/o RG	% Cost Improvement	Total RG Time (s)	Total IG Time (s)	Avg. No of Nodes with RG
12	1970	2265	13.0%	1280	390	7190
10	1547	1832	15.6%	1040	130	6426
8	1161	1445	19.7%	660	44	5551

Table 4.4: Experimental results when varying K.

In our experiment the length of queries is distributed between 1 to K, the maximum query length, following a distribution similar to a normal distribution with a mean value at K/2. Table 4.4 illustrates that as K decreases from 12 to 8, the relative improvement of regrouping increases from 13% to 20%. This is because, as K decreases, query nodes are distributed in a much smaller search space, with more opportunity for grouping. As K decreases from 12 to 8, since the incremental grouping algorithm can search a much smaller query space for a new query, the time spent for each incremental regrouping operation decreases significantly from 39 to 4.4 milliseconds. The time for each regrouping decreases from 1.3 to 0.7 seconds.

(4) Varying O_{input} and O_{update} (orders for allocating appearance frequencies of input files in queries and update frequencies over input files, respectively)

As mentioned in Section 3, the two key factors when regrouping are to select nodes that have many parents and that are updated infrequently. These two factors can conflict in some cases. We want to investigate how this may affect the effectiveness of regrouping.

In this experiment, we investigate, in two specific cases, the correlation between the distribution of queries and the distribution of update frequency. In the first case, both distributions have the same order, (e.g. O_{input} = O_{update} = *Descending)*, which implies that a node with the most

parents is also updated most frequently. This is the case with the highest extent of conflict in choosing nodes to retain. In the second case, the two distributions have the opposite order, (e.g. O_{input} = *Descending*, O_{update} = *Ascending*), which implies that a node with the most parent is also the one updated least frequently. We expect regrouping to spend less time in choosing nodes in the second case than the first one.

O_{input}	O_{update}	Avg. Cost with RG	Avg. Cost w/o RG	% Cost Improvement	Total RG Time (s)	Total IG Time (s)	Avg. No of Nodes with RG
Dsc	Dsc	3507	4208	16.7%	2020	149	7663
Dsc	Asc	1403	1655	15.2%	1040	130	6425

Table 4.5: Experimental results when varying O_{input} and O_{update}.

As expected, the experimental results in Table 4.5 demonstrate that both incremental grouping and regrouping, in the second case, perform much better than the first case. Surprisingly, the relative improvement of regrouping over incremental grouping is almost the same in both situations. The execution time spent on an incremental group optimization remains almost constant in both cases. The time for a regrouping in the second case is about half that of the first case, even though the number of nodes differs slightly. The time difference comes from the fact that in the first case, at each level, nodes with the most parents (and also the least updated ones) will be chosen to keep by the *Level* algorithm, allowing the *Level* algorithm to finish quickly. In the second case, the *Level* algorithm consumes more time select which nodes to retain since nodes with many parents may not be worth keeping.

Summary Overall, the following observations can be drawn consistently from our experiments. First, regrouping consistently improves the grouping quality compared with a pure incremental grouping strategy under various situations. In most cases regrouping obtains about a 16% improvement over a pure incremental grouping strategy. In some cases, a 25% improvement is observed. Second, regrouping usually has a larger improvement in situations that favor grouping. We believe that in real applications many grouping opportunities will exist. For example, many users may pose continuous queries over a few "hot files". Regrouping will be important in such applications. Third, regrouping is efficient. In most cases, each regrouping takes about 1 second in an environment with an average of 7000 nodes in the group plan. Such an extra cost is worthwhile given that continuous queries may be executed many times. Furthermore, since regrouping occurs less frequently than incremental grouping, the total time spent on regrouping can be just an order of magnitude more than time spent on incremental grouping.

In addition, our experiments also show that incremental grouping is fairly effective and efficient under various situations. In most cases, each incremental grouping only takes about 15 milliseconds, obtaining a fairly good quality.

From those observations, we believe a mixed approach of a constant incremental grouping plus a periodical regrouping achieves the real balance between performance and cost.

4.3.2 Timing of Regrouping

In this section, we study the timing of regrouping. Two approaches, *periodic regrouping* and *active regrouping* are studied. In the former approach, regrouping occurs after a fixed number of queries have been added to the system. In the later approach, the run-time status of a continuous query system is monitored against some pre-defined threshold. Regrouping is performed when a triggering condition is satisfied.

(1) Periodic Regrouping

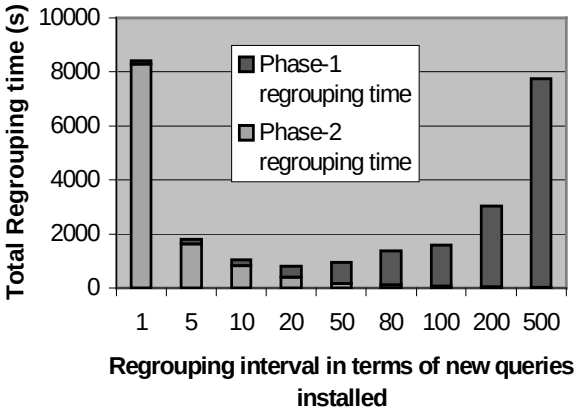

Figure 4.3: Experimental results of regrouping every fixed number of queries

We measure the execution time for each phase of regrouping (Figure 4.3). The total time on phase 1 of regrouping (which constructs links among existing nodes and new nodes since the last regrouping) increases significantly from 120 to 7747 seconds, as the regrouping interval is increased from after every query to every 500 queries. On the other hand, the total time of phase 2 of regrouping (which minimizes the graph using the *Level* algorithm) drops from 8300 to 19 seconds. In our experiments, the total regrouping time reaches a minimum value when regrouping occurs about every 20 queries. This observation perfectly matches our cost analysis and predications in Section 3. We can see that regrouping, when it occurs at the desired rate (every 20 queries in the above example), is 10 times faster than that when it occurs either after every query or every 500 queries. Thus, the timing of regrouping is fairly crucial.

Results from this experiment show that the regrouping quality only degrades slightly, as regrouping frequency varies from every query to every 500 queries. This is

because the query workload contains 10000 queries and the variance is not large enough to cause significant differences in the cost improvement.

(2) Active Regrouping

Even though a periodic regrouping approach is very simple, it has the potential drawback of performing inefficiently. A more efficient approach, *active regrouping*, performs regrouping only when necessary. Such an approach can potentially reduce the overhead of regrouping. The main idea of *active regrouping* is to let the system be monitored on some particular parameters at running time. Either users or the system can specify a threshold for triggering the execution of a regrouping. We use the following experiments to illustrate the potential advantages of using an *active regrouping* approach.

D	Avg. Cost with RG	Avg. Cost w/o RG	% Cost Improvement	Total RG Time (s)	Total IG Time (s)	Avg. No of Nodes with RG
1	1547	1832	15.6%	**1040**	130	6426
2	809	962	15.9%	**2496**	128	3353
3	536	640	16.2%	**4374**	144	2211

Table 4.6: Experimental results when varying D.

In this experiment, we duplicate each of the 10000 queries D times and insert them into the query workload in a random order. Thus, the total number of queries in the workload is D*10000. Table 4.6 shows that the average cost of both incremental grouping and regrouping drops in proportion to 1/D, when the duplication ratio (D) increases from 1 to 3. This is because a duplicate query will match an existing node without adding any new nodes. The total time for incremental grouping is almost the same regardless of D, since it takes almost no time to incrementally group a duplicate query. The total time of regrouping, however, increases significantly, because regrouping is performed after every 10 queries installed in the experiment. Thus, the number of regroupings increases as D increases. Since duplicate queries do not add new nodes to the group plan, it may not be worth regrouping if the group plan has not changed significantly. This experiment shows the drawbacks of a blind, periodic regrouping approach.

D	Avg. Cost with RG	Avg. Cost w/o RG	% Cost Improvement	Total RG Time (s)	Total IG Time (s)	Avg. No of Nodes with RG
1	1547	1832	15.6%	**759**	127	6424
2	808	962	16%	**799**	140	3329
3	536	640	16.3%	**923**	138	2208

Table 4.7: Experimental results of an *active regrouping* approach.

An *active regrouping* approach can help solve this problem. Table 4.7 shows the result of using an *active regrouping* approach over the same query workload used in Table 4.6. In this experiment, regrouping is triggered when the number of new nodes added since last regrouping becomes more than a predefined regrouping threshold (20 nodes in this experiment). Since duplicate queries do not introduce new nodes to the group plan, the number of new nodes added since last regrouping is a much better measure of the changes that have occurred to the actual group plan. Table 4.7 illustrates that the active regrouping approach achieves the same quality as the periodic approach. However, as D, the duplication ratio of a continuous query workload, increases from 1 to 3, the total time spent regrouping increases only very slightly from 759 to 923 seconds, because the number of regroupings performed is almost the same in all three cases. In contrast, the total time spent regrouping in the *periodic regrouping* approach increases from 1040 to 4374 second, as D is increased from 1 to 3 (Table 4.6).

5. Related Work

Previous research on multiple-query optimization [RC88] [Sel86] [RSSB00] [MRSR01] considers only a one-time global query optimization over a batch of queries. Early research on multiple-query optimization [RC88] [Sel86] has focused on finding an optimal query plan for a small number of queries with common sub-expressions. Those approaches usually use exhaustive search approaches and can be very expensive for processing a large number of queries. [RSSB00] proposes some important heuristics to reduce the cost of exhaustive search algorithms. [MRSR01] further applies the multiple-query optimization techniques in [RSSB00] to materialized view selection and maintenance. In contrast, we consider the dynamic regrouping of continuous queries in the presence of a large, dynamic query workload.

Dynamic query re-optimization [BFMV00] [KD98] [UFA98] is another closely related area. In earlier studies, queries are dynamically re-optimized based on run-time information. Such a dynamic scheme is usually better than a traditional query optimization approach, which only uses information available at query optimization time. There are two major differences between our work and theirs. First, while they consider optimizing single queries, the focus of our work is multiple query optimization. Second, re-optimization uses new data statistics collected during query execution, while our regrouping mechanisms are driven by newly-arrived queries in a query workload. Thus, these two approaches are orthogonal and can be applied together.

Our regrouping algorithms use signature nested loop joins for set-containment join predicate. Extensive studies of signature-based join algorithms can be found in [HM97] [RPN+00].

6. Conclusions and Future Work

In this paper, we design and evaluate an efficient and dynamic regrouping approach to optimize a large

continuous query workload. The results from this paper show that when a periodic dynamic regrouping is applied together with incremental grouping [CDTW00], it can further improve a pure incremental grouping method by up to 25% at a very low overhead of regrouping time. Our heuristic-based regrouping is very efficient even when applied to a fairly large group plan with about 6500 join groups. Cost analysis and experimental study are presented to determine a good regrouping interval, which is critical to the efficiency of a periodic regrouping approach. In addition, our results show that active regrouping, which performs regrouping whenever necessary, can significantly reduce the total regrouping time. Overall, we believe a constant incremental grouping plus an occasional dynamic regrouping can achieve a high quality grouping at a fairly low cost.

Dynamic regrouping of large continuous query workloads is still an open research area. In this section, we briefly mention a few of the problems on which we are currently working.

- Accurate Cost Estimation of Nodes: In this paper, for the simplicity purpose, we use the update frequency of a node as the metric of the cost for computing that node. An accurate estimation of the cost of a node can be difficult and tedious. Detailed cost models presented in [CDN02] might be used for the cost estimations. In addition, our phase-2 regrouping algorithms can be extended to incorporate such changes.
- Regrouping with Considerations of Adding New Nodes: Our regrouping algorithm significantly reduces the search space by removing redundant nodes from an incremental group optimization solution. We plan on investigating how adding new intermediate nodes can help regrouping. Preliminary experimental results indicate that considering adding nodes can significantly increase regrouping time with only a slight improvement in performance.
- Memory Constraint: The amount of memory available is another important factor that can significantly affect the time spent regrouping. Our algorithm assumes all nodes can fit in available physical memory. When the number of installed continuous queries becomes very large, the entire query plan may not be able to fit in the physical memory available for regrouping. It would be interesting to re-evaluate our algorithm in such an environment.
- Considering queries with operators other than joins for dynamic regrouping.

Reference:

[BFMV00] L. Bouganim, F. Fabret, C. Mohan and P. Valduriez, “Dynamic Query Scheduling in Data Integration Systems”, ICDE 2000:425434.

[CDN02] J. Chen, D. J. DeWitt, and J. Naughton, “Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries”, to appear, ICDE 2002.

[CDTW00] J. Chen, D. J. DeWitt, F. Tian and Y. Wang, “NiagaraCQ: A Scalable Continuous Query System for Internet Databases”, SIGMOD 2000: 379-390.

[CLR90] T. H. Cormen, C. E. Leiserson and R. L. Rivest, “Introduction to Algorithms”, MIT Press, 1990.

[GM93] Goetz Graefe, William J. McKenna: The Volcano Optimizer Generator: Extensibility and Efficient Search. ICDE 1993: 209-218

[HM97] Sven Helmer, Guido Moerkotte: Evaluation of Main Memory Join Algorithms for Joins with Set Comparison Join Predicates. VLDB 1997: 386-395

[KD98] Navin Kabra, David J. DeWitt: Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. SIGMOD Conference 1998: 106-117

[LPT99] L. Liu, C. Pu, W. Tang, “Continual Queries for Internet Scale Event-Driven Information Delivery”, TKDE 11(4): 610-628 (1999).

[MRSR01] H. Mistry, P. Roy, S. Sudarshan, K. Ramamritham: Materialized View Selection and Maintenance Using Multi-Query Optimization. SIGMOD 2001:307-318.

[RC88] A. Rosenthal and U. S. Chakravarthy, “Anatomy of a Modular Multiple Query Optimizer,” VLDB 1988: 230-239.

[Rou82] Nick Roussopoulos: View Indexing in Relational Databases. TODS 7(2): 258-290 (1982)

[RSSB00] P. Roy, S. Seshadri, S. Sudarshan, S. Bhobe: Efficient and Extensible Algorithms for Multi Query Optimization. SIGMOD Conference 2000: 249-260

[RPNK00] Karthikeyan Ramasamy, Jignesh M. Patel, Jeffrey F. Naughton, Raghav Kaushik: Set Containment Joins: The Good, The Bad and The Ugly. VLDB 2000: 351-362

[Sel86] T. Sellis, “Multiple query optimization,” ACM Transactions on Database Systems, 10(3), 1986.

[TGNO92] D. Terry, D. Goldberg, D. Nichols, and B. Oki, “Continuous Queries over Append-Only Databases”, SIGMOD 1992: 321-330.

[UFA98] T. Urhan, Michael J. Franklin and L. Amsaleg:, ”Cost Based Query Scrambling for Initial Delays”, SIGMOD 1998: 130-141.