Dynamic Query Re-Optimization

Kenneth W. Ng Zhenghao Wang Richard R. Muntz Silvia Nittel
{kenneth,zwang,muntz,silvia}@cs.ucla.edu
Computer Science Department
University of California
Los Angeles, CA 90095-1596

Abstract

Very long-running queries in database systems are not uncommon in non-traditional application domains such as image processing or data warehousing analysis. Query optimization, therefore, is important. However, estimates of the query characteristics before query execution are usually inaccurate. Further, system configuration and resource availability may change during long evaluation period. As a result, queries are often evaluated with sub-optimal plan configurations. To remedy this situation, we have designed a novel approach to re-optimize suboptimal query plan configurations on-the-fly with Conquest — an extensible and distributed query processing system. A dynamic optimizer considers reconfiguration cost as well as execution cost in determining the best query plan configuration. Experimental results are presented.

1 Introduction

Parallelism is important in today's database query processing. Very long-running queries require parallel processing to deliver reasonable performance due to the growing demands for decision support and data mining against massive databases. A workstation farm is a common environment to support parallel computing. During a long query evaluation, changes of system configuration and resource availability are often expected. Furthermore, a priori estimates of the query characteristics against large datasets, e.g., selectivity, are often coarse and inaccurate. As a consequence, traditional optimization methods that optimize a query once before execution are not totally satisfactory. To remedy this situation, we propose a novel approach to dynamically re-optimize suboptimal query plan configurations and thereby significantly improving evaluation performance of long-running queries. The objective is to take advantage of up-to-date cost estimates which come from more precise information on query statistics (e.g., selectivity of a qualification), as well as to adapt to changing system configuration and resource availability (e.g., availability of machines, other workloads, etc.).

We have implemented an extensible query processing system called *Conquest* (CONcurrent QUEries over Space and Time) which handles complex scientific queries involving computationally expensive calculations on large geo-scientific datasets stored in different formats and managed by many different autonomous storage subsystems [23]. A parallel query execution plan (QEP) in Conquest specifies the parallel process of manipulating data and producing results for a submitted query. The subtasks of the QEP are called operators. The *configuration* of a QEP refers to the composition of operators, input-output communications among operators, degree of data parallelism and operator assignment to different processors. Conquest supports operators that compute a function in an object-at-a-time fashion [6]. This design resembles *Volcano's* Open-Next-Close query evaluation paradigm [7] [8] that supports data stream processing. Each execution of an operator is called an *iteration*.

To dynamically determine the best QEP configuration, an optimizer must first know when a re-optimization should be performed. Different from the approach proposed in [16], which re-optimizes a query plan only when a sub-tree of the QEP is completely done (hence no pipelined parallelism is considered in their model), and the approach suggested in [25], which changes the order of join operations if unexpected network delays happen, we use a *triggering approach* that allows more general query re-optimization in an environment fully supporting parallelism. Built-in and user-defined trigger rules are evaluated at run-time so that the dynamic optimizer *actively* performs re-optimization. Dynamic re-optimization mechanisms

monitor situations of interest and, when they occur, trigger an appropriate response in a timely manner. Trigger rules are of the well-known *event-condition-action* form [4] which is straightforward but powerful. The triggering approach is extensible and is especially appropriate in an extensible database system, e.g., object-relational DBMS [13][14][24], as users can arbitrarily introduce new trigger rules. We have discussed the procedure of modifying a QEP configuration on the fly in [19][20]. In this paper, we focus on presenting a triggered dynamic re-optimization method with which the optimizer can promptly react to changes to the execution environment and/or better knowledge of data characteristics at run-time.

1.1 Motivating Example

To better motivate the problem of suboptimal query plans, we performed the following experiment in a workstation farm environment. A distributed version of *cyclone tracking*, an example query drawn from the geo-scientific application domain, has been implemented. Cyclones are indicated by local minima in atmospheric pressure at the earth's surface. We use *Scan* operators to retrieve spatio-temporal surface pressure data and wind records for the prescribed time interval. Sea level pressure records are passed to a *minima* operator that extracts the location of local minimum in sea level pressure data with certain constraints and a *track* operator connects points to form tracks using auxiliary information such as wind direction.

Three SunMicro workstations, X ("margarita", Ultra-10 with 128 MB memory), Y ("greedy", Ultra-30 with 384 MB memory) and Z ("mint", Ultra-30 with 128 MB memory), each of which is running SUNOS 5.6, JDK1.1.6 and OrbixWeb3.0, were used in this experiment. At the query start up time, both machine Y and Z were with heavy workload. Only machine X was lightly loaded and 98% CPU time could be assigned for this query evaluation. After half hour, most CPU intensive jobs on machine Y were done. Therefore, machine Y became available for query evaluation. Machine Z was still extremely heavily loaded for fifteen minutes more. Twenty minutes after all jobs on machine Z were done, machine X happened to be moderately loaded because of some planned jobs, which took about 80 minutes to perform, hence machine X had to spend most CPU time for those jobs. During the evaluation, some occasional jobs, e.g., web browsing, text editing, came in and lasted for less than 2 minutes.

For comparison, we executed three valid execution plans (Figure 1) under the same workload change. We used UCLA AGCM climate model data sets that contain sea level pressure and wind records from 1980 to 1989, whose storage size is about 1.6 GB. The overall execution time was approximately 730 minutes for plan A, 244 minutes for plan B and 217 minutes for plan C respectively. Though query evaluation of plan-B outperformed the execution of plan-A, it is worse than the execution of plan-C. However, it was unlikely the plan-C would be chosen at start-up time since workload on both machine Y and machine Z was high at that moment. Therefore, we can expect that a query execution which performs with plan-A at start-up time and then changes to plan-B or plan-C at an appropriate time will deliver better performance.

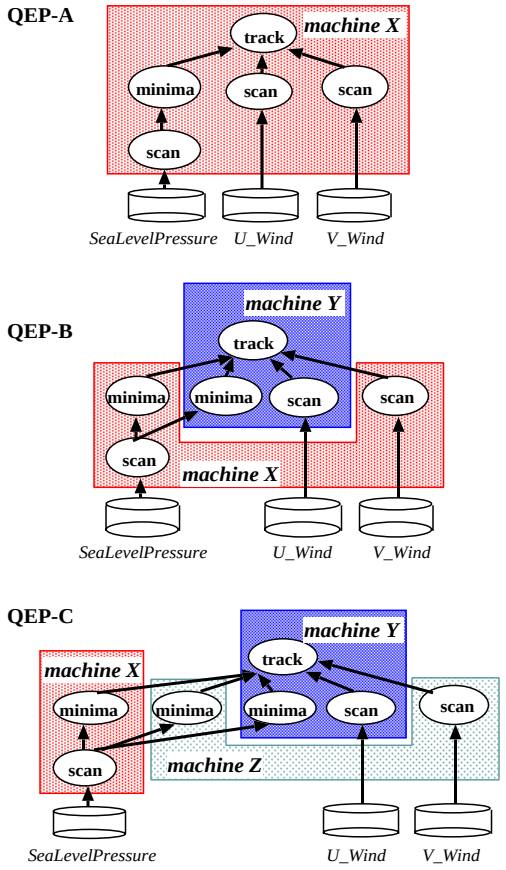


Figure 1. Three query execution plan configurations for cyclone tracking

2 Overview

2.1 Dynamic Re-optimization

Dynamic optimization mechanisms use up-to-date system configuration, system loads, and query execu-

tion information to re-optimize query execution plans on-the-fly. Briefly, system performance and query characteristic metrics are periodically measured during run time. These measured values are used to project the performance trend of the residual query execution. If it is estimated that a new QEP configuration, different from the *current configuration*, will lead to a significant improvement in query performance, a migration strategy from the current plan configuration to the proposed plan is determined and then carried out.

2.1.1 Run-time Measurement

The problem of analyzing the performance of a parallel QEP can be described in two parts: *performance instrumentation* and *performance analysis*. The performance instrumentation is concerned with how to efficiently collect relevant statistics about a computation, while the analysis part focuses on evaluating a variety of statistics and providing useful information (e.g., predicted workload and network traffic in the near future) to the dynamic optimizer.

We consider two types of performance instrumentation: *system instrumentation* and *query instrumentation*. System instrumentation collects statistics on the execution environment such as CPU workload, network traffic, swap space and resource availability. System instrumentation is independent of query execution plans. In other words, different QEPs share the same system statistics and predictions. Query instrumentation focuses on the estimation of query characteristics. For example, the data selectivities of operators are computed on the fly.

2.1.2 Dynamic Re-optimization

We use a triggering approach to implement dynamic re-optimization. When system events occur, e.g., system parameters are measured and updated in the catalog, if particular conditions are satisfied, then the dynamic optimizer is invoked to consider whether a better performance can be achieved by reconstructing the query plan configuration. For example, the system is informed that one workstation participating in the query evaluation is going to be shutdown for regular maintenance, the dynamic optimizer will have to consider where the operator instances running on that machine should be migrated to.

The dynamic optimizer considers the reconfiguration cost in addition to the execution cost of the remainder of the query when making decision about reconfiguration. This additional cost dimension increases the complexity of searching for the best query plan. For instance, a plan with slightly higher residual execution cost (based on current estimates of system configuration and query characteristics) but much lower reconfiguration cost (from the current QEP configuration) could be the better choice since it allows more efficient plan migration. Therefore, only when $residualExecutionCost(newQEP) + reconfigurationCost(currentQEP, newQEP) + threshold < residualExecutionCost(currentQEP)$ returns a *true* value, plan reconfiguration is performed, where *threshold* is a heuristic value pre-defined by system developers to guarantee significant performance improvement.

2.1.3 Query Plan Reconfiguration

A query plan reconfiguration, which is performed by a *reconfigurator*, yields a new, semantically correct configuration for the same query. The process of query plan modification consists of five phases: (1) *operator coordination*, which brings operators into a state in which reconfiguration can be performed without causing deadlock or missing information; (2) *state capture*, which saves the state information including buffered input and output records and execution context of the query execution; (3) *plan modification*, which changes the configuration of the existing query plan; (4) *state restoration*, which initializes the execution environment for the modified QEP, including moving buffered input and output records and reinstating scratchpad context (e.g., with check-point records or other mechanisms to be described later); and (5) *computation resumption*, which restarts the computation after reconfiguration. Phases are executed in parallel whenever possible and options for implementing each phase are chosen based on operator semantics and respective costs.

2.2 Reconfiguration Types

Reconfiguration refers to the change from one query plan configuration to another. We here list considered reconfiguration types. The details can be found in [19]. **Operator Instance Relocation** refers to reassigning an operator instance from one processor to another. **Data stream re-partitioning** modifies the distribution policy for a data stream that feeds different clones of an operator. Generally, each clone runs on a different processor, processes a portion of data stream and maintains individual scratchpad memory. **Modification of the degree of data parallelism** changes the number of clones of an operator. **Operator Replacement** uses a semantically equivalent but different algorithm in place of an implementation. **Algebraic transformation** re-selects an semantically equivalent algebraic expression.

3 Triggered Dynamic Re-Optimization

A triggered dynamic re-optimization monitors the change of system parameters and query characteristics so that an appropriate response can be triggered in a timely manner. Trigger rules are of form **ON** *event* **WHEN** *condition* **CONSIDER** *action* (i.e., <E-C-A specification>)[4]. This allows trigger rules to be fired when an event of interest occurs, e.g., estimates of system parameters and query characteristics are updated, if the condition is satisfied, the dynamic optimizer is invoked to consider possible reconfigurations.

3.1 Trigger Rules

3.1.1 Classification of Triggers

System Triggers are rules that are common to all submitted queries and independent of any query plan execution. For example, whenever a workstation is to be shutdown, all operator instances located on this machine, no matter what queries these operator instances are for, have to be migrated to other machines. On the other hand, newly available resources might also trigger the re-optimization of running QEPs. A system trigger looks like:

```
DEFINE REOPTIMIZATION TRIGGER rule1 FOR SYSTEM
<E-C-A specification>
```

QEP Triggers are rules that are common to all query execution plans. In contrast to system triggers, these trigger rules are associated with a particular QEP execution. For example, in Conquest, buffer queues are used to connect multiple producer operator clones to multiple consumer operator clones. If it is found that a buffer queue is empty most of the time, which can be due to a slow producer or a fast consumer, re-optimization should be triggered to adapt to this situation. A possible reaction is to adjust the data re-distribution policy among the producer clones and the consumer clones. These rules look like:

```
DEFINE REOPTIMIZATION TRIGGER rule2 FOR QEP
<E-C-A specification>
```

Operator Triggers reflect the sensitivity of operators to system parameters and data characteristics. They can be further divided into two sub-classes:

(1) **Built-in Operator Triggers** are with respect to library operators. An example is the table scan operator. If data selectivity is small and an index exists, the operator *index-scan* is used. A built-in trigger might specify the optimizer should **consider** replacing the *index-scan* operator with the *sequential-scan* operator **when** the measured selectivity is higher, by more than some threshold, than the estimated selectivity.

(2) **User-Defined Operator Triggers** are defined by Conquest application developers for user-defined operators. With user-defined triggers, the dynamic optimizer can evaluate whether it is worthwhile to re-optimize the QEP (with respect to particular user-defined operators) under certain circumstances.

Suppose we have a *minima* operator. An operator trigger rule associated with *minima* looks like:

```
DEFINE REOPTIMIZATION TRIGGER rule3
  FOR OPERATOR minima
<E-C-A specification>
```

3.1.2 Specification of Triggers

Triggering Events The most common triggering events are updates to the system parameter measurement (e.g., workload, communication bandwidth) and data characteristics (e.g., operator selectivities). These events happen according to the measurement policy specified for the performance detector. For example, a periodic policy can be used so that performance metrics are measured and updated every 5 seconds. The Conquest triggering events include WORKLOAD, SWAP-MEMORY, COMMUNICATION_BANDWIDTH, and SELECTIVITY, etc.

A triggering event can also be specified as a particular time at which a *known* or *expected* change is scheduled. For example, a discounted broker company such as Charles Schwab or Fidelity provides web trading facilities for customers. The details of customer daily transactions are updated to individual accounts in a batch mode at a scheduled time (only the summary information is updated on-line). Executing such a daily batch process requires a certain amount of system resources. If a long-running query, e.g., regression analysis of Dow Jones blue-chips in ten years, will be in the middle of execution when such a scheduled job starts, then this query plan should be re-optimized at a particular time.

Triggering events can also be due to an external command issued directly from the system administrator. Whenever the system administrator determines that it is necessary to re-optimize the currently executing plan configuration, a command can be sent to the Conquest query manager which interprets the command and raises a triggering event.

Re-Optimization Conditions The condition part of a re-optimization trigger specifies a predicate over the performance metrics collected and estimated at run-time. For example,

```
DEFINE . . .
ON WORKLOAD
```

```
WHEN (CURRENT > 1.2*START) LASTING 10 SECONDS
. . .
```

specifies a re-optimization condition that the workload has been increased at least 20% since last QEP (re-)optimization and the workload has been staying in such a higher level for more than 10 seconds. If the condition part of any re-optimization trigger is satisfied, the dynamic optimizer is invoked for query plan re-optimization.

Reconfiguration Actions The action part of a re-optimization trigger specifies the *suggested* reconfiguration actions. The dynamic optimizer considers these suggested reconfiguration forms to search for the best execution plan under the most updated estimates of system parameters and data characteristics. With respect to the considered reconfiguration forms, the suggested reconfiguration action can be: RELOCATION, REPARTITION, RECLONING, REPLACEMENT, and TRANSFORMATION.

An Example of Trigger Specification

```
DEFINE OPTIMIZATION TRIGGER reopt_minima
FOR OPERATOR minima
ON WORKLOAD
WHEN (CURRENT > 1.2*START) LASTING 10 SECONDS
CONSIDER RELOCATION, RECLONING;
```

The above example specifies a user-defined optimization trigger named *reopt_minima* associated with the user-defined operator *minima*. During execution, the machine on which (a clone of) this operator is being executed is monitored, if the workload has increased 20 percent since last query optimization and stays at that level for more than 10 seconds, the dynamic component of the optimizer will consider to relocate this operator clone from one workstation to another or by changing the number of clones (either increasing or decreasing). If it is found that a better performance can be achieved, a reconfiguration is going to be performed.

3.1.3 Registration

Trigger rules must be registered before they can be used. According to their application domain, trigger rules are grouped into trigger classes of *system*, *QEP* and *operators*. Each trigger is a first-class object in individual class. No re-optimization trigger can be in more than one class. Once registered, triggers are stored in the Conquest system catalog. Users can then use **activate** and **deactivate** command to enable or disable an individual re-optimization trigger and/or the whole class of re-optimization triggers. For example, command **deactivate SYSTEM REOPTIMIZATION TRIGGERS** disables all triggers in system class. Triggers are not evaluated unless they are activated.

3.2 Execution Semantics

The execution semantics of re-optimization triggers is important since the triggering behavior could be complex even for a small set of triggers. The first issue we need to consider is the *granularity* of triggering. There is a choice of firing the trigger after each system parameter (or data characteristics) has been updated or once for the entire set of parameters. Another issue is whether the dynamic optimizer should be invoked immediately if one trigger condition is satisfied or not invoked until all trigger conditions have been evaluated. Finally, when the newly defined and activated triggers start to be evaluated is also an issue. In this section, we describe our approach to run-time trigger evaluation and then the application to query processing with triggered dynamic re-optimization.

3.2.1 Trigger Evaluation

Not all triggers are relevant to a particular query plan configuration. To trigger a dynamic re-optimization, the system considers only a subset of defined and activated triggers. As we have discussed, re-optimization triggers are categorized into three classes: system, QEP and operator. Both system triggers and QEP triggers are applied to a particular plan configuration. In addition, the operator triggers related to the operators used in the plan configuration should also be evaluated. All applied re-optimization triggers compose a trigger set. Since different plan configurations may involve different triggers, such a trigger set must be updated after each plan reconfiguration. The newly defined and activated triggers can then be added to the trigger set and be considered.

The Conquest trigger manager performs the task of evaluating re-optimization triggers. The relationships among events, conditions and actions are illustrated in Figure 2. For those events due to performance measurement, updated data items (i.e., system parameters and query characteristics) are loaded into the working memory. Respective re-optimization triggers are then evaluated in an arbitrarily order by an execution thread of the trigger manager. On the other hand, another trigger evaluation thread maintains a system clock to examine the maturity of time re-optimization triggers. Finally, the third execution thread is responsible for receiving external re-optimization requests re-directed from the query manager. Each time a re-optimization

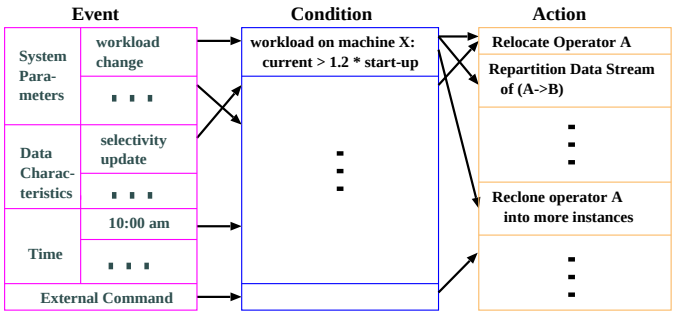


Figure 2. Events, Conditions and Actions

trigger is evaluated, the variables in the condition are bound to data items in working memory. If a condition is satisfied, its associated action results in a *deferred* re-optimization which means the dynamic optimizer is not immediately executed but waits until all conditions have been evaluated. Suggested reconfiguration forms of the satisfied triggers are passed to the dynamic optimizer. Once the dynamic optimizer has started, the trigger manager does not respond to any triggering event until a new configuration plan starts or the current plan configuration is decided to be the best.

3.2.2 Query Processing

Having introduced the triggering mechanisms, we are

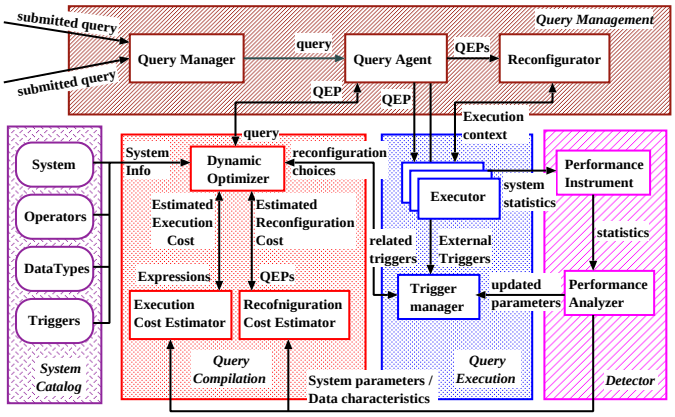


Figure 3. Query Execution

now ready to describe the query processing with dynamic triggered re-optimization (Figure 3):

(1) During query execution, the *detector* dynamically measures system parameters and data characteristics as the submitted query is being evaluated with a particular plan configuration. Once updated, system parameters and data characteristic information are sent to the *trigger manager*.

(2) The *query agent* which monitors the execution of the submitted query listens to the external re-optimization command and if there is any, the query agent redirects such a request to the trigger manager.

(3) The trigger manager maintains three synchronized execution threads to manage different types of triggering events: one thread performs evaluation of activated time triggers; another responds the external re-optimization commands; and the third evaluates activated triggers with respect to performance measurement. If any condition is satisfied, the trigger manager informs the *dynamic optimizer* to re-optimize the QEP with suggested reconfiguration forms after all conditions have been evaluated.

(4) Instead of exhaustively searching for the optimal execution plan for the remainder of the query, the dynamic optimizer considers QEP configurations based on the *current* plan configuration as well as all possible reconfiguration forms passed from the trigger manager. The dynamic optimizer determines the best QEP configuration by considering both execution cost and reconfiguration cost.

(5) If the residual execution cost of the current QEP configuration is greater than the sum of (a) the reconfiguration cost from the current plan to the new one; (b) the estimated residual execution cost of the new plan; (c) a *threshold* which is a heuristic value to prevent reconfiguration that gains insignificant performance improvement, the *reconfigurator* is then requested to perform a cost-effective QEP reconfiguration.

We note that it is not easy to determine the threshold value for making reconfiguration decision. In general, a QEP configuration that involves highly parameter-sensitive operators may require a lower threshold while the one with less parameter-sensitive operators can use a higher threshold to filter out insignificant reconfiguration potentials.

3.3 Experimental Results

3.4 Experiment I

Recall our motivating example, the distributed version of *cyclone tracking*, an example query drawn from the geo-scientific application domain. When the query evaluation started, only machine X was lightly loaded and 98% CPU time could be available for the query evaluation. Both machine Y and Z were with heavy workload at that moment. The workload on machine Y became lighter after half hour as most CPU intensive jobs on machine Y were done. As a result, machine Y

could be considered to participate in the evaluation. Fifteen minutes later, machine Z was also available for evaluating the query. One hour after the query started, machine X happened to be moderately loaded because of some planned jobs, which took about 80 minutes to perform, hence machine X had to spend most CPU time for those jobs. During the evaluation, some occasional jobs, e.g., web browsing, text editing, came in but did not last for more than 2 minutes. We assume that all user tasks have the same execution priority.

With run-time re-optimization, the overall execution time was approximately 2 hours and 36 minutes, including 7.5 minutes (approximately 5%) for performing run-time reconfiguration. The plan configuration changes during evaluation are shown in Figure 4.

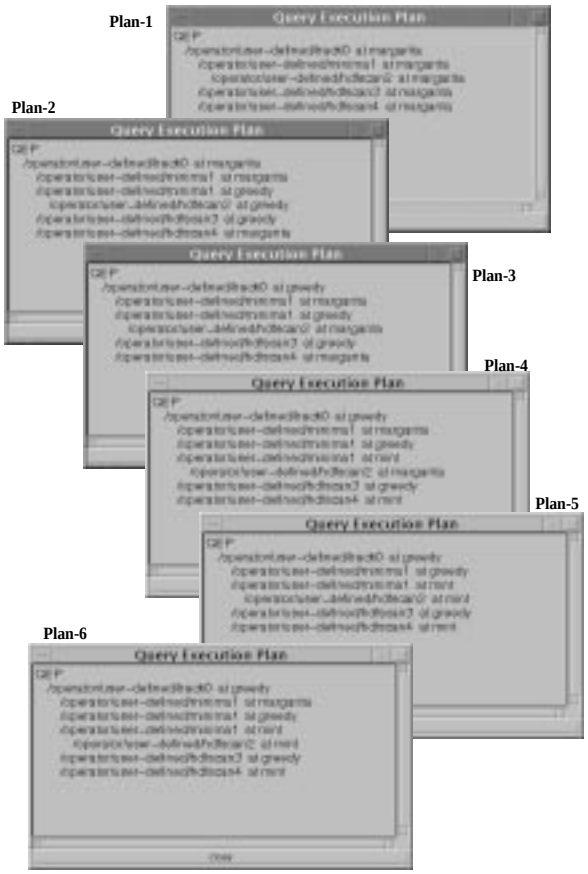


Figure 4. Plan Changes During Cyclone Tracking Query Evaluation

For comparison, we executed the three execution plans presented as our motivating example under the same emulated workload change but with no dynamic reconfiguration. Plan-A, where all operators were assigned to machine X, took 730 minutes. Plan-B and Plan-C took 244 minutes and 217 minutes respectively (Figure 5). The improvement was as high as 78% compared to Plan-A, and 36% and 28% compared to Plan-B and Plan-C. Furthermore, Plan-C can never be chosen by an optimizer that optimizes the query only once before the execution since the high workload on both machine Y and Z prevented Plan-C to be chosen. In contrast, The run-time optimization approach adapts the query configuration to the environment so that it delivered the best performance.

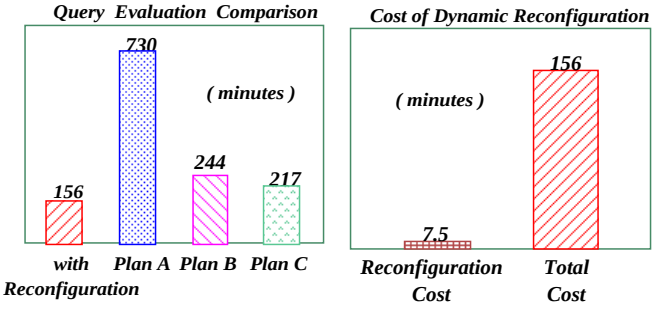


Figure 5. Experimental Results

As a proof of concept, we implemented dynamic reconfiguration mechanisms in a straightforward means which performs each reconfiguration step in sequence. In practice, reconfiguration steps should be performed in parallel. For example, once an operator instance and its all producers become suspended, its execution context can be captured immediately without waiting for the suspension of other operators. Similarly, steps of new operator instance creation and old operator instance destruction can be performed with different threads in parallel. We estimate the reconfiguration cost in our experiment can be decreased at least half if a parallel reconfigurator is implemented, hence the dynamic reconfiguration more beneficial.

3.5 Experiment II

The incurred overhead of dynamic query re-optimization depends on how often QEP reconfigurations occur and how much each QEP reconfiguration costs. The reconfigurations due to "noisy" change of parameters, which can be defined as a sharp but transient value variation, should be avoided. The trigger manager can preserve benefits from dynamic re-optimization by requiring a minimum duration of the new value, i.e., the *lasting interval*.

To study how lasting interval affects the performance in triggered dynamic re-optimization, we implemented the following experiment with the cyclone

tracking application. A 600 second execution segment, which is roughly divided into three 200 second periods, T_1, T_2 and T_3, is examined. During T_1, machine X is lightly loaded (i.e., almost all CPU time can be used by the given query) while machine Y is fairly loaded (i.e., about 50% CPU time can be used); thus query execution plan A in Figure 6 is the best configuration. During T_2, machine X became extremely heavily loaded

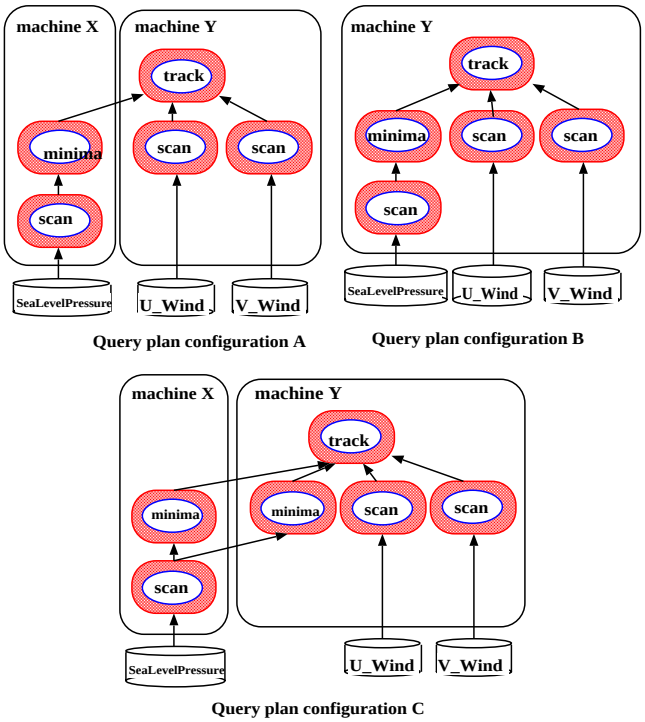
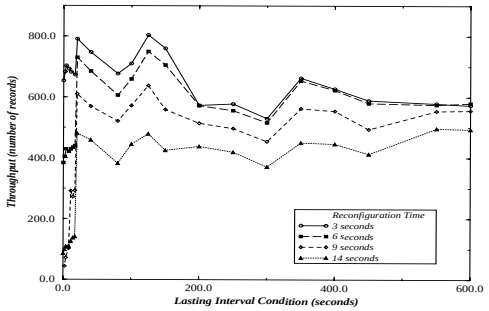


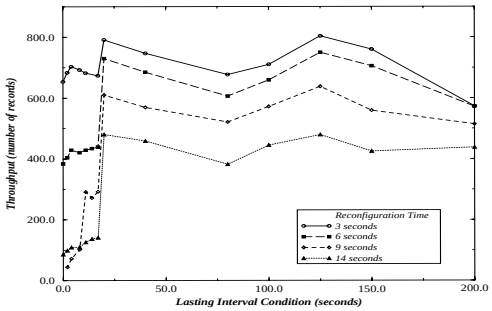
Figure 6. Query Execution Plans Considered in Experiment II

and the workload on machine Y is very light. Therefore, plan B is the best. In T_3, both machine X and machine Y are lightly loaded so that plan C performs best. Therefore, an ideal query execution with triggered dynamic re-optimization starts with plan A, then processes with plan B, and ends with plan C. In addition, these re-optimizations should occur at the starting of T_2 and T_3 respectively. However, the existence of noisy changes makes such an ideal situation impossible. During each execution period, sharp changes (approximate 20 seconds) occur in the middle of each period; hence "incorrect" reconfiguration decisions might be made. To avoid meaningless reconfiguration due to these noisy changes of workload, different lasting intervals are specified in the experiment. We found that if a lasting interval was defined too short, dynamic re-optimization would be too sensitive to any changes to system parameters. However, a long lasting interval would delay a re-optimization and make dynamic query re-optimization less beneficial.

We evaluated the performance by measuring the throughput with different lasting interval specifications. We also adjust reconfiguration cost (by calling additional computation loops) to observe its influence. Our experimental result is shown in Figure 7 where (a) shows the throughput variations when lasting interval values (marked points in the figure) vary from 0 to 600 seconds and (b) enlarges the portion of the first 200 seconds. When the lasting interval was just



(a) Lasting intervals vary from 0 to 600 seconds



(b) Lasting intervals vary from 0 to 200 seconds

Figure 7. Throughput with Different Lasting Intervals of Workload Changes

longer than the duration of a sharp change (about 20 seconds) or the lasting interval was long enough to "ignore" any small changes, the execution performed better than other lasting intervals. Smaller lasting intervals (i.e., less than 20 seconds) resulted in modifying QEP configurations too often while longer lasting intervals (e.g., 150 seconds) triggered the QEP reconfiguration too late so that less benefit were gained. We also note that reconfiguration cost has a significant impact on the performance. Smaller reconfiguration cost performed better.

4 Related Work

Traditional static query optimization techniques are well-developed and particularly useful for those queries that are short running, e.g., on-line transaction processing, but are not adequate for many of today's various application domains where long-running queries ranging from hours to days are common. Most efforts in query optimization concentrate on how to optimize a query plan prior to execution. For example, [3] and [9] proposed an approach to address the problem of parameter changes between compile-time and run-time by introducing a new operator *"choose-plan"* which incorporates a decision procedure to choose among alternative plans at start-up-time using up-to-date knowledge. However, it is required that these parameters which are unbound at compile-time must be defined or estimated no later than start-up time. Adaptive Optimization tries to improve the execution plan for a canned query after each run. With a query feedback mechanism, [2] approximates the attribute value distribution with a curve-fitting function. [17] divides and orders the plan search space into subspaces. A plan is produced based on the current system parameters and the performance of the past generated plans for the same query such that an incremental optimization can be done.

Recently, there is some related work in this area ([1], [16], [25]). In their work, either only limited changes are allowed to QEPs or changes are restricted to built-in functions. For example, [1] allows only the change of built-in *scan* operator *replacement*. [16] proposed mid-query re-optimization of QEPs in a relational database environment. This work mainly addresses the inaccurate estimation of data characteristics such as selectivity. Using statistics collected at run-time, the system reconfigures the unprocessed portion of a QEP, hence improving the performance. Intermediate results are materialized for plan modification. There are major differences between our work and theirs. First, they only reconfigure a QEP at points where no operation is partially done. In other words, no *pipelined* parallelism, or data flow structure, can be active when a reconfiguration is considered. On the other hand, we considered the dynamically re-optimizing QEP configurations in a parallel execution environment. The reason is that users are going to expect full parallel support for all user-defined extensions in an extensible database environment, e.g., ORDBMS [5]. Parallelization is important for performance improvement to long running queries. As a consequence, more reconfiguration forms are considered in our research.

Using a similar reconfiguration technique as in [16] (materializing intermediate results), [25] proposed a two-phase (plan rescheduling and operator synthesis) scrambling method to modify a query plan which is discovered to be suboptimal due to data arrival delays. Phase 1 identifies the non-blocked sub-trees and materializes the results produced by sub-tress. Phase 2 creates a new plan for processing the remainder of the query. Our work considers more generic re-optimization cases and allows user-defined trigger rules to exploit more reconfiguration opportunities.

We note that user-defined operators have been widely studied in recent years ([10], [11], [12], [15], [18], [21], [22]). Yet most published work discusses only optimization *before* query evaluation starts. For example, the *PREDATOR* database system [22], which views the world as an integrated collection of data types, each of which supports a declarative, optimizable query language, optimizes queries in an ORDBMS with enhanced abstract data types [21] before query execution.

5 Summary and Future Work

In this work, we focus on the performance improvement of long running query execution in an extensible and distributed query processing system. We have described an overview of the dynamic re-optimization mechanisms and proposed a novel approach to implement the triggered dynamic re-optimization on-the-fly. The *triggering approach* we use to re-optimize query execution plans when conditions of activated trigger rules are satisfied gives a timely response to the unexpected changes to system parameters and data characteristics. In addition, the trigger manager can also handle scheduled re-optimization requests and external re-optimization commands.

There are several issues with respect to dynamic re-optimization that require further investigation. First, though user-defined optimization triggers can specify the possible reconfiguration forms for dynamic optimizer, this can not help the optimizer to know more about how to *initially* optimize user-defined operators. In an extensible query processing system, a better means to accomplish the optimization task should allow users to anticipate this process in a simple but clear manner. In other words, operator developers should have be able to define their *user-defined optimization rules* and even more. Further, an efficient run-time measurement is a key component for the dynamic optimizer. The sensitivity of a user-defined function to system configuration and data characteristics has great impact in re-optimization decision making. A cost-effective prediction algorithm for system variables and data statistics parameters is vital. Finally, the techniques of dynamic re-optimization can also be applied

to other applications such as fault tolerance, mobile computing. For example, if a query is running on a mobile computing environment where the system configuration may change anytime, dynamic re-optimization can also provide a solution for guaranteed performance.

Acknowledgments The authors would like to thank Dr. Eddie C. Shek for his insightful comments. Thanks are due to all the members of Data Mining Laboratory in Computer Science Department at UCLA for the system implementation.

References

[1] G. Antoshenkov. Dynamic Query Optimization in Rdb/VMS. In *Proceedings of ICDE*, 1993.

[2] C.M. Chen and N. Roussopoulos. Adaptive Selectivity Estimation using Query Feedback. In *Proceedings of ACM SIGMOD*, 1994.

[3] R.L. Cole and G. Graefe. Optimization of Dynamic Query Evaluation Plans. In *Proceedings of ACM SIGMOD*, 1994.

[4] U. Dayal, E. Hanson, and J. Widom. Active Database Systems. In W. Kim, editor, *Modern Database Systems*, pages 434–456. Addison Wesley, 1995.

[5] D. DeWitt. Parallel Object-Relational Database Systems: Challenges and Opportunities. In *Invited Talk, PDIS*, 1996.

[6] F. Fabbrocino, E. Shek, and R. Muntz. The Design and Implementation of the Conquest Query Execution Environment. Technical Report CSD-970029, UCLA, June 1997.

[7] G. Graefe, R.L. Cole, D.L. Davison, W.J. McKenna, and R.H. Wolniewicz. Extensible Query Optimization and Parallel Execution in Volcano. In *Query Processing for Advanced Database Systems*, pages 305–335. Morgan Kaufmann Publishers, San Mateo, California, 1994.

[8] G. Graefe and W.J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of ICDE*, 1993.

[9] G. Graefe and K. Ward. Dynamic Query Evaluation Plans. In *Proceedings of ACM SIGMOD*, 1989.

[10] L.M. Haas and et al. Starburst Mid-Flight: As the Dust Clears. *IEEE TKDE*, 2(1):143–160, 1990.

[11] J.M. Hellerstein and J.F. Naughton. Query Execution Techniques for Caching Expensive Methods. In *Proceedings of ACM SIGMOD*, 1996.

[12] J.M. Hellerstein and M. Stonebraker. Predicate Migration: Optimizing Queries with Expensive Predicates. In *Proceedings of ACM SIGMOD*, 1993.

[13] IBM. *SQL Reference*. http://www.software.ibm.com/data/db2/library, 1998.

[14] Informix. *Illustra User Manual*. http://www.informix.com/answers, February 1998.

[15] M. Jaedicke and B. Mitschang. On Parallel Processing of Aggregate and Scalar Functions in Object-Relational DBMS. In *Proceedings of ACM SIGMOD*, 1998.

[16] N. Kabra and D.J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *Proceedings of ACM SIGMOD*, 1998.

[17] H. Lu, K. Tan, and S. Dao. The Fittest Survives: An Adaptive approach to Query Optimization. In *Proceedings of VLDB*, 1995.

[18] A.P. Marathe and K. Salem. A Language for Manipulating Arrays. In *Proceedings of VLDB*, 1997.

[19] K. Ng, Z. Wang, and R.R. Muntz. Dynamic Reconfiguration of Sub-Optimal Parallel Query Execution Plans. Technical Report CSD-980033, UCLA, 1998.

[20] K. Ng, Z. Wang, R.R. Muntz, and E.C. Shek. On Reconfiguring Query Execution Plans in Distributed Object-Relational DBMS. In *Proceedings of ICPADS*, 1998.

[21] P. Seshadri, M. Livny, and R. Ramakrishnan. The Case for Enhanced Abstract Data Types. In *Proceedings of VLDB*, 1997.

[22] P. Seshadri and M. Paskin. Predator: An OR-DBMS with Enhanced Data Types. In *Proceedings of ACM SIGMOD*, 1997.

[23] E. Shek, R.R. Muntz, M. Mesrobian, and K. Ng. Scalable Exploratory Data Mining of Distributed Geoscientific Data. In *Proceedings of KDD*, 1996.

[24] M. Stonebraker and D. Moore. *Object-Relational DBMSs*. Morgan Kaufmann Publishers, Inc, 1996.

[25] T. Urhan, M.J. Franklin, and L. Amsaleg. Cost-based Query Scrambling for Initial Delays. In *Proceedings of ACM SIGMOD*, 1998.