# Dynamic Reconfiguration of Sub-Optimal Parallel Query Execution Plans

**Kenneth W. Ng  Zhenghao Wang  Richard R. Muntz**

{kenneth,zwang,muntz}@cs.ucla.edu

Computer Science Department

University of California

Los Angeles, CA 90095-1596

September 21, 1998

## Abstract

Existing query optimization methods do not satisfy some of today's query processing requirements. Typically, only coarse or inaccurate estimates of database statistics are available prior to query evaluation. On the other hand, massive database sizes and growing demands for sophisticated processing result in long-running queries in extensible Object-Relational DBMS, particularly in decision support, and in data warehousing analysis applications. Therefore changes in system configuration and resource availability during query evaluation are not unexpected and can result in deteriorated query performance. Considering a parallel query evaluation environment, we propose dynamic reconfiguration of sub-optimal parallel query execution plans (QEPs) to adapt QEPs to the environment as well as to refined estimates of data and query characteristics. To ensure correct query evaluation in the face of modification of the QEP, we propose an algorithm to coordinate the steps in a reconfiguration. We also present a taxonomy of user-defined operators which allow the dynamic optimizer to plan a reconfiguration and systematically evaluate alternatives for execution context checkpointing and restoration. A syntactic extension of SQL to expose the relevant characteristics of user defined functions in support of dynamic reconfiguration is proposed. An example from the experimental system is presented.

# 1 Introduction

Growing demands for decision support and data mining against massive databases can result in very long-running queries that require parallel processing to deliver reasonable performance. During query evaluation, changes of system configuration and resource availability in a workstation farm environment are not uncommon. On the other hand, the introduction of abstract data types (ADTs) and associated user-defined functions (UDFs) in an extensible object-relational DBMS for non-traditional application domains (e.g., time-series data analysis, image processing, and geoscience information systems) makes it more difficult to estimate intermediate data sizes. Furthermore, knowing little about the UDFs operation semantics, query optimizers often treat a UDF sub-expression as an opaque operation that is not considered during optimization. As a consequence, static optimization methods that optimize a query once before execution do not satisfy today's query processing requirements.

A parallel query execution plan (QEP) specifies the parallel process of manipulating data and producing results for a given query. The subtasks of the QEP are called operators. The *configuration* of a QEP refers to the composition of operators, connections among operators, degree of data parallelism and operator assignment to different processors. In this paper, we present a novel approach that improves the performance of query processing in an extensible and distributed Object-Relational DBMS – *dynamic reconfiguring sub-optimal parallel QEPs*. 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.). As a consequence, available computing resources can be more efficiently utilized and the evaluation time of long running queries can be minimized. Reconfiguration includes options such as reassigning tasks to different processors, and changing the degree of data parallelism.

An important part of performing a reconfiguration is to ensure the *correctness* and the *completeness* of query results – which means no *erroneous* result is produced and *all* correct results are generated. Therefore, configuration changes are required to leave the QEP in a consistent state. Informally, a consistent QEP state is one from which the QEP execution can continue and such that prior output plus future output constitute the correct result. The evaluation of a QEP can be viewed as moving the QEP from one consistent state to another.

To address the above issue, we study *operator coordination* and *context management. Operator coordination* here means to bring operators to a state such that modifying the configuration of operators (e.g., migration to a new processor, degree of parallelism, etc.,) will not introduce erroneous results or lose valid results. Coordinating distributed application modules in the face of a system configuration evolution is widely studied in configuration management where loosely-coupled processes are assumed and their state is self-contained except for certain periods when they are interacting with each other ([6][17]). However, a QEP is different in that the computation of an operator relies heavily on the input data that is usually the output of another operator. We here propose an algorithm that is appropriate for a distributed query processing environment.

*Context management*, on the other hand, refers to how an operator's execution context, which can be described as a summary of data processed up to the current time (e.g., the partial sum and count of an average computation), be carried forward to the newly configured operator. Operator semantics can have great impact on context management. By identifying their characteristics relevant to dynamic reconfiguration, we classify operators (both built-in and user-defined functions) and propose different methods for capturing and restoring execution context.

In section 2, we present an overview of our work on supporting dynamic reconfiguration of sub-optimal parallel query execution plans. Section 3 is devoted to the discussion of operator coordination and context management. SQL syntax extensions for UDF registration with support for reconfiguration features are suggested in section 4. The experimental results of reconfiguring QEPs on-the-fly are also presented in this section. After a discussion of related work in section 5, section 6 summarizes the paper and presents a brief outlook on future work.

# 2 Reconfiguration Overview

## 2.1 Parallel Query Execution Plans

In database query processing, a query execution plan (*QEP*) specifies the process of manipulating data and producing results. Parallelism in a workstation farm environment can be harnessed by assigning operators in a QEP to different processors. The communication cost, which is the overhead for sending and receiving data records over the network, can offset the benefits from this parallelism. Therefore, multiple operators may be compositionally grouped into a *fragment* that is executed within a process, hence behaving like a *macro-operator*, to maximize overall performance.

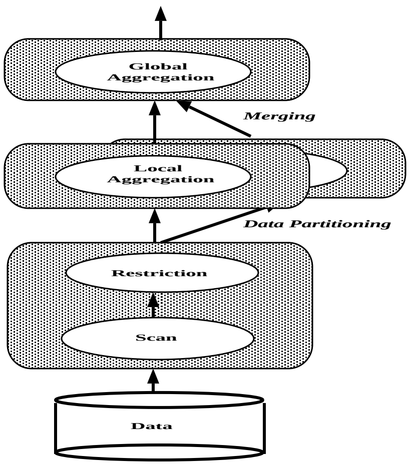


Figure 1: A sample query execution plan

In the following, we use the term *operator* to refer to both *basic* and *macro* operators unless it is necessary to distinguish them. An operator may be cloned into multiple instances for data parallelism, where each *operator instance* (or *clone*) processes a portion of input data (Figure 1).

Basic operators in a QEP can be either *built-in* or *user-defined* functions (UDFs) that implement algebraic operations or enforce physical properties (e.g. sort ordering) of data records. An ORDBMS provides a number of built-in functions that can be used in query expressions. For example, the typical built-in functions available in DB2/SQL include: numeric operators $+, -, *, /$, string operator || (concatenation), SUBSTR (substring), and aggregate functions COUNT, SUM, AVG, MAX, MIN ([12]). Knowledge about an operation and input ordering can be exploited for efficient performance. A simple example from [5]

**SELECT MAX(ORDER_NUMBER) FROM SALES**

illustrates a case in which only the first data object needs to be retrieved by using a descending index on ORDER_NUMBER.

When built-in functions are not sufficient to express complex query requirements, ADTs (Abstract Data Types) are introduced by application developers and additional User-Defined Functions (UDFs) that operate on associated ADTs can be defined and registered. In DB2, for example, a user can define a function *minima* that operates on a two dimension array (i.e., an ADT) to find "*local minima*" (each grid point with a value smaller than its neighbors) as shown in Figure 2. As can be seen from this example, the developer specifies the characteristics of a UDF as a part of the user-defined function registration. Among these characteristics, the user has to specify whether

```
CREATE FUNCTION minima( TwoDimArray, INTEGER, INTEGER,  FLOAT, INTEGER )
RETRUNS SetOfPoints
EXTERNAL NAME 'TwoDimArray!minima'
LANGUAGE C
PARAMETER STYLE DB2SQL
NOT VARIANT
NOT FENCED
NOT NULL CALL
NO SQL
NO EXTERNAL ACTION
NO SCRATCHPAD
NO FINAL CALL;
```

Figure 2: Registration of a new UDF in DB2

"SCRATCHPAD" is used or not. A scratchpad is the memory area assigned to a UDF to "remember" the *context* (or *execution state*). This space is allocated on the first execution of a UDF ( if "SCRATCHPAD" is defined ) and not deleted until the last execution. It allows the summary information concerning previously processed data items to be carried forward to the next execution of the same UDF, and influence the behavior of future computations. Further, an ORDBMS such as Informix Illustra supports operators that compute a function in an object-at-a-time fashion [13]. This design resembles the Open-Next-Close query evaluation paradigm [7][8][4][22] that supports data stream processing. In this paper, we assume that all operators run in a stream processing environment and each execution of an operator is called an *iteration.*

An input object stream can be defined as follows:

1. [] is an input object stream (the empty stream consisting of just the *end of stream* marker);
2. $r[]$ is an input object stream, for any input object r;
3. $rR$ is an input object stream where $r$ is the next available input object followed by input object stream $R$.

An object instance in the data stream can be denoted as $r_p$ where $p$ ($> 0$) is an integer uniquely identifying the position of data object $r$ located in stream $R$. During the evaluation, the function $f$ reads input objects in a sequence dictated by their positions in the stream, produces output objects, and/or modifies the operator scratchpad memory, i.e., its *execution context.* Such a function $f$ is described in terms of two function parameters, $\delta$ and $\alpha$, and an iterative control structure $F$:

$$f(R) = F(s_0, R)$$

$$F(s, []) = \alpha(s, [])$$

$$F(s, rR) = \alpha(s, r)F(\delta(s, r), R)$$

where [] is a marker to indicate *the end of an object stream.*

There is a set of "context" values, $\Sigma = \{\sigma_0, \sigma_1, ...\}$, where $s_0 \equiv \sigma_0$, i.e., the initial context is the null context, the scratchpad context before the UDF processes any input. The *context transition function* $\delta$ maps a (context, input)-pair to a new context, and $\alpha(s, r)$ is the output produced in context $s$ with input $r$. Thus $F$ maps a (context, object stream)-pair into an output object stream.

## 2.2 Dynamic Optimizer

A dynamic optimizer takes advantages of up-to-date system configuration and query execution information to optimize query execution plans on the fly. Briefly, the dynamic optimizer periodically measures system performance and query characteristics metrics during run time. Variation of these variables are then estimated for a number of future time periods such that the "best" QEP at the beginning of each period can be found. If it is estimated that a new QEP, different from the *current QEP*, leads to a significant improvement in query performance, a migration strategy from the current plan to the proposed plan is determined and then executed.

### 2.2.1 Run-time Measurement

The problem of analyzing the performance of a parallel QEP is divided into two parts: *performance analysis* and *instrumentation*. 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 spaces and resources 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.

### 2.2.2 Dynamic Reoptimization

Dynamic reoptimization can be both *event driven* and *time driven*. An event can be a resource configuration change. For example, one workstation participating in the query evaluation is going to be shutdown in 5 minutes for maintenance. On the other hand, the dynamic optimizer periodically reviews the optimality of the running QEP in an execution.

The dynamic optimizer considers the reconfiguration cost in addition to the execution cost of the remainder of the query. This additional cost dimension increases the complexity of searching for the best query plan. For instance, a plan with slightly higher cost (based on current estimates of system configuration and query characteristics) but much lower reconfiguration cost (to other potential QEPs) could be the better choice since it allows more efficient plan migration.

We use a dynamic programming technique to find the plan with maximum estimated throughput for a finite number of future time periods. We describe the algorithm briefly here, and the details can be found in [24] [25]. The (estimated) residual execution time from the point at which the reoptimization starts is divided into fixed length time periods. System statistics and query characteristics at the beginning of each period are estimated based on run-time measurement information. Each period $[t_{i+1} - t_i]$ includes the reoptimization and reconfiguration overheads. The optimizer tries to make its decision by maximizing the aggregate throughput. Starting from the last period, the dynamic programming algorithm recursively computes the maximum expected throughput from the beginning of each period until the end of the last period. In practice, the dynamic optimizer considers a certain number of future time periods rather than *all* periods.

### 2.2.3 Query Plan Modification

A query plan modification, which is performed by a *reconfigurator*, yields a new, semantically correct configuration for the same query. The process of a 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 alteration*, 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 the semantics of operators and respective costs. In this paper, we focus our discussion on the operator coordination and state capture/restoration.

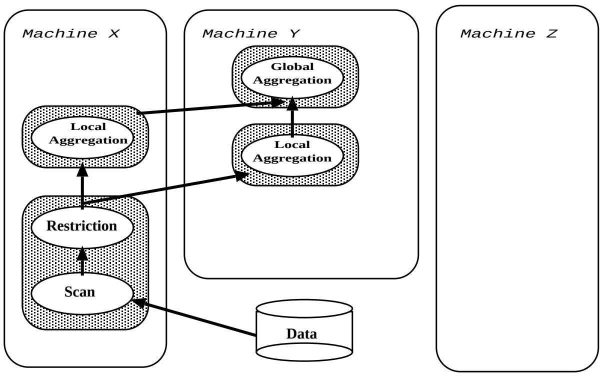


Figure 3: a QEP with data parallelism

### 2.3 Reconfiguration Forms

Assume that we have a parallel QEP as shown Figure 1. The valid forms of plan modification which yield valid QEP configurations that we consider are as follows.

1. *Operator relocation* refers to re-assigning an operator instance from one processor to another. For example, if the workload on machine Y becomes heavier, we might want to move the *local aggregation* UDF (provided it is the bottleneck) to another machine, say $Z$. operator relocation is similar to the process migration facility provided in other systems (e.g. Condor[2], Utopia[26]). The major difference is that the semantics of operators may be exploited in the context of data processing to more efficiently implement migration.

2. *Data stream re-partitioning* modifies the distribution policy for a data stream that feeds different clones of an operator whereas each clone runs on a different processor, processes a portion of data stream and maintains individual scratchpad memory. For example, the *restriction* operator on machine X provides data records to the *local aggregation* operator clones on machine X and machine Y; perhaps alternating between X and Y. Assuming that the optimizer decides to change the data redistribution policy due to the change of workload on machine X (or machine Y), a data stream re-partitioning could use a new policy which sends *one* data record to machine Y, and, alternately, *two* data records to machine X.

3. *Modification of the degree of data parallelism* changes the number of clones of an operator. *Local aggregation* in our example QEP has two clones, each of which processes a portion of the data produced by the *restriction* operator. If *local aggregation* becomes the bottleneck in this QEP and another machine, say Z, becomes available during the query evaluation, then

another clone of *local aggregation* could be created on machine Z and the partitioning policy of the *restriction* operator adjusted to send each clone an appropriate proportion of the records.

4. *Macro-operator regrouping* either splits a macro-operator (which is composed of multiple basic operators) or merges some operators into a new macro. The *scan* and the *restriction* operators on machine X, in our example, are originally grouped as a macro-operator. A split of this macro-operator can enable a migration of the *scan* (or the *restriction*) operator to another available processor.

5. *Operator Replacement* uses a semantically equivalent but different algorithm in place of an implementation. We may, for example, have different algorithms to scan datasets. Typically, an index scan operator, which is semantically equivalent to a sequential scan operator, is more efficient provided the search index is available and the hit ratio meets a certain criteria.

6. *Sub-expression transformation* re-selects an semantically equivalent algebraic expression. For example, if a secondary index is available, we can consider performing *index scan* on the secondary key with different predicates for the *restriction* operator. Or if operations are commutative, we may exchange their order in a pipeline.

# 3 Maintaining Query Execution Plan Consistency

A QEP modification must ensure the final execution results are *correct*, i.e., no *erroneous* result is generated, and *complete*, i.e., *all* correct results are produced. Therefore, a reconfiguration has to be carried out such that the consistency of the query evaluation is not breached. Our approach is to ensure: (1) *Operator Coordination* — the operators which are to undergo change are brought to a state where their configurations can be amended without affecting others; (2) *Context Management* — the states of operators before the reconfiguration can be carried forward to respective operators after the reconfiguration.

## 3.1 Classifying Operators

A classification that characterizes operators such that operator coordination and context management options can be derived is presented as follows.

**Completely Context-Free** At the *end* of a particular iteration, if the execution context of an operator is equivalent to its initial (null) status this operator is said to be at a *context-free point.* Furthermore, if an operator returns to its initial context at the end of *each* iteration, this operator is called *Completely Context-Free.*

Let $R$ be the input data stream, $s_0$ the initial context, $\delta$ a context transition function of operator $f$ such that $s_j = \delta(s_{j-1}, r_j)$. A completely context free operator has the property:

$\forall j,\ s_j = s_0.$

Examples of completely context-free operators include operations such as *filter* and *projection.*

**Periodic Context-Free** operators in this class return to the initial context every $n$ iterations where $n$ is a constant known at query compile-time.

Let $R$ be the input data stream, $s_0$ be the initial scratchpad context, and $\delta$ a context transition function of operator $f$ such that $s_j = \delta(s_{j-1}, r_j)$ where $r_j \in R$. A periodic context-free operator has the property:

$\forall k \geq 0,\ \ s_{kn} = s_0$ where $n > 1$ is a constant.

An example is an aggregate function which computes *30 (business) day* averages of IBM stock closing prices from input that consists of daily closing prices in chronological order.

**Sporadic Context-Free** In contrast to a periodic context-free operator, if the number of iterations to return to the initial scratchpad context is data dependent but bounded by a constant, then the operator is *Sporadic Context-Free.*

Let $R$ be the input stream, $s_0$ the initial scratchpad context, $N > 0$ a constant, and $\delta$ a context transition function of operator $f$ such that $s_j = \delta(s_{j-1}, r_j)$ where $r_j \in R$. A sporadic context-free function has the property:

$\forall j\ s_j = s_0,\ \exists n < N\ \ s.t.\ s_{j+n} = s_0.$

As an example, consider a given input stream of IBM stock closing prices which are irregularly recorded in chronological order. An operator which computes *monthly* closing price averages on this input stream is sporadic context-free.

**Fixed Length Memory** For example, given an input that consists of daily IBM stock closing price records in chronological order, a user-defined aggregate function that computes *30 business day* closing price running averages belongs to this class. Operators in this class have the property that the execution context depends on, at most, the last $n$ input records for some constant $n$, but does not return to any a priori known state.

Let $R$ be the input stream, and $\delta$ a context transition function of operator $f$ such that $s_j = \delta(s_{j-1}, r_j)$ where $r_j \in R$. A fixed length memory operator has the property:

$$\forall j,\ s_{j+n} = \delta(s_{j+n-1}, r_{j+n}) = \delta(\cdots\delta(s_0, r_{j+1}), \cdots, r_{j+n}) \text{ where } n \text{ is a constant.}$$

Since the effect of a record read in each iteration disappears after a fixed number of input records, an equivalent context can be rebuilt by processing or re-processing a fixed number of records.

**Variable Length Memory** In contrast to operators with fixed length memory (i.e., where an input record has a time limited effect on the context), if the effect of an input record on the scratchpad context is data dependent but bounded by a constant, then it is a variable length memory operator.

Let $R$ be the input stream, $N > 0$ a constant, and $\delta$ a context transition function of operator $f$ such that $s_j = \delta(s_{j-1}, r_j)$ where $r_j \in R$. A variable length memory function has the property:

$$\forall j, \exists n < N, \quad s.t. \quad s_{j+n} = \delta(s_{j+n-1}, r_{j+n}) = \delta(\cdots\delta(s_0, r_{j+1}), \cdots, r_{j+n}).$$

An example is an operator that computes 30 day running temperature average from an input stream that consists of irregularly measured temperature records in chronological order.

**Context Non-Rebuildable** All operators that are not known to be in any of the above classes are in this class. For example, an operator that computes the monthly average IBM stock closing prices from an input that consists of *unordered* stock price records is in this class.

### 3.2 Operator Coordination

Performing a reconfiguration should leave the QEP in a consistent state. This requirement can be met if the operators involved in the reconfiguration are *quiesced* before reconfiguring the plan. In this section, we present an algorithm to correctly suspend all operator instances that are involved in the reconfiguration.

A parallel query execution plan can be viewed as a non-cyclic structure composed of a set of operator instances with directed communication channels among them. With respect to the data stream $R$ along a communication channel, the operator instance $P$ that produces the data stream is the *producer* while the operator instance $C$ that consumes this stream is a *consumer*. $C$ is also said to be a consumer of $P$ and $P$ a producer of $C$. With this non-cyclic structure, the root, i.e., the operator instance at the top level, is the ultimate consumer that does not produce a data stream to another operator instance, and the operator instances at the bottom level are ultimate producers that do not consume any data stream from other operator instances.

We determine the involved operators as follows. For *operator relocation*, the operator instance being moved from one machine to another and its producer instances are involved. With respect to *data stream re-partitioning*, the operator instances whose (output) data distribution policies are being changed are involved. A *modification of the degree of data parallelism* involves the operator instances whose clone numbers are being changed and their respective producer instances. *Macro-operator regrouping* involves not only the macro-operators being merged or being split, but also the producers who send data to these macro-operators. The operator instances being replaced and their producers are involved in a reconfiguration of *operator replacement*. When a *sub-expression is transformed*, all operator instances in the sub-expression being transformed and producer instances that feed data streams directly to this sub-expression are involved.

We recall that each process executes an instance of a macro-operator that consists of one or more basic operators. Each $n$-ary operator requires a computation thread, an output thread and $n$ input threads[1]. The computation thread performs computation iterations, the output thread sends data to consumers and the input threads receive data from producers. The output thread starts a *dispatch* operation by sending data records to the consumer operator which is either in the same process (within the same macro-operator) or in a different process (on the same or different machine). Such an operation completes when the output thread receives an *ack* (if the data is successfully put into the consumer input buffer), or a *nak* (if the data is rejected, e.g., due to buffer overflow). For efficiency output records are normally batched for transmission to consumer. A *receive* operation starts when the input thread receives a data transmission request from the

[1]In practice, we implemented input threads with Orbix's *thread-per-request* technology [14]. Therefore input threads are dynamically created upon request for data dispatch and are deleted after each request is processed.

producer. If the data records can be put into the buffer, the input thread returns an *ack*, otherwise, it returns a *nak*. We assume that the output buffer of an operator instance consists of $N_c$ FIFO queues, where $N_c$ is the number of consumer clones, i.e., one queue per consumer.

An operator instance cannot be suspended at any arbitrary time; the optimizer may want an operator to complete a certain number of iterations so that it reaches to a *suspendable point* at which the operator can be paused and reconfigured with lower cost. A *completely context-free* operator has a suspendable point at the end of each iteration, while a periodic context-free or a sporadic context-free operator comes to a suspendable point when it returns to the initial context. A fixed or variable length memory operator can have a suspendable point at the end of each iteration since it will never return to the initial context, and so does a context non-rebuildable operator. We call the time interval between two adjacent suspendable points an *execution interval*. In addition, the output dispatch operation is atomic relative to suspension, i.e., when an operator suspends, there are no partially executed dispatch operations by this operator. Once an operator instance is suspended, it does not initiate any computation iteration nor dispatch operation until the QEP computation is resumed. Because of these requirements, deadlocks and/or livelocks can occur.

A deadlock situation *will* occur if operator instance $A_1$ is blocked waiting for $A_2$ to (a) provide some data (if $A_2$ is $A_1$'s producer, and $A_1$'s input buffer is empty), (b) consume some data (if $A_2$ is $A_1$'s consumer, and $A_1$'s output buffer is full), or (c) acknowledge data transmission (if $A_2$ is $A_1$'s consumer); $A_1$ could be waiting for $A_2$, while $A_2$ is waiting for $A_3$; ...; $A_{n-1}$ is waiting for $A_n$; but $A_n$ is already suspended.

A simple example of a deadlock situation may result if a producer with full output buffer is in the middle of its execution interval (therefore more output data may be generated), but its consumer has already been suspended. Another similar example is when a consumer with empty input buffer is in its execution interval (therefore more input data may be required), but its producer has already been suspended.

To prevent deadlocks caused by pending data transmissions, all transmissions are required to be bounded in time. The input thread of an operator instance returns an *ack* to inform the producer of a successful transmission. The input thread returns a *buffer-full* if the data is rejected because the input buffer is full, and a *suspended* if the data is rejected because this operator is suspended[2]. Notice that the input thread is still alive when the operator instance is suspended, but the input

[2] *buffer-full* and *suspended* are two kinds of *nak*.

buffer is not changed. Further, a *top-down* algorithm in which a consumer will be suspended before any of its producers can prevent the consumer being blocked due to an empty input buffer and suspended producer. As to the deadlock caused by the suspended consumer and output buffer overflow, the computation thread of an operator instance is required not to start an execution interval unless the output buffer queue $i$ has at least $p_i$ empty slots (for all $i$'s), where $p_i$ is the maximum possible number of records this operator instance produces for queue $i$ per execution interval. This arrangement ensures that once an execution interval is started, it can always be completed successfully.

A livelock situation with respect to reconfiguration *might* occur if operator instance $A_1$ needs $A_2$ to consume or produce more data records in order to come to the suspendable point, while $A_2$ has the similar requirement to $A_3$, and so does $A_3$ to $A_4$, $\cdots$, $A_n$ to $A_1$. Because of this cycle, it might never be possible (or may be unacceptably long) for these operator instances to come to their suspendable points simultaneously. (If $A_i$ in the cycle is not involved in the reconfiguration, then its execution interval can be simply viewed as an iteration, and its execution needs $A_{i+1}$ running.) When more than one operator instances are requested to suspend, this cycle might cause livelocks.

An example of a livelock situation is between a producer and a consumer with one record input/output buffers. They depend on each other to come to the suspendable points. When the producer comes to its suspendable point, the consumer might not; and vice versa. If the producer produces $p$ records per execution interval, and the consumer consumes $q$ records per execution interval, and $p$, $q$ are relatively prime, then in the worst case, the producer and consumer needs $q$ and $p$ execution intervals respectively (hence $p * q$ records being processed) to come together to their respective suspendable points.

To avoid livelocks, additional output buffers for suspension are required. Let $p_i$ be the maximum possible number of records an operator instance produces for queue $i$ per execution interval; $q_j$ be the maximum possible number of records consumer clone $j$ consumes per execution interval of clone $j$. The operator instance distributes its $\%_i$ of produced records in an execution interval to queue $i$ and $\%_j$ to queue $j$. The additional output buffer size for queue $i$ should be greater than $LowerBound_i = p_i + \max_j \left(\frac{\%_i}{\%_j} q_j\right)$ to ensure that when consumer clone $i$ is suspended and no longer accepts data, its corresponding queue in the producer's output buffer can have enough room to hold data records that are produced and destined for consumer clone $i$ during the producer's execution intervals when some consumer clones have not yet been suspended. The $LowerBound_i$

slots in output buffer queue $i$ are reserved for "emergency" use. That is, they are empty during normal execution, and are used only if consumer clone $i$ is suspended. The producer knows that a consumer clone is suspended when the *suspended nak* is returned.

The consumer clones may not receive the suspension request at the same time, due to network delay or other reasons. To ensure $LowerBound_i$ slots is sufficient to avoid deadlock or livelock when a producer starts to use "emergency" output buffer slots, a special "emergency" tag is inserted into each output queue. This ensures that all consumer clones are aware that at least one of their clones has been suspended and its input data are being buffered in "emergency" slots, and that they should suspend themselves after their current execution intervals. When an operator is suspended, all special "emergency" tags remaining in the output buffer slots will be deleted.

The coordination algorithm is outlined as follows:

1. The involved operator instances are divided into groups according to connectivity. Each group is a connected subgraph of the QEP, and different groups are not connected.

2. Apply the following steps (3)-(6) to each group.

3. Identify the root operator of the group, and send suspension requests to all involved clones of the root.

4. Each involved clone of the root operator suspends itself after its current execution interval.

5. After all involved clones of the root operator come to their suspendable points, the producers of the root operator are requested to suspend themselves.

6. This process continues recursively until all operator instances in the group come to their suspendable points.

The correctness of the algorithm is argued below:

(1) The algorithm is free from deadlocks. When an operator instance is requested to suspend itself, the other suspended or to-be-suspended operator instances that will affect this operator instance are its consumers and its clones. (a) The operator instance will not be blocked by its consumer, because its output buffer will always have enough space to complete its current execution interval — it has guarded against this before the current execution interval. (b) The operator instance will not be blocked by its clones, because when it starts to consume data tagged as "emergency", it has enough input coming to "feed" it until its next suspendable point — its producer will still have enough output buffer queue space to accommodate any suspended clone. From (a) and (b), we know that any operator instance that is requested to suspend will eventually

come to its suspendable point, i.e., there will not be any deadlock situation.

(2) The algorithm is free from livelocks. When an operator instance is requested to suspend itself, its ability to comply might depend on its producer because it may have to consume some new data from the producer to complete its current execution interval. The producer, in turn, depends on operators down the tree to continue running, but this dependency is not cyclic. Were the output buffer queues too small, the producer would also depend on some consumer clones, which in turn depend on the producer itself, which in turn would depend on some other consumer clones. If this were the case, our algorithm could be livelocked. However, with the "emergency" buffer space, the producer does not have to depend on its consumer clones to continue their execution until all have come to their suspendable points. Therefore, no dependency cycles of pending operator instances can be formed, so the algorithm is free from livelocks.

(3) No intermediate results are missed or duplicated. Once a record is produced, it is stored in the output buffer until a *successful* transmission completes. Therefore no intermediate results are missed. Further, a produced record may have duplicate copies *only* when the consumer has put the received records into its input buffer while the producer has not received the *ack* message yet. However, since a dispatch operation is atomic relative to suspension according to the algorithm and our supporting design for the execution environment, therefore no intermediate results are duplicated. □

After coordinating all involved operator instances, the reconfigurator can save the buffered records and then restore them after modifying the plan. (This save/restore process is different from the context management we will discuss later, where the operator context refers to the summary information of the *processed* records that affects the computation.)

Further, we note that the above algorithm can be optimized such that once all of an operator's producers come to their suspendable points, this operator is in a state where other reconfiguration actions on this operator can be performed — there is no need to wait for all operators have been coordinated. Therefore, different phases in a QEP modification can be performed in *parallel*.

### 3.3 Context Management

Operator context management refers to maintaining the execution context of an operator instance before and after performing a plan reconfiguration. One way to move an operator instance from one process to another is to take a snapshot of the execution context and then restore such an image but this can mean writing additional, potentially complex code for each operator. (Note that if an operator is replaced then the context information is not necessarily in the same form and a transformation would be required.) An alternative that may be available is to *rebuild* an equivalent execution context by (re-)processing a finite number of input records.

In the following, we describe approaches for saving and rebuilding scratchpad context, and discuss the applicability of these approaches to each operator instance with respect to different forms of reconfiguration.

### 3.3.1 Methods

**No-Context** The simplest case is when an operator has no scratchpad context. Recall that in our example of registering a new UDF in DB2, a UDF is possibly registered with option "NO SCRATCHPAD". That means no scratchpad memory is allocated for this UDF at run-time. As a result, it is not necessary to capture and restore any execution context for reconfiguration.

**Waiting** Capturing/restoring the execution context of a running operator instance is not necessary if the scratchpad context is at the same status when the operator instance is initialized. Therefore, a possible strategy is to *delay* the operation of capturing/restoring the execution context until a "context-free" point at which the operator returns to its initial context.

**Check-Point** Checkpointing is a well-known technique (e.g., [2]) we can utilize. For dynamic QEP reconfiguration, instead of doing checkpoints periodically as in other systems, the reconfigurator takes checkpoint record of scratchpad when this option is chosen for context management. Once the involved operator instances are suspended with the coordination algorithm presented previously, the reconfigurator: (a) *takes checkpoint record of scratchpads*; (b) *implements the QEP changes*; (c) *initializes new scratchpads with the checkpoint record.* Then the QEP evaluation is resumed.

We note that the reconfigurator takes checkpoint records only when the operator instance is at the point between two successive iterations (or a suspendable point). If a checkpoint is taken in the middle of computing an iteration, the capture of the processor state information (such as program counter, stack pointer, etc.) is required and hence the options of reconfiguration is limited only to *operator relocation* among a few homogeneous machines.

Checkpointing is particularly suited for the migration of operator instances. However, checkpointing has its limitation since it can not be applied to all types of reconfiguration. For example, a reconfiguration that changes the algorithm may not be able to choose check-pointing method to manage scratchpad context since the newly elected operator may not have the same data structures as the operator once used. Therefore, we propose a *learning* method as a possible alternative to capturing/restoring scratchpad context in situations where checkpointing is inappropriate.

**Learning** Different from a *check-point* operation, which initializes an operator instance by copying the scratchpad context captured from others, context *learning* tries to build up the scratchpad context of a UDF by processing records (Figure 4). Informally, the same data stream flows to both

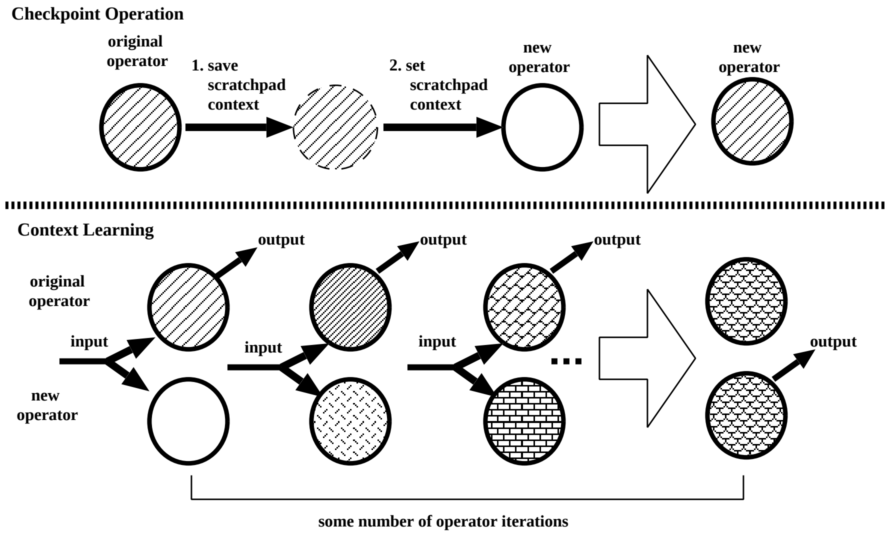


Figure 4: Checkpoint Operation and Context Learning

"old" and "new" operator instances. Once the newly elected operator instance arrives at a state equivalent to the old instance, or we say that the new operator instance is *well-educated*, which means both operator instances will henceforth produce identical outputs, the new operator instance can replace the old one. In the following, we describe how the reconfigurator applies the learning method for context management with respect to different forms of reconfiguration.

When context learning applies, it requires an answer to the question: *how many records do both the old and the new operator instances need to process in order to have equivalent context?* The answer to this question is different for different operators. Given an operator instance $M$ with a fixed length memory of $n$ iterations (or in terms of records), in order to have an equivalent context to $M$, another operator instance $M'$ must process $n$ records.

It is more difficult in the case of an operator instance with variable length memory. Let $M$ be such an operator instance, and $M'$ the new operator instance. The current scratchpad context of $M$ depends on at most last $n$ read records, though $n$ is not a constant known at compile-time. Starting from the current point, we can use a counter $c_1$ to number read records and another counter $c_2$ to number the read records remaining effective in the scratchpad context of $M$. Once $c_1$ is greater than $c_2$, we can claim that $M'$ has established an equivalent execution context to $M$.

For a state non-rebuildable operator, it is hard to determine such execution context equivalence unless the new operator instance re-processes *all* previously read records – this is impractical in almost all cases. Therefore, the learning method is not suitable for this class of operators.

### 3.3.2 Reconfiguration Process

**Operator Relocation** To implement a reconfiguration of moving an operator instance $M$ from one workstation $X$ to another $Y$, the reconfigurator creates a new instance $N$ of the *same* operator on $Y$ when the involved operators are being coordinated. The reconfigurator then saves all records in $M$'s input buffers and output buffer queues once the involved operator instances are suspended. $N$ is connected to $M$'s disconnected producers and consumers. The saved buffer records are then used to initialize $N$'s input buffers and output buffer queues respectively. On the other hand, the context of $N$ is initialized with the method most appropriate to its class category.

As to a *completely context-free* operator, the computation of all involved operator instances are resumed and $M$ is destroyed. No context of $M$ needs to be captured and restored. If a *periodic context-free* or *sporadic context-free* operator is migrated at a context-free point ( by setting a context-free point as the suspendable point while applying the coordination algorithm), the reconfiguration is exactly same to the process of a *completely context-free* operator. Or, a checkpoint operation can be used. The context of $M$ can be checkpointed once the involved operator instances are suspended. The reconfigurator then initializes $N$'s input buffers and output buffer queues with the checkpoint record before the computation is resumed. Lastly, $M$ is destroyed.

Checkpoint method is possibly the best way for migrating a *fixed length memory* operator or a *variable length memory* operator. The process shares the same checkpoint procedure we mentioned in above. An alternative is the *learning* method. After the reconfigurator initializes $N$'s input buffers with $M$'s saved buffer records, $N$ is executed under the *learning* mode. Whenever a new record arrives to $N$, its input thread sends a duplicate to $M$ which is executed "normally" but sends its output to $N$. Instead of sending consumers the produced records from its own output buffer, $N$'s output thread sends consumers the output received from $M$ and simply keeps its own output buffer clean. This process continues until a certain number of input records have been processed. Then $N$'s input thread temporarily rejects any input data until all input/output buffers of both $M$ and $N$ become empty. The execution of $N$ is changed to "normal" mode. In other words, $N$ receives input data normally and sends consumers the produced records from its own output buffer. $M$ is destroyed.

As to a *context non-rebuildable* operator, checkpoint operation is the only applicable method. The process is same to the checkpoint procedure for other operator classes.

**Data Stream Re-partitioning** Data stream re-partitioning changes the data redistribution policy of an operator instance $P$ (the producer) to its consumers $C_i$ ($1 \leq i \leq n$, where $n$ is the number of clones of the consumer operator $C$) without changing the number of consumer clones. We discuss the reconfiguration process as follows.

If $C$ is classified as *completely context-free*, once all involved operator instances are suspended, and buffered records are saved, the reconfigurator (1) changes $P$'s data redistribution policy to all $C_i$; (2) reorganizes records in all $C_i$'s input buffers and in $P$'s output buffer queues to the order when they were produced; these records are then mapped to $C_i$'s input buffers and/or $P$'s output buffer queues according to the new data redistribution policy. The computation can then be resumed and no context needs to be captured.

As to *periodic context-free* or a *sporadic context-free* consumer operators, the process is same to the producedure of *completely context-free* operators if the suspendable points are context-free. The *learning* method (though it may cost more) can also be applied so that the coordination algorithm does not need to suspend the consumer operator instances at their context-free points. The recofigurator creates new producer instance $P^n$ and new consumer instances $C_i^n$ ($1 \leq i \leq n$, where $n$ is the number of clones) when the coordination algorithm is executed. Once the involved operator instances are suspended and buffered records are saved, the reconfigurator changes $P$'s data redistribution policy towards $C$; and reorganizes records in all $C_i$'s input buffers and in $P$'s output buffer to the order when they were produced, then maps these records to $C_i^n$'s input buffers and/or $P^n$'s output buffer queues according to the new data redistribution policy. $P^n$ is connected to $P$'s producers and $C_i^n$s (consumers), $C_i^n$ is connected to $P^n$ (producer) and $C_i$'s consumers while $P$ is disconnected from its producers and all $C_i$ are disconnected from their consumers. $P^n$ and $C_i^n$ is executed under the *learning* mode. Whenever a new record arrive to $P^n$, its input thread sends a duplicate to $P$ which is executed "normally", so is all $C_i$. Each $C_i$ sends its output to the respective $C_i^n$. Instead of sending consumers the produced records from the output buffer, $C_i^n$'s output thread sends consumers the output received from the respective $C_i$ and simply clear its own output buffers. This process continues until $P^n$ has fed each $C_i^n$ a certain number of records. Then $P^n$ no longer duplicates input to $P$ nor accept any input data. When all input/output buffers in all $C_i$ and $C_i^n$ and the output buffer queues of $P$ become empty, the execution of the operator instances $P^n$ and all $C_i^n$ are changed to "normal" mode. The operator instances of $P$ and all $C_i$ are destroyed.

If the consumer operator instances are *fixed length memory* or *variable length memory*, the process of the *context learning* method we described should be used. In general, this reconfiguration form is not applicable for *context non-rebuildable* operators.

**Modification of the degree of parallelism** Modification of the degree of data parallelism is a reconfiguration which changes the number of clones of an operator. A modification of the degree of data parallelism always requires a data stream re-partitioning of the producers.

Let $C$ be the operator whose degree of data parallelism is changed and $C_i$ $(1 \leq i \leq l)$ the $i^{th}$ instance of $C$ where $l$ is the number of clones. $P$ is the operator which feeds data into $C_i$'s and the output of $C_i$'s are sent to $T$, where $P$ and $T$ may have multiple clones and are denoted as $P_j$ and $T_k$ $(j,\ k \geq 1)$. We use a set of new operator instances $C_i^n$ $(1 \leq i \leq m,\quad m \neq l)$ of $C$ to replace the original $C_i$s, and $P_j^n$ to replace $P_j$. All $C_i^n$ and $P_j^n$ are created (if necessary) when the coordination algorithm is executed. We discuss the reconfiguration process as follows.

If the operator $C$ is *completely context-free*, once the involved operator instances are suspended, and buffered records are saved, the reconfigurator (1) changes all $P_j$'s data redistribution policy; (2) reorganizes all records that were in all $C_i$'s input buffers and $P_j$'s output buffer queues to the order when they were produced; and then maps these records to all $C_i^n$'s input buffers and/or $P^n$'s output buffer queues according to the new data redistribution policy. The computation of all involved operator instances are then resumed. All $C_i$s are destroyed and no context needs to be captured.

As to *periodic context-free* or a *sporadic context-free* consumer operators, the process is same to the procedure of *completely context-free* operators when operators are suspended at a context-free point. The *learning* method is applicable so that the coordination algorithm does not need to suspend the consumer operator instances in their context-free points. The recofigurator creates new producer instance $P_j^n$s and new consumer instances $C_i^n$s when the coordination algorithm is executed. Once the involved operator instances are suspended and buffered records are saved, the reconfigurator changes each $P_j$'s data redistribution policy towards $C_i^n$s; reorganizes records in all $C_i$'s input buffers and in $P_j$'s output buffer queue to the order when they were produced, and then maps these records to $C_i^n$'s input buffers and/or $P_j$'s output buffer queues according to the new data redistribution policy. $P_j^n$ is connected to $P_j$'s producers and $C_i^n$s, $C_i^n$ is connected to $P_j^n$ (producer) and $C_i$'s consumers, while $P_j$ is disconnected from its producers and all $C_i$ are disconnected from their consumers. $P_j^n$s and $C_i^n$s are executed under the *learning* mode. Whenever a new record arrives to $P_j^n$, its input thread sends a duplicate to $P_j$ which is executed "normally" and feeds its output to $C_i$s. Each $C_i$ sends its output to the respective $C_i^n$. Instead of sending consumers the produced records from the output buffer, $C_i^n$'s output thread sends consumers the output received from $C_i$ and simply clear its own output buffers. This process continues until each $P_j^n$ has fed each $C_i^n$ a certain number of records. Then each $P_j^n$ no longer duplicates input to $P_j$ nor accepts input data. When all input/output buffers in all $C_i$ and $C_i^n$ and the output buffers of $P_j$s become empty, the execution of all $P_j^n$s and $C_i^n$s are changed to "normal" mode. The operator instances of all $P_j$s and $C_i$s are destroyed.

If the consumer operator instances are *fixed length memory* or *variable length memory*, the *learning* method process we described should be used. In general, this reconfiguration form is not applicable for *context non-rebuildable* operators.

**Macro-operator Regrouping** Operators assigned to the same machine may be grouped in the same process as a macro-operator in order to lower the communication cost between them. An operator relocation may trigger such a reconfiguration. On the other hand, in order to migrate an operator instance which is in a process with other operator instances, we need to split the macro-operator first. In practice, we use a new macro-operator instance to replace multiple connected macro-operator instances for *merging* whereas use multiple connected macro-operator instances to replace a macro-operator instance for *splitting*.

For macro-operator merging, the reconfigurator creates the new operator instances while co-ordination algorithm is being executed. Let $B$ be the connected sub-graph we want to use a macro-operator $N$ to take place. $P_i$s are the producers which feed data into $B$ and $T_j$s are the consumers of $B$. Once the involved operator instances are suspended and buffered records are saved, the reconfigurator initializes the input buffer(s) of operator instances in $N$ with the saved records from operator instances in B. $N$'s consumers are the consumers of the root operator in $B$ and $N$'s producers are the producers of the bottom operators in $B$. $N$ is executed under the *learning* mode. Whenever new records arrive to the bottom operator instances of $N$, the input threads send a duplicate to the bottom operator instances of $B$, which are executed "normally". The root operator instance of $B$ sends its output to the root operator instances of $N$. Instead of sending consumers the produced records from the output buffer, the output thread of the root operator instances of $N$ sends consumers the output received from the root operator instances of $B$ and simply clear their own output buffers. This process continues until a certain number of input records have been fed to $N$. Then the input threads of $N$ no longer duplicate input to the bottom operators in B nor accept any input data. When all input/output buffers in all operator instances of both $B$ and $N$ become empty, the execution of the operator instances of $N$ are changed to "normal" mode. In other words, the output threads of the root operator instances in $N$ send consumers the produced records from its own output buffer. All operator instances of $B$ are destroyed.

In general, this reconfiguration form is not applicable for *context non-rebuildable* operators.

**Operator Replacement** *Operator replacement* uses a new instance $N$ of *another* operator on the same or a different machine $Y$ to replace the original operator instance $M$ on workstation $X$. We note that $N$ must have the same input/output schema to $M$ (otherwise, either its producers or consumers must be replaced with new schema specifications as well — a *sub-expression* replacement). Usually, the internal data structure of $N$ is different from that of $M$ so that checkpoint method is not applicable. Therefore, the context learning method which were described in the paragraph of *operator relocation* should be used except for the classes of context-free operators whose context do not need to be captured and restored provided reconfiguration is taken at a context-free point.

**Sub-expression Transformation** Sub-expression transformation is to replace a sub-QEP $B$ with a new sub-QEP $E$. The producers and the consumers of $B$ do not change their schema.

For sub-expression transformation, the reconfigurator creates the new operator instances of $E$ while coordination algorithm is being executed. Let $P_i$s be the producer which feed data into the bottom operator instances of $B$ and $T_j$s are the consumers of the root operator instances of $B$. Once the involved operator instances are suspended and buffered records are saved, the reconfigurator initializes the input buffers of the bottom operator instances of $E$ with the saved records from the bottom operator instances of B (since their producers do not change, the input schema of the bottom operators in both the new and the old sub-expression should be same). $E$'s consumers are the consumers of the root operator in $B$ and $E$'s producers are the producers of the bottom operators in $B$. $E$ is executed under the *learning* mode. Whenever new records arrive to the bottom operator instances of $E$, the input threads send duplicates to the bottom operator instances of $B$, which are executed "normally". The root operator instances of $B$ send their output to the root operator instances of $E$. Instead of sending consumers the produced records from the output buffers, the output threads of the root operator instances of $E$ send consumers the output received from the root operator instances of $B$ and simply clear their own output buffers. This process continues until a certain number of input records have been fed to $E$. Then the input threads of the bottom operator instances of $E$ no longer duplicate input to the bottom operator instances of $B$ nor accept any input data. When all input/output buffers in all operator instances of both $B$ and $E$ become empty, the execution of the operator instances of $E$ are changed to "normal" mode. In other words, the output thread of the root operator instances in $E$ sends consumers the produced records from their own output buffers. All operator instances of $B$ are then destroyed.

A sub-expression including a non-rebuildable operator can not be replaced, since its context can not be rebuilt.

# 4 Implementation and Performance

This section suggests an extension of SQL syntax supporting the dynamic reconfiguration option. Experimental results are also presented.

### 4.1 Extended Syntax for Function Registration

As mentioned previously, an operator can be either a *built-in* or a *user-defined* function (UDF). In this subsection, we present the SQL syntax extension for the statements that allow the registration of UDFs with support for the reconfiguration features.

The extension is shown in Figure 5 with *italic* text style. The operator class is specified at

```
CREATE FUNCTION <function-name> (<argument type list>)
RETRUNS <data type name>
EXTERNAL NAME <external function name>
[ALLOW RECONFIGURATION WITH
   CLASS { COMPLETELY | PERIODIC | SPORADIC |
            FIXED | VARIABLE | NONREBUILDABLE }
   [ CHECKPOINT <capture function name>, <restore  function name>]
   [ LEARNING < learning function name> ]
   [ ORDER BY {<argument name> [ASC | DESC]}
  [ PARTITION RANGE USING <partitioning function name>]
LANGUAGE <language name>
...
```

Figure 5: Extension of UDF registration

registration. This indicates to the system what possible reconfiguration options can be applied to this UDF. A pair of check-point functions can be defined for capturing and restoring the context information stored in scratchpad. If check-point functions are not specified, then no *check-point* operation will be considered in performing a reconfiguration. The learning function is used to determine whether or not an equivalent context can be established after a certain number of data objects are processed in case the context learning method is used. No context *learning* option will be chosen in performing a reconfiguration if the learning function is not defined. For the classes of *periodic context-free*, *sporadic context-free*, *fixed length memory* and *variable length memory*, the input stream should be in a specified sort ordering with some partitioning policies that may constrain the degree of parallelism. An example of UDF registration is shown in Figure 6 where the user-defined *minima* function is specified as a *completely context-free* UDF and the routines to do checkpoint operations are defined.

### 4.2 Experimental Results

We have implemented the distributed version of *cyclone tracking*, an example query drawn from the geo-scientific application domain. Cyclones are indicated by local minima in atmospheric pressure at the earth's surface. We extract cyclone tracks from spatio-temporal sea-level pressure data field by first extracting local sea-level pressure minima, and then linking them together to form tracks,

```
CREATE FUNCTION minima ( TwoDimArray, INTEGER, INTEGER,  FLOAT, INTEGER )
RETRUNS SetOfPoints
EXTERNAL NAME 'TwoDimArray!minima'
ALLOW RECONFIGURATION WITH
   CLASS  COMPLETELY
   CHECKPOINT  'minimaSaveContext', 'minimaSetContext'
LANGUAGE C
PARAMETER STYLE DB2SQL
NOT VARIANT
NOT FENCED
NOT NULL CALL
NO SQL
NO EXTERNAL ACTION
NO SCRATCHPAD
NO FINAL CALL;
```

Figure 6: An example of UDF registration

i.e., "connecting the points", using auxiliary information such as wind direction.

The query execution plan consists of three operators. Built-in *scan* functions which retrieve surface pressure and wind records for the prescribed time interval in chronological order. These records are passed to a *minima* UDF that is responsible for *extracting* the location of local minimum in sea level pressure with certain constraints. The algorithm chosen is often based on the user's unique definition of the phenomenon. It has to be implemented as a UDF since no precise and universally accepted definition exists for such complex phenomenon and it is not easily expressed in terms of built-in functions. A local minimum in our example is detected at point $(x, y)$ if the sea-level pressure recorded at that local is less than the average value computed using a $5 \times 5$ neighbor grid points centered at $(x, y)$ by a certain amount. A threshold was set to 5.5 millibars to permit the detection of large, shallow low pressure areas. If the spatial resolution of the data set is coarse, the *minima* UDF may also smooth the locations of extracted minima, thus simplifying cyclone tracking by giving a better estimate of the position of the local minima. Given the list of extracted sea-level pressure minima for each time step, the *track* UDF extracts cyclone tracks that last at least three days (scientists may not be interested in short-lived cyclones). The location of two minima at consecutive time steps are considered to be a part of the same cyclone track only if (1) the difference in their spatial coordinates is within a certain threshold (depending on the time step value), user's unique definition e.g. 1/2 grid cell; or (2) their locations are consistent with the direction and velocity of the wind at the 700 millibar level (based on the assumption that wind and cyclone tend to move in the same direction).

Two Ultra-SPARC1 workstations, each of which is running SUNOS 5.5.1 with 124 MB memory, were used in this experiment. At start up time, all UDFs in the QEP are assigned to machine $X$ since machine $X$ was lightly loaded ( 98% CPU time is available for this query ) while machine $Y$

was extremely heavily loaded; shortly after the query started, 56% CPU time of machine *Y* became available and after 40 minutes, only 35% CPU time of machine *X* was available due to other computation intensive jobs; hence the migration of the *scan1* and the *minima* UDF to machine *Y* was triggered. Later, the dynamic optimizer detected that the *minima* UDF consumed over 60% of the CPU time this QEP consumed; it then decided to instantiate one more clone of the *minima* UDF on machine *X*. These reconfigurations are illustrated in Figure 7.

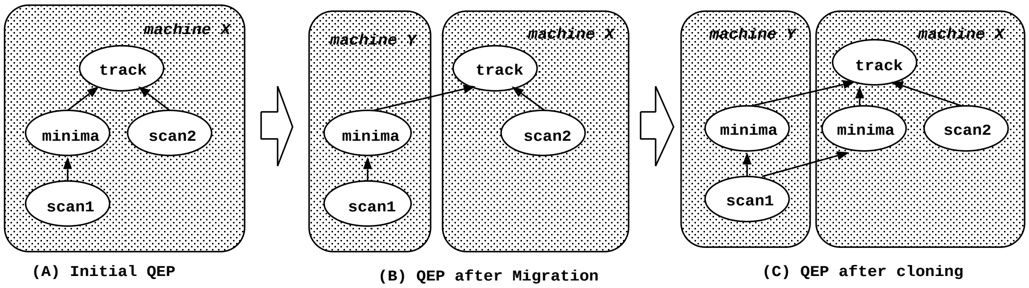


Figure 7: QEP reconfiguration experiment

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 125 minutes. For comparison, we executed the three execution plans under the same workload change without any dynamic reconfiguration. The initial QEP, where all operators were assigned to machine *X*, took 378 minutes. The second and the third execution plans took 202 minutes and 161 minutes respectively. The performance was significantly improved (Figure 8). The improvement was as high as 66.93% compared to the initial plan, and 38.12% and 22.36% compared to the second and the third plans. Note that it was unlikely the second and the third execution plan would be chosen at start-up time since the workload on machine *Y* was extremely high at that moment.

The incurred overhead consists of (1) plan search cost; (2) plan modification cost; (3) environment parameter estimation cost. The plan modification cost, which includes the operator coordination, context management (with checkpoint operation), and QEP re-construction, was measured to be about 3 seconds. The plan search using a dynamic programming algorithm took approximate 30 seconds while the system parameter prediction took about 50 seconds. Therefore, the total overhead in this experiment is less than 90 seconds (or 1%) of the overall execution time.

As a proof of concept experiment, the cyclone tracking QEP structure is straightward, and the

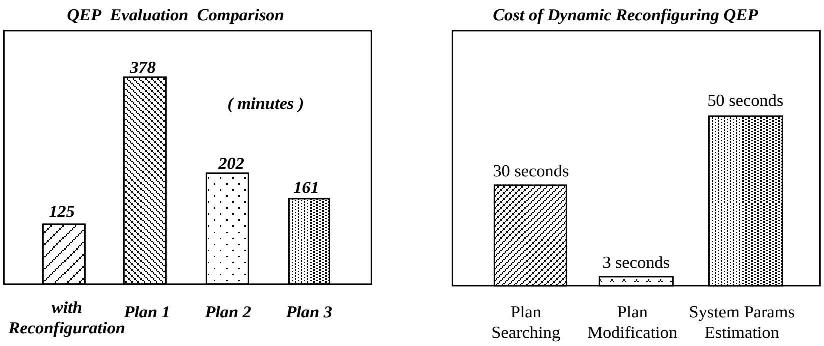


Figure 8: Performance Analysis of the Experiment

checkpoint method, which costs less than the learning method for context management, is used. In practice, since the plan search cost and configuration prediction cost are relative fixed for all QEPs and reconfiguration forms, to keep the reconfiguration cost as low as possible makes the dynamic reconfiguring QEPs more beneficial.

## 5 Related Work

There is some related work in this area ([1], [16], [23]). Yet none of the existing work is completely satisfactory – 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, is considered in their work. On the other hand, we considered the dynamic reconfiguration of QEPs in a parallel execution environment. The reason is that users are going to expect full parallel support for all ADT extensions in an ORDBMS [3]. Parallelization is important for performance improvement to long running queries. As a consequence, more reconfiguration options are considered in our research. Further, we consider reconfiguration involving not only the built-in operators but also user-defined functions in an ORDBMS. Lastly, to materialize intermediate results may not be

practical in the case of the large volume data sets. Therefore, other methods for capturing/restoring execution context are desired.

Using a similar reconfiguration technique as in [16] (materializing intermediate results), [23] 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 reconfiguration cases and provides methods for context capturing and restoring other than intermediate results materialization.

User-Defined Functions are widely studied in recent years ([11], [10], [9], [21], [20], [19], [15]). Yet most published work discusses only optimization *before* query evaluation starts. In [21], the *PREDATOR* database system, which views the world as an integrated collection of data types, each of which supports a declarative, optimizable query language, proposed optimizing queries in an ORDBMS with enhanced abstract data types [20]. A similar effort, but more specific to the data type of multi-dimensional array, can be found in [19], which proposed an algebra for multidimensional array data (AML) such that optimizer can apply rewrite rules to those operators specific to array operations. These efforts also focus their attention on the optimization with UDFs *before* the query evaluation starts. [15] studied the execution of user-defined functions using data parallelism. It is interesting to compare our classification of UDFs for reconfiguration to their classification for parallelization. [15] can be considered as an extension of [18] where the intrinsic characteristics of stream processing were analyzed. Similar reasoning is used in our UDF classification. Context rebuildable UDFs usually require input streams in a certain sort ordering such that the effect of the records read first will disappear before the effect of those records read latter. UDFs that handle unordered input streams usually have irregular execution context patterns.

## 6 Conclusions

In attempting to improve the performance of long running query execution in object-relational database systems, we consider dynamically reconfiguring suboptimal QEPs. An overview of the dynamic optimizer algorithm has been described. Maintaining the QEP state consistency, the key issue to ensure correctly evaluating the QEP when a dynamic reconfiguration happens is addressed in this paper.

We model the characteristics of user-defined functions with their execution context information. Since extensibility is one of the ORDBMS characteristics, such a classification-based approach enables the optimizer to obtain sufficient information about how a QEP might be reconfigured. The coordination algorithm that suspends involved operator instances before performing a reconfiguration is introduced. To support context capturing/restoring, we proposed a novel context management technique (context learning) as the alternative to the common use checkpoint operation.

Some important remaining work with respect to dynamic reconfiguration of QEPs is currently under study. An efficient run-time measurement is a must for the dynamic optimizer. The sensitivity of a user-defined function to system configuration and data characteristics has great impact in reoptimization decision making. The prediction model for system variables and data statistics parameters is a key in triggering the reconfiguration mechanism. On the other hand, the additional dimension of reconfiguration cost increases the complexity in searching for the best query plan. If the execution environment is relatively unstable, for instance, a plan with slightly higher execution cost (based on current system configuration) but much lower reconfiguration cost (to other potential QEPs) could be the best choice since it provides more efficient adaptation.

Though reconfiguring QEPs on-the-fly is motivated by improving long-running query evaluation performance, the techniques can also be applied to other situations. Fault tolerance, for example, is important for the query evaluations which involve large datasets, for example in data warehousing analysis application. To reconfigure a running QEP that becomes invalid due to resource availability provides an alternative for fault tolerant execution. With dynamic reconfiguration, a (relatively) long-running query can more easily be executed in a mobile computing environment where the system configuration may change anytime.

## References

[1] G. Antoshenkov. Dynamic query optimization in Rdb/VMS. In *Proceedings of the 9th International Conference on Data Engineering*, April 1993.

[2] A. Bricker, M. Litzkow, and M. Livny. Condor technical report. Technical Report CS-TR-92-1069, Department of Computer Science, University of Wisconsin, Madison, January 1992.

[3] D. DeWitt. Parallel object-relational database systems: Challenges and opportunities. In *Invited Talk, PDIS*, 1996.

[4] F. Fabbrocino, E. Shek, and R. Muntz. The design and implementation of the conquest query execution environment. Technical Report CSD-970029, UCLA, June 1997.

[5] P. Gassner, G.M. Lohman, K.B. Schiefer, and Y. Wang. Query optimization in the IBM DB2 family. *Data Engineering Bulletin*, 16(4):4–18, December 1993.

[6] K.M. Goudarzi and J. Kramer. Maintaining node consistency in the face of dynamic change. In *Proceedings of ICCDS, Annapolis, Maryland*, pages 62–69, May 1996.

[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 the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria, IEEE Computer Society*, pages 209–218, April 1993.

[9] L.M. Haas, W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B.G. Lindsay, H. Pirahesh, M.J. Carey, and E.J. Shekita. Starburst mid-flight: As the dust clears. *IEEE Transaction on Knowledge and Data Engineering*, 2(1):143–160, 1990.

[10] J.M. Hellerstein and J.F. Naughton. Query execution techniques for caching expensive methods. In *Proceedings of ACM SIGMOD*, pages 423–434, 1996.

[11] J.M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of ACM SIGMOD*, pages 267–276, 1993.

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

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

[14] IONA. *OrbixWeb Programmer's Reference*. IONA Technologies PLC, November 1997.

[15] M. Jaedicke and B. Mitschang. On parallel processing of aggregate and scalar functions in object-relational dbms. In *Proceedings of ACM SIGMOD*, pages 379–389, 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] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions On Software Engineering*, 15(11):1293–1306, November 1990.

[18] T.Y.Cliff Leung and Richard R. Muntz. Query processing for temporal databases. Technical Report CSD-890024, UCLA, April 1989.

[19] A.P. Marathe and K. Salem. A language for manipulating arrays. In *Proc. of the 23rd VLDB Conference, Athens, Greece*, pages 46–55, 1997.

[20] P. Seshadri, M. Livny, and R. Ramakrishnan. The case for enhanced abstract data types. In *Proc. of 1997 VLDB conference*, pages 66–75, 1997.

[21] P. Seshadri and M. Paskin. Predator: An OR-DBMS with Enhanced Data Types. In *Proc. of 1997 ACM SIGMOD International Conference on Management of Data*, 1997.

[22] E. Shek, R.R. Muntz, M. Mesrobian, and K. Ng. Scalable exploratory data mining of distributed geoscientific data. In *Second International Conference on Knowledge Discovery and Data Mining, Portland, Oregon*, August 1996.

[23] T. Urhan, M.J. Franklin, and L. Amsaleg. Cost-based query scrambling for initial delays. In *Proceedings of ACM SIGMOD*, 1998.

[24] Z. Wang. Conquest Dynamic Optimizer Design. Computer Science Department, UCLA, `http://dml.cs.ucla.edu/~zwang/research/Conquest`, July 1998.

[25] J. Yang, K. Ng, F. Fabbrocino, and R. Muntz. Dynamic query optimization with imprecise probabilities. `http://dml.cs.ucla.edu/~kenneth/research/dqo.ps`, Computer Science Department, UCLA, February 1998.

[26] S. Zhou, J. Wang, X. Zheng, and P. Delisle. Utopia: A load sharing facility for large, heterogeneous distributed computer systems. Technical Report CSRI-257, University of Toronto, April 1992.