

Parallelizing User-Defined Functions in Distributed Object-Relational DBMS

Kenneth W. Ng Richard R. Muntz

Computer Science Department

University of California

Los Angeles, CA 90095-1596

{kenneth, muntz}@cs.ucla.edu

Abstract

Full support of parallelism in object-relational database systems (ORDBMSs) is desired. The parallelization techniques developed for relational database systems are not adequate for ORDBMS because of the introduction of complex abstract data types and operations on ordered domains. In this paper, we consider a data stream paradigm and develop a query parallelization framework that exploits characteristics of user-defined functions in a ORDBMS during query optimization. By introducing the concept of windows and abstract data type orderings, we develop a novel approach for parallelizing user-defined functions in a distributed ORDBMS environment. The implementation issues in providing query services in ordered domains are also discussed.

1 Introduction

As time goes by, more and more database vendors agree that Object-Relational DBMSs (ORDBMSs) are the future [24]. Though full support of parallel ORDBMS is expected, the techniques used to parallelize relational database systems are not adequate for ORDBMSs due to the complex abstract data types (ADTs) and user-defined functions (UDFs) [1]. On the other hand, non-traditional database application domains such as time-series data analysis, image processing and geoscience information systems often require data objects in order. As a result, a novel mechanism that supports parallelization of user-defined operations on both ordered and unordered domains is desired.

To fully support parallelism in an ORDBMS, we consider a data stream paradigm and develop a parallelization framework that exploits characteristics of

UDFs in a ORDBMS during query optimization. The concept of user-defined orderings is introduced for stream processing. Further, data parallelism of query execution in a distributed environment involves determining whether the input data stream can be split for concurrent processing by multiple instances of a UDF, and if so, what constraints apply. We introduce the concept of “windows” for modeling the characteristics of data partitioning opportunities available for a UDF. The derived properties of windows on input streams dictate how streams can be properly split and distributed.

The organization of this paper is as follows. In section 2, we present the fundamental concepts of parallel query execution. Section 3 is devoted to the parallelization of user-defined functions. User-defined orderings and stream data distribution are addressed in this section. Implementation issues are discussed in section 4. After a discussion of related work in section 5, section 6 summarizes the paper and presents future work.

2 Modeling Parallel Query Execution

In a stream data model, positions in a data stream are integers $\{1, 2, \dots\}$ that are monotonically increasing. Each position is associated with one and only one data record.

Definition 2.1 A data record r is a record of type $T = t_1 \times t_2 \times \dots \times t_n$ ($n > 0$) where t_1, t_2, \dots, t_n are either system basic data types or abstract data types. The schema of r is written as $\langle a_1 : t_1, a_2 : t_2, \dots, a_n : t_n \rangle$ where a_i is the i^{th} attribute of r . A special null record, denoted as $Null_T$, is associated with the domain of record type T . \square

Definition 2.2 Let R be a set of records of type T . A data stream of type T is a many-to-one function $ds : integer \rightarrow T$ iff (1) for every $r \in R$, there exists

some $i > 0$, such that $ds[i] = r$; and (2) if $ds[i] = r$, then $i > 0$ and $r \in R$. i is a position of r in ds . \square

A record schema contains one or more attributes. While mapping a set of data records (i.e., unordered records) to a data stream (i.e., ordered records), attributes are divided into two groups — *ordering attributes*, which are the determinants of the record ordering, and *ordinary attributes*, which are the other attributes. Similar to sort keys in a physical index file, ordering attributes can be viewed as a logical list. Records are sorted according to a certain ordering (e.g., ascending) of the first attribute, which will result in an ordered sequence of clusters; each cluster consists of records with the same values in the first attribute. Each cluster is ordered on the second attribute, which will result in clusters of records with the same values in the first two attributes. These clusters are ordered on the third attribute, and so on.

In database query processing, a parallel query execution plan (*QEP*) specifies the process of manipulating data and producing results in a distributed environment (e.g., workstation farms). A QEP is composed of operators that implement algebraic operations or enforce physical properties (e.g., sort ordering) of data objects. In addition to assigning operators to different processors, a DBMS may clone an operator for data parallelism, where each *clone* (or *instance*) processes a portion of input data. An operator can be either *built-in* (e.g., $+$, $-$, $*$, $/$ and COUNT, SUM, MAX) or *user-defined*. User-Defined Functions (UDFs) should be registered along with ADTs (Abstract Data Types) for complex query requirements. In DB2 [8], for example, a user can define a *minima* UDF that operates on a two dimensional array (i.e., an ADT) to find “*local minima*” (each grid point with a value smaller than its neighbors) as shown in Figure 1. As can be seen

```
CREATE FUNCTION minima( TwoDimArray, INTEGER, 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 1. Registration of a new UDF in DB2

from this example, among the specified UDF characteristics, the user has to define whether “SCRATCHPAD” is used or not. A scratchpad is the memory area assigned to a UDF clone to maintain the *context* (or *execution state*). This space is allocated when a UDF

clone is initiated (if “SCRATCHPAD” is defined) and not deleted until the last execution of this clone. The context held in the scratchpad memory concerns previously processed records and is carried forward to the next execution. An ORDBMS such as Informix Illustra supports UDFs that run in an object-at-a-time fashion [9]. This design resembles the Open-Next-Close query evaluation paradigm [3][4][2][23] that supports data stream processing. In the rest of this paper, we assume that both built-in and user-defined functions run in a stream processing environment. In addition, we consider a general UDF notion that subsumes column and table user-defined functions.

Data records are read sequentially from an input stream. Very often stream processing requires input streams in a proper order to efficiently perform the computation while reading the input streams only once [11]. Therefore, record orderings can have a significant effect in stream processing performance. Current DBMSs support lexicographical ordering over the domain of strings, numeric ordering over the domain of numbers and chronological ordering over the domain of date/time. However, these built-in orderings based on the results of literal comparisons are not sufficient since abstract data types can be introduced arbitrarily and orderings based on the results of *semantic* comparisons are desired [17]. Further, a UDF may be cloned into multiple copies on processors for data parallelism. Therefore, data streams are split into substreams for evaluation. A number of well-known splitting strategies have been proposed and implemented in relational DBMSs, e.g., round-robin, hashing and range-partitioning [18]. However, many of them do not recognize the need to maintain some degree of the original ordering. To apply these proven techniques, a UDF developer should have some means to notify the optimizer how the UDF semantics affects the possible stream splittings so that the most appropriate strategy can be chosen.

3 Parallelizing UDFs

3.1 ADT Ordering

In a relational DBMS, no matter what the domain of values for an attribute, we can, in principle, compare *values* from the domain, and therefore we can sort these values as a (partially) ordered list[25]. Current ORDBMSs support lexicographical, numeric and chronological orderings for both built-in types and ADTs. For example, DB2 automatically generates functions for value comparisons if the source type of an ADT is in a domain of strings, numbers or date/time [8]. However,

these built-in orderings are not sufficient to represent ADTs' sophisticated semantics. For example, suppose we use an ADT *branch* to describe a nationwide corporation. With respect to the user-defined *.oversee.* ($x.oversee.y$ if x administers y .) and the built-in $<$ relationships, we can have different hierarchies (Figure 2). The optimizer may use a built-in sort operator to order

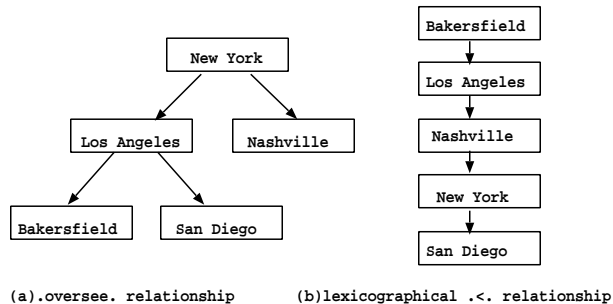


Figure 2. Orderings

instances in the *branch* domain with the lexicographical ordering for some algorithms (e.g., merge-join) but such a built-in sort operator is not able to order the instances to represent the semantic order of the administrative hierarchy which may be required by a UDF. To remedy this situation, we extend value comparisons by allowing *user-defined ADT ordering*.

Definition 3.1 Given a set of records D of abstract data type T , a user-defined method $M : T \times T \rightarrow \text{boolean}$ is a partial ADT ordering on D iff
 (1) $\forall x, y \in D, M(x, y) \wedge M(y, x) \implies x = y$;
 (2) $\forall x, y, z \in D, M(x, y) \wedge M(y, z) \implies M(x, z)$.
 Further, M is a total ADT ordering on D if and only if M is a partial ADT ordering and $\forall x, y \in D$, either $M(x, y)$ or $M(y, x)$. \square

An ADT ordering is often a partial order in practice. For example, with respect to the *.oversee.* comparison mentioned in above, both $[Los\ Angeles, San\ Diego, Bakersfield]$ and $[Los\ Angeles, Bakersfield, San\ Diego]$ are considered to be valid since branches at *Bakersfield* and *San Diego* are not overseeing each other.

To register an ADT ordering, we propose the syntax shown in Figure 3. As an example, the *administration*

```
CREATE ADT ORDERING <ADT ordering name>
ON <abstract data type name>
ORDER BY <registered function name>;
```

Figure 3. Registration of ADT Ordering

ordering can be defined with the statement

```
CREATE ADT ORDERING administration
ON branch ORDER BY oversee;
```

where *oversee* is a registered UDF taking two values, x and y , of the ADT branch type, and returning a boolean value indicating whether x oversees y or not.

3.2 Record Ordering

User-defined record ordering refers to taking a collection of records of the same type and mapping them to the positions of a data stream according to the comparison results of ordering attributes. ADT ordering together with traditional ascending and descending can be used for specifying sorting direction.

We are interested in the effect of different record orderings on UDF performance. Consider, for example, a simple UDF *average* (Figure 4) which lists all branches

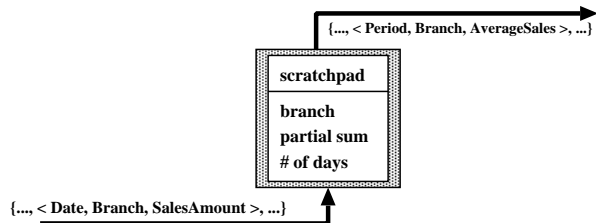


Figure 4. An UDF that computes average sales for each branch

and computes their average sales. The input is a data stream of records containing *business date*, *branch* and *sales amount* and the output is a data stream of *business period*, *branch*, and *average sales amount*. The point here is that the scratchpad contains the execution context (i.e., the partial sum and period length). Suppose *branch* is the ordering attribute of the input stream. The UDF then needs only one pair of registers for storing the partial sum and the length of the business period (i.e., number of days). As soon as an input record of a new branch is read from the data stream, the UDF can output the result for the previous branch. On the other hand, if the data stream is sorted on business date, the UDF¹ may have to keep a pair of registers for each branch. That is, more context information is required. Contrary to this example, for a UDF that outputs a list of all business dates and computes the average sales of all branches on each date, if *business date* is the ordering attribute of the input data stream, less context information is required. Therefore, to best support data stream processing, UDF developers should provide information about how a set of data

¹Here we are taking the view that a UDF is implemented such that that it can handle (internally) different input orderings. A UDF can be registered in different versions which are appropriate for different orderings.

records should be ordered such that the optimizer can properly organize a data stream.

To define a record ordering, a list of ordering attributes and respective directions (i.e., ASC (ascending), DESC (descending) and ADT orderings) need to be specified. Record orderings are associated with both stored database tables and intermediate results in a query evaluation. As for stored database tables, indexes with respect to record orderings are built statically such that the optimizer can have additional data access path options. On the other hand, intermediate results are ordered as a stream at run-time. Record orderings are registered as shown in Figure 5. We note

```
CREATE RECORD ORDERING <record ordering name>
( <ordering-attribute-name> <ordering-attribute-type>,
  [[<ordering-attribute-name> <ordering-attribute-type>], ... ] )
ORDER BY <ordering-attribute-name> { ASC | DESC | <ADT ordering name > }
[...]
```

Figure 5. Registration of Record Ordering

that only ordering attributes are specified in a record ordering. Data record sets that share the same ordering attributes are able to be associated with the same record ordering definition. An example is:

```
CREATE RECORD ORDERING minOrder AS (dd DATE)
ORDER BY dd ASC;
```

To optimize a query execution, built-in sort operators are often inserted into the query plan to order intermediate results of an operator to satisfy another operator’s input requirements [13]. We require explicit specifications of input and output orderings in a UDF registration (Figure 6), which is similar to other extensible database systems (e.g., *Volcano*[3][4]). The

```
CREATE FUNCTION <function-name>
( <argument type list > [ WITH ORDERING <record-ordering list > ]
RETRUNS <data type > [ ASC|DESC|ADT-ordering|record-ordering ]
EXTERNAL NAME <external function name>
...
```

Figure 6. Extension of UDF registration

WITH ORDERING clause specifies the input record ordering requirements. ASC, DESC, ADT ordering or record ordering can be used to indicate the output ordering. The built-in sort operator is extended such that not only the system built-in orderings (usually ASC and DESC) but also user-defined orderings, which are not currently supported in any ORDBMS, can be processed. Furthermore, the definition of a record ordering is decoupled from data record schema so that more than one record ordering can be associated with

the same set of data records hence satisfying different UDF input requirements.

3.3 Windows

Definition 3.2 Given a data stream ds of type T , a substream of size l on ds is a function $ss : integer \rightarrow T$ such that there exists some integer $k \geq 0$ and for all $1 \leq i \leq l$, $ss[i] = ds[k+i]$. ss can be represented as $ds[u;v]$ such that $ss[1] = ds[u]$ and $ss[l] = ds[v]$. The size of a substream is at least 1, i.e., $sizeOf(ss) = v - u + 1 \geq 1$. The set of all substreams of data stream ds of type T is denoted as $sub(T)$. \square

Informally, a *window* (with respect to the semantics of a UDF) is an input substream of bounded size such that the UDF can use this substream to produce one or more output records exclusively. Windows of a data stream may overlap. Window size is often much less than the size of a whole input stream. We are only interested in those *minimal* windows that contains no window with respect to the same set of output records. In the remainder of this paper, windows refer to minimal windows, unless otherwise noted.

Definition 3.3 Let F be a UDF such that $F(ds_{in}) = ds_{out}$ where ds_{in} is the input data stream of type T_{in} and ds_{out} is the output data stream of type T_{out} . A *window* with respect to F is a substream $ds_{in}[u;v]$ of size bounded by a constant $C > 0$, i.e., $v - u + 1 \leq C$, such that $F(ds_{in}[u;v]) = ds_{out}[x;y]$ where $1 \leq x \leq y \leq sizeOf(ds_{out})$. Further, $ds_{in}[u;v]$ is minimal iff $\nexists ds_{in}[u;v']$ such that $ds_{in}[u;v']$ is a substream of $ds_{in}[u;v]$ with $v' < v$ and $F(ds_{in}[u;v']) = ds_{out}[x;y]$. The set of all minimal windows is denoted as W_F . \square

Given two windows of stream ds of type T , $X = ds[u_1;v_1]$ and $Y = ds[u_2;v_2]$, the following relationships between X and Y are considered:

- (a) **equal(X, Y)** if $u_1 = u_2 \wedge v_1 = v_2$;
- (b) **meet(X, Y)** if $v_1 = u_2 - 1$;
- (c) **contain(X, Y)** if $u_1 \leq u_2 \wedge v_2 \leq v_1$;
- (d) **overlap(X, Y)** if $u_1 \leq u_2 \wedge u_2 \leq v_1 \wedge v_1 \leq v_2$;
- (e) **before(X, Y)** if $v_1 \leq u_2 - 1$;
- (f) **start(X, Y)** if $u_1 = u_2 \wedge v_1 \leq v_2$;
- (g) **end(X, Y)** if $v_1 = v_2 \wedge u_1 \leq u_2$.

Definition 3.4 Given a set of windows W_F defined as above and $w_1, w_2 \in W_F$, a linear ordering in W_F can be identified as $w_1 \ll w_2$ iff $meet(w_1, w_2)$ or $overlap(w_1, w_2)$ or $before(w_1, w_2)$. Further, a window successor function is defined as $next(w_1) = w_2$ if (1) not $equal(w_1, w_2)$; and (2) $w_1 \ll w_2$; and (3) $\forall w_3 \in W_F$ such that if $w_1 \ll w_3$, then $w_2 \ll w_3$, otherwise $equal(w_2, w_3)$. \square

3.4 Window Functions

A UDF can be specified in a manner that enables generic data parallelism: (1) *window function*, denoted as $F.window()$, is the function that identifies input windows; (2) *evaluation function*, denoted as $F.eval()$, is the function that evaluates windows and produces results; (3) *merging function*, denoted as $F.merge()$, is the function that merges results from each clone of the evaluation function (Figure 7). A window function is

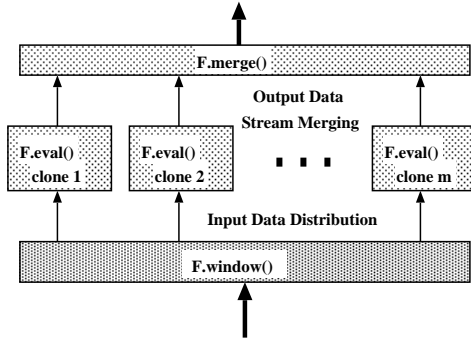


Figure 7. Data Distribution for a UDF

composed of a *window expression* and a *window directive*. A window expression defines the common properties of input records within the same window while a *window directive* specifies the relationship between successor windows. Attributes used in a window definition are called *window attributes*. In the following, we describe the primitives defining a window function. We note that this is not a complete list but sufficient to define most interesting windows.

To specify a window expression, we include the following primitives:

SIZEOF(N) Each window is of size N which is a constant.

SIZEOF(expression) Each window is of size n which is the result of the expression. An expression is constructed with built-in arithmetic and/or scalar operators or some other UDFs, and returns an integer as the result.

WHEN(logical expression) Each record in the same window should satisfy the logical expression, which is constructed with built-in and/or user-defined comparison functions, and returns a boolean as the result. The following keywords are commonly used in a logical expression: (1) *FIRST*, the first record in a window; (2) *CURRENT*, the current record to evaluate; and (3) *PREV*, the previous record in the data stream.

Let W_F be the set of windows with respect to UDF F , the following primitives can be used to define a window directive:

MEET $\forall w_1, w_2 \in W_F$ such that $next(w_1) = w_2 \Rightarrow meet(w_1, w_2)$.

OVERLAP(N) $\forall w_1, w_2 \in W_F$ such that $next(w_1) = w_2 \Rightarrow overlap(w_1, w_2) \wedge w_2[1] = w_1[m - N + 1]$ where m is the size of w_1 and N is a constant.

OVERLAP(expression) $\forall w_1, w_2 \in W_F$ such that $next(w_1) = w_2 \Rightarrow overlap(w_1, w_2) \wedge w_2[1] = w_1[m - n + 1]$ where m is the size of w_1 and n is the result of the expression. An expression is constructed with built-in arithmetic and/or scalar operators or some other UDFs, and returns an integer.

The syntax of window registration is shown in Figure 8. As an example, let us have a simple relation

```
CREATE WINDOW <window name>
([<attribute name> <datatype>, ...])
AS <window expression>
WITH <window directive>;
```

Figure 8. Registration of Window Definition

`salesTable(dd DATE, sales FLOAT)`, to compute the n days running sales average, a window can be defined as follows:

```
CREATE WINDOW running ( n INTEGER )
AS SIZEOF( n ) WITH OVERLAP( n-1 );
```

As an example of the window usage, let *runningAverage* be a registered UDF with input record ordering on *dd*, the SELECT statement that computes 30 days running sales average is:

```
SELECT runningAverage(S.sales) USING running(30)
FROM salesTable S;
```

3.5 Classification of Windows

According to user-defined window expressions and directives, a window function with respect to $ds_{out} = F(ds_{in})$ (where F is a UDF, ds_{out} and ds_{in} are output and input streams respectively) can be classified as follows:

Unit Window All windows identified by a window function in this class are of size 1. That is, the window expression is *SIZEOF(1)*. Examples of operators with such a window function include *filter* and *projection*.

Fixed Length and Disjoint Windows identified by a window function in this class are of window expression *SIZEOF(n)* and window directive *MEET*, where $n > 1$ is known at optimization time. An example is the window function for a UDF computing *30 (business) day* averages of IBM stock prices from input that consists of daily closing prices in chronological order.

Variable Length and Disjoint Windows identified by a window function in this class are of size

bounded by a constant C . The window expression includes $SIZEOF(C)$ and $WHEN(logical\ expression)$, while window directive is of $MEET$. As an example, consider a given input stream of IBM stock closing prices which are irregularly recorded in chronological order. The window function for a UDF computing *monthly* closing price averages on this input stream is of this class.

Fixed Length and Continuous All windows identified by the window functions in this class are of window expression $SIZEOF(n)$ only and window directive $OVERLAP(k)$ where both n and k are constants known at optimization time. For example, given an input that consists of daily IBM stock closing price records in chronological order, the window function of a UDF computing *30 business day* closing price running averages belongs to this class.

Variable Length and Continuous Window functions in this class identify windows bounded by a constant C and overlapped. The window expression is defined as $SIZEOF(C)$, the bound of each window size, and $WHEN(logical\ expression)$, which evaluates records for each window. The window directive is specified as either $OVERLAP(n)$ or $OVERLAP(expression)$. An example is the window function of a UDF that computes 30 day running average from an input stream that consists of irregularly measured temperature records in chronological order.

Unclassified All other window functions that can not be classified into any of the above. For example, the window function of a UDF that computes the monthly average IBM stock closing prices from an input that consists of *unordered* stock price records is in this class.

3.6 Window Recognition

Static Window Recognition For those window functions in *unit window* or *fixed length* classes, the optimizer can analyze windows on input streams at optimization time because they are not data dependent. The window expression is evaluated when the query is compiled so that each position of the input stream is associated with respective windows. As a result, the data distribution mechanism can efficiently assert whether an input record belongs to a certain window.

Dynamic Window Recognition In contrast to the static window recognition, for the window functions in *variable length* classes, the optimizer can not conclude the accurate window sizes at optimization time. In order to assert whether an input record belongs to a window or not, the window expression needs to be evaluated at run-time. The optimizer can estimate the expected sizes according to statistical information and

then choose the most appropriate distribution strategy, as we will discuss soon, to parallelize the data computation. The evaluation results of window expressions are stored for future window size estimation.

3.7 Stream Data Distribution

A number of distribution strategies for data parallelism have been proposed and implemented for relational DBMSs[18]. In the following, we discuss how *hashing*, *round-robin* and *range-partitioning* are applied in stream processing with our window-based approach. We assume that window attributes are consistent with ordering attributes (i.e., window attributes are the first k ordering attributes where k is equal to or less than the number of ordering attributes) since a better performance can be delivered [15].

Hashing and Round-robin Both hashing and round-robin strategies attempt to spread the workload more evenly among processors by distributing records randomly. To split the input data stream with a *unit* window function of a UDF, the distributor does not have additional overhead with either the hashing or round-robin strategies since all windows are of size 1. As a result, the distributor simply sends next available record to UDF clone designated.

For a UDF with *disjoint* window function (either *fixed length* or *variable length*), windows are sequential in the input stream and do not overlap. As a result, no replication overhead is incurred. If windows are *fixed length*, the distributor counts the number of records already sent to a clone. Once the number is equal to the window size, it can switch the distribution to another UDF clone. If windows are *variable length*, the distributor needs to keep more information and evaluate the window expression at run-time. The distributor can switch the destination UDF clone within a period of time for sending C records since the size of each window is bounded by a constant C . As an example, suppose we have a UDF which computes branch average sales of a company from input that consists of daily sales amount in chronological order. The distributor evaluates the given window expression, which compares the branch name of the available record with the branch name of the current window, and sends the record to the respective UDF clone.

For a UDF with *continuous* window function, an input record may belong to more than one window. Therefore, replication overhead will generally be incurred. If windows are *fixed length*, the distributor sends the same record to more than one clone and maintains a table of record counts, which indicates how many records have been sent to each UDF clone.

Once a record count is equal to the fixed window size, that means a complete window has been sent to the respective clone. If windows are *variable length*, the distributor needs to evaluate the window expression at run-time to determine whether a complete window of records has been sent to UDF clones. Sending a complete window to a clone can be done within the time of sending C records where C is the bound of window size. As an example for *variable* and *continuous* windows, suppose we have a UDF computing 30 day moving average sales of a company branch from input that consists of irregular collected sales amount in chronological order. The distributor evaluates the given window expression, which compares the date of the available record with the window sizes, and then sends the record to respective clones.

For a UDF with *unclassified* window function, a UDF can not be cloned unless a merging function is defined. If it is the case, relational distribution strategies can be applied directly. For example, a UDF implementing merge-sort algorithm can be cloned such that each UDF clone sorts a substream and the merging function merges the outputs from each clone. The distributor can send records to each clone based on records instead of windows.

Range Partitioning The range partitioning strategy can be characterized by clustering and processing records with close (or equal) attribute values at the same UDF clone. We call the attributes used for range-partitioning the partition attributes. With respect to input stream $X(I, A, B)$ where I is of integer type to indicate the position of a record in the data stream, A is the list of ordering attributes $[a_1, \dots, a_p]$, and B is the set of ordinary attributes $\{b_1, \dots, b_q\}$. Range-partitioning can be classified into three forms, according to whether the partition attributes are: (1) the first k ordering attributes; (2) some attributes except the first k ordering attributes; (3) both the first k ordering attributes and some other attributes where $(1 \leq k \leq p)$. We discuss each scheme as follows.

Case 1 When the partition attributes are composed of the first k ordering attributes, the data stream is partitioned *horizontally* (Figure 9.a). Record replication may happen at the boundaries between portions for UDFs with fixed or variable length *continuous* window functions. Suppose the input stream is divided into m portions, the number of replicated records is bounded by $(m - 1) * (C - 1)$ where C is the window size of a fixed length continuous window functions, or the bound of window size for variable length continuous window functions. For example, suppose we have an aggregate function which computes 30 business days running average sales of a branch from input that con-

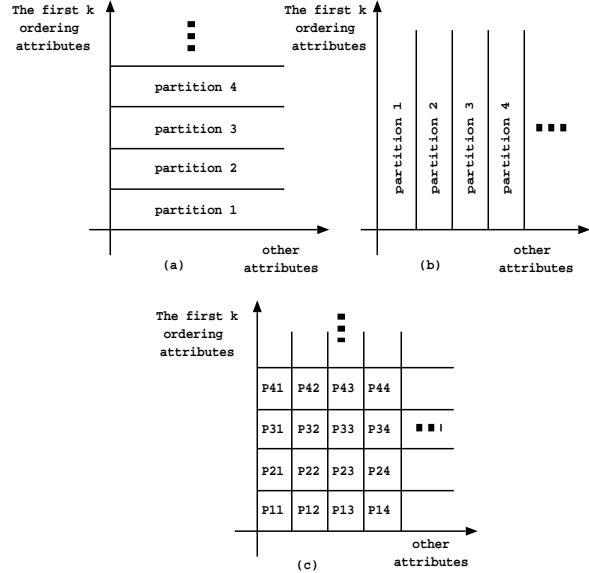


Figure 9. Forms of Range-Partitioning

sists of daily sales amount in chronological order for 360 days. The distributor divides the input stream into 4 portions according to *date*, the number of records in the boundary between two neighbor portions is 29. Therefore, the total number of replicated records is 87.

As for a UDF with *unit* window function, there is no record replication. If the edges of portions are the edges of windows for a UDF with periodic or sporadic disjoint window function, records are not replicated for partitioning.

Case 2 In the case of partition attributes which are attributes other than the first k ordering attributes (Figure 9.b), the distributor has to keep track whether or not complete windows have been sent to appropriate UDF clones for computation. For this partitioning form, once a record is distributed to a particular clone, all records in the same window should be sent to the same clone (otherwise some UDFs may not be able to complete the respective computation). As a result, the possibility of record replication is increased. For example, suppose we have an aggregate function which computes 30 business days running average sales of a branch from input that consists of daily sales amount in chronological order for 360 days. The distributor divides the input stream into 4 portions according to sales volumes. The records in the same window (30 days) may not have sales volumes in the same category, e.g., r_1 and r_2 in window w . Therefore, r_1 and r_2 are distributed to different portions. However, in order to complete the computation of window records, all records in w need to be sent to both destinations where

r_1 and r_2 were sent. In addition, duplicate output may result since w is distributed to different destinations and therefore the elimination of duplicates is required. Because of the potentially redundant and inefficient computation, this form of range-partitioning may not be practical for most UDFs (except those UDFs with *unit* window functions).

Case 3 Another possible form of range-partitioning is where the partition attributes consist of both the first k ordering attributes and some other attributes (Figure 9.c). As we have discussed above, using attributes other than the first k ordering attributes as partition attributes is possible but not practical because of potentially increasing the need for replicating input records and eliminating output duplicates. Therefore, this form of range-partitioning is not appropriate for most UDFs.

4 Implementation Issues

In practice, the overhead of additional sort operations may offset the benefits from parallelizing UDFs with proper ordered input streams. We briefly discuss some implementation solutions as follows.

Building indexes When a user-defined ordering is registered, if it is associated with a stored database table, an index providing alternative path to access records in a certain sequence can be built in advance so that no sort operation at run-time is needed.

Weakening ordering constraint When reading a data stream, a UDF may not require the input sequence to be exactly same as the specified user-defined ordering, which may be registered for more than one UDF. As we have discussed, the set of window attributes, which is UDF semantics related, should be consistent with the first k ordering attributes (where k is less than or equal to the number of ordering attributes). Since our data distribution approach is window-based, as long as the input stream is ordered according to window attributes (i.e., the first k ordering attributes) the ordering requirement is satisfied. We note that such a property may depend on how the UDF is implemented. Therefore, the registry of a UDF has an option to allow this assumption.

Pushing down window attributes The ordering property of window attributes for an operator in a QEP can be “pushed down” to combine with other sort operations or stored table index access whenever possible. Here we assume that a QEP is a tree-like structure. The bottom part of a QEP consists of operations scanning stored databases files. In general, the less (extra) sort operations, the better the performance.

5 Related Work

User-Defined Functions are widely studied in recent years ([7], [6], [5], [22], [21], [14], [10]). However, most of these efforts consider limited or non-parallel execution. For example, developers of IBM DB2 can specify ALLOW PARALLEL or DISALLOW PARALLEL for a user-defined *scalar* function[8]. Our approach allows the parallel execution for more general UDFs. both *inter-* and *intra-* parallelism are discussed in [3]. Different from their work, we identify operator characteristics with window functions such that the optimizer can derive the parallel execution systematically. [20] [11] [12] studied the intrinsic characteristics of stream processing in detail. We have extended their work to allow more general streaming data and UDFs as well as user-defined orderings. It is interesting to compare our classification of window functions to the classification of UDFs for parallelization in [10], which classifies UDFs (operators) according to the allowed partitioning methods, while we analyze window properties and classify window functions according to their patterns in input streams with the extensibility of user-defined ADT orderings. [19] introduces the concept of *limited patience patterns* to model window pattern searching. They use “BLOB” (a special column field) to store time-series data and do not consider the ordering among tuples in a table. Though [17] extended SQL to handle ordered tables, UDFs and parallelism are not considered in their work.

6 Summary and Future Work

The focus of this work has been on parallelizing general UDFs in an ORDBMS environment. We have proposed a novel optimization approach, which is based on a data streaming paradigm, to fully support parallelism as well as dynamic reconfiguration of query execution plans [16] in an ORDBMS. The major points presented in this paper can be summarized as: (1) User-defined orderings are introduced to support data stream processing; (2) Properties of window functions for UDFs have been characterized into classes such that systematic window distributions can be derived to parallelize data computation; (3) Implementation and optimization issues are enumerated for providing query services in ordered domains.

Some issues require further investigation. First, statistical execution information is important in optimizing the dynamic window expression evaluation. To estimate the bound of windows of variable length, historical information is critical. There is also a question of how this information can be obtained efficiently and

summarized in a suitable form for the optimizer. In addition, for long running queries in application domains such as data warehousing statistical analysis, fault tolerance is important as well. A possible approach is to build a checkpoint periodically during run-time. Since each output record is associated with an input window, with checkpointing at the edges of some windows, the query processing can be stopped and restarted. However, checkpoint operations at run-time increases the workload. We need to study the optimal frequency of checkpointing carefully. Further, for a query processing service, a global optimization algorithm that materializes and pipelines intermediate results of common sub-queries can speed up the overall performance.

References

- [1] D. DeWitt. Parallel object-relational database systems: Challenges and opportunities. In *Invited Talk, PDIS*, 1996.
- [2] F. Fabbrocino, E. Shek, and R. Muntz. The design and implementation of the conquest query execution environment. Technical Report CSD-970029, UCLA, June 1997.
- [3] G. Graefe and et al. Extensible Query Optimization and Parallel Execution in Volcano. In *Query Processing for Advanced Database Systems*, 1994.
- [4] G. Graefe and W.J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of ICDE*, April 1993.
- [5] L.M. Haas and et al. Starburst Mid-Flight: As the Dust Clears. *IEEE TKDE*, 2(1), 1990.
- [6] J.M. Hellerstein and J.F. Naughton. Query execution techniques for caching expensive methods. In *Proc. of ACM SIGMOD*, 1996.
- [7] J.M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. of ACM SIGMOD*, 1993.
- [8] IBM. *SQL Reference*. <http://www.software.ibm.com/data/db2/library>, 1998.
- [9] Informix. *Illustra User Manual*. <http://www.informix.com/answers>, February 1998.
- [10] M. Jaedicke and B. Mitschang. On parallel processing of aggregate and scalar functions in object-relational dbms. In *Proc. of ACM SIGMOD*, 1998.
- [11] T.Y.C. Leung. *Query Processing and Optimization in Temporal Database Systems*. PhD thesis, Computer Science Department, UCLA, 1992.
- [12] T.Y.C. Leung and R.R. Muntz. Query processing for temporal databases. Technical Report CSD-890024, UCLA, April 1989.
- [13] G.M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. of ACM SIGMOD*, June 1988.
- [14] A.P. Marathe and K. Salem. A language for manipulating arrays. In *Proc. of VLDB*, 1997.
- [15] K. Ng and R. Muntz. Optimization of user-defined functions in Object-Relational DBMS. Technical Report CSD-990023, UCLA, April 1999.
- [16] K. Ng, Z. Wang, R.R. Muntz, and E.C. Shek. On reconfiguring query execution plans in distributed object-relational dbms. In *Proc. of the International Conference on Parallel and Distributed Systems, Tainan, Taiwan*, December 1998.
- [17] W. Ng and M. Levene. OSQL: An extension to sql to manipulate ordered relational databases. In *Proc. of IDEAS*, April 1997.
- [18] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.
- [19] C.S. Perng and D.S. Parker. SQL/LPP: a Time Series Extension of SQL based on Limited Patience Patterns. Technical Report CSD-980034, Computer Science Department, UCLA, 1998.
- [20] P. Seshadri, M. Livny, and R. Ramakrishnan. Sequence query processing. In *Proc. of ACM SIGMOD*, 1994.
- [21] P. Seshadri, M. Livny, and R. Ramakrishnan. The case for enhanced abstract data types. In *Proc. of VLDB*, 1997.
- [22] P. Seshadri and M. Paskin. Predator: An OR-DBMS with Enhanced Data Types. In *Proc. of ACM SIGMOD*, 1997.
- [23] E. Shek, R.R. Muntz, M. Mesrobian, and K. Ng. Scalable exploratory data mining of distributed geoscientific data. In *Proc. of CKDDM*, 1996.
- [24] M. Stonebraker and D. Moore. *Object-Relational DBMSs*. Morgan Kaufmann Publishers, Inc, 1996.
- [25] Jeffrey D. Ullman. *Principles of Database and KnowledgeBase Systems*. Computer Science Press, 1988.