Optimization of 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

Technical Report: CSD-990023

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 [23]. Though full support of parallel ORDBMS is expected, the techniques used to parallelize relational database systems are not adequate to parallelize the operations in ORDBMSs due to the introduction of complex abstract data types (ADTs) [1]. User-defined functions (UDFs) that are associated with ADTs are usually opaque to a query optimizer and allow little optimization opportunities. On the other hand, in non-traditional database application domains such as time-series data analysis, image processing and geoscience information systems, data manipulation of ordered data objects is not rare. 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, which supports both *bushy* and *pipelined* parallelism, and develop a parallelization framework that exploits characteristics of UDFs in a ORDBMS during query optimization. With respect to ADTs, 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. Finally, we discuss related work in Section 5, and conclusions and future research are included in Section 6.

2 Modeling the Parallel Query Execution

2.1 Data Model

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

Definition 2.1 A data record r is a record of type $T = t_1 \times t_2 \times \cdots \times t_n$ $(n > 0)$ where $t_1, t_2, \cdots, t_n$ are either system basic data types or abstract data types. The schema of r is written as $< a_1 : t_1, a_2 : t_2, \cdots, a_n : t_n >$ 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. □

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 said to be a position of r in ds. □

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.

2.2 User-Defined Functions

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 assign operators to different processors, a DBMS may also clone an operator for data parallelism, where each *clone* (or *instance*) processes a portion of input data. An operator can be either *built-in* or *user-defined*. For example, DB2/SQL provides built-in operators such as $+, -, *, /$, || (string concatenation), and aggregate functions COUNT, SUM, AVG, MAX, MIN ([8]). User-Defined Functions (UDFs) can 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 from

```
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

this example, among the UDF characteristics specified as part of the registration, 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][22] 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.

We note that a regular user-defined function in current ORDBMSs (e.g., DB2) takes arguments of built-in or abstract data types and returns data objects of a certain data type. This arrangement is consistent with all built-in functions so that a user-defined function can appear anywhere a built-in function can [8]. However, it is inconvenient and not efficient for some applications. For example, in order to compute *moving average* of stock closing price on a database with record schema *Date, Closing_Price*, users may want to have a UDF which takes both attributes *Date* and *Closing_Price* of the same tuple as arguments and returns result objects with *Period* and *Moving_Avg* columns. In this research, we consider a general notion of user-defined functions that subsumes regular user-defined functions, column functions and table functions.

2.3 Characteristics

A UDF, whenever possible, is cloned into multiple copies on processors for data parallelism. During evaluation, each UDF clone reads a sub-sequence of input data stream, produces output objects, and/or modifies its execution context. The initial context of each clone is denoted as *null* to indicate the status before processing any input. Output records from UDF clones may need to be merged as final results. Some important characteristics can be pointed out here:

- 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 [16]. As a result, an extensible scheme that supports user-defined orderings is expected in an ORDBMS.

- In order to support data parallelism, data streams are split into substreams. A number of well-known splitting strategies have been proposed and implemented in relational DBMSs, e.g., round-robin, hashing and range-partitioning [17]. However, for ordered streams many of these just do not work — they do not recognize the need to maintain some degree of the original ordering. To apply these proven techniques in object-relational DBMS, a UDF developer should have some means to notify the system how the UDF semantics affects the possible stream splittings so that the optimizer can find the best way to parallelize the execution.

3 Parallelizing UDFs

In this section, we first discuss an extensible ordering scheme for stream processing, and then introduce the concept of *windows* for parallelizing UDFs execution.

3.1 Ordering

A basic assumption of the relational model is that records in a table have no explicit interrelation between them. Though some operations in relational DBMSs such as *merge-join* require input data records are in certain order, such orderings are based on the results of a *literal* comparison (e.g., lexicographical ordering, numeric ordering) instead of *semantics*. To support general user-defined operations, this assumption should be relaxed.

3.1.1 ADT Ordering

In a relational database system, 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 [24]. Current ORDBMSs support lexicographical, numeric and chronological orderings for both built-in and abstract data types. 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 have an ADT *branch* to describe a nationwide corporation. The head office in *New York* runs two regional offices at *Nashville* and *Los Angeles*, and the latter administrates two small branches at *San Diego* and *Bakersfield* respectively. With respect to the user-defined *.oversee*. (x.oversee.y if x administers y.) and the built-in $<$ relationships, we can have different

hierarchies. The optimizer may use a built-in sort operator to order instances in the *branch* domain

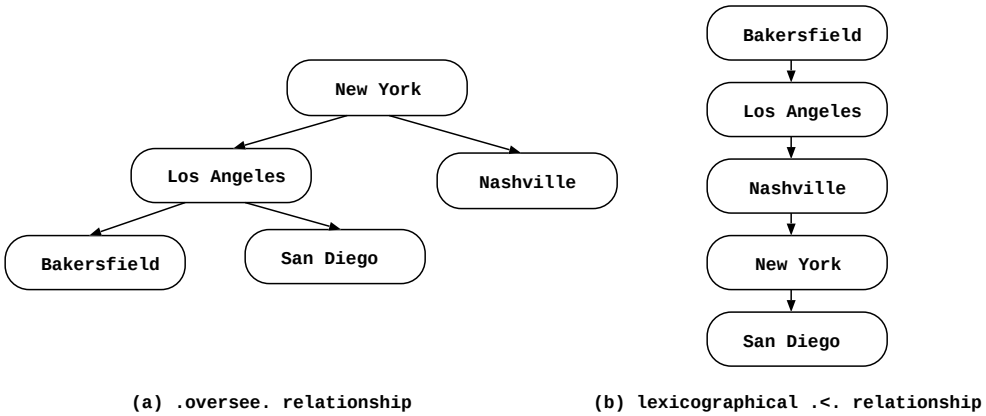

Figure 2: Orderings

with the lexicographical ordering for some algorithms (e.g., merge-join) as shown in Figure 2(b) (or in reverse order). However, the 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 user-defined function. 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 $M : T \times T \rightarrow boolean$ is a partial ADT ordering on D iff

(1) $\forall x, y \in D$, $M(x, y) \wedge M(y, x) \Longrightarrow x = y$;

(2) $\forall x, y, z \in D$, $M(x, y) \wedge M(y, z) \Longrightarrow M(x, z)$.

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 $M(y, x)$. □

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 abstract data type ordering, we propose the syntax shown in Figure 3. As an example, the *administration* ordering can be defined with the statement

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

where *oversee* is a registered user-defined function which takes two values, x and y, of the ADT

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

Figure 3: Registration of ADT Ordering

branch type, and returns a boolean value indicating whether x oversees y or not.

3.1.2 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 specified as the sorting direction of each ordering attribute. A special record ordering, random ordering, that maps each data record in the record collection to one and only one arbitrary position in a data stream is defined to be the default ordering so that data manipulations on ordered domain can be viewed as the general form of the existing set operations in query processing.

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 and computes their average

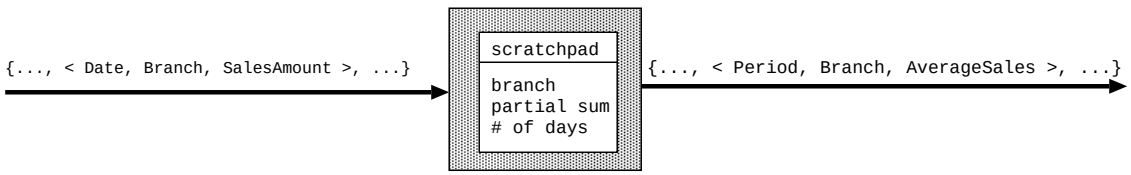

Figure 4: An UDF that computes average sales for each branch

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 UDF[1] may have to keep a pair of registers for each branch. That is, more context information

[1] Here we are taking the view that a UDF is implemented such that that it can handle (internally) different input

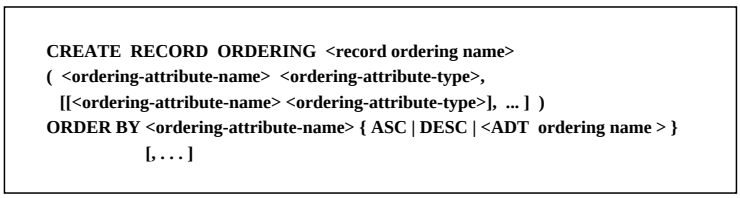

Figure 5: Registration of User-Defined Ordering

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 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. A system built-in sort operator, which may call user-defined ADT ordering functions for ADT orderings, performs record orderings to form a data stream. The optimizer can insert such a built-in operator into a query execution plan, particularly where record orderings are required for a certain UDF.

Record orderings are registered as shown in Figure 5. We note 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 the intermediate results of an operator to fulfill the input requirements of another operator. The sort LOLEPOP (LOw LEvel Plan OPerator) in IBM *Starburst* (or Starwind) project [13] is one of the examples. In an extensible database system, the input requirements and output property

orderings. A UDF can be registered in different versions which are appropriate for different orderings.

of each operator must be registered. For example, *Volcano*, an object-oriented query processing systems [3][4], has such requirements in operator registry. We use the similar approach and require explicit specifications of input and output orderings in a UDF registration (Figure 6). The *WITH*

```
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

ORDERING clause specifies the input sort 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.

3.2 Windows

Definition 3.2 Given a data stream ds of type T, a substream of size l on data stream 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 also 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., $sizeOf(ss) = u - v + 1 \geq 1$. The set of all substreams of data stream ds of type T, $\{ds[1; 1], ds[1; 2], \cdots, ds[2; 2], ds[2; 3], \cdots\}$, is denoted as $sub(T)$. □

Informally, a *window* (with respect to the semantics of a user-defined function) is an input substream of bounded size such that the UDF can use this substream to produce one or more output records. Window size is usually much less than the size of a whole input stream. Further, 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 generally refer to minimal windows, unless otherwise noted.

Definition 3.3 Let F be a user-defined function 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

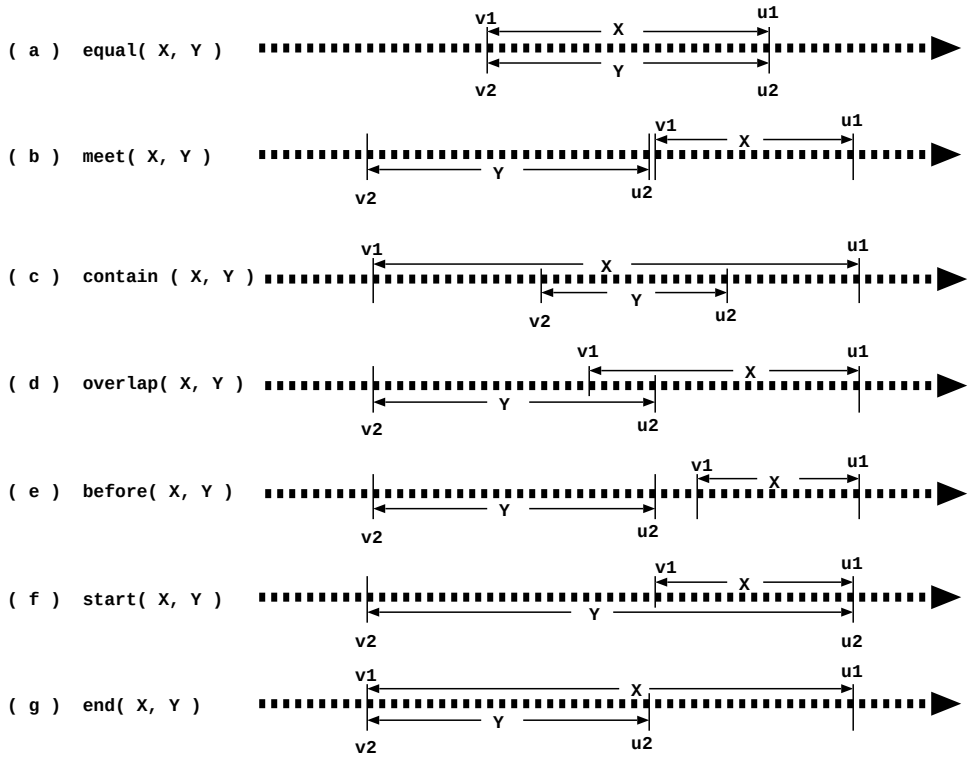

Figure 7: Relationship between two windows

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. □

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 (Figure 7):

(a) **equal**: $equal(X,Y)$ if $u_1 = u_2 \wedge v_1 = v_2$;

(b) **meet**: $meet(X,Y)$ if $v_1 = u_2 - 1$;

(c) **contain**: $contain(X,Y)$ if $u_1 \leq u_2 \wedge v_2 \leq v_1$;

(d) **overlap**: $overlap(X,Y)$ if $u_1 \leq u_2 \wedge u_2 \leq v_1 \wedge v_1 \leq v_2$;

(e) **before**: $before(X,Y)$ if $v_1 \leq u_2 - 1$;

(f) **start**: $start(X,Y)$ if $u_1 = u_2 \wedge v_1 \leq v_2$;

(g) **end**: $end(X,Y)$ 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 $next(w1) = w2$ 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). □

3.3 Window Functions

A user-defined function 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 8).

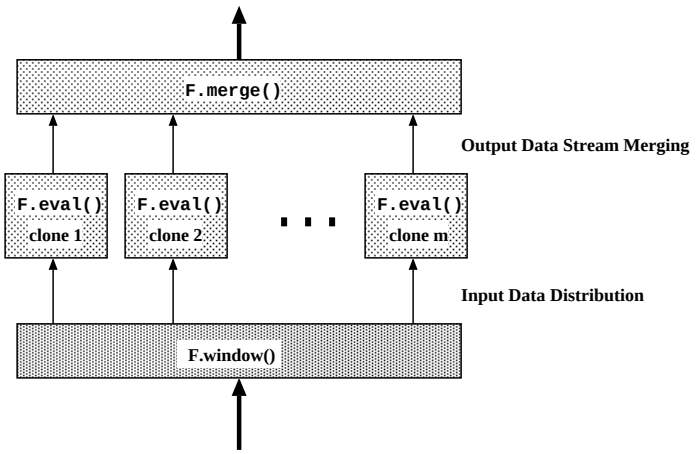

Figure 8: Data Distribution for a User-Defined Function

A window function is 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 of 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 user-defined functions, 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:

  - *FIRST*, the first record in a window;
  - *CURRENT*, the current record to evaluate;
  - *PREV*, the previous record in the data stream.

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

- **MEET**

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

- **OVERLAP(N)**

  $\forall w_1, w_2 \in W_F$ such that $next(w_1) = w_2 \Rightarrow \text{overlap}(w_1, w_2) \wedge w_2[1] = w_1[m - N + 1]$ 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 $next(w_1) = w_2 \Rightarrow \text{overlap}(w_1, w_2) \wedge w_2[1] = w_1[m - n + 1]$ 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 user-defined functions, and returns an integer as the result.

The syntax of window registration is shown in Figure 9. As an example, let us have a simple relation **salesTable(dd DATE, sales FLOAT)**, in order 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 );
```

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

Figure 9: Registration of Window Definition

The *SIZEOF(n)* specifies that the window size is bounded by a constant n and the *OVERLAP(n-1)* indicates that two neighbour windows are overlapped with $n - 1$ records.

Suppose that we have registered *runningAverage* as a user-defined function 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;
```

3.4 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 user-defined function, ds_{out} and ds_{in} are output stream and input stream respectively) can be classified as follows:

Unit Window All windows identified by a window function in this class are of size 1. That is, only *SIZEOF(n)* is defined as the window expression and argument n is known to be 1. Examples of the window functions include those for UDFs such as *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 that computes *30 (business) day* averages of IBM stock closing 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 which computes *monthly* closing price averages on this input stream is in 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 user-defined function that computes *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 classified into any of 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.

3.5 Windows and Ordering

As we have seen, window functions are related to the UDF semantics that requires certain input sort orderings. Window attributes, the attributes used in defining a window function, may consist of (1) ordering attributes only; or (2) ordinary attributes only; or (3) both ordering and ordinary attributes. For our example *average* UDF (Figure 4) with the input data stream of schema <*business date*, *branch*, *sales amount*> and output data stream of <*business period*, *branch*, *average sales amount*> the context information can be characterized as follows:

1. If the input stream is ordered by branch (e.g., lexicographical order) and the average UDF computes the average sales amount for each branch, the ordering attribute is *branch* and the window attribute is also *branch*. The scratchpad context maintains (a) partial sum of the sales amount of the current branch; (b) number of records read for the current branch.

Once a record with different branch is read, (a) is averaged with (b) and an output record is constructed. (a) and (b) is then reset for the new branch.

2. When the input stream is in order by business date and the average UDF computes the average sales amount for each branch, the ordering attribute is *business date* and the window attribute is *branch*. The scratchpad context maintains (a) partial sum of the sales amount for *each* branch; (b) number of records read for *each* branch. At the end of input stream, (a) is averaged with (b) with respect to each branch and output records are constructed.

3. If the average UDF is computing the average sales amount for every month and input stream is in the order of branch, the ordering attribute is *branch* and the window attributes are *branch* and *business date*. The scratchpad context maintains (a) partial sum of the sales amount for *each* month; (b) number of records read for *each* month. Once a record with a different branch is read, (a) is averaged with (b) with respect to each month and output records are constructed.

As illustrated in above example, if the evaluation of a window expression depends on some ordinary attributes, e.g., case (2), the UDF may have to maintain more context information than the case when the evaluation depends on only only ordering attributes, e.g., case (1). Re-ordering the input stream so 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) may deliver better performance. For example, if the input stream is re-ordered with *branch* and then *business date* as ordering attributes. The scratchpad context will need to maintain only (a) a partial sum of the sales amount for the *current* month; and (b) the number of records read for the *current* month. Once a record with a different branch or a different month is read, (a) is averaged with (b) and output record is constructed.

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 in advance. 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. In order to assert whether an input record belongs to a window or not, the window expression needs to be evaluated at run-time. Since all window sizes are bounded by a constant, the optimizer can estimate the expected sizes according to statistical information and then choose the most appropriate distribution strategy, as we will discuss sooner, 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 [17]. Since these strategies were developed for relational DBMS, it is assumed that records are not explicitly inter-related. In this section, we discuss how three typical strategies, *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 in the following discussion since a better performance can be delivered as we have discussed in previous sections.

3.7.1 Hashing and Round-robin

Both hashing and round-robin strategies attempt to spread the workload more evenly among processors by distributing records randomly. In order 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 switches 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 in 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 is 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 which computes 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 UDF clones.

For a UDF with *unclassified* window function, a UDF can not be cloned unless a merging function that finishes the computation based on the partial results from clones is defined. In this case, the traditional record distribution strategy should be applied. For example, a UDF that implements 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 sends records to each clone based on records instead of windows.

3.7.2 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 $[a_1, \cdots, a_p]$, and B is the set of ordinary attributes $\{b_1, \cdots, b_q\}$. 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

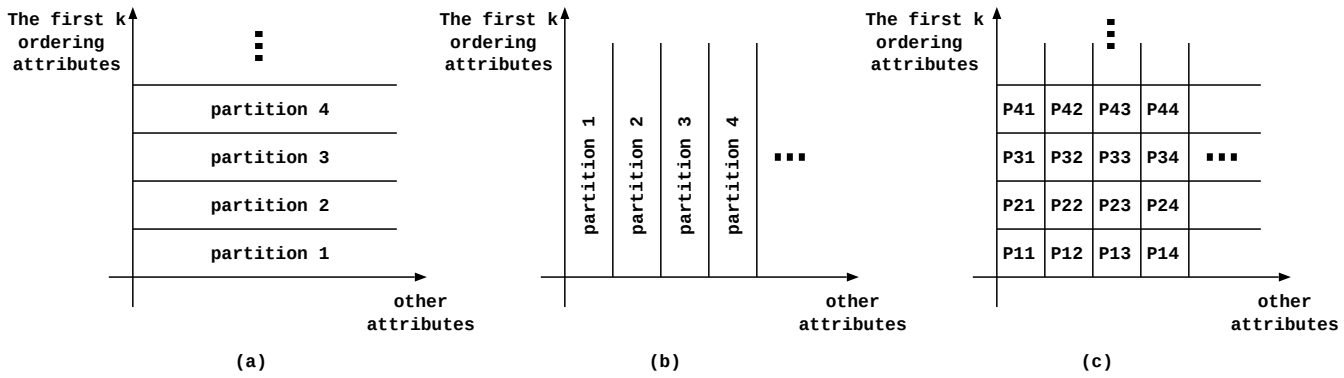

Figure 10: Forms of Range-Partitioning

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 10.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 consists 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 10.b), the distributor has to keep track whether complete windows have been sent to the 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 consequence, 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 in the merge function. 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 that the partition attributes consist of both the first k ordering attributes and some other attributes (Figure 10.c). As we have discussed in 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

The overhead of additional sort operations may offset the benefits from parallelizing UDFs with proper ordered input streams. We now briefly discuss some possible solutions.

Building indexes When user-defined orderings are registered, if such an ordering is associated with a stored database table, index providing alternative path for record access in a certain sequence can then be built in advance such that no sort operation at run-time is necessary.

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. As a result, 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], [21], [20], [14], [10]). 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]. However, most of these efforts focus their attention on *non-parallel* execution. IBM DB2 supports parallel execution of user-defined scalar functions. Developers can specify ALLOW PARALLEL or DISALLOW PARALLEL to notify the optimizer whether a user-defined scalar function can be cloned and executed on multiple processors or not [8]. Our approach allows the parallel execution for more general UDFs.

In [3], both *inter-* and *intra-* parallelism are discussed in object-oriented query processing environment. Different from their work, we identify the characteristics of operators with window functions such that the optimizer can derive the parallel execution systematically. A sequence query processing model is introduced in [19] where the *scope* of a sequence operator is defined as the positions of the input records. [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. [10] studied the execution of UDFs using data parallelism and can also be considered as an extension of [11] and [12]. 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. Different from their approach, we analyze window properties and classify window functions according to their patterns in input streams. Further, users can define their ADT orderings with respect to particular ADTs.

In [18], the concept of *limited patience patterns* is introduced to model window pattern searching.

They use "BLOB" (a special column field) to store the time-series data and do not consider the ordering among tuples in a table. In contrast to their work, we focus on the problem of ordered table (i.e., stream) and define window functions particularly suitable for data parallelism of query execution plans. [16] extended SQL to handle ordered tables, however, user-defined functions that are important part in ORDBMS environment are not considered in their work. In addition, they considered no parallelism support.

6 Summary and Future Work

The focus of this work has been on parallelizing general user-defined functions in an object-relational database 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 [15] in an ORDBMS. The major points presented in this paper can be summarized as follows.

- We have introduced user-defined orderings to support data stream processing in an ORDBMS.
- The properties of window functions for UDFs have been characterized into classes. A systematic window distribution can be derived to parallelize data computation in ordered domains.
- We have also enumerated implementation and optimization issues in providing query services in ordered domains.

There are several issues that require further investigation. First, statistical execution information is important in optimizing the dynamic window expression evaluation. To estimate the bound of windows for window functions in the classes 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 frequencies of checkpointing carefully. Further, for a query processing service, a global optimization algorithm that materializes and pipelines intermediate results of com-

mon sub-queries can speed up the overall performance. A more complicated mechanism may "slow down" a portion of the operations and wait for the intermediate results from another query.

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. Cliff Leung. *Query Processing and Optimization in Temporal Database Systems*. PhD thesis, Computer Science Department, UCLA, 1992.

[12] T.Y.Cliff Leung and Richard 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, Z. Wang, R.R. Muntz, and E.C. Shek. On reconfiguring query execution plans in distributed object-relational dbms. In *Proc. of ICPADS*, December 1998.

[16] W. Ng and M. Levene. OSQL: An Extension to SQL to Manipulate Ordered Relational Databases. In *Proc. of IDEAS*, April 1997.

[17] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.

[18] 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.

[19] P. Seshadri, M. Livny, and R. Ramakrishnan. Sequence query processing. In *Proc. of ACM SIGMOD*, 1994.

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

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

[22] E. Shek, R.R. Muntz, M. Mesrobian, and K. Ng. Scalable exploratory data mining of distributed geoscientific data. In *Proc. of CKDDM*, 1996.

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

[24] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1988.