Implementing Parameterized Range Types in an Extensible DBMS*

Jun Yang Jennifer Widom Paul Brown
Computer Science Department, Stanford University Informix Corporation
{junyang,widom}@db.stanford.edu paul.brown@informix.com

Abstract

A *parameterized type* defines a family of related types. For example, a single parameterized type `ARRAY` provides a common definition for array of `INTEGER`, array of `FLOAT`, and array of rows of a certain row type. SQL3 proposes support for user-defined parameterized types (or *UDPTs* for short), but we have yet to see any full implementation in a commercial DBMS. In this paper, we show how we implemented UDPTs `Range` and `RangeSet` using the DataBlade and Fastpath extensibility features of Informix. We also describe the application of these UDPTs in building a temporal database with TSQL2-like features. We found that implementing temporal database primitives using UDPTs required much less development effort and is more flexible than our earlier approach that used regular, non-parameterized user-defined types. We report some lessons learned in the implementation process, and propose a "wish list" of DBMS extensibility features required for full support of UDPTs. Although our implementation is based on Informix, we expect our experience and findings to be relevant to anyone implementing UDPTs in an extensible DBMS.

1 Introduction

A *parameterized type* defines a family of related types. For example, a single parameterized type `ARRAY` provides a common definition for array of `INTEGER`, array of `FLOAT`, and array of rows of a certain row type. Using parameterized types facilitates code reuse and simplifies software development, as evidenced by wide adoption of C++ templates.

Parameterized types should be familiar to DBMS users and implementors. For example, *collection type constructors* in SQL3, e.g., `SET`, `MULTISET`, and `LIST`, are parameterized types: `SET(LIST(INTEGER))` is a type whose values are sets, and the elements in these sets are lists of integers. The `NUMERIC` type in SQL is a parameterized type with two input parameters: One specifies the precision of the data type, and the other specifies the scale. Note that the input parameters of collection type constructors are types, while the input parameters of the `NUMERIC` type are values.

The SQL standard mandates built-in support for many parameterized types such as those mentioned above [3]. SQL3 also proposes full support for *user-defined* parameterized types [4]. The syntax for a user-defined parameterized type is as follows:

```
CREATE TYPE TEMPLATE <template name> ( {<template parameter declaration>}...  )
  <abstract data type body>
<template parameter declaration> ::= <parameter name> { <data type> | TYPE }
```

*This work was supported by the National Science Foundation under grant IIS-9811947 and by a research grant from Informix Software.

The keyword `TEMPLATE` indicates that the user-defined type is parameterized. The keyword `TYPE` indicates that a parameter is a data type name rather than a data type value. The `<abstract data type body>` is similar to the body of a regular data type definition.

Although all leading relational DBMS vendors support built-in parameterized types and regular user-defined types, many of them do not yet support user-defined parameterized types (or *UDPTs* for short). Informix is among the first to support a form of UDPTs, further discussed in Section 4. In this paper, we show how we implemented two parameterized types, `Range` and `RangeSet` (described in Section 2), using the *DataBlade* and *Fastpath* extensibility features of Informix (Section 4). We also describe the application of these UDPTs in building a temporal database with TSQL2-like features [5] (Section 3). As we will see, implementing temporal database primitives using our UDPTs required much less development effort and is more flexible than our earlier approach that used regular, non-parameterized user-defined types [7]. Finally, we report some lessons learned in the implementation process (Section 4), and propose a "wish list" of DBMS extensibility features required for full support of UDPTs (Section 5). Although our implementation is based on Informix, we expect our experience and findings to be relevant to anyone implementing UDPTs.

2 Description of Parameterized Range Types

Consider any type `T` supporting a `compare()` function that defines a total ordering on the values of type `T`.[1] Our extension supports parameterized type `Range(T)`, which defines ranges, or intervals, whose begin and end points are values of type `T`. We call `T` the *base type* of `Range`. Here are some important features of the `Range` type:

- *Text input/output functions.* These functions convert `Range` values to and from their external string representations, using the text input/output functions of the base type to convert range endpoints. For example, the following strings represent valid `Range(DATE)` values: `'[2000-10-01,2000-11-24)'` means "from October 1, 2000 to November 24, 2000, including October 1 but excluding November 24." `'[1974-04-10,+inf)'` means "from April 10, 1974 to forever." `'()'` indicates an empty range.
- *Infinity.* We do not assume that the base type supports the notions of positive infinity (denoted by "`+inf`") and negative infinity (denoted by "`-inf`"). They are implemented by the `Range` type itself.
- *Closed and open endpoints.* A range endpoint is either closed (denoted by a square bracket) or open (denoted by a parenthesis). For a *discrete* base type, i.e., one that supplies a `successor()` function, it suffices to support one type of endpoint since the other type can be inferred. For example, assuming that `DATE` has a granularity of one day, we can rewrite the half-open range `'[2000-10-01, 2000-11-24)'` as the fully closed range `'[2000-10-01,2000-11-23]'`. However, to handle the general case where the base type may be non-discrete, we support both closed and open endpoints. For example, in the case of `Range(FLOAT)`, there is no equivalent of `'[10.01,11.24)'` that is closed at both endpoints.
- *Binary send/receive functions.* These functions transfer binary representations of `Range` values between different platforms across the network. Like text input/output functions, binary send/receive functions use their counterparts in the base type to handle the range endpoints.

[1] More precisely, `compare(t1, t2)` returns 1 if `t1` is greater than `t2`, 0 if `t1` is equal to `t2`, or -1 if `t1` is less than `t2`.

Case	relation(X,Y)	relation(Y,X)
X: empty Y: empty	EMPTY_EMPTY	EMPTY_EMPTY
X: empty Y: not empty	EMPTY_NOTEMPTY	NOTEMPTY_EMPTY
X: xxx Y: yyyyyy	BEFORE	AFTER
X: xxx Y: yyyyyy	MEETS	MET_BY
X: xxx Y: yyyyyy	OVERLAPS_BEFORE	OVERLAPS_AFTER
X: xxx Y: yyyyyy	STARTS	STARTED_BY
X: xxx Y: yyyyyy	CONTAINED_IN	CONTAINS
X: xxx Y: yyyyyy	FINISHES	FINISHED_BY
X: xxxxxx Y: yyyyyy	EQUALS	EQUALS

Figure 1: Relationships between two Ranges.

- *Relationships between Ranges.* We provide a function `relation()` that determines the relationship between two ranges. There are a total of sixteen cases, illustrated in Figure 1. These cases are more detailed than *Allen's operators for intervals* [1] and also handle empty ranges. For convenience, we also provide other functions that can be defined in terms of `relation()`. For example, `overlap(X,Y)` returns true if and only if there exists a value of the base type within both ranges `X` and `Y`; `contains(X,Y)` returns true if and only if every value of the base type within range `X` is also within range `Y`.

  Function `relation()` uses the `compare()` function provided by the base type to compare range endpoints. In addition, if the base type provides a `successor()` function, it is used to determine `MEETS` and `MET_BY` relationships. For example, if the granularity of `DATE` is one day, then `relation('(-inf,1974-04-09]', '[1974-04-10,+inf)')` will return `MEETS`, because `successor('1974-04-09')='1974-04-10'`. However, since `FLOAT` does not provide a `successor()` function, `relation('(-inf,4.09]','[4.1,+inf)')` returns `BEFORE`.

- *Operations on Ranges.* Two examples are `union(X,Y)`, which returns the smallest possible range containing both `X` and `Y`, and `inter(X,Y)`, which returns the largest possible range contained in both `X` and `Y`. Operations may use base type functions `compare()` and `successor()` either directly or indirectly through `relation()`.

- *R-tree indexing.* We support R-tree index on any table column whose type is `Range(T)`, provided that base type `T` supplies a `distance()` function that computes the distance between two values of type `T`. This `distance()` function is used by R-tree support routines to evaluate different ways of grouping ranges together by comparing the sizes of overall bounding ranges.

The second parameterized type we implemented, `RangeSet(T)`, defines sets of maximal, non-overlapping ranges of type `Range(T)`. `RangeSet` has the same requirements on base type `T` as `Range`, i.e., `T`

provides text input/output, binary send/receive, `compare()`, and optionally, `successor()` and `distance()`. Here are some features of the `RangeSet` parameterized type:

- *Text input/output and binary send/receive functions.* These functions are analogous to their `Range` counterparts. We give several `RangeSet(DATE)` strings as examples to illustrate the overall concept of `RangeSet`: `'{[2000-10-01,2000-11-24],[2001-01-29,2001-03-02]}'` means "from October 1, 2000 to November 24, 2000, and from January 29, 2001 to March 2, 2001." `'{}'` indicates an empty `RangeSet`. The text input function automatically *normalizes* `RangeSet` values by merging ranges that overlap or meet each other. For example, `'{[1974-04-10,1975-05-12), [1975-05-12,+inf)}'` becomes `'{[1974-04-10,+inf)}'`, i.e., "from April 10, 1974 to forever."
- *Relationships between and operations on `RangeSets`.* We support boolean relationship predicates `overlap()`, `contains()`, `equal()`, etc., with the expected semantics. We also support common operations on `RangeSets` such as `union()` (normalized union of two `RangeSets`), `inter()` (intersection), and `diff()` (difference). Functions `union()` and `inter()` are overloaded to work with both `RangeSet` and `Range`. However, one subtlety is that `RangeSet union()` and `Range union()` have different semantics when inputs are disjoint:

  ```
  union('{[2000-10-01,2000-11-24]}','{[2001-01-29,2001-03-02]}')
      = '{[2000-10-01,2000-11-24],[2001-01-29,2001-03-02]}'
  union('[2000-10-01,2000-11-24]','[2001-01-29,2001-03-02]')
      = '[2000-10-01,2001-03-02]'
  ```

  This difference stems from the fact that `RangeSet` is closed under union, but `Range` is not. (Recall that we defined `union(Range,Range)` to be the smallest possible `Range` containing both inputs.)
- *Aggregates over `RangeSets`.* An aggregate function for `RangeSet` takes a collection of `RangeSets` as input and computes a single `RangeSet` as output. For example, `group_union()` and `group_inter()` compute the union and the intersection of a collection of `RangeSets`, respectively.
- *Casts to and from `Range`.* A cast from `RangeSet` to `Range` converts a `RangeSet` to the smallest possible `Range` that contains all ranges in the `RangeSet`. A cast from `Range` to `RangeSet` converts a `Range` to a `RangeSet` containing the `Range`, or an empty `RangeSet` if the `Range` is empty.

3 Application in Temporal Databases

In previous work we implemented temporal database support as a "hard-coded" extension to an extensible relational DBMS [7]. Specifically, we identified five temporal data types and implemented them in Informix as regular user-defined types. These types are:

- *`Chronon`*: a specific point in time.
- *`Span`*: a duration of time between two `Chronons`, either positive or negative.
- *`Instant`*: either a `Chronon` or a `NOW`-relative time whose interpretation depends on the current time.
- *`Period`*: a pair of `Instants`, one marking the start of the period and the other marking the end.
- *`Element`*: a set of maximal, non-overlapping `Periods`.

In the prototype implementation [7], all five data types have a built-in granularity of one second. Obviously, a temporal database implementation should provide more options. For example, employment history only needs a day granularity, but the timing of a nuclear fission needs a nanosecond granularity. Another

limitation of the prototype is that Periods (and hence Elements) are always made of Instants. Although Instants are required for the most general case, in many practical cases timestamps will never be NOW-relative. In these cases, we should be able to use Periods and Elements made of Chronons in order to avoid the unnecessary overhead of NOW-relative Instants.

With our implementation based on regular user-defined types, extending the prototype to handle the problem mentioned in the previous paragraph gets tedious very quickly: If we want Instants with a different granularity, or Periods and Elements made up of Chronons instead of Instants, we must implement separate Period and Element types, even though the code is nearly identical to the code for the default Period and Element.

With parameterized range types, the situation is much better. The default Period is simply Range(Instant), and the default Element is RangeSet(Instant). Instant can be implemented as a parameterized type, whose base type is any Chronon-like type with arbitrary granularity. Starting from SQL type DATE, we can immediately instantiate Instant(DATE), Range(DATE), Range(Instant(DATE)), RangeSet(DATE), and RangeSet(Instant(DATE)), without writing any extra code. To handle a different time granularity, we write one base type and automatically get five more derived types. We can also write functions to convert between temporal types of different granularities (e.g., RangeSet(DATE) and RangeSet(DATETIME)) or between one temporal type that is NOW-relative and one that is not (e.g., Range(Instant(DATE)) and Range(DATE)). Overall, development and extensions are greatly simplified.

As a simple example of using Range and RangeSet to implement a temporal database, consider a Circulation table that keeps track of book circulation for a library:

```
CREATE TABLE Circulation
(book CHAR(20), borrower CHAR(20), due_date DATE, period Range(Instant(DATE)));
```

We store the due date of a book as a DATE, and the actual loan period as a Range(Instant(DATE)). We first create an index on the period attribute:

```
CREATE INDEX idxCirculation ON Circulation(period);
```

Now we insert a tuple specifying that Jack borrowed *The Shining* on November 24, and the book will be due on December 24:

```
INSERT INTO Lending VALUES
('The Shining','Jack','2000-12-24','[2000-11-24,NOW]');
```

Next let us ask a query that is a temporal self-join: Who had possession of two books at the same time, and both of them eventually became overdue? Return the book information and the period in which they were both checked out:

```
SELECT c1.*,c2.*,inter(c1.period,c2.period)
FROM Circulation c1, Circulation c2
WHERE c1.book <> c2.book AND c1.borrower = c2.borrower
AND overlap(c1.period, c2.period)
AND CAST c1.due_date AS Instant(DATE) <= end(c1.period)
AND CAST c2.due_date AS Instant(DATE) <= end(c2.period);
```

Finally let us ask a query that does temporal *coalescing* [2]: when was each book checked out?

```
SELECT book, group_union(CAST period AS RangeSet(Instant(DATE)))
FROM Circulation GROUP BY book;
```

As we can see from this example, with the features provided by parameterized types Range and RangeSet, it is easy to create indexes on temporal columns and write fairly complicated temporal queries.

4 Implementation of Parameterized Range Types

We implemented the parameterized range types described in Section 2 as a DataBlade for Informix Dynamic Server 2000. A DataBlade module implements a set of related data types and their support routines. Once installed in a database server, features of the DataBlade become an integral part of the server. In the following, we describe in more detail the Informix features that we used in implementing `Range` and `RangeSet`.

- *Constructor types.* `Range` and `RangeSet` are created as *constructor types* in Informix. With a constructor type `CT`, we can instantiate a type `CT(BT)` for any base type `BT`, including a user-defined data type. Therefore, a constructor type is essentially a parameterized type with one argument that specifies a data type. Informix does not yet support the most general form of UDPTs as proposed by SQL3, but constructor types probably cover many common uses of parameterized types.[2]
  Another approach to implementing `RangeSet` would be to define `RangeSet(T)` as `SET(Range(T))` using the SQL3 collection type `SET`. However, `SET` does not automatically enforce that ranges are maximal and non-overlapping, and the built-in `SET` relationships and operations do not have same semantics as those for `RangeSet`. Therefore, we decided to implement `RangeSet` as a separate constructor type.
- *User-defined functions.* Informix allows user-defined functions to be written in C and compiled into dynamically loadable executables. The `CREATE FUNCTION` statement specifies the signature of the function and its location in the executable. Parameterization can be achieved by declaring function input and output types to be constructor types with no base types. For example, the signature of the range overlap function is "`bool overlap(Range,Range)`." This function handles all instantiated `Range` types. However, Informix does not yet support function templates. In other words, we cannot declare the function that extracts the right endpoint of a range as "`T end(Range(T))`," where `T` is a template parameter. This lack of template support makes it difficult to declare generic constructor and accessor functions. Fortunately, Informix supports function polymorphism, so we can explicitly declare each instantiation, e.g., "`DATETIME end(Range(DATETIME))`," "`DATE end(Range(DATE))`," etc. All declarations can still share a common implementation in C.
- *Fastpath.* As we have seen in Section 2, functions on parameterized types frequently need to call functions on the base types. Function `relation(Range,Range)`, for example, needs to call `compare()` and possibly `successor()` on the base type of its input arguments. If `compare()` does not exist for the base type, an error should be generated (when `relation()` is passed a `Range(BLOB)` argument, for example). On the other hand, if `successor()` does not exist, `relation()` simply knows that it should not return `MEETS` or `MET_BY`. *Fastpath* is an interface that allows a user-defined function to resolve, locate, and execute functions that may reside in other dynamic libraries. The ability to execute other functions is essential for generic functions, such as `relation()`, which work with different base types. The ability to perform function resolution is also necessary, because generic functions do not know what the actual base types are until run-time.
- *The `MI_FPARAM` structure.* A generic function can find out what the actual base types are at runtime through a *`MI_FPARAM`* structure, which is passed in automatically by the Informix server as an additional input argument. The `MI_FPARAM` structure contains detailed information about the inputs

[2] At the time of this writing, constructor types in Informix remain an undocumented feature, although Informix products, such as the TimeSeries DataBlade, reply on this feature extensively.

and output of the function, such as type identifiers (from which we can determine the base types), length, precision and scale (if applicable), and whether an input is `NULL`. The `MI_FPARAM` structure also provides a way to store state information between invocations of the function for the duration of a single SQL statement. For example, once we use Fastpath to look up certain functions on a base type, we can then store the function handles as part of the state. Thus, base type function resolution only happens once when the function is first invoked. This optimization is crucial for performance because function resolution is a fairly expensive operation.

- *Other features.* Informix supports user-defined casts and aggregates. It also provides built-in R-tree support for any type that implements a set of R-tree support functions (`union()`, `inter()`, `size()`) and strategy functions (`overlap()`, `equal()`, `contains()`, `within()`). All these features work well with constructor types.

5 Conclusion, Wish List, and Future Work

In conclusion, Informix already provides core support for UDPTs, and we used this support to implement two parameterized types, `Range` and `RangeSet`. Through our experience of using these parameterized range types to implement temporal database primitives, we found parameterized types to be extremely effective in reducing development and maintenance effort. We also found that the development process for UDPTs closely resembles that of non-parameterized user-defined types, from a DataBlade developer's perspective. We did not have to learn any new interface specific to parameterized types; rather, existing features find new uses in supporting parameterized types. For example, when first introduced in Postgres [6], Fastpath was used primarily to improve performance since it allows calls directly to the database backend, bypassing parsing, validation, and optimization. Now, we find Fastpath to be essential for generic functions. The `MI_FPARAM` structure allows a function to store state information, which turns out to be crucial in reducing overhead of run-time function resolution.

Nevertheless, more features are needed in order to support all features of SQL3-style UDPTs:

- *More general parameterized types.* Constructor types take only a single parameter that refers to a type. It would be desirable to have parameterized types with multiple template parameters, and to allow parameters to refer to either a type or a value, as discussed early in Section 1.
- *Template functions.* Currently, we can declare a generic function such as `overlap(Range,Range)`, but in some ways it is too relaxed: It requires run-time type checking in order to catch the case when we pass in one `Range(INT)` and one `Range(DATE)` as arguments. For compile-time checking, we must explicitly declare `overlap(Range(INT),Range(INT))`, `overlap(Range(DATE), Range(DATE))`, etc., for all possible base types. What we really need is a template function `overlap( Range(T),Range(T))`, where `T` is a template parameter. Constructor and accessor functions are also best declared using templates, as discussed in Section 4.
- *More compile-time analyses.* Related to the previous bullet, template functions would enable compile-time type checking. For example, it would be possible to reject a call to `overlap(Range(INT), Range(DATE))` as an error at query compilation time instead of run-time. Furthermore, it would be desirable to do compile-time checking on the availability of certain base type functions required by a template function. The list of required base type functions could be specified as a part of the template function declaration. Then, the template function would not need to do run-time function

resolution. For example, `overlap(Range(BLOB),Range(BLOB))` would be caught as a compile-time error rather than a run-time error, because `BLOB` `compare()` is not available.

One issue that we have not yet addressed in our work is support for query optimization involving parameterized types and template functions. A template function does not have a constant cost, because it frequently calls functions on base types, whose costs may vary greatly for different base types. Selectivity of a boolean template function may also depend on selectivities of certain functions on base types. The statistics collection routine of a parameterized type may need to call the statistics collection routines for base types. Informix already provides an infrastructure that allows users to provide cost and selectivity estimation routines for user-defined functions, and statistics collection routines for user-defined types. As future work, we plan to implement these routines for `Range` and `RangeSet`, and investigate whether the infrastructure provides adequate support for query optimization involving parameterized types and template functions.

Acknowledgements

We are grateful to Michelle Cheng, Peter Mork, and Huacheng C. Ying from Stanford University, for their help in implementing the temporal DataBlade and its graphical query interface; and to Kevin Brown from Informix, for his help in using constructed types.

References

[1] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[2] M. H. Böhlen, R. T. Snodgrass, and M. D. Soo. Coalescing in temporal databases. In *Proc. of the 1996 Intl. Conf. on Very Large Data Bases*, pages 180–191, Bombay, India, September 1996.

[3] C. J. Date and H. Darwen. *A User's Guide to the Standard Database Language SQL*. Addison-Wesley, Reading, Massachusetts, 1997.

[4] ISO. Working draft, database language SQL — part 2: Foundation, December 1998.

[5] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, Boston, Massachusetts, 1995.

[6] M. Stonebreaker and L. A. Rowe. The design of Postgres. In *Proc. of the 1986 ACM SIGMOD Intl. Conf. on Management of Data*, pages 340–355, Washington, D.C., May 1986.

[7] J. Yang, H. C. Ying, and J. Widom. TIP: A temporal extension to Informix. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, page 596, Dallas, Texas, May 2000.