TOOBIS

TOQL Specification

Deliverable T33TR.1

UNIVERSITY OF ATHENS
NATIONAL TECHNICAL UNIVERSITY OF ATHENS
01-PLIROFORIKI S.A.
O2 Technology

December 1996

Table of contents

Table of contents ____ *i*

Table of figures ____ *v*

1. Specification ____ *6*

1.1 Introduction/Design Objectives ____ **6**

1.2 Data types for Time Representation ____ **7**

1.2.1 Literals ____ 7

A. Instants ____ 7

B. Intervals ____ 9

C. Periods ____ 9

D. Period sets ____ 11

1.2.2 Functions ____ 13

1.2.3 Predicates ____ 18

1.2.4 Operators ____ 20

1.2.5 Automatic type and granularity conversions ____ 23

A. Type conversions ____ 23

B. Granularity conversions ____ 24

1.3 A Sample Database ____ **25**

1.4 Queries on Temporal Data ____ **26**

1.4.1 Expressions for valid state objects not allowing overlapping timestamps and valid event objects ____ 30

1.4.2 Expressions for valid state objects that allow overlapping timestamps ____ 31

1.4.3 Expressions for transaction time objects ____ 32

1.4.4 Expressions for bitemporal objects not allowing overlapping valid timestamps and bitemporal event objects ____ 33

1.4.5 Expressions for bitemporal objects that allow overlapping valid timestamps ____ 35

1.4.6 Examples ____ 37

1.4.7 The type of the result of the subscript operators ____ 38

1.4.8 Support for evolved and deleted values ____ 39

1.4.9 Extracting object states ____ 40

1.4.10 Semantic ambiguity resolution ____ 41

1.5 Referencing Object Variants ____ **42**

1.6 Conversion Between Temporal and Snapshot Values ____ **44**

A. The *snapshot* modifier ____ 44

B. The *valid* modifier 44
C. The *transaction* modifier 46
D. The *bitemporal* modifier 46

1.7 Temporal Joins 47
1.7.1 The type of the result of the temporal join 48
1.7.2 The value of the result of a temporal join 49
A. Temporal join between two valid state objects not allowing overlapping 50
B. Temporal join between two valid state objects allowing overlapping 50
C. Temporal join between two valid event objects 50
D. Temporal join between a valid state object not allowing overlapping and a valid state object allowing overlapping 51
E. Temporal join between a valid state object not allowing overlapping and a valid event object 51
1.7.3 Temporal joins involving more than two objects 51

1.8 Restructuring Operators 52
1.8.1 Converting period timestamping variants to instant timestamping 52
1.8.2 Producing maximal timestamps 54

1.9 Aggregation 57
1.9.1 Partitioning a single temporal object into variant subsets 58
1.9.2 Combining variants from multiple temporal objects 63

1.10 TOQL and User Requirements 65

2. Design 68

2.1 Introduction 68

2.2 Architecture for the TOQL Processor 68

2.3 Query Processing 72
2.3.1 Syntactical transformations 72
A. Literals 72
B. Functions 73
C. Predicates 74
D. Operators 74
E. Type conversions 75
F. Retrieval of temporal instance properties 76
G. Extracting object states 78
H. Referencing object variants 79
I. Conversion between temporal and snapshot values 80
J. Temporal joins 81

K. Restructuring operators 82
L. Aggregation 83
2.3.2 Auxiliary methods 86
2.3.3 Implementation of TOQL on top of an OQL v. 1.1 compliant OODBMS 87
2.4 Using TOQL in C++ 88
3. Conclusions 90
Appendix A: TOQL BNF syntax 91
1. Axiom 91
2. Basic 91
3. Simple Expression 92
4. Comparison 92
5. Boolean expression 92
6. Constructor 93
7. Accessor 93
8. History accessor 94
9. Timestamp Accessor 94
10. Collection expression 94
11. Select Expression 95
11. Set expression 95
12. Conversion 95
13. Temporal Modifiers 96
14. Temporal join 96
15. Restructuring operator 96
Appendix B: Language completeness with respect to the user requirements 97
1. Delta Application 97
Query A 97
Query B 98
Query C 99
Query D 99
Query E 99

Query F ____ 100
Query G ____ 100
Query H ____ 100

2. GlaxoWellcome Application ____ 101

Query A: Belonging, existence, of one event to a period ____ 101
Query B: Comparisons of dates, translation of dates by intervals ____ 101
Query C: Comparisons of periods ____ 101
Query D: Calculation of interval ____ 102
Query E: Comparisons of values on a period, or at different dates ____ 102
Query F: Average of a value on a period, or on an interval ____ 102
Query G: Search for some periods or intervals that satisfy a criterion ____ 103
Query H: Manipulation of relative time ____ 103

Appendix C: Snapshot application compatibility ____ 104

1. Accessing Temporal Properties ____ 104

2. Accessing Objects with Temporal Properties ____ 105

3. A Compatibility Example ____ 106

Appendix D: Temporal Selection ____ 110

The *when* Clause vs. Subscripting ____ 110

Appendix F: References ____ 113

Table of figures

Figure 1 - Specifying literals of type Instant 8
Figure 2 - Specifying literals of type Interval 9
Figure 3 - Specifying literals of type Period 11
Figure 4 - Specifying literals of type Period Set 12
Figure 5 - Operations on intervals 20
Figure 6 - Operations on instants 21
Figure 7 - Operations on periods 22
Figure 8 - Operations on period sets 23
Figure 9 - Expressions for valid state objects not allowing overlapping and valid event objects 30
Figure 10 - Expressions for valid state objects that allow overlapping timestamps 32
Figure 11 - Expressions for transaction time objects 33
Figure 12 - Expressions on bitemporal objects not allowing overlapping valid timestamps and bitemporal event objects 34
Figure 13 - Expressions on bitemporal objects that allow overlapping valid timestamps 36
Figure 14 - Type of the result of the tstruct operator 48
Figure 15 - Equivalent representations of temporal information 55
Figure 16 - TOQL capabilities against user requirements 67
Figure 17 - TOQL processor architecture 69
Figure 18 - The TOQL processor modules 69
Figure 19 - Overhead introduced for lexical analysis, syntax analysis and rewriting 71
Figure 20 - Percentage of the introduced overhead to overall query execution time 71
Figure 21 - Transformations for literal support 73
Figure 22 - Transformations for function support 73
Figure 23 - Transformations for predicate support 74
Figure 24 - Transformations for standard comparison operators support 74
Figure 25 - Transformations for operator support 75
Figure 26 - Transformations for implicit conversion support 76
Figure 27 - Transformations used for supporting functions count, first and last for temporal objects 76
Figure 28 - Transformations for subscript support 77
Figure 29 - Transformations for supporting functions valid and transaction on the results of subscript operators 78
Figure 30 - Transformations for supporting functions valid and transaction on the results of subscript operators 78
Figure 31 - Transformations for object state extraction 79
Figure 32 - A sample snapshot database schema 106
Figure 33 - Sample snapshot database contents 106
Figure 34 - Database contents after the introduction of temporal semantics 108

1. Specification

1.1 Introduction/Design Objectives

In this section we give the functional specification of TOQL. The new features are given in bold, to make reading easier. In the language extension which is presented below, the following goals were adopted:

1. *TOQL is an upwards compatible extension of OQL.*

   Every statement that is valid in OQL v. 1.2 ([ODMG95]) should be valid in TOQL, so as to allow applications that do not incorporate temporal semantics to function on top of the extended DBMS. TOQL also provides extensions for the management of temporal data. These extensions must adhere to the overall syntax of OQL.

2. *Temporal and non-temporal data should be treated uniformly*.

   Temporal data[1] should not present different behaviour than non-temporal data. They should appear like "normal" data, equipped with some special functionality, accessible via methods and language constructs.

3. *TOQL should provide complete temporal functionality*.

   The extended language must provide adequate functionality to support the needs of the applications. Timestamps should be accessible and temporal operations (e.g. temporal selection and temporal join) must be provided.

4. *Temporal query syntax and semantics should be clear.*

   Despite the fact that temporal queries have an inherent degree of complexity, query syntax should be kept simple and semantics must be clear.

5. *Query evaluation must preserve the updatability of the results to the maximum extent possible.*

   The default functionality of the language should be oriented towards returning updatable objects, thus the schema of query results should match the schema of the actual database objects. This is compliant to the ODMG semantics, under which no implicit schema transformations ever take place. However, the user must be able to specify and retrieve arbitrary object schemata.

[1] T*emporal data* are objects and instance properties (attribute or relationship) which have valid time semantics, transaction time semantics or both.

1.2 Data types for Time Representation

TOQL provides with facilities to manipulate the data types which are used for the representation of time, namely *INTERVAL, INSTANT, PERIOD*, and *PERIOD_SET*[2]. A TOQL query may contain literals of these types as well as functions, predicates and operators which can be applied on these data types. In the following paragraphs these features of TOQL are described.

1.2.1 Literals

A. Instants

A literal of type *INSTANT* may be specified by using the reserved word `INSTANT`, followed by a string literal containing the actual value. The string literal is, in turn, followed by a granularity specification. Finally, the granularity specification may be followed by a calendar specification.

If the calendar specification is missing, the default one is used, which is the Gregorian. The granularity specification, if present, should designate a granularity that is valid for the calendar in which the instant is expressed; if the granularity specification is missing, the default granularity for the calendar is used (in the Gregorian calendar, the default granularity is *second*).

The format of the string literal following the keyword *INSTANT* is calendar dependent. In the Gregorian calendar, the string literal of an instant specification is *YYYY-MM-DD HH:MM:SS*, where the first part (i.e. the portion before the space) represents the date and the second part represents the time. Only the *year* part is mandatory, whereas all other fields are optional. However if some field is present, all other fields between the year and the specified field must be entered (e.g. if the *day* part is specified, the *month* field is mandatory).

If the value in the string literal contains more information than is needed in the specified granularity (e.g. in the Gregorian calendar the string literal contains a month specification, whereas a granularity of year is specified for the value), the extra information is ignored; if less information is supplied, the missing parts are filled

[2] Each of these types actually hosts a *family of types*, with each member of the family having a different granularity. The following discussion applies to all members of each family. Rules for granularity conversion are given as needed.

using default values (in the Gregorian calendar, the default value for months and days is 1; for hours, minutes and seconds, the default value is 0).

The string literal may contain one of the words `BEGINNING`, `FOREVER` and `NOW`, designating the minimum, maximum and the current timestamp, respectively. In particular, the word `NOW` may be followed by an opening and a closing parenthesis (`NOW()`), in which case the computation of the value of the system clock is deferred until the actual value of the instant is needed. This feature is useful only when the instant is used as an argument to an update method, since in all other cases the instant will be bound to the value of the system clock at the query's execution time, and is thus equivalent to NOW. Figure 1 presents some examples of *Instant* literals specification.

Literal	Value
INSTANT '1990' YEAR CALENDAR Gregorian	An instant for the year 1990 of the Gregorian calendar.
INSTANT '1990-01-01' MONTH	The instant *January 1st, 1990*, in the Gregorian (default) calendar with a granularity of month. The day specification is ignored, since it is redundant for instants with a granularity of month.
INSTANT '1990-01' DAY	The instant *January 1990*, in the Gregorian (default) calendar with a granularity of day. The missing part (day specification) is filled with a default value (1), thus the result is the instant *January 1st, 1990*.
INSTANT 'NOW' MINUTE	The current date and time, expressed in a granularity of minute in the Gregorian (default) calendar.
INSTANT 'Spring 1996' SEMESTER CALENDAR Academic	An instant corresponding to the Spring semester of the academic year 1996. It is expressed in the "Academic" calendar, having a granularity of "Semester".

Figure 1 - Specifying literals of type *Instant*

B. Intervals

A literal of type *INTERVAL* may be specified by using the reserved word `INTERVAL`, followed by a string literal containing the actual value. The string literal is, in turn, followed by a granularity designation which may be followed by a calendar specification.

If the calendar specification is missing, the Gregorian calendar is used as a default. The granularity specification, if present, should designate a granularity which is valid for the calendar in which the instant is expressed (year, month, day, hour, minute or second, in the Gregorian calendar); if the granularity specification is missing, the default granularity for the specified (or default) calendar is used. Figure 2 presents some examples of *Period* literals specification.

Literal	Value
INTERVAL '10' DAY CALENDAR Gregorian	An interval of 10 Gregorian calendar days.
INTERVAL '5' YEAR	An interval of 5 years in the Gregorian (default) calendar.
INTERVAL '2' SEMESTER CALENDAR Academic	An interval of two semesters in the academic calendar.

Figure 2 - Specifying literals of type *Interval*

C. Periods

Literals of type *PERIOD* may be specified by using the reserved word `PERIOD`, followed by a string literal containing the actual value. The string literal is, in turn, followed by a granularity specification (in the Gregorian calendar, acceptable granularities are *year*, *month*, *day*, *hour*, *minute* and *second*). The granularity specification may be followed by a calendar designation.

If the calendar specification is omitted, the Gregorian calendar is used as default. The granularity specification if present, should designate a granularity which is valid for the calendar in which the instant is expressed (year, month, day, hour, minute or second, in the Gregorian calendar); if the granularity specification is missing, the default granularity for the specified (or default) calendar is used.

The string literal must have a value which starts with a left square bracket (‘`[`’), end with a right parenthesis (‘`)`’) and contains two comma-separated literals l_i and l_j of type *INSTANT*, which must be expressed in the period’s calendar. Only the ‘string value’ part of the instant-type literals should be specified, i.e. the reserved word `INSTANT`, the granularity specification and the calendar designation are omitted. A value of type *PERIOD* is considered to include all instants from l_i up to, but not including l_j.

The period literal specification is considered valid only if l_j designates a time instant which follows l_i. Note that this check is performed after converting each instant to the designated granularity, so the period literal `PERIOD '[1990-01, 1990-04)' YEAR` is invalid, since, after converting each endpoint instant to a granularity of year, the second instant is equal to the first one. If any of the endpoints is equal to `NOW()`, then the following additional conditions must be met:

- if the value of the first instant is `NOW()`, the value of the second instant must be `FOREVER`.
- if the value of the second instant is `NOW()`, the current value of the system clock should be greater than the first instant.

Figure 3 presents some examples of *Period* literals specification.

Literal	Value
PERIOD '[1990, 1991)' YEAR CALENDAR Gregorian	The year 1990 of the Gregorian calendar.
PERIOD '[1990-01, 1992-07)' MONTH	A 30-month period, starting from January 1990 and ending at July 1992 (the last month is not included in the period). The period is expressed in the Gregorian calendar (the default)
PERIOD '[1990, 1991-04)' MONTH CALENDAR Gregorian	An 15-month period, starting from January 1990 and ending at April 1991 in the Gregorian calendar. The information missing from the first instant (the month specification) is filled with a default value (1).
PERIOD '[NOW, 2000-01-01)' DAY	A period starting from the current time and ending at the first day of year 2000. The period is expressed in the Gregorian (default) calendar.
PERIOD '[Winter 1996, Spring 1997)' CALENDAR Academic	A period starting at the Winter semester of the academic year 1996 and ending at the Spring semester of the academic year 1997.

Figure 3 - Specifying literals of type *Period*

D. Period sets

Period set literals may be specified by using a constructor, in the same way that set literals may be defined in OQL. The constructor is named `period_set`, it is followed by a granularity specifier and a calendar designation. The constructor accepts a list of values of type *PERIOD*, which are the elements of the period sets. For each argument only its string literal part needs to be specified, i.e. the `PERIOD` keyword and the granularity designation may be omitted. However, the TOQL processor accepts full period specifications for any of the arguments to the constructor.

If the calendar specification is omitted, the default calendar is used, which is the Gregorian one. The granularity specification, if present, should designate a granularity which is valid for the calendar in which the period set is expressed (year, month, day, hour, minute or second, in the Gregorian calendar); if the granularity specification is missing, the default granularity for the specified (or default) calendar is used. All periods that are included in the constructor should be expressed in the calendar of the period set.

All arguments to the constructor are converted to the designated granularity and overlapping or adjacent arguments to the constructor are merged into a single period value, within the period set. Figure 4 presents some examples of *Period set* literals specification.

Literal	Value
period_set('[1990, 1991)', '[1992, 1994)') YEAR	A period set containing two intervals. The default calendar (Gregorian) is used for the period set.
period_set(PERIOD '[1990, 1991)' YEAR) DAY CALENDAR Gregorian	A period set of granularity day, containing the period '[1990-01-01, 1991-01-01)' (the result of casting the period '[1990, 1991)' to a granularity of day. The period set is expressed in the Gregorian calendar.
period_set('[1990, 1991)', '[1991, 1992)') YEAR	A period set with one element, corresponding to the years 1990 and 1991. Two arguments are listed in the constructor, but since they are adjacent, they are replaced by a single period ('[1990, 1992)') in the result value.
period_set('[Winter 1996, Spring 1997)') SEMESTER CALENDAR Academic	A period set with a single period element, expressed in the Academic calendar.

Figure 4 - Specifying literals of type *Period Set*

1.2.2 Functions

TOQL introduces new functions which can be applied on data of type *INSTANT*, *PERIOD* and *PERIOD SET*. These functions allow for the construction of values of type *PERIOD* and *PERIOD SET* (functions *period*, *intersection*, *merge*, and function *period_set*, presented in the previous section), event and period extraction (functions *begin*, *end*, *first* and *last*; for Gregorian instants, functions *year, month, day, hour, minute, second* are also provided) and computation of the duration of periods and period sets (function *duration*).

These functions are defined at the data model level as class methods, and thus may be called using the standard method invocation syntax (e.g. *p1->duration()*), but the functional notation is more user friendly. The TOQL processor will arrange for the necessary transformations. Functions *intersection*, *merge* and *duration* are not really needed, since the same results may be computed using expressions, but it is estimated that they will be used quite frequently, so they are provided in order to facilitate more compact query formulation.

The signatures of the new functions, along with a brief description of their functionality is presented in the following paragraphs.

- **`PERIOD period(in Instant I1, in Instant I2)`**

  Period is a constructor function. It accepts two arguments of type *INSTANT* and returns a result of type *PERIOD*, starting at *Instant1* and ending at *Instant2*. (Note that, since the periods are open right, instant *Instant2* is not included in the period.) Both arguments must have the same granularity, which determines the result's granularity. If *Instant1* does not occur later on the time axis than *Instant1*, the *NIL* value is returned.

  Examples:

  period(INSTANT '1994-01' MONTH, INSTANT '1995-01' MONTH) = PERIOD '[1994-01, 1995-01)' MONTH

  period(INSTANT '1995-01' MONTH, INSTANT '1994-01' MONTH) = *NIL*

  period(min(*I1*, *I2*), max(*I1*, *I2*)) returns a period which is delimited by the instants *I1* and *I2*. The earliest instant is used as the period's start, whereas the latest one is used as the period's end.

- **`PERIOD intersection(in PERIOD P1, in PERIOD P2)`**

  When applied on two arguments of type *PERIOD*, the **intersection** yields a result of the same type. The result contains all time points which are common to the two arguments and its granularity is the same as the function's arguments' granularity[3]. If the two arguments do not have common points, the **intersection** function returns the *NIL* value.

  Examples:

  intersection('[1994-01, 1995-01)', '[1994-06, 1995-06)') = '[1994-06, 1995-01)'

  intersection('[1994-01, 1995-01)', '[1995-06, 1996-01)') = *NIL*

- **`PERIOD_SET intersection(in PERIOD_SET PS1, in PERIOD_SET PS2)`**

  When applied on two arguments of type *PERIOD SET*, the **intersection** yields a result of the same type. The result contains all time points which are common to its two arguments, represented as non adjacent and non overlapping periods. If the two arguments do not have common points, the **intersection** function returns the empty period set. The granularity of the result is the same as the function's arguments' granularity.

  Examples:

  intersection(period_set('[1994-01, 1995-01)', '[1996-01, 1997-01)') MONTH, period_set('[1994-06, 1995-06)') MONTH) = {'[1994-06, 1995-01)'}

  intersection(period_set('[1994-01, 1995-01)') MONTH, period_set('[1995-06, 1996-01)') MONTH) = {} (empty period set)

- **`PERIOD merge(in PERIOD P1, in PERIOD P2)`**

  When the **merge** function is applied on two arguments of type *PERIOD* it yields a result of the same type. The result contains all time points which are included in either argument, and its granularity is the same as the arguments' granularity. If the two arguments are neither overlapping, nor adjacent and thus the result cannot be represented using a single value of type *PERIOD*, the **merge** function returns the *NIL* value.

  Examples:

  merge('[1994-01, 1995-01)', '[1994-06, 1995-06)') = '[1994-01, 1995-06)'

[3] The cases where the arguments have different granularities are presented in section 1.2.5.

merge('[1994-01, 1995-01)', '[1995-06, 1996-01)') = *NIL*

- **`PERIOD_SET merge(in PERIOD_SET P1, in PERIOD_SET P2)`**

  When the **merge** function is applied on two arguments of type *PERIOD SET* it yields a result of the same type. The result contains all time points which are included in either argument, expressed as non overlapping and non adjacent periods. The granularity of the result is the same as the arguments' granularity.

  Examples:

  merge(period_set('[1994-01, 1995-01)') MONTH,
  period_set('[1994-06, 1995-06)') MONTH) = {'[1994-01, 1995-06)'}

  merge(period_set('[1994-01, 1995-01)') MONTH,
  period_set('[1995-06, 1996-01)') MONTH) = {'[1994-01, 1995-01)',
  '[1995-06, 1996-01)'}

- **`short year(in Instant I)`**

  Function **year** accepts an argument of type *INSTANT* and returns an integer, which corresponds to the year part of the argument. The argument must be an instant of the Gregorian calendar.

  Example: year(INSTANT '1994-01' MONTH) = 1994

- **`short month(in Instant I)`**

  Function **month** accepts an argument of type *INSTANT* and returns an integer, which corresponds to the month part of the argument. If the argument does not contain a month part (because its granularity is coarser than *MONTH*), the corresponding default value (1) is returned. The argument must be an instant of the Gregorian calendar.

  Examples:

  month(INSTANT '1994-05' MONTH) = 5

  month(INSTANT '1994' YEAR) = 1

- **`short day(in Instant I)`**

  Function **day** accepts an argument of type *INSTANT* and returns an integer, which corresponds to the day part of the argument. If the argument does not contain a day part, the corresponding default value (1) is returned. The argument must be an instant of the Gregorian calendar.

  Example: day(INSTANT '1994-01-08 04:59:01' SECOND) = 8

- **`short hour(in Instant I)`**

  Function **hour** accepts an argument of type *INSTANT* and returns an integer, which corresponds to the hour part of the argument. If the argument does not contain a hour part, the corresponding default value (0) is returned. The argument must be an instant of the Gregorian calendar.

  Example: hour(INSTANT '1994-01-08 04:59:01' SECOND) = 4

- **`short minute(in Instant I)`**

  Function **minute** accepts an argument of type *INSTANT* and returns an integer, which corresponds to the minute part of the argument. If the argument does not contain a minute part, the corresponding default value (0) is returned. The argument must be an instant of the Gregorian calendar.

  Example: minute(INSTANT '1994-01-08 04:59:01' SECOND) = 59

- **`short second(in Instant I)`**

  Function **second** accepts an argument of type *INSTANT* and returns an integer, which corresponds to the seconds part of the argument. If the argument does not contain a seconds part, the corresponding default value (0) is returned. The argument must be an instant of the Gregorian calendar.

  Example: second(INSTANT '1994-01-08 04:59:01' SECOND) = 01

- **`INSTANT begin(in PERIOD P)`**

  When applied on an argument of type *PERIOD* the **begin** function returns an instant, which corresponds to the earliest time point contained in its argument. The granularity of the result is the same as the argument's granularity.

  Example: begin(PERIOD '[1994-01, 1995-01)' MONTH) = '1994-01'

- **`INSTANT begin(in PERIOD_SET PS)`**

  When applied on an argument of type *PERIOD SET* the **begin** function returns an instant, which corresponds to the earliest time point contained in its argument. The granularity of the result is the same as the argument's granularity.

  Example: begin(PERIOD '[1994-01, 1995-01)' MONTH) = '1994-01'

- **`INSTANT end(in PERIOD P)`**

  When applied on an argument of type *PERIOD*, the **end** function returns an instant of the same granularity as the function argument. The value of the argument is equal to the ending boundary of the period.

Example: end(PERIOD '[1994-01, 1995-01)' MONTH) = '1995-01'.

- **`INSTANT end(in PERIOD_SET PS)`**

  When applied on an argument of type *PERIOD SET*, the **end** function returns an instant of the same granularity as the function argument. The value of the result is equal to the ending boundary of the last period included in the argument.

  Example: end(period_set('[1994-01, 1995-01)', '[1995-06, 1997-01)') MONTH) = '1997-01' (note that since the period type is *closed left-open right* the time point 1997-01 is *not* considered to belong to the period).

- **`INTERVAL duration(in PERIOD P)`**

  When the argument of function **duration** is of type *PERIOD*, the function returns the argument's duration, computed at the argument's granularity (years, months, etc.).

  Example: **duration**(PERIOD '[1994-01, 1995-01)' MONTH) = INTERVAL '12' MONTH.

- **`INTERVAL duration(in PERIOD_SET PS)`**

  When the argument of function **duration** is of type *PERIOD SET*, the function returns the sum of the durations of all the periods that its argument contains. The number of instants is computed at the argument's granularity.

  Example: **duration**(period_set('[1994, 1995)', '[1996, 1997)') YEAR) = 2

- **`PERIOD first(in PERIOD_SET PS)`**

  Function **first** accepts an argument of type *PERIOD SET* and returns a result of type *PERIOD*, which has the same granularity as the function argument. The value of the result is equal to the earliest period which is included in the argument. If the argument is the empty period set, the value *NIL* is returned.

  Example: **first**(period_set('[1994, 1995)', '[1996, 1997)') YEAR) = '[1994, 1995)'

- **`PERIOD last(in PERIOD_SET PS)`**

  Function **last** accepts an argument of type *PERIOD SET* and returns a result of type *PERIOD*, which has the same granularity as the function argument. The value of the result is equal to the latest period which is included in the argument. If the argument is the empty period set, the value *NIL* is returned.

  Example: **last**(period_set('[1994, 1995)', '[1996, 1997)') YEAR) = '[1996, 1997)'

In addition to the functions presented above, TOQL introduces a syntactic construct to facilitate explicit conversion of types used for time representation to a different granularity. In order to convert a datum of type *INSTANT*, *INTERVAL*, *PERIOD* or *PERIOD SET* to a different granularity, the *CAST* construct is used as follows:

CAST datum TO granularity_specification

where *granularity_specification* is a designation of the desired granularity. The conversion is performed according to the following rules:

- when casting an instant to a coarser granularity, the extra information is dropped; when converting to a finer granularity, the missing information is filled with default values, as described in paragraph 1.2.1.
- when casting a period to some granularity, the starting end point of the period is always converted to the target granularity, as described above. However, the rules for converting the ending instant, depend on whether the original granularity is finer or coarser that the target one:
  1. if the target granularity is finer, then the ending instant is converted to it using the procedure described for instant conversion.
  2. if the target granularity is coarser, then the last instant that is actually contained in the period is computed (note that since the period is *open right*, the last instant included in the period is the predecessor of the right end point), it is converted to the target granularity by dropping the excessive information, and then its successor is used as the ending instant of the result of the conversion.

  These rules guarantee that no conversion leads to an illegal period and preserve the intuitive semantics of granularity conversion.
- when casting a period set to another granularity (finer or coarser), each of the periods included of the period set is converted to the designated granularity. If the target granularity is coarser than the original, the conversion may lead to the merging of periods, within the period set.

1.2.3 Predicates

TOQL supports all comparison operators defined in OQL and introduces new predicates which facilitate testing of the relative position of instant, period and period set values. The new predicates are described in the following paragraphs (i_1 and i_2

denote instant values; p_1 and p_2 denote period values; ps_1 and ps_2 denote period set values):

- **`p1 overlaps p2`**

  `ps1 overlaps ps2`

  The **overlaps** predicate may be applied on two operands of type *PERIOD* or *PERIOD SET*. It evaluates to *true*, when there exists one time point which is included in both operands, otherwise it evaluates to *false*.

  Examples:

  PERIOD '[1994-01, 1995-01)' MONTH **overlaps** PERIOD '[1994-06, 1995-06)' MONTH yields *true*

  PERIOD '[1994-01, 1995-01)' MONTH **overlaps** PERIOD '[1995-06, 1996-01)' MONTH yields *false*

- **`i1 precedes i2`**

  `p1 precedes p2`

  `ps1 precedes ps2`

  The **precedes** predicate may be applied on two operands of type *INSTANT, PERIOD* or *PERIOD SET*. It evaluates to *true*, when all instants included in the left operand are located before all instants included in the right operand, on the time axis.

  Examples:

  INSTANT '1990-01' MONTH **precedes** INSTANT '1991-01' MONTH yields *true*

  PERIOD '[1994-01, 1995-01)' MONTH **precedes** PERIOD '[1995-06, 1996-01)' MONTH yields *true*

  PERIOD '[1994-01, 1995-01)' MONTH **precedes** PERIOD '[1994-06, 1995-06)' MONTH yields *false*

- **`p1 contains p2`**

  `ps1 contains ps2`

  The **contains** predicate may be applied on two operands of type *PERIOD* or *PERIOD SET*. It evaluates to *true*, when every instant included in the right operand is also included in the left operand, otherwise it evaluates to *false*.

Examples:

PERIOD '[1994-01, 1996-01)' MONTH **contains** PERIOD '[1995-01, 1995-06)' MONTH yields *true*

PERIOD '[1994-01, 1995-01)' MONTH **contains** PERIOD '[1994-06, 1995-06)' MONTH yields *false*

- **`p1 meets p2`**

  The **meets** predicate may be applied on two operands of type *PERIOD*. It evaluates to *true*, when the end of the left operand is equal to the start of the right operand, otherwise it evaluates to *false*.

  Examples:

  '[1994-01, 1995-01)' **meets** '[1995-01, 1995-06)' yields *true*

  '[1994-01, 1995-01)' **meets** '[1994-06, 1995-06)' yields *false*

1.2.4 Operators

Standard arithmetic and set theoretic operators may be used in order to perform calculations on data types used for time representation. In the following figures, the allowable operations on the different types of data used for time representation are described.

Left operand	Operator	Right operand	Result	Description
	-	interval	interval	Negation of the interval's duration.
	+	interval	interval	Unary plus, returns the value of the operand.
interval	+	interval	interval	Adds the duration of the operands.
interval	-	interval	interval	Computes the difference of the operands' duration.
interval	*	number	interval	Multiplies the interval's duration by the designated number.
number	*	interval	interval	Equivalent to *interval * number*.
interval	/	number	interval	Divides the interval's duration by the designated number.
interval	/	interval	number	Computes how many times the left operand's duration is greater than the right operand's duration.

Figure 5 - Operations on intervals

Left operand	Operator	Right operand	Result	Description
instant	+	interval	instant	Computes an instant which is moved on the time axis by the duration specified by the designated interval.
interval	+	instant	instant	Equivalent to *instant + interval*.
instant	-	interval	instant	Equivalent to *instant + (- interval)*.
instant	-	instant	interval	Computes an interval whose duration is equal to the number of instants between the left and the right operand.

Figure 6 - Operations on instants

Left operand	Operator	Right operand	Result	Description
period	+	interval	period	Adds the value of the interval to the instants marking the starting and ending points of the period.
period	-	interval	period	Equivalent to *period + (- interval)*.
interval	+	period	period	Equivalent to *period + interval*.
period	>>	interval	period	Computes a period with the same starting instant as the left operand, but the ending instant is moved towards the end of the time axis by the duration specified by the designated interval. This operator extends the period by the specified interval.
period	<<	interval	period	Computes a period with the same starting instant as the left operand, but the ending instant is moved towards the beginning of the time axis by the duration specified by the designated interval. This operator shrinks the period by the specified interval.
period	union or +	period	period	Equivalent to *merge(p1, p2)*.
period	except or -	period	period	Computes a period which includes all time points contained in the left operand but not in the right operand. If the remaining time points are not consecutive, and thus cannot be represented with a single period value, *NIL* is returned.
period	intersect or *	period	period	Equivalent to *intersection(p1, p2)*.

Figure 7 - Operations on periods

Left operand	Operator	Right operand	Result	Description
period_set	union or +	period_set	period_set	Equivalent to *merge(ps1, ps2)*.
period_set	except or -	period_set	period_set	Computes a period set which includes all time points contained in the left operand but not in the right operand
period_set	intersect or *	period_set	period_set	Equivalent to *intersection(ps1, ps2)*.
period_set	+	interval	period_set	The + operator is applied to each period in the period set.
period_set	-	interval	period_set	The − operator is applied to each period in the period set.

Figure 8 - Operations on period sets

1.2.5 Automatic type and granularity conversions

In contexts where one of the functions, predicates and operators described in paragraphs 1.2.2, 1.2.3 and 1.2.4 is used with arguments which do not match exactly one of the signatures presented in these paragraphs, the TOQL processor introduces automatic type conversions, so as to be able to invoke an appropriate function. Additionally, all operations on which two or more objects representing time (instants, intervals, periods or period sets) are involved, require all their operands to be expressed in the same granularity. If this is not the case, the TOQL processor arranges for automatic conversions, in order to align the operands to the same granularity. In the following paragraphs, these features of TOQL are presented.

A. Type conversions

Instants are considered isomorphic to trivial periods, i.e. periods which contain only one time point, thus if an instant is used where a period is expected, the instant *I1* of granularity *G1* is converted to the period, whose beginning is *I1* and its end equal to *I1* + *INTERVAL '1' G1* (G1 is the granularity designation).

Periods are considered isomorphic to singleton period sets, i.e. period sets containing only one period; thus, if a period *P1* with granularity *G1* is used where a period set is expected, the period is converted to a period set whose value is equal to *period_set(P1) G1*.

Combining the two conversion rules presented above, instants should be considered isomorphic to period sets; thus, if an instant is used where a period set is expected, the

instant is first converted to a period and then to a period set, following the appropriate conversion rules.

A final conversion rule which is applied by the TOQL processor is casting of string values to instants or periods: if a string is used where an instant, an interval or a period is expected, TOQL tries to convert the string value to the appropriate type. In these conversions, a granularity of *second* and the Gregorian calendar is used, so as to provide compatibility with the TODL defaults.

Examples:

period1 meets *instant1* → *period1 meets period(instant1, instant1 + interval '1' G1)*
(*G1* is the granularity of *instant1*).

instant1 + period1 → *period(instant1, instant1 + interval '1' G1) + period1*
(*G1* is the granularity of *instant1*).

period_set1 + period1 → *period_set1 + period_set(period1) G1*
(*G1* is the granularity of *period1*).

period_set1 + instant1 → *period_set1 + period_set(period(instant1, instant1 + INTERVAL '1' G1)) G1*
(*G1* is the granularity of *instant1*).

B. Granularity conversions

TOQL adopts the *left operand granularity semantics* for granularity conversion, i.e. if two arguments of different granularities are used in a function, predicate or operator, both arguments are converted to the granularity of the left operand before the expression is evaluated. As a consequence, operators like *+, union* and *intersection* are not commutative, when applied to operands of different granularities. This approach, however, is more flexible than the *finest granularity semantics* (which is used in most programming languages), since the latter requires explicit type casts when computation at a coarser level is required, whereas using the *left operand granularity semantics*, it suffices to rearrange the arguments.

As an exception to this rule, if some operation involves an anchored quantity of time (instant, period or period set) and an unanchored quantity of time (interval), the unanchored quantity of time is not casted to the granularity of the anchored quantity, but the operation is performed directly on the different granularities. This allows the

operation to compute the correct result when the mapping between the original and the target granularity of the interval is irregular. For example, consider the operation

INSTANT '1990-01-31' DAY + INTERVAL '1' MONTH

If the interval (1 month) is casted to the granularity of *day*, it will yield *31*, and adding 31 days to the date 'January 31, 1990' will result to 'March 3, 1990', which is not the expected result. Performing the operation with different argument granularities allows to interpret the unanchored quantity of time relatively to the anchored one, and thus producing the correct result (in this case, 'February 28, 1990').

1.3 A Sample Database

Before we proceed to the description of the extended functionality of TOQL, regarding temporal data, we present a sample database, which will be used in the examples throughout the following paragraphs. The database contains information about dairy products and the factories in which they are manufactured. Its schema is described via the following TODL statements:

```
interface   Product
(extent     Products
 key        ProductName)
{
// Instance properties
            attribute String ProductName;
            attribute String Description;
            attribute float AvgSales valid state overlaps
                  granularity month;
            attribute List<String> Ingredients transaction;
            attribute Interval LifeTime valid state granularity day;
            relationship Set<Factory> manufactured valid state
                   granularity month transaction
                   inverse Factory::manufactures;
}

interface   Factory
(extent     Factories
 key        Owner, Location)
{
// Instance properties
            attribute String Owner;
            attribute String Location;
            attribute String Manager valid state granularity month;
            attribute Long Turnover valid state granularity month;
            relationship Set<Product> manufactures valid state
                   granularity month transaction
                   inverse Product::manufactured;
}
```

In the following, we will assume that the database has the following contents:

Obj001

ProductName: ‘Life Orange Juice’
Description: ‘Orange juice. Sold in 330ml, 1 lt and 2 lt packages’
AvgSales: {(value: 1000, VT: [1995-01, 1996-01)), (value: 1300, VT: [1995-06, 1996-06)),
(value: 1400, VT: [1996-01, 1997-01)), (value: 1600, VT: [1996-06, 1997-06))}
Ingredients: {(value: {‘Concentrated Orange Juice’, ‘Water’, ‘Sugar’, ‘E210’}, TT: [1995-01-01, 1995-06-01)),
(value: {‘Concentrated Orange Juice’, ‘Water’, ‘Sugar’}, TT: [1995-06-01, 1996-01-01)),
(value: {‘Concentrated Orange Juice’, ‘Water’, ‘E210’}, TT: [1995-06-01, *UC*))}
LifeTime: {(value: 18, VT: [1995-01-01, 1995-06-01)), (value: 9, [1995-06-01, 1996-06-01)),
(value: 12, [1996-06-01, *forever*))}
Manufactured: {(value: {Ref<Obj005>}, VT: [1995-01-01, 1996-06-01), TT: [1994-12-01, *UC*)),
(value: {Ref<Obj005>}, VT: [1996-06-01, 1997-01-01), TT: [1994-12-01, 1995-11-16)),
(value: {Ref<Obj005>, Ref<Obj006>}, VT: [1996-06-01, 1997-01-01), TT: [1995-11-16, *UC*)),
(value: {Ref<Obj006>}, VT: [1996-07-01, 1997-06-01), TT: [1996-04-10, *UC*))}

Obj002

ProductName: ‘Champion’
Description: ‘Chocolate milk with honey and malt. Sold in packages of 330 ml.’
AvgSales: {(value: 600, VT: [1994-01, 1994-06)), (value: 500, VT: [1994-06, 1996 -09)),
(value: 450, VT: [1994-06, 1995-01)) }
Ingredients: {(value: {‘Cow’’s Milk’, ‘Honey’, ‘Chocolate’, ‘Malt’, ‘Color’}, TT: [1994-01, 1994-05)),
(value: {‘Cow’’s Milk’, ‘Honey’, ‘Chocolate’, ‘Malt’}, TT: [1994-05, 1995-02))}
LifeTime: {(value: 6, VT: [1994-01-01, 1994-06-01)), (value: 9, [1994-06-01, 1995-01-01)),
(value: 12, [1995-06-01, *forever*))}
Manufactured: {(value: {Ref<Obj005>}, VT: [1994-01-01, 1996-01-01), TT: [1993-11-07, 1994-10-10)),
(value: {Ref<Obj005>}, VT: [1994-06-01, 1995-01-01), TT: [1994-10-10, *UC*))}

Obj005

Owner: ‘Delta Dairy S.A.’
Location: ‘St. Stefanos’
Manager: {(value: ‘Stefanou’, VT: [1994-01, 1995-01)), (value: ‘Nikou’, VT: [1995-01, 1996-09)}
Turnover: {(value: 8000, VT: [1995-01, 1996-01)), (value: 9000, [1996-01, 1996-06))}
Manufactures: {(value: {Ref<Obj002>}, VT: [1994-01-01, 1996-01-01), TT: [1993-11-07, 1994-10-10)),
(value: {Ref<Obj002>}, VT: [1994-06-01, 1995-01-01), TT: [1994-10-10, *UC*)),
(value: {Ref<Obj001>}, VT: [1995-01-01, 1997-01-01), TT: [1994-12-01, *UC*))}

Obj006

Owner: ‘3E Canning Co.’
Location: ‘Korinthos’
Manager: {(value: ‘Andreou’, VT: [1993-06, 1995-04)), (value: ‘Ioannou’, VT: [1995-04, 1997-01))}
Turnover: {(value: 5000, VT: [1994-01, 1994-09)), (value: 6000, VT: [1995-01, 1995-07))}
Manufactures: {(value: {Ref<Obj001>}, VT: [1996-06-01, 1997-01-01), TT: [1995-11-16, 1996-04-10)),
(value: {Ref<Obj001>}, VT: [1996-06-01, 1997-06-01), TT: [1996-04-10, *UC*))}

1.4 Queries on Temporal Data

The simplest query returns all the objects in an extent (e.g. the *Products* extent):

```
Products
```

The type of the result of this query is *Bag<Product>*. The result contains all product instances, as recorded in the database. Note that no filtering of information is performed in such a query, thus the valid time and transaction time attributes of each selected object contain the whole history. The “masking” of *logically deleted*

information (i.e. values of transaction time instance properties that pertain to past states of the database) and variants with past or future valid times is expected to be encapsulated in the *behaviour of the respective objects* (i.e. valid, transaction time or bitemporal data). This means that the default "get value" method of a transaction time (or bitemporal) object should return only the values which have not been logically deleted, and the default "get value" method of a valid time (or bitemporal) object should return only the values whose valid timestamp overlaps the current time point. Two other approaches to this schema may be identified, but both of them possess undesirable properties:

1. the first approach is to replace each transaction time (or bitemporal) object by the current snapshot of the object. This leads to having different memory and database schemata (the database schema is a set of timestamped values whereas the memory schema is a single value), and subsequently results that are not updatable objects.
2. the second approach is to eliminate object variants whose transaction timestamp end is not equal to *UC*, from all transaction time and bitemporal data. This can be achieved in two ways:
   A. by transforming the query to a different form, which ungroups each *Product* object, eliminates the logically deleted object variants using appropriate functions and then reconstructs *Product* objects:

```
select Product(ProductName: p->ProductName,
               Description: p->Description,
               LifeTime: p->LifeTime->current(),
               Ingredients: p->Ingredients->current(),
               manufactured: p->manufactured->current()))
from Products as p
```

   In this case, however, each one of the returned object is assigned a new object identifier, and thus cannot be used to update the corresponding database object.

   B. in order to preserve object updatability, an *update method* may be created for transaction time and bitemporal data, which eliminates logically deleted versions. In this case, the query would be transformed to:

```
select p from Products as p
where p->LifeTime->eliminate_past_states() and
      p->Ingredients->eliminate_past_states() and
      p->manufactured->eliminate_past_states()
```

(the *eliminate_past_states()* method for transaction time objects might look like:

```
Boolean TransactionTime::eliminate_past_states() {
for each variant
     if (variant.timestamp.end != UC)
           this->delete_variant(variant);
     return true;
}
```

)

Although this approach produces updatable objects, it has the undesirable side effect of physically deleting the past states of the object and thus is unacceptable.

The individual products may also be retrieved by referencing the distinct members of the *Products* extent:

```
select p from p as Products
```

The behaviour of the query is identical to the previous one, i.e. it returns a set of product instances, as recorded in the database.

The current value of individual temporal instance properties may be retrieved by listing them in the select list. For example, the query 'For each product, fetch its name, the current lifetime and the list of ingredients' can be stated as

```
select p->ProductName as ProductName
       p->LifeTime as LifeTime,
       p->Ingredients as Ingredients
from products as p
```

Although in the database schema attributes *LifeTime* and *Ingredients* are declared to have valid time semantics and transaction time semantics, respectively, this query returns only the current value for these attributes. This provides compatibility for applications that have been developed on top of a snapshot database schema, which at some later point has been modified to incorporate temporal semantics. For a more complete description on snapshot applications compatibility, see Appendix C.

If valid time attribute which allows variants with overlapping timestamps is listed in the select list, then a set of structures is returned, each one containing the value and the valid timestamp of a variant whose valid timestamp overlaps with the current time point (since all of them can be considered *current* information). For example, if the query

```
select p->AvgSales as AvgSales
from Products as p
where p->ProductName = 'Life Orange Juice'
```

is evaluated on 31/10/1996, it will return all the estimations about the average sales of product “Life Orange Juice”, for any period that includes October 1996, and is stored in the database, i.e. the set

{(value: 1400, VT: [1996-01, 1997-01)), (value: 1600, VT: [1996-06, 1997-06))}

The complete history of the attributes having valid time, transaction time or bitemporal semantics may be retrieved if the modifier ***valid***, ***transaction*** or ***bitemporal*** is prepended to the attribute, respectively. Thus, in order to retrieve the complete history of the *AvgSales*, *LifeTime* and *Ingredients* attributes, the previous query would be restated as:

```
select p->ProductName as ProductName,
       valid p->AvgSales as AvgSales,
       valid p->LifeTime as LifeTime,
       transaction p->Ingredients as Ingredients
from Products as p
```

In this case, the type of the result would be *bag<struct(ProductName: string, AvgSales: float valid state overlaps granularity month, LifeTime: interval valid state granularity day, Ingredients: list<string> transaction)>*. This means that the ***valid***, ***transaction*** and ***bitemporal*** modifiers return the histories of their operands as they are stored in the database; the application may use methods provided by the interface of valid time, transaction time and bitemporal data, in order to extract specific values or iterate over the variants. When applied to bitemporal data the ***valid*** and ***transaction*** modifiers convert them to valid or transaction time objects, respectively. In the first case, the transaction time history is dropped and the result contains only the current database state. If the ***transaction*** modifier is prepended to a bitemporal object, the variants with past and future valid times are dropped and, for the remaining variants, the valid timestamp is removed, resulting thus to a transaction time object. The ***transaction*** modifier may not be used with bitemporal objects that allow overlapping of valid timestamps.

Temporal objects (valid time, transaction time and bitemporal objects) may also be treated as *indexed collections*, orthogonally to lists, sets, bags and arrays supported by OQL. This means that a number of expressions designating a single variant or a set of variants within a temporal object is provided. In the following paragraphs, these expressions are described.

1.4.1 Expressions for valid state objects not allowing overlapping timestamps and valid event objects

In the following figure, the allowable expressions for valid time objects that do not allow variants with overlapping timestamps are described. The type of the result of each operation is determined as follows:

1. a single value of type *T*, if the operation results to a single variant. (*T* is the timestamped type)
2. *set<struct(Value: T, VT: timestampType)>*, if the operation results to more than one variants. *timestampType* is set to *period* for valid state objects not allowing overlapping timestamps and *instant* for valid event objects. The calendar and granularity of the *VT* field are derived from the respective characteristics of the operand's valid time.

Expression	Description
count(*valid_obj*)	Returns the number of variants contained in the valid time object
first(*valid_obj*)	Returns the variant of the element with the smallest[4] valid timestamp.
last(*valid_obj*)	Returns the variant with the greatest valid timestamp
valid_obj[number]	Returns the variant whose rank within the object matches the subscript. Variants are ordered with respect to their timestamps.
valid_obj[n1:n2]	Returns the variants whose ranks are included between numbers *n1* and *n2*.
valid_obj[instant]	Returns the variant whose valid timestamp overlaps with the designated instant.
valid_obj[period]	Returns the variants whose valid timestamp overlap with the designated period. If a variant's valid timestamp overlaps partially with the specified period (i.e. it contains some instants that are included in *period* and some instants that are not), this variant appears in the result, but its timestamp is set to *intersection(VT, period)*, where *VT* is the variant's valid timestamp.

Figure 9 - Expressions for valid state objects not allowing overlapping and valid event objects

When an instant or a period is used as a subscript, it may be preceded with the ***valid at*** designation, e.g. *valid_obj[valid at INSTANT '1990-01-01' DAY]*.

[4] A *total order* is defined for periods. Period $p1 = [p1_b, p1_e)$ is less than $p2 = [p2_b, p2_e)$ if one of the following conditions are true:
1. $p1_b < p2_b$
2. $p1_b = p2_b \wedge p1_e < p2_e$

If an expression yields a single variant, then the result is subject to the following operations:

a) function *valid* may be applied to the result, so as to return the associated timestamp (e.g. *valid(first(valid_obj))*).
b) the modifier *weighted* (or *weighted valid*) may be prepended to the expression returning the single variant. In this case, the value of the variant is multiplied by the duration of the variant's timestamp, in order to produce the final result. This modifier may be used only when the multiplication operation is meaningful for the type of the variant's value.

1.4.2 Expressions for valid state objects that allow overlapping timestamps

In the following figure, the allowable expressions for valid state objects that allow variants with overlapping timestamps are described. The schema of the result of each operation is determined as follows:

1. a single value of type *T*, if the operation results to a single variant. (*T* is the timestamped type)
2. *set<struct(Value: T, VT: period)>*, if the operation results to more than one variants. The calendar and granularity of the *VT* field are derived from the respective characteristics of the operand's valid time.

Expression	Description
count(*valid_obj)*	Returns the number of variants contained in the valid time object
first(*valid_obj*)	Returns the variant with the smallest valid timestamp.
last(*valid_obj*)	Returns the variant with the greatest valid timestamp
valid_obj[number]	Returns the variant whose rank within the object matches the subscript. Variants are ordered with respect to their timestamps.
valid_obj[n1:n2]	Returns the variants whose ranks are included between numbers *n1* and *n2*.
valid_obj[instant]	Returns the variants whose valid timestamps overlap with the designated instant.
valid_obj[period]	Returns the variants whose valid timestamp is contained in the designated period. Variants whose valid timestamp overlaps partially with the specified period, do not appear in the result.
valid_obj[distinct period]	Returns the variant whose valid timestamp is equal to the designated period.

Figure 10 - Expressions for valid state objects that allow overlapping timestamps

When an instant or a period is used as a subscript, it may be preceded with the ***valid at*** designation, e.g. *valid_obj[valid at INSTANT '1990-01-01' DAY].*

If an expression yields a single variant, then the result is subject to the following operations:

a) function *valid* may be applied to the result, so as to return the associated timestamp, e.g. *valid(first(valid_obj))*.

b) the modifier *weighted* (or *weighted valid*) may be prepended to the expression returning the single variant. In this case, the value of the variant is multiplied by the duration of the variant's timestamp, in order to produce the final result. This modifier may be used only when the multiplication operation is meaningful for the type of the variant's value.

1.4.3 Expressions for transaction time objects

In the following figure, the allowable expressions for transaction time objects are described. The schema of the result of each operation is determined as follows:

1. a single value of type *T*, if the operation results to a single variant. (*T* is the timestamped type)

2. *set<struct(Value: T, TT: period)>*, if the operation results to more than one variants. The calendar and granularity of the *TT* field are derived to the system's settings for transaction timestamp representation.

Expression	Description
count(*trans_obj*)	Returns the number of variants contained in the transaction time object
first(*trans_obj*)	Returns the variant of the element with the smallest transaction timestamp.
last(*trans_obj*)	Returns the variant with the greatest transaction timestamp.
trans_obj[number]	Returns the variant whose rank within the object matches the subscript. Variants are ordered with respect to their timestamps.
trans_obj[n1:n2]	Returns the variants whose ranks are included between numbers *n1* and *n2*.
trans_obj[instant]	Returns the variant whose transaction timestamp overlaps with the designated instant.
trans_obj[period]	Returns the variants whose transaction timestamps overlap with the designated period. If a variant's transaction timestamp overlaps partially with the specified period, this variant appears in the result, but its timestamp is set to *intersection(TT, period)*, where *TT* is the variant's transaction timestamp.

Figure 11 - Expressions for transaction time objects

When an instant or a period is used as a subscript, it may be preceded with the ***current at*** designation, e.g. *trans_obj[current at INSTANT '1990-01-01' DAY]*.

If an expression yields a single variant, then the result is subject to the following operations:

a) function *transaction* may be applied to the result, so as to return the associated timestamp (e.g. *transaction(first(trans_obj))*).

b) the modifier *weighted* (or *weighted transaction*) may be prepended to the expression returning the single variant. In this case, the value of the variant is multiplied by the duration of the variant's timestamp, in order to produce the final result. This modifier may be used only when the multiplication operation is meaningful for the type of the variant's value.

1.4.4 Expressions for bitemporal objects not allowing overlapping valid timestamps and bitemporal event objects

In the following figure, the allowable expressions for bitemporal objects that do not allow overlapping valid timestamps and bitemporal event objects are described. The

result type for each operation is given, except for the cases that a single variant is returned, in which case the type of the result is *T* (*T* is the timestamped type).

Expression	Description
count(*bitemp_obj)*	Returns the number of variants contained in the bitemporal object.
bitemp_obj[valid at instant]	Returns a *set<struct(Value: T, TT: period)>*, containing all variants whose valid time overlaps with the designated instant.
bitemp_obj[current at instant]	Returns a *set<struct(Value: T, VT:timestampType)*, containing all variants whose transaction time overlaps with the designated instant. For bitemporal state objects not allowing overlapping *timestampType* is *period*, whereas for bitemporal event objects *timestampType* is *instant*.
bitemp_obj[valid at instant1, current at instant2]	Returns the variant whose valid and transaction timestamps overlap with *instant1* and *instant2*, respectively.
bitemp_obj[valid at period]	Returns a *set<struct(Value: T, VT:timestampType, TT: period)>* containing the variants whose valid timestamps overlap with the designated period. For bitemporal state objects not allowing overlapping *timestampType* is *period*, whereas for bitemporal event objects *timestampType* is *instant*. Variants whose valid timestamp overlaps partially with the designated period, appear in the result with truncated valid timestamps.
bitemp_obj[current at period]	Returns a *set<struct(Value: T, VT:timestampType, TT: period)>*, containing the variants whose transaction timestamps overlap with the designated period. For bitemporal state objects not allowing overlapping *timestampType* is *period*, whereas for bitemporal event objects *timestampType* is *instant*. Variants whose transaction timestamp overlaps partially with the designated period, appear in the result with truncated transaction timestamps.
bitemp_obj[valid at period1, current at period2]	Returns a *set<struct(Value: T, VT:timestampType, TT: period)>* containing the variants whose valid and transaction timestamps overlap with *period1* and *period2*, respectively. For bitemporal state objects not allowing overlapping *timestampType* is *period*, whereas for bitemporal event objects *timestampType* is *instant*. Variants whose valid or transaction timestamp overlaps partially with the respective period, appear in the result with truncated timestamps.

Figure 12 - Expressions on bitemporal objects not allowing overlapping valid timestamps and bitemporal event objects

In the expressions where the ***valid at*** and ***current at*** designations are listed together, their order may be reversed. Note that it is allowed to specify both designations using

an *instant* with one of them and a *period* for the other (e.g. *bitemp_obj[valid at INSTANT '1990' YEAR, current at PERIOD '[1990-1991)' YEAR]*). In this case, the instant is casted to a period, using the rules presented in section 1.2.5, and then the expression is evaluated.

If an expression yields a single variant, then the result is subject to the following operations:

a) functions *valid* and *transaction* may be applied to the result, in order to return the associated valid or transaction timestamp respectively, e.g. *valid(bitemp_obj[valid at INSTANT '1990-01' DAY, current at INSTANT '1990-01' DAY])*.

b) the modifiers *weighted valid* or *weighted transaction* may be prepended to the expression returning the single variant. In this case, the value of the variant is multiplied by the duration of the variant's timestamp, in order to produce the final result. This modifier may be used only when the multiplication operation is meaningful for the type of the variant's value.

1.4.5 Expressions for bitemporal objects that allow overlapping valid timestamps

In the following figure, the allowable expressions for bitemporal objects that allow variants with overlapping valid timestamps are described. The result type for each operation is given, except for the cases that a single variant is returned, in which case the type of the result is *T* (*T* is the timestamped type).

Expression	Description
count(*bitemp_obj)*	Returns the number of variants contained in the bitemporal object
bitemp_obj[valid at instant]	Returns a *set<struct(Value: T, VT: period, TT: period)>* containing the variants whose valid timestamps overlap with the designated instant.
bitemp_obj[current at instant]	Returns a *set<struct(Value: T, VT: period)>*, containing all variants whose transaction time overlaps with the designated instant.
bitemp_obj[valid at instant1, current at instant2]	Returns a *set<struct(Value: T, VT: period)>*, containing all variants whose valid and transaction timestamps overlap with the respective instants.
bitemp_obj[valid at period]	Returns a *set<struct(Value: T, VT: period, TT: period)>* containing the variants whose valid timestamps overlap with the designated period. Variants whose valid timestamp overlaps partially with the designated period do not appear in the result.
bitemp_obj[valid at distinct period]	Returns a *set<struct(Value: T, TT: period)>*, containing the variants whose valid timestamps are equal to the designated period.
bitemp_obj[current at period]	Returns a *set<struct(Value: T, VT: period, TT: period)>* containing the variants whose transaction timestamps overlap with the designated period. Variants whose transaction timestamp overlaps partially with the designated period appear in the result with a truncated timestamp.
bitemp_obj[valid at period1, current at period2]	Returns a *set<struct(Value: T, VT: period, TT: period)>* containing the variants whose valid and transaction timestamps overlap with the designated periods. Partially overlapping valid and transaction timestamps are handled as described above.
bitemp_obj[valid at distinct period1, current at period2]	Returns a *set<struct(Value: T, VT: period, TT: period)>* containing the variants whose valid timestamp is equal to *period1* and whose transaction timestamps overlap with *period2*. Partially overlapping valid and transaction timestamps are handled as described above.

Figure 13 - Expressions on bitemporal objects that allow overlapping valid timestamps

If an expression yields a single variant, then the result is subject to the following operations:

a) functions *valid* and *transaction* may be applied to the result, in order to return the associated valid or transaction timestamp respectively, e.g. *valid(bitemp_obj[valid at DISTINCT INSTANT '1990-01' DAY, current at INSTANT '1990-01' DAY]).*

b) the modifiers *weighted valid* or *weighted transaction* may be prepended to the expression returning the single variant. In this case, the value of the variant is multiplied by the duration of the variant's timestamp, in order to produce the final result. This modifier may be used only when the multiplication operation is meaningful for the type of the variant's value.

1.4.6 Examples

The following queries can be answered by TOQL, using the expressions described in paragraphs 1.4.1 through 1.4.5:

Which was the life time of product "Life Orange Juice" when it was first introduced?

```
select first(valid p->LifePeriod)
from Products as p
where p->ProductName = "Life Orange Juice"
```

(the expression *(valid p->LifePeriod)[1]* may also be used, instead of *first(valid p->LifePeriod)*).

Which was the life time of product "Life Orange Juice" when it was first introduced and when was this life time changed?

```
select first(valid p->LifePeriod) as LifePeriod,
       end(valid(first(valid p->LifePeriod))) as ChangePoint
from Products as p
where p->ProductName = "Life Orange Juice"
```

Here, we can see that function ***valid*** can be applied on a variant that has been extracted from a valid time object, in order to access its valid timestamp. This function may also be applied on variants of bitemporal objects. Function ***transaction*** is also supported and can be applied on variants extracted from transaction time or bitemporal objects.

Which was the life time of product "Life Orange Juice" on 1994-01-01?

```
select (valid p->LifePeriod)[INSTANT '1994-01-01' DAY]
from Products as p
where p->ProductName = "Life Orange Juice"
```

How has the life time of product "Life Orange Juice" evolved during 1994?

```
select (valid p->LifePeriod)[PERIOD '[1994-01-01, 1995-01-01)' DAY]
from Products as p
where p->ProductName = "Life Orange Juice"
```

Which was the life time of "Champion" milk when it was withdrawn from the market?

```
select last(valid p->LifePeriod)
from Products as p
where p->ProductName = "Champion"
```

Which ingredients were used for "Champion" milk throughout its lifetime?

```
select transaction p->Ingredients
from Products as p
where p->ProductName = "Champion"
```

1.4.7 The type of the result of the subscript operators

As described in paragraphs 1.4.1 through 1.4.5, if a subscript operator that (potentially) selects more than one variants is applied on a temporal object, the result of the operator is a set of structures, with each structure containing the variant value and one or two timestamps, depending on the subscript operator and the time dimensions supported by the temporal object.

A different approach to the return type would be that in these cases a temporal object would be returned. The timestamped type of the temporal object would be identical to the operand's timestamp type, while the resulting temporal object would support one or two temporal dimensions. Although this solution seems more "natural", the semantics of transaction time dimension disallow its adoption. These semantics state that an object's transaction time starts at the instant the object was inserted in the database and ends at the instant the object was removed from it (deleted or updated). Instance properties cannot be removed from the database; they can only change their value (possibly to *nil*), and thus a variant must always exist in a temporal instance property which has a transaction timestamp with right boundary equal to *UC*.

If the subscript operators return temporal objects, it is possible for the user to formulate a query that selects a transaction time object which does not include current information. This object may then be assigned to a temporal instance property using host language code, or via an update method in a query, as shown in the following example:

```
select p->set_ingredient_history((transaction p->Ingredients)
                        [period '[1990-01-01, 1991-01-01)' day])
from Products as p
```

(The *set_ingredient_history* method used in this example sets the whole history of the attribute *Ingredient* of the receiving object to be equal to the history included in the argument). In order to avoid such problems, the subscript operators on transaction time objects return sets, so such assignments are not possible. The *set* return type has been extended to subscript operators operating on valid time and bitemporal objects too, for uniformity reasons.

1.4.8 Support for evolved and deleted values

By default, all subscript operators disregard the variants which are tagged as *deleted* ([TR31TR.1]) and consider only the variants that are tagged as "evolved" or "current". However, access to the deleted variants is provided by appending to the subscripted expression one of the keywords *evolved, deleted* or *all*. Keyword *evolved* is the default, thus its presence does not modify the result type or values that are described in paragraphs 1.4.1 through 1.4.5. If, however, the expression is followed by the keyword *deleted* or the keyword *all*, the result type and values are modified as follows:

1. the result type is *set<struct(value: T, VT: timestampType, TT: period, flag: variant_type)>*. *T* is the timestamped type, *timestampType* is either *instant* or *period*, depending on whether the temporal object's valid time is designated as *event* or *state*, respectively and *variant_type* is the enumeration *(current, evolved, deleted)*. One of the fields *VT* and *TT* may not occur in the result schema, if the result of the respective operator does not include the corresponding field, as described in paragraphs 1.4.1 through 1.4.5.
2. each structure in the result set corresponds to a variant of the temporal instance property. If the subscripted expression is followed by the *deleted* specification, only variants that are tagged as *deleted* will be considered for the formulation of the result value; if the *all* designation is used, instead of the *deleted* specification, all variants will be considered. The *flag* field for each member of the result set will indicate the status of the respective variant (*current, evolved* or *deleted*).

According to this description, the query

```
select (valid p->LifePeriod)[INSTANT '1994-01-01' DAY] all
from Products as p
where p->ProductName = 'Life Orange Juice'
```

will return a bag containing a single element. This element will be a set of structures, Each structure in the result set has a value of type *interval*, a timestamp of type *period* and a flag of type *variant_type*. These structure correspond to all variants in the valid history of the "Life Orange Juice" product's life period, whose valid timestamp overlaps with the designated instant (1994-01-01). The status of each variant (*evolved* or *deleted*) is reported in the *flag* field.

1.4.9 Extracting object states

If a class has been declared to have valid time and/or transaction time semantics, the expressions described in paragraphs 1.4.1 through 1.4.5 may be used with expressions whose value is an instance of the class. For example, if the *Factory* class declaration were

```
interface   Factory
(extent     Factories
 key        Owner, Location) valid state granularity day transaction;
{
// Attributes
            attribute String Owner;
            attribute String Location;
            attribute String Manager;
            relationship Product manufactures
                  inverse Product::manufactured;
}
```

then the queries

```
select (bitemporal state f)[valid at INSTANT '1995-01-01' DAY,
                            current at INSTANT 'NOW' DAY]
from Factories as f
```

and

```
select (bitemporal state f)[current at INSTANT 'NOW' DAY,
                  valid at PERIOD '[1995-01-01, 1995-06-01)' DAY]
from Factories as f
```

could be used to retrieve the states of the factories, as recorded in the current database state for the instant January 1, 1995 and the period of the first five months of 1995, respectively. Thus, the syntax for invoking a state extraction operation on an object is

```
(time_dimension state temporal_object)extraction_operation
```

where *time_dimension* may be one of the keywords *valid*, *transaction* and *bitemporal*, and *extraction_operation* must be a subscript operator which is applicable to objects of the specified time dimension(s). Keyword *valid* may be used only for valid time and bitemporal objects; keyword *transaction* is applicable only to transaction time objects, bitemporal event objects and bitemporal state objects not allowing overlapping of valid timestamps; keyword *bitemporal* is allowed only for bitemporal objects.

The type of the result of a state extraction operation will be either an object state, or a set of structures, with each structure holding an object state and a timestamp (or two timestamps), depending on the number of states returned by the expression: if the expression yields exactly one state, the result type will be an object state; if the expression yields more than one states, the result type is a set of structures. The

number of timestamps depends on the valid and transaction time semantics of the operand and the subscript operator. The rules for determining the number of timestamps are analogous to the rules described in paragraphs 1.4.1 through 1.4.5.

According to the above, the first example query returns a bag of factory states, with each state corresponding to the state of each factory on 1995-01-01, as recorded in the current state of the database. The second example query returns a bag of sets. Each set corresponds to a factory, and contains the factory states within the first five months of year 1995, as recorded in the current state of the database.

The allowable expressions and the respective return types for a specific temporal class depend on the temporal characteristics that have been defined for this class, and are in coherence with the allowable expressions and the return types for temporal instance properties.

The same modifiers (i.e. *time_dimension state*) may be prepended to temporal objects when they are used as an argument to one of the functions *count*, *first* and *last*. In these cases, the functions return number of states in the object, the state with the smallest timestamp and the state with the greatest timestamp, respectively. Functions *first* and *last* may not be used when *time_dimension* is set to *bitemporal*.

1.4.10 Semantic ambiguity resolution

In expressions involving temporal objects ambiguity may arise from the usage of member accessor operators (`.` and `->`), temporal modifiers (*valid*, *transaction* and *bitemporal*), the subscript operator (`[]`) and specific functions (*count*, *first*, *last*), since the operators may be meaningful both for the temporal and the snapshot dimension of the arguments. For example, the query

```
select count(p->Ingredients)
from Products as p
where p->ProductName = "Life Orange Juice"
```

may be interpreted as “fetch the number of ingredients currently used for product Life Orange Juice” (the function applies to the snapshot dimension of the *Ingredients* attribute) or “fetch the number of times that the ingredients of product Life Orange Juice have changed” (the function applies to the temporal dimension of the *Ingredients* attribute). This ambiguity is resolved by introducing a semantic rule, stating that in order for the function or operator to apply to the temporal dimension of its argument, the argument must be preceded with a temporal modifier; otherwise, the

function applies to the snapshot dimension. This semantic rule preserves snapshot reducibility, since the semantics of snapshot queries do not change if the database schema has been enriched with temporal semantics.

Note, however, that the temporal modifiers have lower precedence than the member accessor and the subscript operators, so parentheses must be used in order to specify the correct order of evaluation. Thus, in order to retrieve the first list of ingredients for the product "Life Orange Juice" that has been recorded into the database, the following query must be issued:

```
select (transaction p->Ingredients)[1]
from Products as p
where p->productName = "Life Orange Juice"
```

If the parentheses were omitted from this query (i.e. the select list read *transaction p->Ingredients[1]*), attribute *Ingredients* would be converted to its snapshot value, the subscript operator would then be applied to it. Finally, the attempt to evaluate the *transaction* modifier on the string result of the subscript operator would result to a semantic error, since temporal modifiers are not applicable on strings.

1.5 Referencing Object Variants

A temporal instance property is actually a collection of values, each one having one or two associated timestamps (valid and/or transaction time). Since OQL allows collections to be used for variable definition in the *from* clause as well as in *collection expressions* (universal quantification, existential quantification and membership testing), TOQL allows temporal instance properties to be used for the same purpose.

When a variable is defined in terms of a temporal object, it iterates over the different variants which are stored in the temporal object. All variants are considered, regardless of their valid or transaction timestamp. The defined variable may be used as argument to the functions ***valid*** and ***transaction*** so as to return the corresponding timestamp (function ***valid*** may be applied to valid time and bitemporal objects, whereas function ***transaction*** may be applied to transaction time and bitemporal objects). TOQL supports all forms of variable declaration that are defined in OQL v. 1.2, thus a variable iterating over the variants of a temporal object may be defined in the *from* clause using the syntax

```
temporal_object as identifier
```

The ***as*** keyword is optional and may thus be omitted. In the existential and universal quantification constructs, a variable is defined using the ***in*** keyword. Consequently, an existential quantification query that involves a variable defined on a temporal object has the form

```
exists identifier in temporal_object: query
```

whereas a universal quantification query involving a variable defined on a temporal object is written as

```
for all identifier in temporal_object: query
```

Temporal objects may be used in the place that a collection is allowed in a membership testing query (`item in collection`). In this case, each variant of the temporal object is tested, in order to determine if its value is equal to the left hand side value. If any variant is found to have the specified value, the expression evaluates to *true*, otherwise it evaluates to *false*. Finally, temporal objects may be used as right hand side queries in *composite predicates*. If *e1* and *e2* are expressions with *e2* being a temporal object and *e1* having the snapshot type of *e2*, and *relation* is a relational operator (=, !=, <, <=, > or >=) then e_1 *relation some* e_2, e_1 *relation any* e_2 and e_1 *relation all* e_2 are expressions whose value is of type boolean. Expressions e_1 *relation some* e_2 and e_1 *relation any* e_2 evaluate to *true* if the operator *relation* holds between expression e_1 and the value of at least one variant of e_2. Expression e_1 *relation all* e_2 evaluate to *true* if the operator *relation* holds between expression e_1 and the values of all variants of e_2.

Examples

1. Select the products that have ever had a life time equal to15 days:

```
select p from Products as p
where exists l in valid p->LifeTime: l = INTERVAL '15' DAY
```

   This query may also be formulated using membership testing as follows:

```
select p from Products as p
where INTERVAL '15' DAY in valid p->LifeTime
```

   The same query may be stated using a composite predicate:

```
select p from Products as p
where INTERVAL '15' DAY = any valid p->LifeTime
```

2. For each product, list its name and life time, along with the corresponding period:

```
select p->ProductName as ProductName,
       l as LifeTime,
       valid(l) as LifeTimePeriod
from Products as p, valid p->LifeTime as l
```

3. For each product, list its name and its ingredients, if the ingredients contained 'E210'. List also the period during which the ingredients were used.

```
select p->ProductName as ProductName,
       p->Ingredients as Ingredients,
       transaction(Ingredients) as TT
from Products as p, transaction p->Ingredients as Ingredients
where exists Ingredient in Ingredients: Ingredient = 'E210'
```

1.6 Conversion Between Temporal and Snapshot Values

A. The *snapshot* modifier

TOQL provides the ability to convert temporal objects to a set of snapshot values, using the ***snapshot*** modifier. The ***snapshot*** modifier may be prepended to any query returning a temporal object and effectively drops all timestamps, returning only plain values. For example, the query

```
select p->ProductName as ProductName,
       max(snapshot p->LifeTime) as MaxLifeTime
from Products as p
```

yields a set of tuples. Each tuple holds the name of an product, and the maximum lifetime it had ever had.

The type of the result depends on the type returned by the query following the ***snapshot*** modifier. If the type returned by the query is *<type> temporal_spec*, then the type of the result is *Bag<type>*. (*temporal_spec* is any specification designating valid time and/or transaction time semantics.)

B. The *valid* modifier

The ***valid*** modifier may be used to convert collections to valid time objects. The syntax of the ***valid*** modifier is

```
valid valid_type [overlaps] [granularity] [calendar]
                 '['query : expression']'
```

where *query* is any query returning the desired collection and *expression* is a query providing the timestamps. The *valid_type* designation should be either *state* or *event* while *granularity* and *calendar* are used to define the desired granularity and calendar of the valid timestamp. Finally, the optional ***overlaps*** designation can be used to specify that overlapping valid timestamps are allowed within the valid time object (the default is to disallow overlapping timestamps). This designation may only be used when *valid_type* is state.

The collection returned by *query* is converted to a valid time object: each element of the collection is converted to an object variant, within the valid time object. The timestamp of each variant is computed by evaluating the *expression*, which must yield a value of type *INSTANT* or *PERIOD*. For example, the query

```
valid state [select Ingredient
             from transaction aProduct->Ingredients as Ingredient :
             transaction(Ingredient)]
```

produces a valid time object, with each variant corresponding to a snapshot of the transaction time history of the *Ingredients* attribute. The valid timestamp of each variant is set to be equal to the transaction timestamp of the original variant. Effectively this query converts the transaction time object to a valid time object. Note that the query following the colon (**:**) may reference variables that are declared in the query preceding the colon. This applies only to variables declared in the top level *from* clause of the left hand side query, and not to variables that are declared in subqueries.

The type of the result depends on the type returned by the query enclosed in the square brackets (denoted as *<type>* in the following), the *valid_type* designation, the existence of the *overlap* specifier and the granularity and calendar specification. The rules for determining the result type are as follows:

- if *valid_type* is ***state***, the ***overlaps*** designation is absent and granularity *G* and calendar *C* are specified, then the result type is *<type> valid state granularity G calendar C*. An error is raised if *expression* produces overlapping periods.
- if *valid_type* is ***state***, the ***overlaps*** designation is present and granularity *G* and calendar *C* are specified, then the result type is *<type> valid state overlaps granularity G calendar C*. An error is raised if *expression* produces identical periods.
- if *valid_type* is ***event***, the ***overlaps*** designation is absent and granularity *G* and calendar *C* are specified, then the result type is *<type> valid event granularity G calendar C*. An error is raised if *expression* produces identical instants.
- if *valid_type* is ***event*** and the ***overlaps*** designation is present, an error is raised.

In all cases, the calendar specification may be omitted, in which case it defaults to the Gregorian calendar. Similarly, if the granularity specification is absent, the default granularity for the specific calendar is used. The default granularity for the Gregorian calendar is *second*. Finally, *<type>* is not allowed have temporal characteristics, in any case.

C. The *transaction* modifier

The ***transaction*** modifier may be used to convert snapshot objects to transaction time objects. Its syntax is

```
transaction '[' query ']'
```

Where *query* is any query returning a single object (the query may actually return a collection of objects -e.g. a set- in which case the collection is treated as a single object of type *collection*). The result contains a single variant, whose value is set to the result of the query, whereas its timestamp is set to *[NOW, UC)*. Note that there is no provision to set the transaction time of the variant, since the transaction time reflects the time period that facts were known to the database, thus it is not reasonable to "force" past or future knowledge. For example, if the query

```
select transaction [p->ProductName] from Products as p
```

is issued against the database presented in section 1.3 on 1996/08, the result will be a set containing the following two elements:

```
{(value: 'Life Orange Juice', TT: [1996/08, UC))}
{(value: 'Champion', TT: [1996/08, UC))}
```

Each element of the set is a transaction time object.

The type of the result of the ***transaction*** modifier depends on the type returned by the *query* that is enclosed in square brackets. If the type returned by the *query* is *<type>*, then the type of the result of the ***transaction*** modifier will be *<type> transaction*. *<type>* is not allowed to have temporal semantics.

D. The *bitemporal* modifier

The ***bitemporal*** modifier may be used to convert collections to bitemporal objects. The syntax of the ***bitemporal*** modifier is

```
bitemporal valid_type [overlaps] [granularity] [calendar]
                      '['query : expression']'
```

and its functionality is analogous to the ***valid*** modifier, except that it produces bitemporal objects, instead of valid time objects. The transaction timestamp of each variant is set to *[NOW, UC)*.

The rules for determining the type of the results are analogous to the rules applied for the ***valid*** modifier, except that bitemporal objects are produced, instead of valid time objects.

1.7 Temporal Joins

Temporal joins are needed when the information stored in two (or more) temporal objects must be combined. Consider the query 'For each factory, list its owner, location and turnover, along with the name of the manager of the corresponding period'. The information about the factory's turnover is recorded in the *turnover* instance property, whereas information about the managers of factories is stored in the *Manager* instance property. Both of these instance properties have valid time semantics.

The query may be answered in TOQL using object variant referencing, as illustrated below:

```
select f->Owner as Owner, f->Location as Location,
       t as TurnOver, m as Manager,
       intersection(valid(t), valid(m)) as time
from Factories as f, valid f->Turnover as t, valid f->Manager as m
where valid(t) overlaps valid(m)
```

The query above may produce many elements for each factory, one for each combination of turnover and manager with overlapping valid time periods. The schema of the result may be changed so as to produce one element for each factory, consisting of three fields, the first two being the factory's owner and location and the third one being a valid time object of type *struct(Turnover: integer, Manager: string) valid state granularity month*. This can be achieved using the ***valid*** modifier as follows:

```
select f->Owner as Owner, f->Location as Location,
    Info: valid state [select t as Turnover, m as Manager
                       from valid f->Turnover as t,
                            valid f->Manager as m
                       where valid(t) overlaps valid(m)) :
                       intersection(valid(t), valid(m))]
from Factories as f
```

The query as presented above is quite verbose, and since it is anticipated that such queries will be frequent, TOQL provides a special operator, ***tstruct***, which may be used to express temporal joins between temporal objects. Using the ***tstruct*** operator, the query above may be expressed as

```
select f->Owner as Owner, f->Location as Location,
       tstruct(Turnover: valid f->Turnover,
                Manager: valid f->Manager) as Info
from Factories as f
```

Thus, the syntax for the ***tstruct*** operator is similar to the syntax of the standard ***struct*** operator, i.e.

```
tstruct(identifier: query {, identifier: query})
```

where each query evaluates to a temporal object. Note that the *tstruct* operator constructs a new temporal object, by setting the variant values and the associated timestamps. Timestamp setting, however, is only allowed on the valid time dimension, since the transaction time axis reflects the evolution of the database through time, and consequently the construction of past or future transaction timestamps should not be permitted. Thus, only valid time objects may be combined using the *tstruct* operator; bitemporal objects must be first converted to valid time objects by prepending the keyword *valid* to the corresponding object (resulting in dropping the transaction time dimension) or using the *valid* modifier, in order to construct arbitrary timestamps for each variant; transaction time objects must be converted to valid time objects, using the *valid* modifier.

1.7.1 The type of the result of the temporal join

The type of the result of the ***tstruct*** operator depends on the types of its arguments. In the following table, the cases of joining two temporal objects using the *tstruct* operator are summarised.

query1 \ query2	valid state <T2>	valid state overlap <T2>	valid event <T2>
valid state <T1>	valid state <cT>	valid state overlap <cT>	valid event <cT>
valid state overlap <T1>	valid state overlap <cT >	valid state overlap <cT>	Not allowed
valid event <T1>	valid event <cT>	Not allowed	valid event <cT>

Figure 14 - Type of the result of the *tstruct* operator

(The notation *cT* used in the table above, denotes the type *struct(id1: T1, id2: T2).*)
The user is allowed to cast the result of the temporal join to a temporal class that has been declared to the database through TODL, using the standard OQL notation for type conversion, i.e. *(class_name)query*. In this case, the composite type is set to the snapshot class that corresponds to the temporal class and has been implicitly declared by the TODL processor. Note that such a conversion may be mandatory, if the underlying OODBMS and data model do not allow “on-the-fly” construction of arbitrary temporal classes.

Both arguments to the *tstruct* operators must use the same calendar for their valid timestamps; the granularity, however, need not be the same and, in the case that the

arguments' granularities do not match, the granularity of the result's valid time is set to the granularity of the left argument's valid time. Note that since conversion to coarser granularities may produce identical timestamps out of different ones (e.g. timestamps PERIOD '[1990-01, 1990-02)' MONTH and PERIOD '[1990-04, 1990-06)' MONTH both map to the same period when converted to granularity YEAR, namely PERIOD '[1990, 1991)') it is possible that such a conversion will lead to an attempt to construct an illegal valid time object i.e.

- a valid state object not allowing overlapping, containing variants with overlapping timestamps.
- a valid state object allowing overlapping, containing variants with identical timestamps.
- a valid event object, containing variants with identical timestamps.

In all these cases, an exception is raised to indicate the error.

As shown in Figure 14, the result's valid time characteristics depend on the valid time characteristics of the arguments. The rules for determining the type of a temporal join's result are derived from the semantics of the different valid time object types:

- valid state objects not allowing overlapping store information that applies to period level and instant level as well.
- valid state objects allowing overlapping store information that applies to period level only.
- valid event objects store information that applies to instant level only.

In all cases, in order for the result of a temporal join to incorporate some *level of information* (period or instant), this level of information must be included in both operands. This is the reason for which temporal joins between valid state objects allowing timestamp overlapping and valid event objects is disallowed, since these types do not have a common level of information representation.

1.7.2 The value of the result of a temporal join

The value of the result of a temporal join with two arguments is discussed in the following paragraphs. Analogous rules apply when more than two temporal objects are joined via the ***tstruct*** operator.

A. Temporal join between two valid state objects not allowing overlapping

If *VO1* and *VO2* are two valid state objects that do not allow overlapping of valid timestamps and have types *valid state<T1>* and *valid state<T2>* respectively, then *tstruct(id1: VO1, id2: VO2)* produces a valid state object that does not allow overlapping of valid timestamps and has type *valid state<struct(id1: T1, id2: T2)>*. A variant *v* with value *struct(id1: x1, id2: x2)* and valid timestamp $valid_v$ belongs to the resulting object if the following conditions are met:

1. a variant v_1 with value *x1* and valid timestamp $valid_{v1}$ belongs to *VO1*.
2. a variant v_2 with value *x2* and valid timestamp $valid_{v2}$ belongs to *VO2*.
3. **intersection**($valid_{v1}$, $valid_{v2}$) = $valid_v$.

B. Temporal join between two valid state objects allowing overlapping

If *VO1* and *VO2* are two valid state objects that do allow overlapping of valid timestamps and have types *valid state overlaps<T1>* and *valid state overlaps<T2>* respectively, then *tstruct(id1: VO1, id2: VO2)* produces a valid time object that allows overlapping of valid timestamps and has type *valid state overlaps<struct(id1: T1, id2: T2)>*. A variant *v* with value *struct(id1: x1, id2: x2)* and valid timestamp $valid_v$ belongs to the resulting object if the following conditions are met:

1. a variant v_1 with value *x1* and valid timestamp $valid_{v1}$ belongs to *VO1*.
2. a variant v_2 with value *x2* and valid timestamp $valid_{v2}$ belongs to *VO2*.
3. $valid_{v1} = valid_v$ and $valid_{v2} = valid_v$.

C. Temporal join between two valid event objects

If *VO1* and *VO2* are two valid event objects and have types *valid event<T1>* and *valid event<T2>* respectively, then *tstruct(id1: VO1, id2: VO2)* produces a valid event object of type *valid event<struct(id1: T1, id2: T2)>*. A variant *v* with value *struct(id1: x1, id2: x2)* and valid timestamp $valid_v$ belongs to the resulting object if the following conditions are met:

1. a variant v_1 with value *x1* and valid timestamp $valid_{v1}$ belongs to *VO1*.
2. a variant v_2 with value *x2* and valid timestamp $valid_{v2}$ belongs to *VO2*.
3. $valid_{v1} = valid_v$ and $valid_{v2} = valid_v$.

D. Temporal join between a valid state object not allowing overlapping and a valid state object allowing overlapping

If *VO1* is a valid state object that does not allow overlapping of valid timestamps and has type *valid state<T1>* and *VO2* is a valid state object that allows overlapping valid timestamps and has type *valid state overlaps<T2>*, then *tstruct(id1: VO1, id2: VO2)* produces a valid state object of type *valid state overlaps<struct(id1: T1, id2: T2)>* that allows overlapping timestamp. A variant *v* with value *struct(id1: x1, id2: x2)* and valid timestamp $valid_v$ belongs to the resulting object if the following conditions are met:

1. a variant v_1 with value *x1* and valid timestamp $valid_{v1}$ belongs to *VO1*.
2. a variant v_2 with value *x2* and valid timestamp $valid_{v2}$ belongs to *VO2*.
3. ($valid_{v1}$ **contains** $valid_v$ or $valid_{v1} = valid_v$) and $valid_{v2} = valid_v$.

Analogous rules apply when the left operand is a valid state object that allows overlapping and the right operand is a valid state object that doesn't allow overlapping timestamps.

E. Temporal join between a valid state object not allowing overlapping and a valid event object

If *VO1* is a valid state object that does not allow overlapping of valid timestamps and has type *valid state<T1>* and *VO2* is a valid event object that has type *valid event<T2>*, then *tstruct(id1: VO1, id2: VO2)* produces a valid event object of type *valid event<struct(id1: T1, id2: T2)>*. A variant *v* with value *struct(id1: x1, id2: x2)* and valid timestamp $valid_v$ belongs to the resulting object if the following conditions are met:

1. a variant v_1 with value *x1* and valid timestamp $valid_{v1}$ belongs to *VO1*.
2. a variant v_2 with value *x2* and valid timestamp $valid_{v2}$ belongs to *VO2*.
3. ($valid_{v1}$ **contains** $valid_v$ or $valid_{v1} = valid_v$) and $valid_{v2} = valid_v$.

Analogous rules apply when the left operand is a valid event object and the right operand is a valid state object that allows overlapping timestamps.

1.7.3 Temporal joins involving more than two objects

The ***tstruct*** operator may be applied to any number of temporal objects to_1, to_2, ..., to_n with types $<T_1>$ vtc_1, $<T_2>$ vtc_2, ..., $<T_n>$ vtc_n, respectively (T_i denotes the

timestamped type, while vtc_i represents the valid time characteristics of the corresponding objects). In this case, the type of the resulting objects is determined as follows:

1. the timestamped type is *struct(id_1: T_1, id_2: T_2, ..., id_n: T_n)*, where id_i are the identifiers used in the ***tstruct*** operator.
2. The granularity of the valid timestamps is set to the granularity of the valid timestamps of the leftmost object.
3. the valid time semantics of the result are set to the valid time characteristics with the highest precedence, amongst the involved ones.

1.8 Restructuring Operators

Restructuring operators facilitate the formulation of different equivalent representations of temporal data. Two restructuring operators are provided, with the first one converting period-timestamped variants to instant-timestamped ones, while the second restructuring operator allows for selection of the time axis on which maximal timestamps will be produced. These operators are discussed in the following paragraphs.

1.8.1 Converting period timestamping variants to instant timestamping

For valid state objects not allowing overlapping transaction time objects and bitemporal objects, a restructuring operator is provided that breaks down each valid or transaction time period to individual instants, producing thus many object variants out of a single one. The syntax for invoking this restructuring operator is

`temporal_object(partition` *`time_axis`* `as instant)`

where *time_axis* is either *valid* or *transaction*. *time_axis* may be *valid* if the temporal object has valid state semantics and does not allow overlapping for valid timestamps. The value *transaction* is allowed for *time_axis* if the temporal object has transaction time semantics.

Applying the restructuring operator *(partition valid as instant)* on an object of type *T valid state G1 calendar C1* (*G1* is the granularity of valid timestamps) produces a result of type *set<struct(Value: , VT: Instant)>*, with instants having granularity *G1* and calendar *C1*. An element (v_r, vt_r) belongs in the result set if a variant (v_{var}, vt_{var}) occurs within the valid time object and the following conditions are met:

- $v_r = v_{var}$
- vt_{var} **contains** vt_r

If the same operator is applied on a bitemporal object, the result schema is extended so as the elements of the resulting set include one extra field with type *Period* and name *TT*. The value of this field, for each element of the result set, is equal to the transaction timestamp of the respective variant.

The restructuring operator *(partition valid as instant)* is not applicable on valid state and bitemporal state objects that allow overlapping on valid timestamps, and an attempt to apply this operator on such an object raises a semantic error. This restriction is imposed for semantic purposes, since the values of the variants in valid state and bitemporal state objects allowing overlapping of valid timestamps apply only on period level, and not on instant level.

Similarly, applying the *(partition transaction as instant)* operator on an object of type *T transaction*, results in a set of type *set<struct(Value: T, TT: Instant)>*. An element (v_r, tt_r) belongs in the result set if a variant (v_{var}, tt_{var}) occurs within the transaction time object and the following conditions are met:

- $v_r = v_{var}$
- tt_{var} **contains** tt_r

If the same operator is applied on a bitemporal object, the result schema is extended so as the elements of the resulting set include one extra field with name *VT*. The type of this field is identical to the type of the valid timestamps of the bitemporal object. The value of this field, for each element of the result set, is equal to the valid timestamp of the respective variant.

Example:

For the factory owned by “Delta Dairy S.A.” and located at “St. Stefanos”, list the manager for each individual month.

```
select MonthInfo
from Factories as f,
     (valid f->Manager)(partition valid as instant) as MonthInfo
where f->Owner = "Delta Dairy S.A" and f->Location = "St. Stefanos"
```

Since the value of the *Manager* field for the object containing information on the factory owned by “Delta Dairy S.A.” and located at “St. Stefanos” is the valid state object

```
{(value: 'Stefanou', VT: [1994-01, 1995-01)),
 (value: 'Nikou', VT: [1995-01, 1996-09)}
```

(see section 1.3), the result of this query will be equal to the following set:

```
{(Value: 'Stefanou', VT: 1994-01),  (Value: 'Stefanou', VT: 1994-02),
                ...,                (Value: 'Stefanou', VT: 1994-12),
 (Value: 'Nikou', VT: 1995-01),     (Value: 'Nikou', VT: 1995-02),
                ...,                (Value: 'Nikou', VT: 1996-08)}
```

Finally, we note that if the distinguished value *forever* appears as the end of a valid timestamp, it is substituted by the value of the current timestamp, when conversion to instant timestamping is performed. Indeed, the special value *forever* is frequently used in valid timestamps having the "until changed" semantics, and not as actually designating the end of the calendar. Thus, in order to avoid computation of unnecessary (and numerous) variants, this substitution is enforced. If the user wants to formulate variants which extend to the end of the calendar, he/she must use explicitly the last timestamp in the calendar, instead of the distinguished timestamp *forever*.

1.8.2 Producing maximal timestamps

When information evolution is tracked with respect to both time axes, information may be represented using different equivalent forms. For example, consider the case that a contract is signed with a factory to manufacture some product during the period *[94, 96)*, and this fact is recorded in the database on *93-12*. Later, the contract is renegotiated, and its duration is changed to *[94, 95)*; the database is updated on *94-04*. The temporal data described above can be stored in the *Manufactured* relationship under many equivalent representations, e.g.

```
{(value: {Ref<factory>}, VT: [94, 96), TT: [93-12, 94-04)),
 (value: {Ref<factory>}, VT: [94, 95), TT: [94-04, UC))}          (1)
```

```
{{value: {Ref<factory>}, VT: [94, 95), TT: [93-12, UC)),
 (value: {Ref<factory>}, VT: [95, 96), TT: [93-12, 94-04))}       (2)
```

```
{{value: {Ref<factory>}, VT: [94, 95), TT: [93-12, 94-04)),
 (value: {Ref<Factory>}, VT: [94, 95), TT: [94-04, UC)),          (3)
 (value: {Ref<factory>}, VT: [95, 96), TT: [93-12, 94-04))}
```

Representation (1) has maximal valid timestamps, whereas representation (2) has maximal transaction timestamps. Representation (3) does not use maximal timestamps on either axis The different coverings of the temporal information, are depicted graphically in Figure 15.

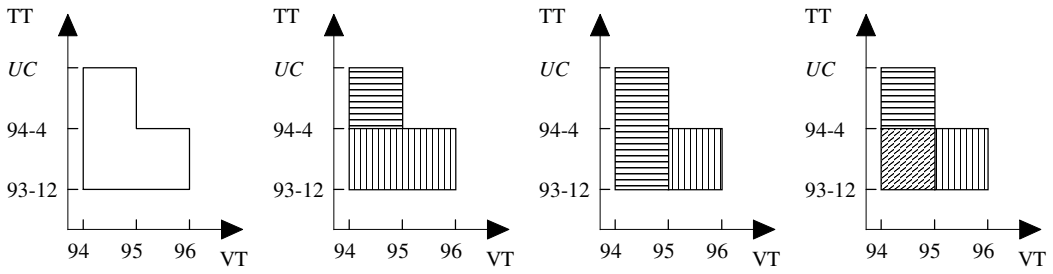

Figure 15 - Equivalent representations of temporal information

The data model may choose any of these representations to store temporal information into the database, but different queries may have different representation needs, in order to produce correct results. For example, consider the following queries:

1. Fetch the products for which contracts have ever been made to be manufactured by the factory for more than 18 months.
2. Fetch the products for which information about the production status during 1994 has remained constant for more than 6 months.

The first query can be expressed in TOQL as

```
select p
from Products as p
where exists m in p->Manufactured:
                duration(VALID(m)) > INTERVAL '18' MONTH
```

but it will only yield correct results if evaluated against representation (a) (indeed, if evaluated against representations (b) or (c), the product will not be selected). The second query can be expressed in TOQL as

```
select p
from Products as p
where exists m in p->Manufactured:
                begin(valid(m)) <= INSTANT '1994' YEAR and
                end(valid(m)) >= INSTANT '1995' and
                duration(transaction(m)) > INTERVAL '6' MONTH
```

The query will return correct results if evaluated against representation (b) (indeed, if evaluated against representations (a) and (c), the product will not be selected). Since TOQL cannot automatically determine which representation should be used, it is clear that the user should allow to choose whether maximal valid or maximal transaction timestamps will be used for the representation of temporal data in query processing. This facility is provided via a second restructuring operator, whose syntax is

`temporal_object(partition` *`time_axis`* `as period)`

where *time_axis* may be either *valid* or *transaction*. When the operator is applied to an object of type *T valid state G1 calendar C1 transaction* (*G1* is the granularity of the

valid timestamps), it modifies the variants of the temporal object so as to produce maximal times on the designated axis and produces a result of type *set<struct(Value: T, VT: period, TT: Period)>*. An element *(v_{res}, vt_{res}, tt_{res})* appears in the result set, if a variant with the same value and equal timestamps appears in the restructured form of the temporal object.

Using this restructuring operator, the two queries presented above may be stated as:

Query 1:

```
select p
from Products as p
where exists m in (p->Manufactured)(partition valid as period):
                duration(m->VT) > INTERVAL '18' MONTH
```

Query 2:

```
select p
from Products as p
where exists m in (p->Manufactured)(partition transaction as period):
                begin(m->VT) <= INSTANT '1994' YEAR and
                end(m->VT) >= INSTANT '1995' and
                duration(m->TT) > INTERVAL '6' MONTH
```

Syntactical changes are kept minimal between the initial and the final form of the query. Apart from the addition of the invocation of the restructuring operator, the only change is that variant timestamps are accessed as structure members, while in the original forms functions *valid* and *transaction* were used for the same purpose. This is necessary, as the type of the collection, on terms of which the variable in the *exists* clause is defined has been modified from *T valid state G1 calendar C1 transaction* to *set<struct(Value: T, VT: period, TT: period)>*. This modification is necessary, since, if the result of the partitioning operator were identical to the operand type, it would have to retain the data model settings for temporal information representation (in order to maintain consistency), thus restructuring would not be possible.

For symmetry reasons, the *(partition valid as period)* operator is applicable to valid state objects not allowing overlapping, and the *(partition transaction as period)* operator is applicable to transaction time objects. In these cases, the result schema is modified, so as each element of the result set does not include the field *TT* or the field *VT*, respectively. Since the data model will have probably chosen that objects with a single time dimension shall have maximal timestamps over this dimension, the operator will have no effect in such cases, apart from changing the type of the operand.

1.9 Aggregation

OQL v. 1.2 introduces the *group by* clause, which partitions the result of a *select/from/where* query into subsets, each one containing elements that have identical values in one or more properties. If the *group by* clause is used, the result schema of the query is modified so as to include one field for each one of the properties listed in the *group by* clause, indicating thus the values which are common to the members of the subset and one set-type field named *partition*, that contains the actual members of the subset.

For example the query

```
select p
from Products as p
group by p->Description like '*milk*' as IsMilk
```

selects all products from the *Products* extent and splits them into two subsets, depending on whether the product description contains the word *milk* or not. The elements of the first subset share the property that the expression *p->Description like '*milk*'* that is specified in the *group by* clause evaluates to *true* for each one of them, while the same expression evaluates to *false* for each member of the second subset. The query's result schema is *bag<struct(IsMilk: boolean, partition: bag<Product>)>*, where *partition* stores all the elements of the subset, whereas *IsMilk* stores the common value of the *group-by* expression for all members of the subset.

OQL allows for computation of *aggregate values* over the subsets produced using the *group by* clause. These values are computed using the *count, sum, avg, min* and *max* functions in the *select list*. For each such function used in the *select list*, the result schema is extended by one field, in order to accommodate the function's result. This result is evaluated by applying the function to the members of the corresponding subset. For example the query

```
select IsMilk, partition as TheProducts, count(partition) as NumProds
from Products as p
group by p->Description like '*milk*' as IsMilk
```

again partitions the objects in the *Product* extent into two subsets, depending on whether their description includes the word *milk* or not, but additionally computes the number of products in each subset and returns the tally into the *NumProducts* field of the result schema (the result schema is now is *bag<struct(IsMilk: boolean, TheProducts: bag<Product>, NumProducts: integer)>*).

Finally, after subset formulation has taken place, some subsets may be filtered out of the final result, depending on whether they satisfy a condition or not, using the *having* clause. The condition in the *having* clause is evaluated against the elements of each subset (the members of the *partition* field of each result structure) and the result structure appears in the result only if the condition evaluates to *true*. Thus the query

```
select IsMilk, partition as TheProducts, count(partition) as NumProds
from Products as p
group by p->Description like '*milk*' as IsMilk
having avg(select p1->LifeTime
           from partition as p1) > INTERVAL '12' DAY
```

partitions the objects in the *Products* extent as described above, but includes each subset in the final result only if the average current life time of the objects included in it exceeds 12 days.

Since TOQL operates on database schemata that contain temporal information, it should extend this functionality by allowing for partitioning of temporal information into subsets with respect to the time axis, as well as aggregate computation over the elements of each subset, and subset filtering. TOQL provides two types of temporal partitioning. The first type allows for splitting of a single temporal object into variant subsets, each pertaining to a specific portion of the time axis. The second partitioning method allows for combination of variants of multiple temporal objects that pertain to a specific portion of the time axis into a single set. These two types of partitioning, along with the methods for partition filtering and aggregate value computation are described in the following paragraphs.

1.9.1 Partitioning a single temporal object into variant subsets

The first type of partitioning splits a single temporal object into sets of variants, with each set containing the variants which pertain to a specific period. Partitioning may be performed either on the valid time or on the transaction time axis. The user specifies the desired time axis and an interval, which is used as the basic partitioning unit for the chosen time axis. For example an interval of 1 year specifies that the chosen time axis will be partitioned into segments with duration equal to one calendric year and variants will be included into some partition, if they contain information pertaining to the associated time segment.

Each basic partitioning unit may be extended towards the beginning and/or the end of the time axis by a specified duration, allowing thus for flexible time window

formulation. For example, the user may choose a basic partitioning unit of one year and an extension towards the end of the time axis equal to two years, in order to compute aggregates over three-yearly periods, with each period starting at the beginning of each year.

The syntax for invoking this form of partitioning is

```
TemporalObject(partition time_axis as interval_query
      [leading interval_query] [trailing interval_query] [calendar])
```

i.e. the temporal object is followed by a parenthesised expression which starts with the *partition* keyword and specifies the time axis on which the splitting operation will be performed (*valid* or *transaction*) and the basic partitioning unit. The expression contains two optional fields that may be used to specify the extension of the basic partitioning unit towards the beginning and the end of the time axis (the *leading* and *trailing* clauses, respectively). The time dimension specified in the partitioning expression must occur within the partitioned temporal object and the granularity and calendars of all interval queries must be identical, but not necessarily equal to the characteristics of the temporal object's partitioned time axis (e.g. the temporal object may have valid time semantics with granularity of *day*, whereas partitioning may be performed on a yearly basis). Finally, the *calendar* clause, if present, specifies that the starting point for the calculation of the time windows is the beginning of the calendar; if this clause is absent, the starting point is the smallest time instant occurring within the timestamps of the partitioned axis of the temporal object.

This form of aggregation is useful for producing analytical reports over periods, for information that is stored in a temporal object. For example the query "for all products, retrieve the name and the evolution of their life time for each calendric year" could be expressed in TOQL as

```
select p->ProductName as Name,
    (valid p->LifeTime)(partition valid as INTERVAL '1' YEAR calendar)
                                                    as YearlyLifeTime
from Products as p
```

The *YearlyLifeTime* field in this query's result schema is a set of structures, with each structure corresponding to a partition of the valid time axis with duration equal to one year. Each structure contains two member fields. The first member is named *TimeSlice*, its type is *Period* and contains the portion of the time axis to which the information of this structure pertains. The granularity and calendar of the *TimeSlice* field is derived from the respective characteristics of the timestamps in the partitioned

time axis. The second member is a set named *Partition*, and contains the values and the valid timestamp of the variants associated with the portion of the valid time axis stored in the respective *TimeSlice* field. The schema of the *Partition* member in this example thus is *bag<struct(Value: interval, VT: period)>* and the query result's schema is *bag<struct(Name: string, YearlyLifeTime: bag<struct(TimeSlice: period, Partition: bag<struct(Value: interval, VT: Period)>)>*. If the database contents are as described in section 1.3 and the query is evaluated on December 15, 1996, the result of the query will be:

```
{(Name: 'Life Orange Juice',
 YearlyLifeTime: {(TimeSlice: [1995-01-01, 1996-01-01),
                  Partition: {(Value: 18, VT: [1995-01-01, 1995-06-01)),
                              (Value:  9, VT: [1995-06-01, 1996-06-01))}),
                 (TimeSlice: [1996-01-01, 1997-01-01),
                  Partition: {(Value: 18, VT: [1995-06-01, 1996-06-01)),
                              (Value: 12, VT: [1996-06-01, forever))})}),
  (Name: 'Champion',
   YearlyLifeTime: {(TimeSlice: [1994-01-01, 1995-01-01),
                     Partition: {(Value:  6, VT: [1994-01-01, 1994-06-01)),
                                 (Value:  9, VT: [1994-06-01, 1995-01-01))}),
                    (TimeSlice: [1995-01-01, 1996-01-01),
                     Partition: {(Value: 12, VT: [1995-06-01, forever))})
                    (TimeSlice: [1996-01-01, 1997-01-01),
                     Partition: {(Value: 12, VT: [1996-06-01, forever)})})})}
```

In the general case, if *TO* is a temporal object, the schema of the *Partition* member resulting from a partitioning operation is a structure, whose fields are determined as follows:

- the structure always contains a field named *Value* whose type is identical to the temporal object's timestamped type.
- if *TO* has valid time semantics, the structure contains a field named *VT* that stores the variant's valid time. The type of the *VT* field is *Period*, if *TO* has valid state semantics; if, however, *TO* has valid event semantics, the type of the *VT* field is set to *Instant*. In all cases, the granularity and calendar of the *VT* field are identical to the respective characteristics of *TO*'s valid timestamps.
- if *TO* has transaction time semantics, the structure contains a field named *TT* which stores the variant's transaction time. The type of the *TT* field is always *Period* with the default calendar and granularity used by the system for the transaction time dimension.

In the above example, three additional characteristics of the partitioning operator are demonstrated. The first characteristic is that the distinguished value *forever* in valid timestamps is substituted by the value of the current timestamp for the purposes of

partitioning. Indeed, the special value *forever* is frequently used in valid timestamps having the "until changed" semantics, and not as actually designating the end of the calendar, so, in order to avoid computation of unnecessary (and numerous) partitions, this substitution is enforced. If the user wants to formulate partitions which extend to the end of the calendar, he/she must use explicitly the last timestamp in the calendar, instead of the distinguished timestamp *forever*.

The second characteristic of the partitioning procedure demonstrated in the example above is the handling of period-timestamped variants, whose timestamps are not fully contained within a single segment of the time axis. When variants are timestamped using periods, it is possible that some variant timestamp is not fully contained within a single portion of the time axis, as these portions are defined by the basic partitioning unit and the *leading*, *trailing* and *calendar* clauses. The way that these variants are handled depends on the semantics of the partitioned object:

- for objects having valid state semantics and not allowing overlapping, the value of the variant appears in every partition whose *TimeSlice* field overlaps with the variant's timestamp, since the information in such variants is applicable to subperiods and instants.
- for objects having valid state semantics and allowing overlapping, the variant does not appear in any partition, since the information in such variants is meaningless for instants and subperiods.
- for transaction time objects, the value of the variant appears in every partition whose *TimeSlice* field overlaps with the variant's timestamp.

Note that partial containment situations do not arise for objects with valid event semantics, since instants are either contained in some period, or they are not. In all cases, if the variant appears within some partition, it retains its original timestamp. We choose to keep the original timestamp, instead of using only the part of it that overlaps with the specific portion of the time axis for two reasons: firstly, if the intersection of the periods is used, there is no way for the user to reconstruct the original timestamp; if however, the original timestamp is used, the user may compute the intersection, by applying the *intersection* function to the timestamp and the respective *TimeSlice* field. Secondly, as stated above, the distinguished value *forever* is treated similarly to value *NOW* for the purposes of partition formulation, thus indications for its presence are not provided by the *TimeSlice* fields. If the intersection

of the periods is used, these indications are not provided by the individual timestamps either. Using thus the original timestamps does not result to any loss of information and allows computations to be made on either the original timestamp or the portion of the original timestamp associated with the respective time window.

Since the result of the partitioning operation is a set, it may be used in the *from* clause, so as to define a variable iterating over the members of the set. This variable may be used in conditions within the *where* clause, so as to facilitate subset filtering, or in expressions in the *Select*-list, in order to compute aggregate values over the variants. For example, the user may request an analytical report for the lifetimes of the products for each even year, which can be computed using the following query:

```
select p->ProductName as Name,
    (select variant
     from (valid p->LifeTime)(partition valid as INTERVAL '1' YEAR calendar)
                                                                as variant
     where year(begin(variant->TimeSlice)) mod 2 = 0) as YearlyLT
from Products as p
```

If the user wants to compute the average life time for each product over each year, without taking into account the duration that each life time applied within each year, he/she may issue the following query:

```
select p->ProductName as Name,
    (select variant->TimeSlice as TimeSlice,
            avg(select LT->Value
                from variant->Partition as LT) as AvgLifeTime
    from (valid p->LifeTime)(partition valid as INTERVAL '1' YEAR calendar)
                                                    as variant) as YearlyLt
from Products as p
```

Finally, the basic partitioning unit, as well as the interval expressions in the *leading* and *trailing* clauses need not be constant for all objects. For example, if the declaration of class *Product* includes a field *ReportingPeriod* of type *Interval*, the query

```
select p->ProductName as Name,
       (valid p->LifeTime)(partition valid as p->ReportingPeriod calendar)
                                                              as ReportLT
from Products as p
```

can be issued to produce analytical reports for history of the lifetime for products in the database, using a potentially different reporting period for each product. For instance, the reporting period for the “Life Orange Juice” product may be set to 1 year, whereas the reporting period for the “Champion” product could be set to 15 months.

1.9.2 Combining variants from multiple temporal objects

The second type of partitioning allows for combination of variants from multiple temporal objects into groups, with each group pertaining to a specific portion of either the valid or the transaction time axis. Groups may be filtered, depending on whether they satisfy some condition, and aggregate values may be computed over the elements of each group. This form of partitioning is provided via a special form of the *group by* clause, in which the grouping expression is a time dimension (valid time or transaction time), with an associated basic partitioning unit and, optionally, unit extensions towards the beginning or the end of the time axis. Group filtering and aggregate value computation is performed using the standard OQL mechanisms, i.e. the *where* clause and aggregate functions, respectively. The alternative syntax of the *group by* clause is

```
group by time_axis interval_query [leading interval_query]
         [trailing interval_query] as identifier
```

with *time_axis* being either *valid* or *transaction*. This form of the *group by* clause is available only when the *from* clause defines a single variable iterating over a set of temporal objects. The time dimension designated in the *group by* clause must occur in the objects resulting from the *select/from/where* query. The granularity and calendars of all interval queries must be identical; the granularity must not necessarily match the characteristics of the temporal object's partitioned time axis (e.g. the temporal objects may have valid time semantics with granularity of *day*, whereas partitioning may be performed on a yearly basis). If the partitioning expression is *time_axis interval_query [optional_time_windows] as identifier*, then the result of the grouping is *set<struct(identifier: period, Partition: bag<struct(Value: T, VT: period, TT: period)>)>*, with *T* being the timestamped type returned by the *from* clause. (One of the fields *VT* and *TT* may be missing, if the *from* clause defines a variable iterating over a collection of transaction time objects or a collection of valid time objects, respectively. If the *from* clause defines a variable iterating over a collection of valid event objects, the *VT* field will be of type *Instant*, instead of *Period*.)

Each element of the returned collection contains the information pertaining to a specific portion of the time axis. Variants that occur in any member of the set defined in the *from* clause and contain information pertaining to the specific period of the designated time axis, appear as elements in the *Partition* field of the corresponding

structure. After grouping has been performed, the *having* clause is applied (if present) to eliminate groups not meeting some user-specified condition and, finally, the *select*-list is evaluated. Both the *having* clause and the expressions in the *select*-list may reference the fields *identifier* and *Partition*. Handling of the distinguished value *forever* in valid timestamps and behaviour in the presence of variants whose timestamps are partially contained within a time window are as described in paragraph 1.9.1.

For example, the user may want to compute the average yearly lifetime of fresh products, which can be evaluated using the following query:

```
select TimeSlice, avg(select x from Partition as x) as Average
from (select valid p->LifeTime from Products as p) as LT
group by valid INTERVAL '1' YEAR as TimeSlice
```

The *group by* clause causes the restructuring of the collection used in the *from* clause, so as to create groups, each one containing two fields. The first field is named *TimeSlice* (as specified in the *group by* clause) and designates the portion of the time axis with which this structure is associated. The second field is named *Partition* (which is a system-generated name) and contains all the variants that occur in any of the valid time objects which belong to the bag defined in the *from* clause, and have a valid timestamp which overlaps with the respective portion of the time axis. For each such group, the *select*-list is evaluated, so as to produce the final query result.

If the user wanted to report only on products whose current life time exceeds 10 days, the following query could be issued:

```
select TimeSlice, avg(select x from Partition as x) as Average
from (select valid p->LifeTime from Products as p) as LT
where LT[INSTANT 'NOW' DAY] > INTERVAL '10' DAY
group by valid INTERVAL '1' YEAR as TimeSlice
```

As in standard OQL, the *where* clause is applied before the *group by* clause, so the undesired temporal objects are filtered out before grouping is performed.

If the user wanted to report only on periods that start on an odd year, the following query could be issued:

```
select TimeSlice, avg(select x from Partition as x)
from (select valid p->LifeTime from Products as p) as LT
group by valid INTERVAL '1' YEAR as TimeSlice
having year(begin(TimeSlice)) mod 2 = 1
```

As in standard OQL, the *having* clause is applied after grouping is performed.

Finally, the basic partitioning unit as well as the interval expressions in the *leading* and *trailing* clauses must be identical for all elements of the collection specified in the

from clause, thus only interval literals are allowed in the alternative form of the *group by* clause and not generic queries.

1.10 TOQL and User Requirements

TOQL covers all temporal needs that have been identified in the user requirements analysis phase and reported in [TR22TR.1]. The specific needs are listed in the following figure. Where appropriate, a brief description is given on how TOQL meets the specific needs.

User requirements needs	Supported?	Comments
Temporal dimension		
valid time	✓	
transaction time	✓	
Typology of objects		
snapshot objects	✓	
valid time objects	✓	
transaction time objects	✓	
bitemporal objects	✓	
Temporal structure (reasoning)		
temporal point	✓	Operations provided on instants
period	✓	Operations provided on periods
set of temporal points	✓	Standard OQL bag/set operators
set of periods	✓	Period sets
Granularity and calendar		
calendar	✓	Gregorian. Arbitrary user-defined calendars may be used.
granularities	✓	Year, month, day, hour, minute, second in the Gregorian. Arbitrary in user-defined calendars.
Temporal queries		
comparisons of dates	✓	Operators on instants
comparison and construction of periods	✓	Operators on periods and constructor functions
mapping of dates to intervals	✓	Operators on instants; implicit conversion of instants to periods
calculation of intervals	✓	Via operators and constructor functions
relative time manipulation	✓	Relative timestamps
conversion of snapshot objects to valid time, transaction time and bitemporal	✓	*valid*, *transaction* and *bitemporal* modifiers
extraction of an object's valid time and transaction time	✓	Functions *valid* and *transaction*
construction of the valid time related to a result object	✓	Via the *valid* and *bitemporal* modifiers
selection of values according to a period	✓	Subscript operators and object variant referencing
selection of periods satisfying a criterion	✓	Object variant referencing
selection of a valid time object from a valid time object	✓	Subscript operators
temporal grouping	✓	Temporal objects are temporally grouped
temporal ordering selection	✓	Sorting on instants and periods
aggregate functions	✓	Standard OQL aggregation and temporal forms of aggregation
extrapolation or calculation of values according to a period		The application needs to define its own extrapolation policy
temporal modal operators	✓	The standard OQL *exists* and *for all* predicates may be applied to object variants

Continued from the previous page

User requirements needs Supported? Comments

Continued on the next page

Temporal queries		
conversion of valid time objects to snapshot, transaction or bitemporal	✓	*snapshot*, *transaction* and *bitemporal* modifiers
conversion of transaction time objects to snapshot, valid or bitemporal	✓	*snapshot*, *valid* and *bitemporal* modifiers
conversion of bitemporal objects to snapshot, valid and transaction time	✓	*snapshot*, *valid* and *transaction* modifiers

Figure 16 - TOQL capabilities against user requirements

TOQL provides additional features which were not identified in the user requirements analysis phase, in order to be temporally complete. Examples of these features are temporal joins, the *leading* and *trailing* clauses in aggregation etc.

2. Design

2.1 Introduction

In this section, the design of the TOQL query processor is presented. The following goals were adopted in the design of the processor:

1. *full temporal functionality should be delivered.*

   The developed TOQL processor should provide all the temporal characteristics described in the specification of TOQL.

2. *full snapshot DBMS compatibility*.

   The TOQL processor must accept "plain" OQL queries and evaluate them, in order to provide the desired level of compatibility for snapshot applications.

3. *performance*.

   High performance is a crucial non-functional requirement for applications and the TOQL processor must contribute to this direction. Additionally, it is desirable that snapshot queries on snapshot data are not penalised with any overhead.

4. *portability*.

   The implementation of the TOQL processor should not be bound to specific characteristics of the underlying system; rather it should be portable to every ODMG-compliant object oriented DBMS.

2.2 Architecture for the TOQL Processor

The TOQL processor will be implemented as a software module functioning on top of the OQL processor of O_2. TOQL queries will be intercepted and transformed to OQL queries, which will be submitted to the OQL processor for evaluation. The results returned by the OQL processor will be forwarded to the user, or the application which has issued the TOQL query. This scheme is illustrated in Figure 17.

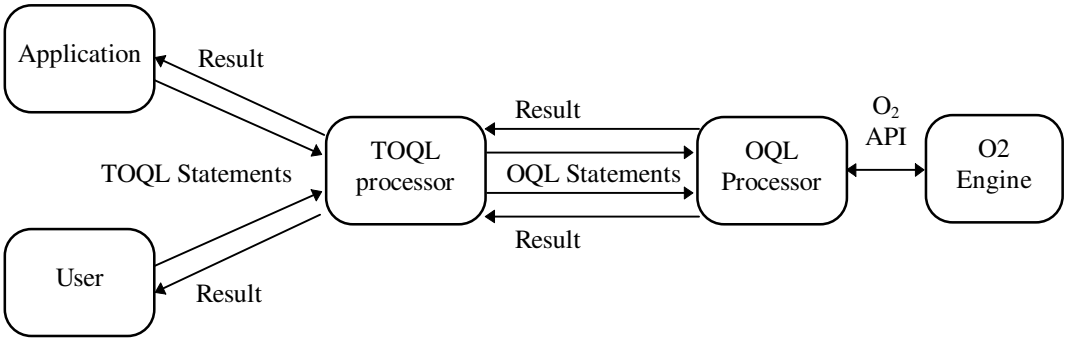

Figure 17 - TOQL processor architecture

The TOQL processor itself is further subdivided into smaller software modules, with each one of them being responsible for one stage of the processing of the query, as illustrated in Figure 18. These modules are described in the following paragraphs.

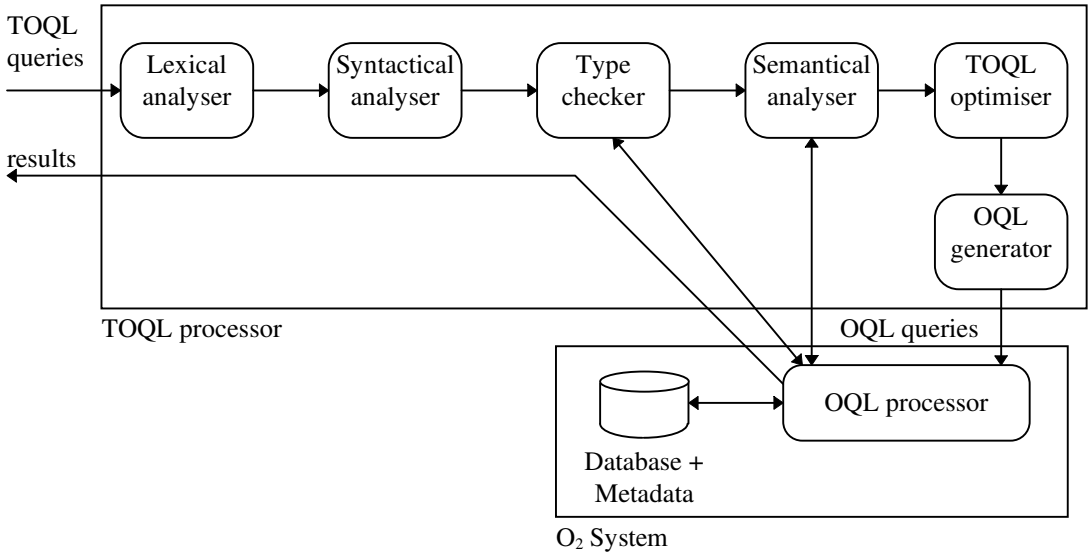

Figure 18 - The TOQL processor modules

1. the *lexical analyser* processes the query which is submitted to the TOQL processor and splits it to *lexical tokens*, with each one of them corresponding to an element of TOQL. If the query contains a substring which is not a valid element of TOQL, an error condition is raised and further processing is aborted.
2. the *syntax checker* reads the stream of *lexical tokens* produced by the lexical analyser and checks whether this forms a valid TOQL query. The syntactical analyser generates a *syntax tree*, which describes the structure of the TOQL query.

If the query is not found to conform to the TOQL syntax rules, an error is reported and query processing is terminated.

3. the *type checker* traverses the syntax tree, checking whether the operation designated in the query are legal, with respect to the types of their arguments. In order to perform this task, the *type checker* must be able to access information about the objects stored in the database, as well as the method signatures. These data are obtained by querying the database metadata, which are maintained by the TODL processor and stored in the O_2 system.

   Additionally, the *type checker* arranges for the insertion of the appropriate type conversions (as described in paragraph 1.2.5), when they are not directly supported by the data model. If an operation with illegal argument types is designated (the types of the arguments to the operation do not match one of the legal signatures neither directly, nor after the application of type conversion rules), the *type checker* module raises an error and the processing of the TOQL query is aborted.
4. the *semantical analyser* module determines the actual operations that must be invoked, in order to evaluate the query. These operations cannot always be directly determined from the syntax rules or the type checking but rather from the overall query structure. For example, if a valid time instance property appears in the *select*-list and is not preceded by the keyword *valid*, then, according to the TOQL specification, only its current value (as opposed to its full history) should be retrieved. The *semantical analyser* will arrange for the invocation of the appropriate method which will extract the current value of the object. The data which are necessary to perform the semantical analysis are obtained by querying the database metadata.
5. the TOQL optimiser arranges for the elimination of redundant operations (such as coalescing a valid time object which is already coalesced) or the compacting of operations wherever possible (e.g. two consecutive type conversions -implicit or explicit- on the same datum may be replaced by a single one).
6. the *OQL generator* module processes the syntax tree and the data structures produced by the *type checker* and the *semantical analyser* modules and formulates an OQL query, which will yield the desired results.

This design scheme enables the fulfilment of the goals outlined in section 2.1. Full temporal support can be achieved by query transformations and method invocations,

which are described in more detail in paragraph 2.3. Compatibility with the snapshot DBMS (O_2) is provided, since TOQL is a superset of OQL, thus every statement that is legal in OQL, is also legal in TOQL. Additionally, TOQL has been designed to provide the maximum degree of compatibility for snapshot applications, even if they operate on a temporally enriched database schema (for more details, see Appendix C). The adopted two-layer schema does not imply poor performance. Admittedly, this approach introduces one new function (query rewriting) and leads to the duplication of three other functions, namely lexical analysis, syntax analysis and type checking, which will be carried out by both the TOQL query processor and OQL query processor. (Semantical analysis and optimisation performed by the TOQL query processor do not overlap with the corresponding actions taken by the OQL processor, since they focus on different aspects; for example, the optimiser of the TOQL processor handles the temporal aspects of the operations, whereas the optimiser of the OQL processor focuses on indexes and physical storage of data.) However, the overall overhead imposed by this duplication is negligible: measurements performed using the ORES parser ([ORES94]) which performed a similar task, showed that the time needed for the lexical analysis, syntax analysis and query rewriting stages of SQL queries ranges from 0.003 seconds for simple *select-from-where* queries, up to 0.17 seconds for huge, multi-line queries (the query which needed 0.17 seconds for these stages was 110 lines long and was sized over 2 KBytes; queries of this size are not really common). In all cases, the time needed for these three stages does not exceed the 0.08% of the overall query execution time. These measurements are graphically depicted in the following two diagrams.

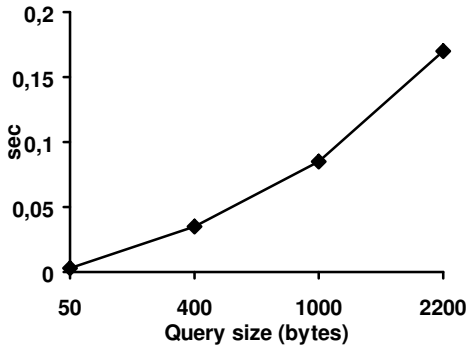

Figure 19 - Overhead introduced for lexical analysis, syntax analysis and rewriting

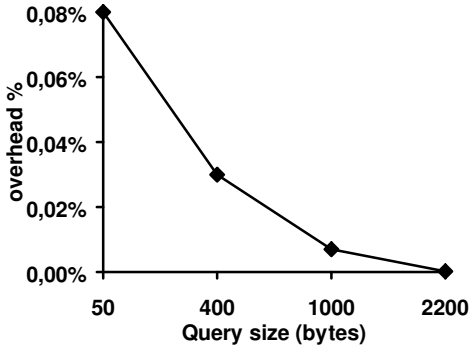

Figure 20 - Percentage of the introduced overhead to overall query execution time

The ORES implementation did not contain a distinct type checker, so the overhead imposed by the type checker could not be measured; however it is estimated that the overhead imposed by the type checker should remain low, and it will not exceed the sum of the overheads imposed by the other modules.

Another aspect that usually affects the performance of extensions implemented outside some DBMS is that processing must take place both in the server side and the client side, necessitating thus the need for numerous data transfers between the client and server; however in the O_2 system, processing takes place on the client side only, so data are transferred once from the O_2 server to the client, without imposing any overheads.

Moreover, the two-layer scheme guarantees that snapshot applications operating on snapshot data do not pay a performance penalty, because of the introduction of temporal support: these applications may submit their queries directly to the O_2 system's OQL processor, so their performance will not be affected.

Finally, implementing the TOQL processor on top of the O_2 database system allows it to be easily ported on top of any ODMG compliant database system, since it does not use any special features of O_2. This would be impossible if the TOQL processor was integrated into the O_2 system, since the TOQL processor would be tightly bound to special characteristics of the O_2 system.

2.3 Query Processing

As stated in the previous section, the TOQL processor analyses the queries submitted to it and transforms them to equivalent OQL queries, which are subsequently fed into the O_2 system's OQL processor. In the following sections, the syntactical transformations employed by the TOQL processor and the auxiliary methods which are used within the rewritten queries are presented.

2.3.1 Syntactical transformations

A. Literals

Each interval, instant, period or period set literal is transformed to an invocation of the corresponding class's constructor functions. The TOQL processor splits up the literal specification to the corresponding value and granularity fields, calls the constructor of the corresponding class, so as to produce a new object. Period set literals, which are

specified via a constructor function (*period_set*) are transformed to an invocation of the corresponding class's constructor; in this case, however, the periods comprising the period set are grouped in a set, before they are passed to the constructor, in order to avoid variable length argument lists. Examples of the transformations employed for literal support are illustrated in the following figure.

TOQL syntax	Transformed syntax
`INTERVAL '5' DAY`	`interval()->set_value('5', 'day')`
`INSTANT '1990-01' MONTH`	`instant()->set_value('1990-01', 'month')`
`PERIOD '[1990, 1991)' YEAR`	`period()->set_value('[1990, 1991)', 'year')`
`period_set('[1990, 1991)', '[1993, 1995)') YEAR`	`period_set()->set_value( set('[1990, 1991)','[1993, 1995)'), 'year')`

Figure 21 - Transformations for literal support

Note that the transformations described above are subject to change, in order to match exactly the internal structure of the instant, interval, period and period set objects.

B. Functions

Since the O_2 system allows only class member functions to be used in OQL, the functions described in section 1.2.2 will be implemented as member functions of the corresponding classes and the TOQL processor will arrange for the transformation of the function call syntax to a member function invocation. The first argument of the function will always be the object through which the member function will be invoked. The remaining arguments (if any) will be passed as parameters to the member function.

The *CAST* syntactic construct is also mapped to a member function invocation: the datum following the *CAST* keyword will be the object through which the member function will be invoked and the target granularity will be used as an argument to the member function invocation. Examples of the transformations used for function calling and explicit granularity casting support are illustrated in the following figure.

TOQL syntax	Transformed syntax
`begin(some_period)`	`(some_period)->begin`
`month(some_instant)`	`(some_instant)->month`
`merge(period1, period2)`	`(period1)->merge(period2)`
`CAST some_period TO DAY`	`(some_period)->cast('day')`

Figure 22 - Transformations for function support

In the transformed syntax, the object through which the member function is invoked is enclosed in parentheses, in order to avoid operator precedence problems.

C. Predicates

Within the data model, predicates will be implemented as member functions to the corresponding classes. The TOQL processor will transform the infix predicate notation *operand1 predicate operand2* to a member function invocation construct. In these transformations, the left operand will be the object through which the member function will be invoked, whereas the right operand will be passed as a parameter to the member function. Examples of the transformations used for the implementation of predicates are illustrated in the following figure.

TOQL syntax	Transformed syntax
`instant1 precedes instant2`	`(instant1)->precedes_inst(instant2)`
`period1 overlaps period2`	`(period1)->overlaps_per(period2)`
`period_set1 contains period_set2`	`(period_set1)->` `overlaps_ps(period_set2)`

Figure 23 - Transformations for predicate support

In the transformed syntax, the object through which the member function is invoked is enclosed in parentheses, in order to avoid operator precedence problems.

The same transformation technique will be used for implementing standard comparison operators (i.e. >, >=, < and <=) for classes *instant* and *interval*. Each of these operators will have a corresponding member function (e.g. `greater`, `greater_equal`, `less` and `less_equal`, respectively), and the TOQL processor will transform comparisons via standard operators to invocations of these member functions, as depicted in the following figure.

TOQL syntax	Transformed syntax
`instant1 > instant2`	`(instant1)->greater(instant2)`
`interval1 <= interval2`	`(interval1)->less_equal(interval2)`

Figure 24 - Transformations for standard comparison operators support

These transformations are required because the O_2 system does not allow usage of user-defined C++ operators in OQL queries.

D. Operators

TOQL introduces new operators on intervals, instants, periods and period sets, as described in paragraph 1.2.4. However, since the O_2 system disallows the usage of

user-defined C++ operators in OQL queries, these operators will be implemented as member functions within the corresponding classes and the TOQL processor will arrange for the transformation of the infix operator syntax to the equivalent member function invocation.

For unary operators (unary plus and unary minus), the operand will be the object through which the member function will be invoked, and no additional parameters will be supplied in this function invocation. For binary operators, the left operand will be the object through which the member function will be invoked, whereas the right operand will be a parameter to this invocation. An exception to this rule is the operator `number * interval`: in the transformed version of this operator, the right operand (i.e. the interval) is the object through which the member function is invoked, whereas the left operand (i.e. the number) is passed as a parameter to this method invocation. Examples of the transformations used for the implementation of operators are illustrated in the following figure.

TOQL syntax	Transformed syntax
`- some_interval`	`(someinterval)->uminus`
`interval1 + interval2`	`(interval1)->plus_int(interval2)`
`instant1 - interval1`	`(instant1)->bminus_int(interval1)`
`number1 * interval1`	`(interval1)->multiply(number1)`

Figure 25 - Transformations for operator support

In the transformed syntax, the object through which the member function is invoked is enclosed in parentheses, in order to avoid operator precedence problems.

E. Type conversions

When an implicit type conversion is needed, the TOQL processor implements it by calling a member function of the object that must be converted. Periods require only one such function, which allows for their conversion to period sets and instants have one member function that converts an instant to a period. Conversions from instants to period sets are implemented by first converting the instant to a period and subsequently converting the result to a period set. If instants have a member function that directly converts instants to period sets, the TOQL optimiser will replace the two successive member function calls by a single call to this member function. Examples of the transformations used for the implementation of implicit transformations are illustrated in the following figure.

TOQL syntax	Transformed syntax
`instant1 + period1`	`((instant1)->to_period)->plus(period1)`
`period_set1 + period1`	`(period_set1)->plus(period1->to_period_set)`
`period_set1 + instant1`	`(period_set1)->plus(instant1->to_period->to_period_set)`

Figure 26 - Transformations for implicit conversion support

F. Retrieval of temporal instance properties

When a temporal instance property is used in a query and is not preceded by any of the *valid*, *transaction* and *bitemporal* keywords, it is automatically converted to the corresponding snapshot value. This is achieved by replacing the reference to the temporal property with a member function call, which returns the current value of the temporal object (i.e. the variant whose valid and transaction timestamps overlap with the current instant). For example, if the query

```
select p->ProductName as ProductName,
       p->LifeTime as LifeTime,
       p->Ingredients as Ingredients
from Products as p
```

is submitted against the database schema described in section 1.3, it will be rewritten by the TOQL processor as

```
select p->ProductName as ProductName,
       (p->LifeTime->current_value) LifeTime,
       (p->Ingredients->current_value) as Ingredients
from Products as p
```

Analogous transformations are used in relationships, too.

When temporal properties are preceded with the corresponding *valid*, *transaction* or *bitemporal* keyword, this transformation is not applied; in these cases, the TOQL processor simply drops the preceding keyword.

The TOQL processor uses additional transformations when the *count*, *first* and *last* functions or the subscript operator are used with temporal instance properties. Functions *count*, *first* and *last* are transformed to member function calls which accept no parameters, as shown in the following figure.

TOQL syntax	Transformed syntax
`count(bitemporal bitemp_obj)`	`(bitemp_obj)->count`
`first(valid valid_obj)`	`(valid_obj)->first`
`last(transaction trans_obj)`	`(trans_obj)->last`

Figure 27 - Transformations used for supporting functions *count*, *first* and *last* for temporal objects

Subscript operators are also mapped to calls to member functions. The temporal instance property on which the subscript is applied, is used as the object through which the member function is invoked, whereas the subscripts are passed as parameters to the member function. For valid time and bitemporal objects that allow overlapping of valid timestamps, the usage of the DISTINCT specifier results to the invocation of a different member function. Examples of the transformations employed to implement the subscript operators are shown in the following figure.

TOQL syntax	Transformed syntax
`valid_obj[n1]`	`(valid_obj)->get_value_num(n1)->value`
`trans_obj[n1:n2]`	`(trans_obj)->get_value_2num(n1, n2)`
`valid_obj[instant1]`	`(valid_obj)->get_value_inst(instant1) ->value`
`trans_obj[period1]`	`(trans_obj)->get_value_per(period1)`
`o_valid_obj[DISTINCT period1]`	`(o_valid_obj)->get_value_dper(period1) ->value`
`bitemp_obj[VALID AT instant1]`	`(bitemp_obj)->get_valid_hist_inst (instant1)`
`bitemp_obj[CURRENT AT period1]`	`(bitemp_obj)->get_trans_hist_per (period1)`
`bitemp_obj[VALID AT instant1, CURRENT AT instant2]`	`(bitemp_obj)->get_hist_2inst(instant1, instant2)`
`o_bitemp_obj[CURRENT AT period1, VALID AT DISTINCT period2]`	`(o_bitemp_obj)->get_dvalue_per(period2, period1)`

Figure 28 - Transformations for subscript support

Analogous transformations are applied when the subscripted expression is followed by one of the specifiers *deleted* or *all*. The presence of any of these specifiers results to the invocation of a different member function, which returns the desired variants in its result (the variants tagged as *deleted* or all variants, respectively).

The results of subscript operators returning a single variant can be used, according to the specifications of TOQL, as arguments to functions *valid* and *transaction*, in order to access the corresponding timestamps. When such a function reference is encountered, it is modified by the TOQL query processor to a call to the corresponding function, followed by an access to the valid timestamp or the transaction timestamp field. Examples of the transformations employed to implement the *valid* and *transaction* functions for subscript operations are given in the following figure.

TOQL syntax	Transformed syntax
`valid(valid_obj[n1])`	`(valid_obj)->get_value_num(n1)->VT`
`transaction(trans_obj[instant1])`	`(trans_obj)->get_value_inst(instant1) ->TT`
`valid(o_valid_obj[DISTINCT period1])`	`(o_valid_obj)->get_dvalue_per(period1) ->VT`
`valid(bitemp_obj[CURRENT AT instant1, VALID AT instant2])`	`(bitemp_obj)->get_dvalue(instant2, instant1)->VT`

Figure 29 - Transformations for supporting functions *valid* and *transaction* on the results of subscript operators

Modifiers *weighted valid* and *weighted transaction* are also handled by syntactical transformations. When such a modifier is used, the TOQL processor modifies it to a piece of OQL code which accesses the variant's value and multiplies it with the duration of the associated timestamp. Examples of this procedure are illustrated in the following figure.

TOQL syntax	Transformed syntax
`weighted valid_obj[n1]`	`(valid_obj->get_value_num(n1)->value * duration(valid_obj->get_value_num(n1)->VT))`
`weighted trans_obj[inst1]`	`(trans_obj->get_value_inst(inst1)->value * duration(trans_obj->get_value_inst(inst1) ->TT)`

Figure 30 - Transformations for supporting functions *valid* and *transaction* on the results of subscript operators

G. Extracting object states

Object state extraction is handled similarly to the application of subscript operations. Each request to extract states of some object possessing temporal semantics as a whole using the square bracket operator (`[]`), is mapped to a call to a member function of the respective object. The object whose variants are extracted is used as the object through which the member function is invoked, whereas the subscripts are passed as parameters to the member function. For valid time and bitemporal objects that allow overlapping of valid timestamps, the usage of the *distinct* specifier results to the invocation of a different member function. Examples of the transformations employed to implement the subscript operators are shown in the following figure.

TOQL syntax	Transformed syntax
`valid_obj[n1]`	`valid_obj->get_state_num(n1)->value`
`trans_obj[n1:n2]`	`trans_obj->get_state_2num(n1, n2)`
`valid_obj[instant1]`	`valid_obj->get_value_inst(instant1) ->value`
`trans_obj[period1]`	`trans_obj->get_value_per(period1)`
`o_valid_obj[DISTINCT period1]`	`o_valid_obj->get_dvalue_per(period1) ->value`
`bitemp_obj[VALID AT instant1]`	`bitemp_obj->get_valid_hist_inst (instant1)`
`bitemp_obj[CURRENT AT period1]`	`bitemp_obj->get_trans_hist_per (period1)`
`bitemp_obj[VALID AT instant1, CURRENT AT instant2]`	`bitemp_obj->get_hist_inst(instant1, instant2)`
`o_bitemp_obj[CURRENT AT period1, VALID AT DISTINCT period2]`	`o_bitemp_obj->get_dvalue_per(period2, period1)`

Figure 31 - Transformations for object state extraction

H. Referencing object variants

A query referencing object variants is transformed by the TOQL processor to a query using standard OQL constructs and features of the temporally enhanced data model. If a variable *var* is defined in a *from* clause to iterate over the variants of a temporal instance property *t_prop*, the TOQL processor applies the following transformations:

1. the temporal instance property in the *from* clause is substituted by a call to the data model method which returns the history ([TR31TR.1]) of the temporal property. This method returns the history as a set of structures, with each structure containing the value of the variant its valid timestamp and/or its transaction timestamp, depending on whether the temporal instance property has valid time semantics, transaction time semantics, or both. These three fields of these structures will be denoted in the following as *value*, *VT* and *TT*, respectively.

   Since the result of this method invocation is a *set*, OQL allows the definition of a variable which iterates over the elements of this set.
2. each reference to the variable *var* within the scope of the *from* clause that does not occur within functions *valid* and *transaction* is substituted by the notation *var->value*.
3. each occurrence of the function *valid(var)* within the scope of the *from* clause is replaced by the notation *var->VT*.
4. each occurrence of the function *transaction(var)* within the scope of the *from* clause is replaced by the notation *var->TT*.

According to these transformation rules, the query

```
select xLifeTime as LifeTime, valid(xLifeTime) as When
from Products as p, valid p->LifeTime as xLifeTime
where p->ProductName = "Life Orange Juice"
  and xLifeTime > INTERVAL '12' DAY
```

which selects the periods during which the life time of the product “Life Orange Juice” was longer than 12 days along with actual life time, will be rewritten by the TOQL processor as

```
select xLifeTime->value as LifeTime, xLifeTime->VT as when
from Products as p, (p->LifeTime).get_history as xLifeTime
where p->ProductName = "Life Orange Juice"
  and (xLifeTime->VT).greater(interval().set_value('12', 'day'))
```

I. Conversion between temporal and snapshot values

TOQL provides four modifiers (*snapshot*, *valid*, *transaction* and *bitemporal*) for converting between temporal and snapshot values. The syntactical transformations employed in order to implement these transformations are described in the following paragraphs.

The *snapshot* modifier

The *snapshot* modifier is implemented by retrieving the full history of the temporal instance property using the appropriate data model method and subsequently selecting the *value* field from the structures in the resulting set. For example, the query

```
select p->ProductName as ProductName,
       max(snapshot p->LifeTime) as MaxLife
from Products as p
```

is transformed to

```
select p->ProductName as ProductName,
       max(select toql_var001->value
           from (p->LifeTime).get_history as toql_var001) as MaxLife
from Products as p
```

The *valid* modifier

In order to evaluate the *valid* modifier, the TOQL processor first identifies the type returned by the query which is enclosed in the brackets. This type, along with the valid type specification (*state* or *event*) and the presence of the *overlaps* designation are used to determine the type which is returned by the *valid* modifier. The TOQL processor arranges for a call to the target type’s class constructor, using query enclosed in the brackets, together with the expression following the colon (**:**) to formulate an argument to it. An example of the overall transformation is depicted below. The query

```
valid state granularity day [select p->productName
                             from Products as p:
                             period '[1990-01-01, 1997-01-01)' DAY]
```

is transformed to

```
string_vstate_day().set_value(select p->ProductName as Value,
          period().set_value('[1990-01-01, 1997-01-01)', 'day' as VT)
          from Products as p)
```

(In this example we assume that the name of the class used to store data of type *string valid state granularity day* is *string_vstate_day*.)

The *transaction* modifier

In order to evaluate the *transaction* modifier, the TOQL processor first identifies the type returned by the query which is enclosed in the brackets. The TOQL processor arranges for a call to the target type's class constructor, using query enclosed in the brackets to formulate an argument to it. An example of the overall transformation is depicted below. The query

```
transaction [select p->productName from Products as p
             where p->productName = 'Life Orange Juice']
```

is transformed to

```
string_trans().set_value(select p->productName
                         from Products as p
                         where p->productName = 'Life Orange Juice')
```

(In this example we assume that the name of the class used to store data of type *string transaction* is *string_trans*.)

The timestamp of the variant is always set to *[NOW(), UC)*, and this is handled by the *set_value* method, thus the TOQL processor does not pass to the class constructor any information on the timestamp to be used.

The *bitemporal* modifier

The TOQL processor handles the *bitemporal* modifier similarly to the *valid* modifier. Only the target class is changed, so as to select a class that holds bitemporal data, instead of valid time data.

J. Temporal joins

Temporal joins are implemented through query rewriting. The TOQL processor initially determines the result schema, based on the schemata of the joined queries and the identifiers enclosed in the parentheses following the ***tstruct*** keyword (which are used for naming the fields in the resulting temporal object). The TOQL processor arranges for a call to the constructor of the result class, followed by a call to a *set_value* function, using an appropriate argument. This argument is calculated by declaring two OQL variables that iterate over the variants of the two temporal objects

and uses a *where* clause to filter out the pairs of variants whose corresponding timestamps do not fulfil the criteria that pertain to the specific type of join. For example, the query

```
select f->owner as Owner, f->Location as Location,
       tstruct(Manager: valid f->Manager,
                Turnover: valid f->Turnover) as Info
from Factories as f
```

is rewritten as

```
select f->owner as Owner, f->Location as Location,
       string_int_valid_state().set_value(
         select toql_var001->value as Manager,
                toql_var002->value as TurnOver) as value,
              (toql_var001->VT)->intersection(toql_var002->VT) as VT
         from (f->Manager).get_full_history as toql_var001,
              (f->Turnover).get_full_history as toql_var002
         where (toql_var001->VT)->overlaps(toql_var002->VT)) as Info
from Factories as f
```

(string_int_valid_state is used in this example as the name of a class that stores valid state information, with each variant being a structure with two fields; the first field is of type *string* and the second field of type *integer*.)

If the user provides an explicit cast for the result of the temporal join, the TOQL processor determines the associated snapshot type by querying the database metadata, and arranges for converting the value part of the elements of the argument of the *set_value* method to this snapshot type. For example, if *MgrProfit* is a valid time class containing two member fields *Manager* and *Turnover* and *MgrProfitSnapshot* is the respective snapshot class generated by the TODL processor, the query

```
select f->owner, Location: f->Location as Owner,
       (MgrProfit)tstruct(Manager: valid f->Manager,
                          Turnover: valid f->Turnover) as Info
from Factories as f
```

is rewritten as

```
select f->owner as Owner, f->Location as Location,
       MgrProfit().set_value(
         select MgrProfitSnapshot(Manager: toql_var001->value,
                                  Turnover: toql_var002->value) as value,
              (toql_var001->VT)->intersection(toql_var002->VT) as VT
         from (f->Manager).get_full_history as toql_var001,
              (f->Turnover).get_full_history as toql_var002
         where (toql_var001->VT)->overlaps(toql_var002->VT)) as Info
from Factories as f
```

K. Restructuring operators

Restructuring operators are implemented via calls to auxiliary methods. Creation of instant-timestamped information out of period-timestamped information is facilitated by the *restructure_valid_instant* method, which is supported by valid state objects not

allowing overlapping and bitemporal state objects not allowing overlapping, and the *restructure_transaction_instant* method, which is supported by transaction time objects and bitemporal objects. These methods do not accept any arguments and return a set containing structures that correspond to instant-timestamped variants on the valid time axis or the transaction-time axis, respectively.

Formulation of maximal timestamps on the valid time axis is facilitated by the *restructure_valid_period* method, which is supported by valid state objects not allowing overlapping and bitemporal state objects not allowing overlapping, and the *restructure_transaction_period* method, which is supported by transaction time objects and bitemporal objects. These methods do not accept any arguments, and return a set containing structures that correspond to variants with maximal timestamps on the valid time axis or the transaction time axis, respectively. If valid state objects not allowing overlapping or transaction time objects store variants with maximal timestamps on their unique time axis, then the respective methods for these classes may simply convert their arguments to variant sets. This may also be true for one of these methods for bitemporal objects.

In all cases, the TOQL processor rewrites invocations of the restructuring operations to calls to the respective methods, e.g. the call

```
(f->Manager)(partition valid as instant)
```

is rewritten by the TOQL processor as

```
(f->Manager).restructure_valid_instant
```

L. Aggregation

TOQL introduces additional aggregation methods, allowing for partitioning of temporal objects into variant subsets, each containing information pertaining to a specific portion of some time axis. Each subset may contain variants from a single temporal object or variants from multiple temporal objects.

The first type of partitioning is implemented via auxiliary methods: when such a partitioning is required, a member function of the temporal object is called, and the desired duration of the basic partitioning unit, along with the unit extensions towards the beginning and the end of the time axis are passed as parameters to it (if some unit extension is not specified, a zero-length interval is used in the place of the respective parameter). For example, the query

```
valid_time_object(partition valid as INTERVAL '12' MONTH
                  trailing INTERVAL '3' MONTH)
```

is transformed to the following method invocation:

```
(valid_time_object).mk_valid_partition(interval().set_value('12', 'month'),
                                       interval().set_value('0', 'month'),
                                       interval().set_value('3', 'month'))
```

The method *mk_valid_partition* is part of the behaviour of temporal objects and is responsible for formulating and returning the expected result, as described in paragraph 1.9.1 and returning it. A similar method is provided for partitioning over the transaction time axis. Once the partitioning has been performed, subgroup filtering and aggregate value computation may be performed using standard OQL methods.

The second partitioning method is more complex, since it requires the combination of variants that belong to multiple temporal objects. Fetching all the involved objects in memory, so as to process them within a method that will formulate the result, as in the previous case is not feasible, since the data volume may be excessively high. In order to avoid such problems, the TOQL processor combines the partitioning method introduced for the previous case with query rewriting, involving standard OQL aggregation, in order to perform the aggregation described in paragraph 1.9.2. Recall that the generic form of a query requesting this form of aggregation is

```
select select-list
from collection-of-temporal-objects as variable
where item-condition
group by time-axis basic-unit [leading leading-interval]
                  [trailing trailing-interval] as identifier
having group-condition
```

and in such a query, first the collection specified in the *from* clause is formulated and then its elements that do not satisfy the condition stated in the *where* clause are eliminated. The variants of the remaining temporal objects are then restructured, in order to formulate the subgroups, and subgroups not meeting the condition specified in the *having* clause are eliminated. Finally, the *select-list* is evaluated, in order to produce the final result.

The steps described above are followed by the TOQL processor in order to evaluate the query. The first step is to formulate the collection specified in the *from* clause and apply the condition specified in the *where* clause, which is performed using the query

```
select * from collection-of-temporal-objects as variable
where item-condition
```

The result of this query is a collection of temporal objects, which will be denoted as *c1* in the following. The TOQL processor defines a variable iterating over the

elements of this collection, invokes the partitioning method for each element and finally selects the members of each subcollection returned by the partitioning method, together with the associated time period. This is performed using the query

```
select toql_var02->TimeSlice as TimeSlice, toql_var03 as Variant
from c1 as toql_var01,
     toql_var001->mk_axis_partition(basic-unit, leading-interval,
                                    trailing-interval) as toql_var02,
     toql_var02->partition as toql_var03
```

The result of this query is a collection of structures, with each structure containing two fields. The first field is a period, corresponding to a window of the time axis on which aggregation will be performed. The second field is a variant containing information on that window of the time axis. The result of this query will be denoted as *c2* in the following.

The final step is to group together variants pertaining to the same window of the time axis, filter out the variant subgroups that do not meet the condition specified in the *having* clause and compute the values in the original *select-list*, which will be the query result. This is accomplished via the query

```
select select-list
from (select TimeSlice as identifier,
             (select toql_var05->variant
              from partition as toql_var05) as partition
      from c2 as toql_var04
      group by c2->TimeSlice as TimeSlice) as toql_var06
where group-condition
```

In this query, the *group by* clause partitions the variants into subsets, with respect to the time axis window to which each variant pertains. The *select-list* of the same *select/from/where* query arranges so that the schema returned by the original *group by* clause is set to the expected one. Finally, the outermost select performs subset filtering, applying thus the *having* clause, and lastly, the requested *select-list* is evaluated. Gluing together the three steps, we obtain the overall query transformation that is applied for the second type of temporal aggregation:

```
select select-list
from (select TimeSlice as identifier,
             (select toql_var05->variant
             from partition as toql_var05) as partition
      from (select toql_var02->TimeSlice as TimeSlice,
            toql_var03 as Variant
            from (select *
                  from collection-of-temporal-objects as variable
                  where item-condition) as toql_var01,
                  toql_var001->mk_axis_partition(basic-unit,
                              leading-interval,
                              trailing-interval) as toql_var02,
                  toql_var02->partition as toql_var03) as toql_var04
      group by c2->TimeSlice as TimeSlice) as toql_var06
where group-condition
```

2.3.2 Auxiliary methods

The syntactical transformations employed by the TOQL processor, in order to implement the temporal extensions to OQL, are complemented with auxiliary methods. These methods are used for:

- the implementation of restructuring operators. Methods *restructure_valid_instant* and *restructure_transaction_instant* implement transformation of period-timestamped variants to instant-timestamped variants, whereas methods *restructure_valid_period* and *restructure_transaction_period* arrange for production of maximal timestamps on the valid time axis or the transaction time axis, respectively. None of these methods accept arguments. Methods *restructure_valid_instant* and *restructure_valid_period* are supported by valid state objects not allowing overlapping of valid timestamps and bitemporal objects not allowing overlapping of valid timestamps. Methods *restructure_transaction_instant* and *restructure_transaction_period* are supported by transaction time objects and bitemporal objects.
- the implementation of partitioning in temporal forms of aggregation. Valid time objects support the *mk_valid_partition* method while transaction time objects support the *mk_transaction_partition* method. Bitemporal objects support both methods.

  Each of the methods *mk_valid_partition* and *mk_transaction_partition* accept three arguments of type interval and one argument of type boolean. The first argument is the basic partitioning unit of the respective time axis. The second argument defines an extension of this partitioning unit towards the beginning of the time axis, corresponding thus to the interval designated in the *leading* subclause of the

partitioning clauses. The third argument defines an extension of the basic partitioning unit towards the end of the time axis, corresponding thus to the interval designated in the *trailing* subclause of the partitioning clauses. If the basic partitioning unit should not be extended towards the beginning or the end of the time axis, the corresponding parameter to the method should be a zero length interval. Finally, the fourth argument indicates whether time window offsets should be computed starting from the beginning of the calendar or from the smallest timestamp within the temporal object.

Both methods return a set of structures, with each structure containing two fields. The first field designates the time window, defined by the basic partitioning unit and the two extensions; this field is named *TimeSlice*. The second field is named *Partition* and contains all the variants of the temporal object that contain information pertaining to the respective time window.

2.3.3 Implementation of TOQL on top of an OQL v. 1.1 compliant OODBMS

TOQL, as described in section 1.2 to 1.9 is a consistent extension to the more recent standard of OQL, namely version 1.2 ([ODMG95]). In version 1.2, some syntactic constructs of version 1.1 ([Cattell94]) are not valid, since they have been replaced with SQL-92 resembling syntactic constructs. Examples of such syntactic constructs are the select list (a *struct* construct is now implicit if more than one attributes are listed; names are assigned to members of the select-list using the *as name* notation; the asterisk (*) can be used as a select list), the grouping mechanism (the *group identifier in query by...* query has been replaced by the *group by* clause), the sorting specification (the *order by* clause is used instead of the *sort identifier in query by...* construct) etc. Although the syntactic changes in OQL v. 1.1 are substantial, the modular design of the TOQL processor allows for porting the TOQL processor on top of either an OQL v. 1.1 compliant OO-DBMS or on top of an OQL v. 1.2 compliant OO-DBMS with minimal effort. More specifically, only the *OQL Generator* module needs to be modified for such a porting, so as to emit version-specific code for syntactical constructs that are not identical in both language specifications. The only feature of OQL v. 1.2 that cannot be directly supported on top of an OQL v. 1.1 conferment OO-DBMS is the generic sorting with arbitrary sorting order. However, two special cases can be handled in such an occasion:

1. if any numeric attribute is requested to be sorted in descending order, the corresponding expression in the *order by* clause can be negated (using the unary minus operator) and sorting can be performed with ascending order, with respect to the resulting expression.
2. if all attributes must be sorted in descending order, then the opposite sorting can be requested from the underlying OO-DBMS and then the resulting list can be inverted using an auxiliary method.

2.4 Using TOQL in C++

TOQL queries may be submitted from within a C++ program. In order to submit a TOQL query for evaluation, a C++ program has to perform the following actions:

1. create a link to the database by connecting to the temporal OODBMS using the *d_TSession::begin* member function, and opening a database using the *d_TDatabase::open* member function. *d_TSession* and *d_TDatabase* are temporal specialisations of classes *d_Session* and *d_Database* ([O_2 96]), respectively, and the functionality of these two member functions is similar to the member functions *d_Session::begin* and *d_Database::open* ([O_2 96]). These two methods need to be invoked only once for each execution of the program.
2. formulate the query text within an instance of the class *d_TOQL_Query*. *d_TOQL_Query* is a temporal specialisation of class *d_OQL_Query* ([O_2 96]). The query text must conform to the syntax and semantic rules of TOQL and may contain *bind arguments* in any place that a literal value is allowed. Bind arguments have the form `$i`, where *i* is a positive integer and can be bound to specific values using the << C++ operator, e.g.
   ```
   my_query << "Johnes" << "100000";
   ```
3. invoke the function *d_toql_execute*. This function accepts two arguments. The first argument must be an instance of the class *d_TOQL_Query* whose bind arguments (if any) must be all bound to specific values. The result of the query is stored in the second argument. The type of the second argument must match the type returned by the query. The prototype of the *d_toql_execute* function is
   ```
   int d_toql_execute(d_TOQL_Query &q, T &result);
   ```
   and is resembling to the standard ODMG function for executing queries, whose prototype is

```
int d_oql_execute(d_OQL_Query &q, T &result);
```

Steps (2) and (3) may be performed any number of times within an application, once step (1) has been successfully completed. The TOQL C++ interface is quite similar to the OQL C++ interface. Further details may be found in [O_2 96].

3. Conclusions

In this report we gave the specification and design of a temporal extension to OQL v.1.2. The extension is consistent with OQL, in that it preserves its syntax and semantics. The design of the TOQL processor allows for portability over any ODMG compliant DBMS. Query evaluation is based on query rewriting and method invocation, allowing thus for exploitation of the underlying OODBMS's optimisation mechanisms.

Appendix A: TOQL BNF syntax

The TOQL syntax will be presented in a rather informal BNF. The following notation is used:

- {symbol} means a sequence of 0 or n symbols
- [symbol] means an optional symbol
- **keyword** is a terminal of the grammar
- xxx_name has the syntax of an identifier
- xxx_literal is self explanatory, e.g. “a string” is a string_literal
- bind_argument stands for a parameter when embedded in a programming language

1. Axiom

```
query_program ::= {define_query;} query
define_query ::= define identifier as query
```

2. Basic

```
query ::= nil
query ::= true
query ::= false
query ::= integer_literal
query ::= float_literal
query ::= character_literal
query ::= string_literal
query ::= instant_literal
query ::= interval_literal
query ::= period_literal
query ::= entry_name
query ::= query_name
query ::= bind_argument[5]
```

[5] A bind argument allows to bind expressions from a programming language to a query when embedded into this language

```
query ::= from_variable_name
query ::= (query)
```

3. Simple Expression

```
query ::= query + query[6]
query ::= query - query
query ::= query * query
query ::= query / query
query ::= - query
query ::= query mod query
query ::= abs (query)
query ::= query || query
```

4. Comparison

```
query ::= query comparison_operator query
query ::= query like string_literal
comparison_operator ::= =
comparison_operator ::= !=
comparison_operator ::= <
comparison_operator ::= <=
comparison_operator ::= >
comparison_operator ::= >=
```

5. Boolean expression

```
query ::= not query
query ::= query and query
query ::= query or query
operator && is synonymous with and
operator || is synonymous with or
```

[6] The operator + is also used for list and array concatenation

```
operator ! is synonymous with not
```

6. Constructor

```
query ::= type_name ([query])
query ::= type_name (identifier: query
          {, identifier: query})
query ::= struct(identifier: query {, identifier: query})
query ::= set([query {, query}])
query ::= bag([query {, query}])
query ::= list([query {, query}])
query ::= array([query {, query}])
query ::= period_set([query {, query}])
```

7. Accessor

```
query ::= query dot attribute_name
query ::= query dot relationship_name
query ::= query dot operation_name([query {, query}])
dot ::= . | ->
query ::= query [query]
query ::= query [query : query]
query ::= query [distinct query]
query ::= query [valid at [distinct] query]
query ::= query [current at query]
query ::= query [valid at [distinct] query,
                 current at query]
query ::= query [current at query,
                  valid at [distinct] query]
query ::= first(query)
query ::= last(query)
query ::= function_name(query {, query})
```

8. History accessor

```
query ::= valid query
query ::= transaction query
query ::= bitemporal query
```

9. Timestamp Accessor

```
query ::= valid(query)
query ::= transaction(query)
```

10. Collection expression

```
query ::= for all identifier in query: query
query ::= exists identifier in query: query
query ::= exists (query)
query ::= unique (query)
query ::= query in query
query ::= query comparison_operator quantifier query
quantifier ::= some
quantifier ::= any
quantifier ::= all
query ::= count(query)
query ::= count(*)
query ::= sum(query)
query ::= min(query)
query ::= max(query)
query ::= avg(query)
```

11. Select Expression

```
query ::= select [distinct] projection_attributes
          from variable_declaration
               {, variable_declaration}
          [where query]
          [group by partition_attributes]
          [having query]
          [order by sort_criterion {, sort_criterion}
projection_attributes ::= projection {, projection}
projection_attributes ::= *
projection ::= query [as identifier]
variable_definition ::= query [[as] identifier]
partition_attributes ::= projection {, projection}
partition_attributes ::= time_axis query [leading query]
                                         [trailing query]
time_axis ::= valid
time_axis ::= transaction
sort_criterion ::= query [ordering]
ordering ::= asc
ordering ::= desc
```

11. Set expression

```
query ::= query intersect query
query ::= query union query
query ::= query except query
operator + is synonymous with union
operator - is synonymous with except
operator * is synonymous with intersect
```

12. Conversion

```
query ::= listtoset(query)
```

```
query ::= element(query)
query ::= flatten(query)
query ::= (class_name)query
query ::= valid (state | event) [overlap] [granularity]
          [calendar calendar_spec] [query : query]
query ::= transaction [query]
query ::= bitemporal (state | event) [overlap]
          [granularity] [calendar calendar_spec]
          [query : query]
```

13. Temporal Modifiers

```
query ::= valid [state] query
query ::= transaction [state] query
query ::= bitemporal [state] query
```

14. Temporal join

```
query ::= tstruct(identifier: query
                  {, identifier: query})
```

15. Restructuring operator

```
query ::= query (partition time_axis as restruct_spec)
restruct_spec ::= instant
restruct_spec ::= period
```

Appendix B: Language completeness with respect to the user requirements

The purpose of this appendix is to verify that the syntax and semantics of TOQL are rich enough to support the needs of the two pilot applications. The queries and reports that have been identified in the user requirements analysis process ([T22TR.1]) have been extracted and the TOQL queries which produce the appropriate results are given in the following paragraphs. It must be noted that the schema on which these queries operate may not be the schema that will be used by the applications; the final form of the latter will be determined within the application development task, using the methodology produced by task T23. If the final database schema is not as described in [T32TR.1], then the queries presented below will have to be modified accordingly.

1. Delta Application

The queries producing the reports required for the DELTA application ([T22TR.1] pp. 2.4) are given below. The queries are based on the schema described in [T32TR.1]. Some of the queries are lengthy and look complicated, but it must be pointed out that these queries actually produce reports. In a relational system, the production of these reports would require a number of SQL statements and considerable amount of host language (or 4GL) code.

Query A

All distributors per product type. The report is listing for a specific period the total or incremental (in 15-day or monthly) sales performances. The quantities printed are: last year, target, this year, difference (last year - this year) and deviation (last year - target).

TOQL query

```
select product, distributor,
    (select sum(x->qty) from partition as x
     where not (prodData->timestamp overlaps $1) as lastYear,
    element(select p
        from (valid d->hasTarget)[end($1)])->targetData->products as p
        where p = prod) as target
    (select sum(x->qty) from partition as x
     where prodData->timestamp overlaps $1) as thisYear,
    lastYear - thisYear as difference,
    target - lastYear as deviation
from (select d as distributor, batchItem->p as prod,
             batchItem->Quantity as qty,
             batchItem->timestamp as tstamp
     from Distributors as d,
      (valid d->delivers)[twoYears] as packages
      packages as package,
      (valid package->packageData->products)[valid(packages)]
                                    as batchItem) as as prodData
group by prodData->prod as product,
         prodData->distributor as distributor
```

Notes:

1. `$1` is a placeholder for the desired period.
2. the *twoYears* identifier used in the query is actually a substitute for the expression

   ```
   period((begin($1) - INTERVAL '1' YEAR), end($1))
   ```

   This substitution is used for query readability purposes.

Query B

For customers and distributors: quantities, values and average values for a certain period of time.

TOQL query

```
select c->customerName as customerName,
       (select productName,
               sum(select x.Quantity
                   from partition as x) as totalQty,
               avg(select x.Quantity * x.price
                   from partition as x) as avgValue,
               sum(select x.Quantity * x.price
                   from partition as x) as totalValue
        from (select prd->p->productName as productName,
                     prd->p->price[valid(mt)] as price,
                     prd->Quantity as Quantity
              from (valid c->maintains)[$1] as mt,
                ((valid mt->stockData)[valid(mt)])->products as prd)
                                                          as p1
        group by p1->productName as productName) as Info
from Customers as c
```

Notes:

1. `$1` is a placeholder for the desired period.
2. The query for distributors is not given; it is quite similar to query (A) and can easily be derived from it.

Query C

For a given distributor, customer sales involving quantities of products. This report refers to 15-day and monthly period totals.

TOQL query

```
select productName, TimeStamp
       sum(select x->qty from partition as x) as dailyQty
from (select pr->p->productName as productName,
             pr->Quantity as Quantity,
             CAST pr->timestamp TO DAY as timeStamp
      from DeliveryPackages as dp,
           dp->packageData->products as pr
      where dp->deliveredBy->distributorName = $1) as prods
      group by prods->productName as productName,
               prods->timeStamp as TimeStamp
order by productName, timeStamp
```

Notes:

1. `$1` is a placeholder for the desired distributor name.
2. This query returns to the application the daily totals per product, sorted by product name and day of sale. Calculation of 15-day totals is trivial, using minimum amount of host language code. Monthly totals can be calculated without any need for application code if the valid timestamp of variable *pr* is casted to *month* instead of *day*.

Query D

History (archived). Sales reports, involving sales per day for distributor, customer, product type.

TOQL query

```
select dt->isProduct->productName as productName
       dt->quantityLoaded - dt->quantityReturned as qty,
       dt->returnedAt as ReturnDate
from DailyTotals as dt
order by productName, ReturnDate
```

Notes:

The query above retrieves the data for a “per product” report. “Per distributor” and “per customer” reports are produced similarly.

Query E

History report for all routes. The customers assigned to distributors over time.

TOQL query

```
select d->distributorName as distributorName,
       valid d->makesVisit as distributorVisits,
       valid valid d->serves as distributorServes
from Distributors as d
```

Query F

Report for the daily loading. Involves date, unit, route, distributor, product type, quantities, value.

TOQL query

```
select d->distributorName as distributorName,
       tstruct(route: valid d->makesVisit,
                products: valid d->delivers) as dailyData
from Distributors as d
```

Query G

Aggregate reports of turnover for 15-day or monthly intervals. The reports indicate the following customer categories: all, old, new, lost to competition.

TOQL query

```
select TStamp,
       sum(select x->batchPrice from partition as x) as TurnOver,
       Partition
from (select sum(select prod->Quantity *
                        prod->p->price->[deliv->timestamp] *
                        (1.0 - deliv->discount)) as batchPrice,
             cast (deliv->timestamp) to MONTH) as tstamp
      from CustomerDelivery as cd, cd->deliveryData as deliv
      where (cd->receivedBy)->customerType = $1)
group by turnData->tstamp as TStamp
```

Notes:

1. `$1` is the desired customer type (old, new, lost to competition). If the report should include all customers, the *where* clause should be omitted.
2. for 15-day periods, the corresponding period number within a year may be extracted using an appropriate function.

Query H

What is the stock of the customers, every day?

TOQL query

```
select c->customerName as cust, valid c->maintains as stock
from Customers as c
```

2. GlaxoWellcome Application

The queries producing the reports required for the GlaxoWellcome application ([TR22TR.1] pp. 3.10-3.11) are given below. The queries are based on the schema described in [TR32TR.1].

Query A: Belonging, existence, of one event to a period

Does there exist, at least, an undesirable event, occurring between visit 1 and visit 2?

TOQL query

```
count(select valid(s)
      from ObservationNotebooks as on,
           valid on->functRespTest as frt
      where on->ofPatient->patientName=$1 and
            exists s in valid on->undesirableEvent:
                     valid(valid on->frt[0]) precedes valid(s) and
                     valid(s) precedes valid(valid on->frt[1])) > 0
```

Note:

`$1` is a placeholder for the desired patient name.

Query B: Comparisons of dates, translation of dates by intervals

Does there exist a concomitant treatment with a beginning date inferior to the visit 1, at least from 28 days?

TOQL query

```
count(select p->patientName
      from Patients as p
      where p->patientName = $1 and
          exists ct in valid p->observes->concomitant:
            valid(ct) < valid(valid(p->observes->functRespTest)[0]) -
                        INTERVAL '28' DAY) > 0
```

Note:

`$1` is a placeholder for the desired patient name.

Query C: Comparisons of periods

Is the period of treatment included between visits 1 and 5?

TOQL query

```
count(select p->patientName
      from Patients as p
      where p->patientName = $1 and
         for all frt in (valid p->observes-> functRespTest)[0:4]:
            period(p->observes->trialTreatm->timestamp,
                   p->observes->trialTreatmp->timestamp +
                              p->observes->trialTreatm-> realPeriod)
                     contains valid(frt)) > 0
```

Note:

`$1` is a placeholder for the desired patient name.

Query D: Calculation of interval

Is the length of hospitalisation inferior or equal to the delay between the first and the fifth visit?

TOQL query

```
element(select on->stat4Week->hospitalisationDur <=
                (valid(valid on->functRespTest[4]) -
                        valid(valid on->functRespTest[0]))
        from ObservationNotebooks as on
        where on->ofPatient->patientName = $1)
```

Note:

`$1` is a placeholder for the desired patient name.

Query E: Comparisons of values on a period, or at different dates

Which is the percentage of patients having an increase of VEMS, between J0 and J14, of 10% with regard to the theoretical value?

Which is the difference between J0 and J28 about the observed values and the theoretical values?

TOQL query

```
count(select on->ofPatient
      from ObservationNotebooks as on
      where (valid on->functRespTest[2])->vemsObs -
                  ((valid on->functRespTest)[0])->vemsObs ≥
            1.1 * (((valid on->functRespTest)[2])->vemsTheory -
                  ((valid on->functRespTest)[0])->vemsTheory) /
(select count(p) from Patients as p)
```

```
select ((valid on->functRespTest)[4])->vemsObs -
              ((valid on->functRespTest)[0])->vemsObs as observed,
        ((valid on->functRespTest)[4])->vemsTheory -
              ((valid on->functRespTest)[0])->vemsTheory as theoret
from ObservationNotebooks as on
where on->ofPatient->patientName = $1
```

Note:

In the second query, `$1` is a placeholder for the desired patient name.

Query F: Average of a value on a period, or on an interval

What are the average values week by week of expiratory debit?

TOQL query

```
select weekly->TimeSlice as TimeSlice,
       (select n->expiratoryDebit
        from weekly->partition as n) as AvgVal
from Patients as p,
   (valid p->fillsOut)(partition valid as interval '7' day) as weekly
where p->patientName = $1
```

Note:

`$1` is a placeholder for the desired patient name.

Query G: Search for some periods or intervals that satisfy a criterion

What is the time to obtain an increase of VEMS, with regard to J0, of 10% with regard to the theoretical value?

TOQL query

```
select period(valid((valid on->frt)[0])), min(valid(frt))
from ObservationNotebooks as on,
     valid on->functRespTest as frt
where on->ofPatient->patientName = $1 and
      frt->vemsTheory ≥
             1.1 * ((valid on->functRespTests[0])->vemsTheory)
```

Note:

`$1` is a placeholder for the desired patient name.

Query H: Manipulation of relative time

Is the date of visit 2, superior of 7 days to the date of visit 1, more or less one day?

TOQL query

```
element(select valid((valid on->functRespTest)[1]) -
               valid((valid on->functRespTest)[0])) >=
                              INTERVAL '6' DAY and
               valid((valid on->functRespTest)[1]) -
               valid((valid on->functRespTest)[0]) <=
                              INTERVAL '8' DAY)
        from ObservationNotebooks as on
        where on->ofPatient->patientName = $1)
```

An alternative way to answer this query is:

```
element(select valid(valid on->frt[1]) + INTERVAL '6' DAY =
                              valid(valid on->frt[1]) or
               valid(valid on->frt[1]) + INTERVAL '7' DAY =
                              valid(valid on->frt[1]) or
               valid(valid on->frt[1]) + INTERVAL '8' DAY =
                              valid(valid on->frt[0]))
        from ObservationNotebooks as on,
        where on->ofPatient->patientName = $1)
```

Note:

In both alternatives, `$1` is a placeholder for the desired patient name.

Appendix C: Snapshot application compatibility

Two of the major design goals of TOQL are *upward compatibility with OQL* and *complete temporal functionality*. These two goals may be described in more detail as follows:

1. *Temporally unaware applications should function on top of a temporal object schema, with the minimum amount of changes.*

   If an existing (snapshot) database is converted, so as to incorporate temporal semantics, applications developed prior to the conversion (*temporally unaware applications*) should run on top of the updated database schema, with the minimum number of changes. In particular, changes to OQL queries and to the host language code should be avoided.

2. *Temporally aware applications should be able to exploit the enriched functionality offered by the temporal schema.*

   Applications that have been developed after the conversion to a temporal schema (*temporally aware applications*), should be able to deal with the temporal aspects of data, and must have access to an enriched object interface, which allows the manipulation of the objects' variants.

In the following paragraphs, it is described how TOQL achieves these two goals. The retrieval of individual temporal properties and objects containing temporal properties is discussed, and an example of a schema modification to include temporal characteristics is presented.

1. Accessing Temporal Properties

When a temporal property is used in a TOQL query, the default behaviour is to use its current value (i.e. the value of the variant whose valid and transaction timestamps overlap with the current instant). For example, the query

```
select p->LifeTime
from Products as p
where p->ProductName = 'Life Orange Juice'
```

will retrieve only the current value for the life time of the product "Life Orange Juice", although the stored attribute actually contains the full history of the product's life time. Analogous conversions are performed for transaction time and bitemporal attributes. The conversion is applied regardless of the attribute's position within the

query, i.e. it performed on all attributes appearing in the *select*-list, the *from* clause, the *where* clause, the *group by* clause, the *having* clause and the *order by* clause.

The implicit conversion guarantees that temporally unaware applications may access temporally enriched data with no modifications to queries. Temporally aware applications are not deprived of the capability to access valid/transaction time histories of temporal attributes; these histories are accessible through the *VALID*, *TRANSACTION* and *BITEMPORAL* modifiers and the subscript notation ([]). For instance, an application willing to access the complete history of the "Life Orange Juice" product's life time, should use the query

```
select valid p->LifeTime
from Products as p
where p->ProductName = 'Life Orange Juice'
```

2. Accessing Objects with Temporal Properties

When an object containing temporal properties is used in a TOQL query (practically, such an object may appear only in the *SELECT* list), no conversion is performed on it. The object identifier will be selected and returned to the application, which is consistent with the behaviour of OQL. For example, the query

```
select p
from Products as p
```

will return an object identifier for all products appearing in the *Products* extent, i.e. the set

```
{Obj001, Obj002}
```

Once the object identifiers have been returned to the application, the latter may manipulate the corresponding, by invoking the corresponding operations. Since object manipulation is performed entirely via operations, snapshot application compatibility is not jeopardised, as long as the operation signatures remain intact, regardless of changes made to the internal implementation and representation details of the object. However, temporally unaware applications will have to be recompiled, in order to incorporate the updated object definition. This seems unavoidable in an object-oriented environment, since the memory image of an object must match the database schema, in order to have object updatability.

The programmer may equip the temporally enriched class with additional operations which provide access to the temporal characteristics of the individual attributes. These

operations may be invoked by the temporally aware applications, when access to the valid/transaction time histories of the objects is required.

3. A Compatibility Example

Consider a snapshot database containing information about diary products and factories (its schema is depicted in Figure 32 and sample contents are illustrated in Figure 33; details for *Factory* objects are omitted for brevity reasons). On top of this database, a temporally unaware application has been developed, which performs the following functions:

```
interface   Product
(extent     Products
 key        ProductName)
{
// Attributes
            attribute String ProductName;
            attribute String Description;
            attribute Set<String> ingredients;
            attribute Interval LifeTime;
            relationship Set<Factory> manufactured
                         inverse Factory:manufactures;
}
```

Figure 32 - A sample snapshot database schema

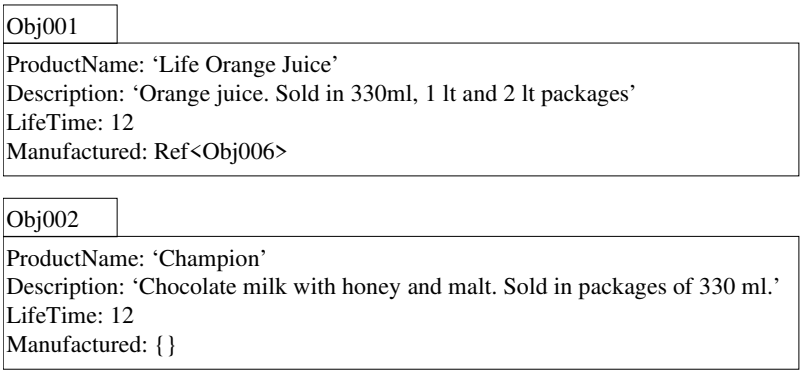

Figure 33 - Sample snapshot database contents

1. determines which is the lifetime of a specific program by using the query

```
select p->LifeTime
from Products as p
where p->ProductName = $1
```

($1 is a placeholder for the product name). The query yields a set intervals containing zero or one elements. The element may be manipulated as any datum of type *interval*. Evaluating this query for patient "Life Orange Juice" produces the result

```
12
```

(we assume that all queries are evaluated on November 1, 1996).

2. determines the names of the products having life times exceeding a certain amount of days, using the query

```
select p->ProductName
from Products as p
where p->LifeTime > $1
```

($1 is a placeholder for the lower limit of the product's life time). The query yields a set of strings having an arbitrary number of elements. Each of these elements may be manipulated as any datum of type *string*. Evaluating this query for an interval equal to INTERVAL '10' DAY will produce the result

```
{'Life Orange Juice', 'Champion'}
```

3. produces a report of product names along with their life times. Data are fetched into memory using the query

```
select p from Products as p
```

The desired values are extracted from the *Product* objects using the *getName* and *getLifeTime* operations and are displayed on the screen (or printed) using standard library procedures. The outcome of this action is

```
Life Orange Juice     12 days
Champion              12 days
```

4. allows for modification of the life time of a product. The product is retrieved into memory using the query

```
select p
from Products as p
where p->ProductName = $1
```

and the product's life time is modified using the *setLifeTime* operation. Evaluating the query for product "Champion" fetches into memory the corresponding object, and after updating its life time to INTERVAL '10' DAY, the outcome of operation (3) should change to

```
Life Orange Juice     12 days
Champion              10 days
```

At some point, e.g. on *1997-01-01*, the database schema is enriched with temporal characteristics and two more attributes are added (*AvgSales* and *Ingredients*), as described in section 1.3. After the conversion to the temporal schema, the database contents are as follows:

Obj001

ProductName: ‘Life Orange Juice’
Description: ‘Orange juice. Sold in 330ml, 1 lt and 2 lt packages’
AvgSales: {}
Ingredients: {(value: {‘Concentrated Orange Juice’, ‘Water’, ‘Sugar’, ‘E210’}, TT: [1996-01-01, *UC*))}
LifeTime: {(value: 12, VT: [1996-01-01, *forever*))}
Manufactured: {value: {Ref<Obj006>, VT: [1996-01-01, *forever*), TT: [1996-01-01, *UC*))}

Obj002

ProductName: ‘Champion’
Description: ‘Chocolate milk with honey and malt. Sold in packages of 330 ml.’
AvgSales: {}
Ingredients: {(value: {‘Cow’’s Milk’, ‘Honey’, ‘Chocolate’, ‘Malt’, ‘Color’}, TT: [1997-01-01, *UC*))}
LifeTime: {(value: 12, VT: [1997-01-01, *forever*))}
Manufactured: {(value: {}, VT: [1997-01-01, *forever*), TT: [1997-01-01, *UC*))}

Figure 34 - Database contents after the introduction of temporal semantics

After the conversion, all four functions of the snapshot application continue to work as expected, without any modification (apart from recompilation, so as to include the updated object and operation definitions):

1. the first function continues to work, since the temporal attribute *p->LifeTime* is converted to the corresponding current snapshot value, before it is passed back to the application, within the query result. The results returned to the application are identical to the ones that would be returned, if the query result was evaluated against a snapshot schema.
2. the second function continues to work, since the temporal attribute *p->LifeTime* appearing in the *where* clause is converted to the corresponding current snapshot value, before the truth value of the condition is evaluated. The results returned to the application are identical to the ones that would be returned, if the query result was evaluated against a snapshot schema.
3. the third function continues to work, since the objects, as stored in the database are returned to the application, and the method *getLifeTime* returns only the current value for the product’s life time. Note that it’s the programmer’s responsibility to ensure that the functionality of the methods in the snapshot object’s interface does not change in the process of introducing temporal characteristics.
4. the fourth function continues to work, since the query returns an updatable object, which is modified through the function *setLifeTime* (which now adds the specified value to the temporal attribute's history, with an appropriate timestamp). The object holding information for patient “Champion” will be changed to

Obj002
ProductName: ‘Champion’ Description: ‘Chocolate milk with honey and malt. Sold in packages of 330 ml.’ AvgSales: {} Ingredients: {(value: {‘Cow’’s Milk’, ‘Honey’, ‘Chocolate’, ‘Malt’, ‘Color’}, TT: [1997-01-01, *UC*)} LifeTime: {(value: 12, VT: [1997-01-01, 1997-01-10)), (value: 10, VT: [1997-01-10, *forever*)} Manufactured: {(value: {}, VT: [1997-01-01, *forever*), TT: [1997-01-01, *UC*)}

(assuming that the modification was performed on 1997-01-01) and, after the modification, the second operation will produce the following result

```
Life Orange Juice     12 days
Champion              10 days
```

which is identical to the result that the same functions of the snapshot application would produce if it functioned on top of a snapshot schema.

Appendix D: Temporal Selection

In [T33TR.1, early] the ***when*** clause was described as part of TOQL. In the past few months, a discussion was held between the project partners and the O_2 representative in the ODMG board was contacted, in order to assess:

- the functionality and the syntax of the *when* clause, along with alternative ways to provide the same (or enhanced) functionality.
- the possibility of promoting the *when* clause to the ODMG board for standardisation

The O_2 representative in the ODMG board clearly stated that it is highly unlikely that the *when* clause, since it is a major change to the OQL syntax and, moreover, the *where* clause is generally sufficient to provide all the necessary restrictions, if explicit usage of temporal objects is permitted. This approach has been adopted in the updated specification of TOQL, and, in the following paragraphs, a discussion can be found, comparing the *when* clause, as specified in [T33TR.1, early] with the approach that has been adopted in the updated specification.

The *when* Clause vs. Subscripting

The syntax of the ***when*** clause, as described in [TR33TR.1, early] is

```
when when-restriction {, when-restriction}
```

where *when-restriction* is defined as

```
variable in (valid | transaction) temporal_object: query
```

Such a clause has the effect of defining a *variable*, which iterates over the valid time (or transaction time) timestamps of the variants of *temporal_object* and, for each timestamp the *query* is evaluated. If, for some timestamp, the query evaluates to *true*, the corresponding variant is considered in further query processing, otherwise the variant is disregarded. This approach provided the ability to compare valid and transaction timestamps with instants and periods. In the absence of a *when* clause, the default behaviour of the query was to consider all variants whose transaction timestamp overlaps with the current instant (the variants that have not been logically deleted), regardless of their valid timestamps. The same rules applied to variables that are defined to iterate over temporal objects (the ***from*** *temporal_object as identifier* construct).

In the updated specification, TOQL provides the ability to select variants of a temporal object, based on their valid and/or transaction timestamps by using the more user-friendly and intuitive notation of subscripting. A temporal object may be treated as a *collection* which can be indexed by numbers, instants or periods. When a number is used to index a temporal object, it specifies the rank of the desired variant (variants are ordered with respect to their valid or transaction timestamps). When a temporal object is indexed by an instant or a period, the subscript designates the time window for which information is desired. In the absence of subscripts, the default behaviour for temporal objects is that one variant is considered (the variant whose valid and transaction timestamps overlap with the current instant). This behaviour may be altered by preceding the temporal object with one of the *valid*, *transaction* or *bitemporal* modifiers. The behaviour of iterations has also been modified, and variables that are defined to iterate over temporal objects, actually iterate over all variants, regardless of their valid and transaction timestamps.

The adopted approach is more flexible than the one originally proposed, since it allows for temporal selection to be used for a temporal object in any place that referencing a temporal object is allowed, rather than "once-per-*select/from/where*-clause", as the previous approach did. For example, the current approach allows the queries "For each product whose lifetime on January 1996 exceeded 12 days, fetch the complete life time history" and "For each product whose current lifetime is less than 20 days, fetch its lifetime on January 1995" as depicted below:

```
select valid p->LifeTime
FROM Products as p
where p->LifeTime[INSTANT '1996-01'] > PERIOD '12' DAY

select p->LifeTime[INSTANT '1995-01']
from Products as p
where p->LifeTime < PERIOD '20' DAY
```

Neither of these queries could be expressed using the *when* clause, as described in [TR33TR.1, early]: if the *when* clause were used, the restriction would apply both to the temporal object in the *where* clause and the temporal object in the *SELECT* list, and since different criteria are required for each temporal object, the desired result could not be computed.

The *when* clause, on the other hand, facilitated the expression of conditions on the duration of the timestamps, which cannot be expressed using subscripts. In the absence of the *when* clause, such constraints can be modelled by iterating over the

variants of the temporal object and explicitly testing the duration of the timestamps using the *where* clause. Thus, the query "for the product 'Life Orange Juice' list the ingredients which were used for more than 12 months" can be expressed using the *when* clause as

```
select p->ingredients
from products as p
where p->ProductName = 'Life Orange Juice'
when t in transaction p->ingredients: duration(t) > period '12' month
```

whereas without the *when* clause it must be formulated using a slightly more complicated syntax:

```
select ing
from products as p, transaction p->ingredients as ing
where p->ProductName = 'Life Orange Juice'
  and duration(transaction(ing)) > period '12' month
```

(Another difference between the two queries is that the first one returns a transaction time object containing the respective variants, whereas the second query returns a set, containing the values of the variants; the timestamps need to be explicitly included in the *select* list.)

The queries identified from the user requirements do not need to test the duration of the timestamps, so they remain unaffected from the removal of the *when* clause. If, however, extra queries are identified that do need such constraints, and the application programmers request for easy means to express them, the implementation team will consider incorporating the *when* clause in the TOQL processor, as a TOOBIS addition to the standards extension proposal that will be submitted to the ODMG board.

Appendix F: References

[Cattell94] R. Cattell (Editor), "The Object Database Standard ODMG-93" (Release 1.1), Morgan Kaufmann Publishers, San Francisco, California.

[O2 96] O2 Technology, "ODMG C++ Binding Guide" (for release 4.6), 1996

[ODMG95] The ODMG Board, "Object Query Language", available at URL *http://www.odmg.org/odmg93/updates/chap4x.ps*.

[ORES94] ORES Project (ESPRIT III P7224), "Deliverable D4: Implementation of Valid Time SQL", edited by 01 Pliroforiki, University of Athens, Information Dynamics and Agricultural University of Athens, April 1994.

[TR22TR.1] TOOBIS Project Deliverable T22TR.1 "Requirements Analysis", edited by Glaxo S.A., Delta S.A., Matra Cap Systems, 01 Pliroforiki S.A., University of Sorbone, 1996.

[TR31TR.1] TOOBIS Project Deliverable T31TR.1 "TODM Specification", edited by Matra Cap Systems and O_2 Technology, 1996.

[TR32TR.1] TOOBIS Project Deliverable T32TR.1 "TODL Specification", edited by University of Athens, 01 Pliroforiki S.A., National Technical University of Athens and O_2 Technology, 1996.

[TR33TR.1, early] TOOBIS Project Deliverable T33TR.1 (early release) "TOQL Specification", edited by University of Athens, 01 Pliroforiki S.A. and O_2 Technology, 1996.