A Uniform Approach to Processing Temporal Queries

Umeshwar Dayal
Digital Equipment Corporation
Cambridge Research Laboratory
One Kendall Square
Cambridge, MA 02139, USA
dayal@crl.dec.com

Gene T.J. Wuu
Bell Communications Research
444 Hoes Lane
Piscataway, NJ 08854, USA
wuu@ctt.bellcore.com

Abstract

Research in temporal databases has mainly focused on defining temporal data models by extending existing models, and developing access structures for temporal data. Little has been done on temporal query processing and optimization. In this paper, we propose a uniform framework for processing temporal queries, which builds upon well-understood techniques for processing non-temporal queries. We start with an object-oriented model, and rely on its rich type system to model complex temporal information. The same query language is used to express temporal and non-temporal queries uniformly. A major benefit to this approach is that temporal query processing can be smoothly extended from an existing (non-temporal) query processing framework. For the purpose of query processing, we describe an object algebra, into which queries are compiled. Since the object algebra resembles the relational algebra, familiar relational query optimization techniques can be used. However, since the physical representation of temporal data and access methods differ from those of non-temporal data, new algorithms must be developed to evaluate the algebraic operators. We demonstrate that temporal queries can be processed and optimized under the existing query processing framework.

1 Introduction

Research in temporal databases [STSN88] has mainly been concentrated on two areas: (1) defining temporal data models and query languages by extending existing models and languages, e.g. relational [SNOD87, CLTA85, GAYE88, TAGA89, NAAH89, GAMC91], entity-relationship [ELWU90, ROSE91], complex object [KAFE90], and object-oriented [WUDA92, SUCH91]; and (2) developing access structures for temporal data [ELMA90, ELMA91, AHN86, SEGU89, GUSE91, KOST89, LUM84, LOSA89, ROSE87]. Little work has been reported on temporal query processing and optimization. The slow progress on temporal query processing can be attributed to two reasons: (1) there has been little agreement on the temporal model and language, and (2) the lack of a suitable algebraic framework in which temporal query processing techniques can be developed.

For the purpose of query processing, a query algebra serves as a target language into which high level query languages are compiled. Using the properties of the algebra, the query optimizer transforms the given query into an equivalent algebraic expression from which a more efficient execution plan can be constructed. Among the factors used by the optimizer to determine an efficient execution plan are the underlying physical representation of data, any available access paths such as indexes, and a cost model associated with the execution of the algebraic operators.

In the past, several temporal algebras have been proposed [CLTA85, GAYE88, TAGA89, TUCL90]. These temporal algebras are high level, and are very powerful, indeed, for expressing complex temporal queries. However, they are not suitable for query optimization for two reasons. First, the temporal operators are defined at too high a level. The *select* operator in [GAYE88], for example, admits a complex Boolean expression (a set predicate consisting of combinations of temporal expressions) as its selection condition. It is not clear how to implement such operators efficiently using the various proposed temporal data representations and temporal access methods. Second, these high level algebras define many new operators that are significantly different from the familiar operators of relational algebra. Little is known about their algebraic properties and transformation rules, which are essential to the query optimizer. Adding to the problem is that many proposed temporal query languages are significantly different from their non-temporal counterparts; consequently, the mapping between the temporal query language and an algebra becomes more complicated, and has not been well studied.

In this paper, we propose a uniform framework for pro-

Proceedings of the 18th VLDB Conference Vancouver, British Columbia, Canada 1992

cessing temporal queries that builds upon well-understood techniques for processing non-temporal queries. It is based on the uniform temporal model proposed in [WUDA92]. We start with an object-oriented model, OODAPLEX [DAYA89], (which itself is an object-oriented extension to the DAPLEX functional model [SHIP81]). We rely on the inherent rich type system of OODAPLEX to model complex temporal information. We treat time points as abstract objects, and define a type hierarchy of time types to support various notions of time. The OODAPLEX type system supports various parameterized types, like set, multiset, tuple, and function. These type constructors are essential in modelling temporal information, such as intervals, temporal properties, and temporal relationships. To introduce additional time-related semantics to the system, we define several temporal functions (e.g., lifespan) and temporal constraints. Such temporal extension is achieved without modifying the base OODAPLEX model.

Since all the properties and behaviour of objects, temporal and non-temporal, are uniformly modelled by the same type system, *no* special language constructs are needed to express temporal queries. We demonstrated in [WUDA92, WUDA91] that a wide range of complex temporal queries can be expressed, very naturally and uniformly, in OODAPLEX. These queries include temporal quantification, aggregation, versioned object manipulation, and the familiar temporal operators (e.g., *when, shift*) introduced by previous models. Again, this is achieved without modifying the OODAPLEX language.

There are several benefits to this uniform approach: users need not learn a new data model and language for temporal applications[1]; and temporal query processing can be smoothly extended from the existing query processing framework.

For OODAPLEX, we had developed an algebra, OOAlgebra, as the target language for query compilation [DAYA89]. Since OOAlgebra resembles the relational algebra, the familiar relational query optimization techniques can be used. However, a few extensions (e.g., for dealing with function application, user-defined methods, and inheritance) were necessary. In adapting these techniques to temporal queries, some further extensions are needed. Since the physical representation of temporal data and access methods differ from those of non-temporal data, new algorithms must be developed to evaluate the algebraic operators. Also, the cost model has to be correspondingly extended (although we do not address this problem in this paper). However, by adopting this uniform approach, a single framework can be used to optimize temporal and non-temporal queries. While we describe the approach in the context of OODAPLEX and OOAlgebra, we believe that the techniques developed here can be applied to other models and query languages.

The paper is organized as follows. Section 2 gives an overview of the OODAPLEX model and language. Section 3 describes OOAlgebra and the mapping between OODAPLEX and OOAlgebra. Section 4 describes the temporal extension from the base model. Section 5 presents the physical representation of temporal data and a temporal indexing structure. Finally, Section 6 discusses various techniques for temporal query processing.

[1]Users still need to know of the various time types and new temporal constraints.

2 OODAPLEX Data Model and Language

2.1 OODAPLEX Data Model

OODAPLEX [DAYA89] is an object-oriented data model based on the DAPLEX functional model [SHIP81]. It supports the essential ingredients of an object-oriented data model [ATKI89]: objects with identity, encapsulation through abstract data types, complex object types, subtyping, inheritance, and polymorphic functions with late binding.

An *object* models a distinct real-world or abstract entity. The basic characteristic of an object is its distinct *identity*, which is immutable, persists for the life time of the object, and is independent of the object's properties or behaviour. All objects are abstract in that they consist of an interface (the object's *type*) and an implementation (the object's *representation*).

Properties of objects, relationships among objects, and operations on objects are all uniformly modelled by *functions*, which are *applied* to objects. A function, too, has an interface (the name of the function and a *signature* that defines the input and output arguments of the function) and an implementation (or *body*). A function may have zero or more input arguments and zero or more output arguments.

Objects that have similar properties and behaviour are grouped into *types* (e.g., *integer, person, ship*). A type specifies a set of functions that can be applied to instances of the type. The implementation of a type defines the representation of its instances and the bodies of the functions specified by the type.

Types are themselves objects of type *TYPE*. Two special functions, *types* and *extent*, which map between types and instances, are provided. The *types* function returns, for a given object, the types of which the object is an instance. The *extent* function returns, for a given type, the set of instances of the type currently in the database.

In addition to the primitive, built-in types (such as *integer, real, boolean, string*) and user defined types, OODAPLEX supports constructed (parameterized) types: set, multi-set, and tuple, denoted as $\{T\}$, $< T >$, and $[A_1 : T_1, \ldots, A_n : T_n]$, respectively. In addition, OODAPLEX supports the function type constructor, which (as we shall see in Section 4) is particularly important to temporal modelling and manipulation. A function type is denoted $(D \rightarrow R)$; D is called the *domain type* and R is called the *range type*. An instance of the function type is a partial function mapping from $extent(D)$ to $extent(R)$.

Subtypes of types may be defined, forming *type (is_a) hierarchies*. If S is a subtype of T, then the following properties hold (a) inclusion: $extent(S) \subseteq extent(T)$ and (b) substitutability: a function that expects an instance of T as input argument will also accept an instance of S for that argument. An example is shown below.

```
type employee is object
   function name (e:employee → n:string)
   function salary (e:employee → s:money)
   function dept (e:employee → d:department)
type manager is employee
   function controls (e:manager → d:{project})
type department is object
   function name (d:department → n:string)
   function loc (d:department → c:city)
   function members (e:department → m:{employee})
   function mgr (e:department → m:manager)
```

Here *manager* is a subtype of *employee* and has the function *controls* defined for it, in addition to the functions *name*, *salary*, and *dept* that it inherits from the supertype. OODAPLEX supports multiple inheritance and polymorphism, but we omit the details here.

2.2 OODAPLEX Query Language

OODAPLEX is an extension of the DAPLEX functional query language [SHIP81, SMIT83]. It is a powerful language that combines in a uniform way the retrieval and update of individual objects and aggregates of objects, associative retrieval and navigation, the manipulation of existing objects and the creation of new objects. For details refer to [DAYA89].

In this paper, we focus on a conjunctive, existential subset of OODAPLEX queries that are nested loop programs of the form:

```
for each $x_1$ in $X_1$ where $B_1(\overline{x_1})$
    for each $x_2$ in $X_2$ where $B_2(\overline{x_2})$
        ...
        for each $x_n$ in $X_n$ where $B_n(\overline{x_n})$
            [ $A_1 : t_1, A_2 : t_2, \ldots, A_k : t_k$ ]
        end
        ...
    end
end
```

The x_i are distinct *variables*; and $\overline{x_i}$ is a list of variables involving only $x_1, \ldots, x_i$. The A_i are distinct *labels*. Each *term* t_j is a constant, a variable in $\overline{x_n}$, or a *function application expression* $f(\overline{u})$, where f is a function, and $\overline{u}$ is a list of terms involving only variables in $\overline{x_n}$. Each *set term* X_i is either a named set S, or a function application expression $f(\overline{u})$, where f is a set-valued function, and $\overline{u}$ is a list of terms involving only variables in $\overline{x_{i-1}}$. Each *Boolean expression* B_i is a conjunctive, existential well-formed formula over free variables $\overline{x_i}$. The conjuncts can be Boolean-valued function application expressions, some of which may be written in the special infix forms (< *term* > < *op* > < *term* >) or (< *term* > in < *set_term* >); or existential formulas of the form (*for some y in* < *set_term* >)(*Q*) where *Q* is a conjunctive, existential well-formed formula over free variables y and $\overline{x_i}$.

This query retrieves a multiset of tuples of the type $[A_1 : t_1, A_2 : t_2, \ldots, A_k : t_k]$ specified by the target list. There's one tuple for each vector of values $\overline{x_i}$ in $X_1 \times \ldots \times X_n$ that simultaneously satisfy the qualifications B_i. An example OODAPLEX query follows.

```
for each d in extent(department) where
   loc(d) = "Boston" and
   for some e in members(d) (salary(e) > 100K)
      [N: name(d), S: salary(manager(d))]
end
```

We chose to focus on this particular subset of OODAPLEX in this paper because it is analogous to SQL (each "for each" loop corresponds to a SQL query block). Hence, this subset provides a good basis for discussing the extensions to conventional query processing techniques that are needed for temporal queries, while allowing us to ignore several problems of query processing in object-oriented databases [MITC91]. For example, in this paper we ignore problems of user-defined types and operations, complex object construction, estimating costs of user-defined methods, and late binding. Those are important problems in their own right, but orthogonal to the issues we discuss here. We do, however, consider the issues of object identity, inter-object references (via functions), navigation (function application, function composition), complex object traversal, and inheritance.

3 OOAlgebra and Query Transformation

3.1 OOAlgebra

The algebra serves two purposes: (a) to define the semantics of OODAPLEX and other query interfaces; and (b) to be the target language for query compilation. While OODAPLEX is based on the function application paradigm, OOAlgebra adopts a relational perspective. In fact, it extends the relational algebra to deal with objects, functions, and subtyping. Here, we only provide a brief overview of the algebra; the reader is referred to [DAYA89, MADA86] for details.

The algebra operates on aggregates (tuples, sets, and multisets). Individual (i.e., non-aggregate) objects are manipulated as if they were 1-tuples. Tuples of objects are, quite naturally, treated as tuples [2]. Sets [3] are treated as sets of tuples (i.e., as relations). Functions are also treated as sets of tuples (viz., their extents) [4].

Figure 1 shows some of the relations corresponding to the schema of the previous section.

3.2 Operations of OOAlgebra

The operations are polymorphic, so that they work on individual tuples and on sets of tuples. However, in this paper, we assume that the inputs and outputs are sets. In the following, $\underline{R}$, $\underline{R1}$, $\underline{R2}$ are relations, L is a list of attribute labels, $\overline{Lin_i : Lout_i}$ is a list of pairs of attribute labels, and B is a Boolean-valued function. When we wish

[2] It is convenient to think that these tuples contain the identifiers of the corresponding objects.

[3] Most operations apply to sets and multisets, so henceforth we will not distinguish between them when there is no risk of ambiguity.

[4] "Computed" functions may result in infinite relations, but that is not a problem since we do not expect to materialize their complete extents.

The *employee* type

salary:

employee	salary
emp1	25K
emp2	35K
emp3	25K

dept :

employee	department
emp1	dept2
emp2	dept3
emp3	dept3

name :

employee	string
emp1	John
emp2	Mary
emp3	Tom

extent(*employee*) :

employee
emp1
emp2
emp3

The *manager* type

controls :

manager	project
emp2	proj2
emp2	proj3
emp3	proj4

extent(*manager*) :

manager
emp2
emp3

Figure 1: Relational Mapping

to emphasize the tuple type of a relation, we use the notation $R[L_1 : T_1, \ldots, L_n : T_n]$, where the L_i are labels and the T_i are types.

The operations *project*(R, L), *select*(R, B), and *product*$(R1, R2)$ are similar to their relational algebra counterparts. The only difference is that *project* returns one tuple per tuple of R, and hence in general produces a multirelation even if R were a relation. The operation *dupelim_project* removes duplicates following a *project*.

The operation *rename*$(R, \overline{Lin_i : Lout_i})$ returns R with each attribute label Lin_i replaced by the respective $Lout_i$.

The usual set operations *union*, *intersection*, and *difference* and their "outer" variants [CODD79] are defined for relations (and multirelations [DAYA82, KLGO85]).

Other operations corresponding to theta_joins and their outer variants (assymetric and symmetric) can be analogously defined.

The main workhorse operation in OOAlgebra is the *apply_append*. In [DAYA89], we defined this operation as the left-outer natural join. The result of *apply_append*$(R1, R2)$ has all the attributes of $R1$ plus those attributes M of $R2$ that are not in $R1$. It consists of every tuple of $R1$ concatenated with the M-value of every tuple of $R2$ that matches on their common attributes; in addition, the unmatched tuples of $R1$ are padded out with *null* values. In practice, this operation is used to mimic the application of a function $R2$ to every member of a finite set $R1$. The common attributes of $R1$ and $R2$ are the input arguments of function $R2$, and the remaining attributes M are the output arguments of $R2$. Informally, the function $R2$ is evaluated over each tuple of $R1$, and the output arguments are then concatenated onto the input arguments. Since $R2$ may be a partial function in general, we used the left-outer natural join. However, for the subset of OODAPLEX that we consider in this paper, we can assume that the *apply_append* operation in fact produces the natural join.

Since we use the algebra to describe query processing, we must say something about how the operations are implemented. *Select*, *project*, *product*, and *join* may be implemented as their relational counterparts are. The *apply_append* operation may be implemented using a variety of algorithms, depending on how the function $R2$ and the set $R1$ are implemented. A stored function may be implemented by associating the input and output arguments as fields in a record (i.e., by directly materializing the result of $R2$ with each record of $R1$); in this case, the *apply_append* can be implemented by a simple file scan. Instead of being associated in the same record, the output objects (result of applying $R2$) may be clustered on physical storage close to the input objects (of $R1$); in this case, the implementation consists of scanning the physical storage units (e.g., pages in an area). Alternatively, the input objects may include references to (object identifiers of) the corresponding output objects. If the references are implemented by physical pointers, the *apply_append* may be implemented by scanning the input objects, and then following the pointers. Alternatively, if the pointers are logical, any join algorithm (e.g., sort-merge or indexed join) may be used. For computed functions, the *apply_append* is typically implemented by scanning (iterating over) the input objects, and invoking the function's method once in each iteration.

3.3 Mapping from OODAPLEX to OOAlgebra

The mapping of an OODAPLEX query into an equivalent OOAlgebra program follows the same general approach used to map SQL queries into relational algebra prior to optimization. First, the set X_1 in the outermost loop is materialized, and restricted (via *select* operations) by the qualification B_1. Then, it is *joined* with the set X_2 and restricted by the corresponding qualification B_2, and so on, working from the outermost loop to the innermost. If there are any existentially quantified variables in any qualification, then the ranges of these variables are also *joined* in, and then these variables are eliminated by doing a *dupelim_project*. Finally, the result is obtained by *projecting* over the target list [$A_1 : t_1, A_2 : t_2, \ldots, A_k : t_k$]. (See [DAYA82, DAYA87] for details.)

There are a few complications, however. First, the qualifications and target list may include function application terms. Because we represent functions by relations, these terms must be mapped to *apply_append* operations. Thus, resolving each qualification entails more than just a selection; *apply_appends* may first be necessary. Similarly, before *projecting* on the target list, *apply_appends* may be necessary to bring all the relevant attributes into a single relation.

Second, the set terms that define the ranges of loop variables may be either named sets or function application terms. The latter are mapped to *apply_append* operations. For the former, the mapping algorithm constructs a *product* of this set with the intermediate result produced so far; the optimizer subsequently tries to replace these *products* by less expensive operations such as *joins* or *apply_appends*.

Third, because the *apply_append* operation matches labels, operands may have to be *renamed*. For every expression of the form x *in* X, the label of the column of x's type in X must be changed to x. Also, for every expression of the form y *in* $f(x)$, the result of the *apply_append* must be *renamed* so that its output column of y's type is labelled y.

Fourth, the mapping algorithm must deal with inheritance. Thus, if S is a subtype of T, variable s ranges over a set X_s of objects of type S, and f is a function defined for T, then the function application term $f(s)$ is mapped into *apply_append*(X_s, f). Since the relation f associates object identifiers of T objects with output objects, and S objects have the same identifiers as T objects, the *apply_append* operation correctly matches tuples of X_s with tuples of f. Note that only some *renaming* may be necessary to match the input argument label in f with s.

Instead of formally stating the algorithm, we illustrate it on the example OODAPLEX query of the previous section.[5]

```
T1  := extent(department)[D:department]
T2  := a_a(T1[D: department],
          loc[D: department, L:city])
T3  := select(T2[D:department, L:city],
          equals(L, "Boston"))
T4  := a_a(T3, members[D:department, E:employee])
T5  := a_a(T4[D:department, L:city, E:employee],
          salary[E: employee, S: money])
T6  := select(T5[D:department, L:city, E:employee,
              S:money], greater_than(S, 100K))
T7  := dupelim_project(T6, [D])
T8  := a_a(T7, name[D:department, N:string])
T9  := a_a(T8[D:department, N:string],
          mgr[D:department, M:manager])
T10 := rename(salary[E: employee, S:money], [E:M])
T11 := a_a(T9[D:department, N:string, M:manager),
          T10[M:employee, S:money])
T12 := project(T11[ D:department, N:string,
              M:manager, S:money], [N, S])
```

T2 associates cities with departments via the *loc* function, enabling the selection in T3, which corresponds to the first conjunct of the qualification. T4 associates employees with each department-city tuple via the *members* function, and T5 then associates the salary of each employee, enabling the selection in T6. Since E is existentially quantified, duplicates must be eliminated in T7 (otherwise a Boston department with more than one employee earning over 100K will appear more than once in the result). This completes the processing of the qualification. To compute the target list, T8, T9, and T11 associate each selected department with its name, its manager, and the salary of its manager. The renaming in T10 is necessary to correctly match the label of T9's *manager* attribute with the *employee* label in the *salary* function, so that the manager and his salary can be associated. Finally, projecting on *name* and *salary* yields the result in T12.

[5]In the examples, we abbreviate *apply_append* as a_a.

Of course, a query optimizer may rearrange the operations to construct an equivalent plan that is cheaper to evaluate. The transformations used are analogous to those for relational optimization, since the operations have similar properties. For instance, *selections* and *projections* may be moved ahead of *apply_appends* and *joins*; e.g., the *selections* in T3 and T6 can be done before any of the preceding operations. Some *apply_appends* may be unnecessary; e.g., if we know that an inclusion dependency holds from the D column of *loc* to the D column of *extent*(*department*), then steps T1 and T2 can be eliminated, and *loc* used directly in step T3. The order of *apply_appends* may be permuted. Some *product* operations introduced by the mapping algorithm that are followed by *selections* may be replaced by *joins* or *apply_appends*.

4 Temporal Extension from the Base OODAPLEX

In this section, we show how the rich object-oriented type system can be used to model temporal information. As an example, we model temporal employees, temporal departments, and temporal relationships between them using the function type constructor.

```
type employee is object
    function name (e:employee → n:string)
    function salary (e:employee →
                        s: (t:time → s:money))
    function dept (e:employee →
                        f: (t:time → d:department))
type department is object
    function name (d:department → n:string)
    function members (e:department →
                        f: (t:time → m:{employee}))
    function mgr (e:department →
                        f: (t:time → m:manager))
```

In our approach, time is modelled by abstract data types. We first define a generic object type, *time*, that carries the most general semantics of time. The > operation is defined for the *time* type. Various different notions of time required by specific applications (linear vs. branching, discrete vs. dense, events, or versions) can be extensibly defined as subtypes of *time*. Instances of the time types have the semantics of time points. Since the model supports polymorphism (overriding), different implementations (and representations) may exist for the same time types while the interfaces of these types remain the same. For each time type, a set type is defined; instances of these set types are point sets. The usual set operations and predicates like $\in$, $\subset$, $=$, $\cap$, and $\cup$, are inherited, but may be overridden due to different implementations of the time types.

To incorporate the additional time-related semantics into the system, OODAPLEX defines several new functions and constraints. [WUDA92] describes the maintenance of these functions and constraints. In this paper, we only discuss the constraints that are relevant to query processing. Other constraints like temporal referential integrity constraints are described in [WUDA92].

The function *lifespan*(o), where o is an object, returns the set of time points during which o existed. *Lifespan* is a polymorphic function. It also accepts a type T or a database DB as it input paramenter. The function *lifespan*(T) returns the union of lifespans of all objects in *extent*(T). The function *lifespan*(DB) returns the union of lifespans of all types in DB.

In temporal databases, an object can be of different types at different times. Consider the following subtypes of *employee*.

```
type manager is employee
   function controls (e:manager →
                        f: (t:time → d:{project}))
type technician is employee
   function owns (e:technician →
                        f: (t:time → d:{tool}))
```

An employee may have been a technician *for a while* and may *later* have been promoted to a manager. Therefore, the lifespan of the employee as an instance of type *manager* (or *technician*) can be different from that of the same employee as type *technician*. We denote the lifespan of an object o as an instance of type T as *lifespan*(o/T).

A basic constraint exists between lifespans of objects, types, and the database DB, as defined below. Let o be an object of type T in database DB.

(C1) $lifespan(o/T) \subseteq lifespan(T) \subseteq lifespan(DB)$

For a partial function g, we use the notation $|g|$ to denote the subset of g's domain over which g is defined. For a temporal attribute f_i of a temporal object o (e.g., *salary* of *employee*), the following constraint is maintained:

(C2) $|f_i(o)| \subseteq lifespan(o)$

In temporal databases, the notion of inheritance is also temporally extended. Let o be an object of both a subtype T and a supertype P. The following constraint must hold.

(C3) $lifespan(o/T) \subseteq lifespan(o/P)$

Because temporal objects can move in and out of the extent of a type T (e.g., *manager*) at different times, the temporal model defines another function, *t_extent*$(P)(t)$, called the *temporal extent function*, that returns a set of objects o of type P that existed at time t. For example, *t_extent*(*employee*)(t) returns the set of employee objects that existed at time t. We formally define the temporal extent function by the following constraint.

(C4) $o \in t\_extent(P)(t) \Longleftrightarrow t \in lifespan(o/P)$

The non-temporal function *extent*(P) defined in the base OODAPLEX model is still available in temporal databases. It actually maintains all objects of type P that have *ever* existed in the database. The inclusion semantics of the base OODAPLEX model remains the same: if T is a subtype of P, then *extent*$(T) \subseteq$ *extent*(P). The relationship between *extent* and *t_extent* is defined by the following constraint.

(C5) $o \in extent(P) \Longleftrightarrow \exists t \in lifespan(o)$
$o \in t\_extent(P)(t)$

It can be derived from (C3) and (C4) that, for a pair of subtype T and supertype P, and a time point t,

(C6) $t\_extent(T)(t) \subseteq t\_extent(P)(t)$

(C6) defines the meaning of temporal inclusion. Due to (C4) and (C5), both the *t_extent* and *extent* functions can be derived from the *lifespan* function. Therefore, there is no need to physically maintain all three functions in the database. As discussed later in Section 5.1, only *lifespan* is physically maintained in the database.

When applying an inherited function, (C2) requires that the result of such application be temporally restricted. Consider the salary function defined over the employee type. The result of the function, when applied to an *employee* object, is a salary function defined for the duration of the employee's lifespan. When applied to the object as an instance of type *manager*, the result cannot be the entire salary history of the employee, since this would violate the (C2) constraint. Instead, the result should be the restriction of the temporal *salary* function to the lifespan of the object as a *manager*.

5 Temporal Data Representation and Indexing

5.1 Physical Representation of Temporal Data

In OOAlgebra, individual objects are represented by *object identifiers* (*oid*'s), Functions are treated as relations in the first normal form. As shown in Figure 2, the *name* function with the type *employee → string* is represented as a relation.

A function type that returns another function as its result, e.g., $T_1 \rightarrow (T_2 \rightarrow T_3)$, can be represented by two separate relations corresponding the two functions. However, to simplify the representation, we transform such a function type into another equivalent form $(T_1 \times T_2) \rightarrow T_3$, a function with two input arguments and one output argument. Hence it is represented as a single relation with three columns corresponding to T_1, T_2, and T_3. As shown in Figure 2, temporal attributes like *salary* or *dept* are represented as single relations with three columns.

Figure 2 shows a *canonical* representation of a temporal Company database which represents the OOAlgebra view of data. The physical data representation may or may not be identical to the canonical form. For example, the *name*, *salary*, and *dept* functions may be physically stored as a single file, and each function can be implemented by a view that projects on the respective columns. Different physical representations may affect the execution plan produced by the query optimizer. In this paper, we assume that the physical representation is identical to the canonical OOAlgebra representation.

Conceptually in OODAPLEX and OOAlgebra, time points are viewed as individual objects. Physically storing these time points as individual objects would not be efficient for the storage system. Most temporal data storage models use intervals (pairs of begin and end points) to represent sets of consecutive time points.

In our temporal storage structure, an arbitrary set of time points (a relation with one column) is physically represented as a set of non-overlapping intervals in a relation.

Each tuple in the relation contains exactly one interval. For example, the point set, {1, 2, 7, 8, 9} is represented by two physical tuples $<[1,2]>$ and $<[7,9]>$. For simplicity we assume, in the following examples, that the time domain is discrete and bounded, hence closed intervals are enough to represent arbitrary sets. Nevertheless, the same representation technique also applies to dense or continuous time domains using combinations of closed and open ended intervals for arbitrary sets.

For relations containing a time column, we use the same technique. For example, the physical tuple < emp1, [0,19], 20K > in the *salary* relation is really a compact representation for the twenty conceptual tuples corresponding to twenty individual time points, {< emp1, 0, 20K >, < emp1, 1, 20K >, ..., < emp1, 19, 20K >} Again, the time column contains at most one interval.

We assume that each tuple in the storage system is uniquely identified by a *tuple identifier*, which should not be confused with object identifiers. In Figure 2, we only show tuple identifiers for the salary relation, since they will be used later for describing the TIME index.

For each temporal object type, P, the database keeps one relation for $lifespan(P)$. The relation has two columns maintaining the mapping from *oid*'s of P to time, as shown in Figure 2. The extent of P can be easily computed or materialized by projecting on the P column of the lifespan table. Note that the (C2) constraint is indeed maintained in the database.

OODAPLEX defines a temporal extent function, $t\_extent(P)(t)$, which gives the set of instances of P that exist at time t. The $t\_extent(P)$ table can be computed (or materialized) from the $lifespan(P)$ table using the (C4) definition. We can think of the $lifespan(P)$ as a table "grouped by" P's *oid*'s, and the $t\_extent(P)$ table as the same table "grouped by" time points. The materialized view of $t\_extent(employee)$, in its flattened first normal form, is shown in Figure 2. As we shall discuss later, the TIME index is an ideal data structure to compute the $t\_extent(P)$ relation from the $lifespan(P)$ relation.

We now discuss the representation of the temporal inheritance hierarchy. Consider a pair of subtype S and supertype P. Functions defined for S, e.g., the *controls* function of the the manager subtype, are represented in the same way as for other types, as shown in Figure 2. The table $lifespan(manager)$ shows that the lifespan of the employee *emp2* as a manager is {[13,21], [39,now]}. This means that *emp2*, was promoted to manager and demoted from manager twice during her career with the company. Note that the (C3) constraint on *emp2* is indeed maintained in the database, i.e.,

$$lifespan(emp2/manager) = \{[13,21],[39,now]\} \subseteq \{[13,now]\} = lifespan(emp2/employee)$$

5.2 The TIME index

Temporal queries often impose search conditions on time attributes as well as on non-temporal attributes. Traditional indexing methods like B+ trees can only be used for indexed values that are totally ordered. Such index methods cannot be used for interval values, since a total ordering

The *employee* type

salary:

tid	employee	time	salary
s1	emp1	[0,19]	20K
s2	emp1	[20,29]	22K
s3	emp1	[30,now]	25K
s4	emp2	[13,44]	30K
s5	emp2	[45,now]	35K
s6	emp3	[9,19]	22K
s7	emp3	[20,24]	25K

name :

employee	string
emp1	John
emp2	Mary
emp3	Tom

dept :

employee	time	department
emp1	[0,34]	dept1
emp1	[35,now]	dept2
emp2	[13,65]	dept2
emp2	[66,now]	dept3
emp3	[9,24]	dept3

extent(*employee*) :

employee
emp1
emp2
emp3

t_extent(*employee*):

time	employee
[0,8]	emp1
[9,12]	emp1
[9,12]	emp2
[13,24]	emp1
[13,24]	emp2
[13,24]	emp3
[24,now]	emp1
[24,now]	emp2

lifespan(*employee*) :

employee	time
emp1	[0,now]
emp2	[13,now]
emp3	[9,24]

The *department* type

dname:

department	string
dept1	Shoes
dept2	Clothes
dept3	Appliance

lifespan(*department*) :

department	time
dept1	[13,now]
dept2	[20,now]
dept3	[0,67]

mgr:

department	time	manager
dept1	[13,21]	emp2
dept1	[22,38]	emp3
dept2	[39,now]	emp2
dept3	[51,67]	emp3

The *manager* type

controls :

manager	time	project
emp2	[13,21]	proj1
emp2	[39,67]	proj2
emp2	[39,67]	proj3
emp3	[5,now]	proj4

lifespan(*manager*) :

manager	time
emp2	[13,21]
emp2	[39,now]
emp3	[22,38]
emp3	[51,67]

Figure 2: The Company Database

does not exist for interval values. In [ELMA90, ELMA91], we proposed an access structure, called the *TIME index*, to support interval-based search. The goal of a temporal search is to find, given a time range R, the tuples whose time columns overlap with R. As a special case, R can be just a single time point.

In the following, we give an overview of the TIME index. It can be used to index any table in Figure 2 that has a time column. The TIME index plays an important role in the temporal query processing techniques proposed in this paper. Other temporal access structures like [KOST89, SEGU89, GUSE91] may also be used to perform time-based search.

The idea behind the TIME index is to maintain a set of linearly ordered *change points*. A change point pt_i is a time point which either (a) starts an interval in the time column or (b) follows immediately the end point of an interval. Consider the salary table. The set of change points are {0, 9, 13, 20, 25, 30, 45, now+1}. These points are called change points because the state of the salary table changes only at these points. For example, the state of the salary table changes at 9, 20, and 25, because *emp*3 started its salary history at 9, got its salary raise at 20, and terminated at 25. If a TIME index is built for the *lifespan*(*employee*) table, the change points in the TIME index correspond to the creation or termination times of *employee* objects.

We use a regular B+ tree to index these linearly ordered change points. For each change point pt_i at the bottom level of the B+ tree, the TIME index maintains a bucket $B(pt_i)$ containing identifiers of all tuples that existed at pt_i, i.e., their time intervals contain pt_i. Consider the change point 30. Its bucket $B(30)$ contains the tuples $s3$ and $s4$, since only these two tuples existed at point 30 (i.e., their time intervals cover point 30). The TIME index for the salary relation is shown in Figure 3.

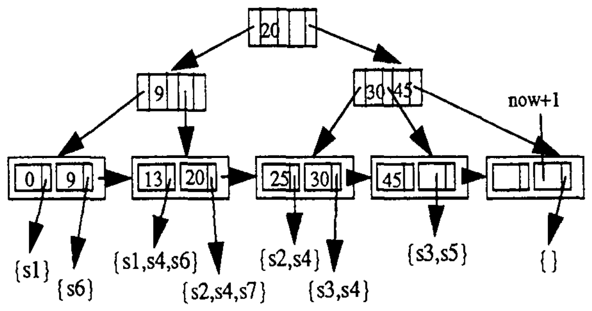

Figure 3: The TIME Index for the *salary* Relation

A bucket $B(pt_i)$ contains all temporal information at change point pt_i. It is important to note that, after the temporal information changes at a change point, it remains the same for the entire salary table until the next change point. Therefore to retrieve temporal data at an arbitrary point p, the following search algorithm is used:

```
Search the B+ tree for $p$;
If $p$ is found
  the temporal information is found in the bucket $B(p)$;
else
  find the change point $q$ in the B+ tree that
  immediately precedes $p$, and $B(q)$ contains
  the temporal information at $p$.
```

The above search algorithm can be extended to support range search as follows. Consider the search range $R = [t_s, t_e]$. First, perform a range search on the B+ tree to find the set S of all change points within R. If t_s is not a change point in the B+ tree, add the change point q that immediately precedes t_s to S. The reason we need to add q to S is because the the temporal information at t_s is recorded at q. After S is retrieved, compute the following to retrieve all temporal information in the range R.

$$\bigcup_{p_i \in S} B(P_i)$$

The basic TIME index is not very efficient in space, since a tuple (e.g., s4 in Figure 3) with a large interval may appear repeatedly in all buckets whose change points are within the interval. To eliminate the duplicate tuple identifiers in these buckets, an incremental scheme is proposed in [ELMA90]; that is, to keep full buckets at the leading entry of each leaf node in the B+ tree, and to keep only incremental changes at the following buckets. A complete description of the TIME index, its enhancements, and their performance results are reported in [ELMA90, ELMA91, ELMA92].

6 Temporal Query Processing

In this section, we show the translation between OODAPLEX temporal queries and OOAlgebra queries, and discuss techniques for processing temporal queries. OODAPLEX and OOAlgebra treat time as points and sets of points. The notion of points or point sets is logical, since they are physically represented as intervals in the storage system, rather than as individual points. Depending on the physical representation and available indices, the algorithms for evaluating the algebraic operators may vary. In any case, the results of these algorithms must fully conform to the semantics of the algebraic operations. We will describe several algorithms for implementing certain important OOAlgebra operators that involve time attributes. We also informally argue that these algorithms conform to the semantics of OOAlgebra.

6.1 Temporal Apply_append Operation

In this subsection, we use the following query to describe algorithms for implementing the *apply_append* operation involving time attributes.

OODAPLEX Query 1:

```
for each e in extent(employee)
   for each t in { [1..3], [18..25] }
      [N:name(e), T:t, S:salary(e)(t)]
   end
```

end

OOAlgebra Query 1:

```
T1 := { [1,3], [18,25] }
T2 := product(extent(employee)[E:employee], T1[T:time])
T3 := a_a(T2[E:employee, T:time],
        name[E:person, N:string])
T4 := a_a(T3[E:employee, T:time, N:string],
        salary[E:employee, T:time, S:numeric])
T5 := project(T4[E:employee, N:string, T:time, S:numeric],
            [N, T, S])
```

T1 has only two tuples containing the two intervals $< [1..3] >$ and $< [18..25] >$. The *product* operator is implemented using the same algorithms as for non-temporal data. The result of T2 is shown in Figure 4. It can be easily argued that the result conforms to the semantics of the Cartesian product between employees and logical points. There are two *apply_append* operations in the OOAlgebra translation. The first one (T3) applies the *name* function over a non-temporal input argument, *employee*; and it requires no new techniques for its implementation. However, the second *apply_append* operation (T4) joins tuples based on both temporal (*time*) and non-temporal (*employee*) attributes; and therefore it requires new algorithms.

We call an *apply_append* operation that is based on at least one common time attribute a *temporal apply_append* operation. Based on the *apply_append* semantics, the evaluation of a temporal *apply_append*(X, Y) is described as follows. Assume that X and Y have schemes (A, NT, T) and (NT, T, B), respectively, where NT is the non-temporal common column(s) and T is the temporal common column(s). The result R of the temporal *apply_append* is defined as follows.

```
let $t$ be a tuple in $X$
let $q$ be a tuple in $Y$
the tuple $< t.A, t.NT, t.T \cap q.T, q.B >$ is in $R$
    if and only if $t.NT = q.NT$ and $t.T \cap q.T \neq \emptyset$
```

It is not difficult to argue that this evaluation conforms to the semantics of the *apply_append* operator, since the time columns of t and q agree on all points (and only those points) in $t.T \cap q.T$. Based on this definition, the temporal *apply_append* of Query 1 is evaluated to the result (T4) shown in Figure 4.

The above temporal *apply_append* operation can be implemented by various algorithms, depending on available index structures on X and Y. A straightforward implementation uses a modified nested loop algorithm, as described below. For each tuple in X, scan through the entire Y table to find all tuples that meet the "matching" condition, and then produce concatenated tuples accordingly, i.e., such that the resulting time is the intersection of the time intervals in the two tuples being joined.

Other algorithms may also be developed to take advantage of efficient access structures. For example, an algorithm proposed in [ELMA90] uses a two-level index tree, which combines the TIME index and the non-temporal index into one index tree. If the two-level index is available on Y, it can be used to find tuples in Y that match tuples in X. Algorithms based on the sort-merge or hash-join approaches may also be developed for temporal *apply_append*. However it is beyond the scope of this paper to elaborate on the various algorithms and their associated costs.

T2 :

employee	time
emp1	[1,3]
emp1	[18,25]
emp2	[1,3]
emp2	[18,25]
emp3	[1,3]
emp3	[18,25]

T4 :

employee	name	time	salary
emp1	John	[1,3]	20K
emp1	John	[18,19]	20K
emp1	John	[20,25]	22K
emp2	Mary	[18,25]	30K
emp3	Tom	[18,19]	22K
emp3	Tom	[20,24]	25K

Figure 4: Intermediate Results of Query 1

6.2 Temporal Selection using the TIME Index

We have discussed various methods for implementing temporal *apply_append*. We now consider the following query to explore some temporal query optimization alternatives.

OODAPLEX Query 2:

```
for each e in extent(employee) where
    for some t in [2 .. 5] (salary(e)(t) > 35K)
        [N: name(e)]
end
```

OOAlgebra Query 2:

```
T1 := {[2, 5]}
T2 := product( extent(employee)[E:employee],
            T1[T:time])
T3 := a_a(T2[E:employee, T:time],
        salary[E:employee, T:time, S:numeric])
T4 := select(T3, greater_than(S, 35K))
T5 := a_a(T4, name[E:employee, N:string])
T6 := dupelim_project(T5[E:employee, T:time,
                    S:salary, N:string], [N])
```

We will show the second and third operations (in T2 and T3) in Query 2 can be transformed into one single equivalent operation, which can be executed more efficiently using the TIME index. By an associative rule analogous to that for join operations, the two steps are transformed to

```
T2' := a_a(extent(employee)[E:employee],
        salary[E:employee, T:time, S:numeric])
T3' := a_a(T1[T:time],
        T2'[E:employee, T:time, S:numeric])
```

Since an inclusion dependency holds from the E column of *salary* to the E column of *extent(employee)*, step T2' can be eliminated and *salary* can be directly used in T3'. (The same transformation can be applied to Query 1.)

```
T3" := a_a(T1[T:time],
          salary[E:employee, T:time, S:numeric])
```

The merged *apply_append* in T3" joins the two relations only by the common time attribute. This is a special case of the temporal *apply_append* described earlier. The TIME index can be directly used to efficiently compute the result. For each interval in T1 (in the case of Query 2, there is only one interval), use the TIME index to find all tuples in the salary relation that satisfy the time overlap condition, and then produce the concatenated tuples accordingly, i.e., such that the resulting time is the intersection of the two overlapping time intervals.

Note that the new *apply_append* is basically a *select* operation with a range search condition, viz., "t in [2..5]". It commutes with the following *select* operation that has a search condition on the salary value, greater_than(S, 35K). This commutativity property provides further alternatives for query optimization. Depending on the selectivities of the two *select* operations and available access structures on time or salary attributes, the order of evaluation of these two operations can be reversed.

6.3 Temporal Inheritance

Since additional constraints, (C2) and (C3), are defined for temporal inheritance (see Section 4), the processing of inheritance must be extended. This is illustrated by the following query, which retrieves the salary history for the current manager of the Clothes department.

OODAPLEX Query 3:

```
for each d in extent(department)
            where name(d) = "Clothes"
   [salary(mgr(d)("now"))]
end
```

OOAlgebra Query 3:

```
T1 := a_a(extent(department)[D:department],
          name[D:department, N:string])
T2 := select(T1, equals(N, "Clothes"))
T3 := a_a(T2[D:department, N:name],
          mgr[D:department, T:time, M:manager])
T4 := select(T3, equals(T, "now"))
T5 := a_a(T4, lifespan(manager)[M:manager,T:time])
T6 := rename(salary[E:employee, T:time, S:numeric],
             [E:M])
T7 := a_a(T5[D:department,N:name,M:manager,T:time],
          T6[M:employee,T:time,S:numeric])
T8 := project(T7[D:department,N:name,M:manager,
              T:time,S:numeric], [T, S])
```

Due to the extended semantics of temporal inheritance, applying an inherited temporal function, e.g., *salary*, needs special consideration. By (C2) and (C6), the salary of the selected manager must be restricted to the lifespan of the employee as manager. The *apply_append* operation in T5 associates the lifespan information with manager, which is then used in the apply_append of T7 to restrict the salary history to just the lifespan of the selected manager.

6.4 Complex Queries, Quantification, Aggregation

As shown in [WUDA92], complex temporal queries may involve multiple time variables, quantification (existential or universal) over time, and temporal aggregation. Some of these complex queries cannot be easily expressed in other existing temporal query languages. In our uniform approach, not only can we express such queries naturally, but we can also process them effectively.

OODAPLEX Query 4:

```
for each e in extent(employee) where name(e) = "John"
   for each t in lifespan(e) where salary(e)(t) < 20K
      [T: t, M: name(mgr(dept(e)(t))(t))]
   end
end
```

Query 4 retrieves all managers for whom John has worked when his salary was less than $20K, and the time periods during which he worked for these managers. The processing of such a complex query requires no extensions to the techniques described above, and hence we omit the OOAlgebra translation.

As discussed in [WUDA92], quantification and aggregation (and grouping) in temporal databases can be expressed over two orthogonal dimensions: the object dimension and the time dimension. Using the full power of OODAPLEX (not just the conjunctive, existential subset defined in Section 2), we can easily express such queries. We now discuss how they can be processed using existing techniques and the TIME index.

Several techniques have been proposed in the literature for efficiently processing (non-temporal) queries with aggregates and quantifiers [DAYA83, CEGO85, DAYA87, BULT87]. These techniques introduced some new algebraic operators and their transformation rules. We can easily import these operators into OOAlgebra. Then, because OODAPLEX uniformly expresses temporal and non-temporal queries, these same techniques can be used to process temporal queries that have quantifiers and aggregates. (However the implementation of these operators may need to be modified to take advantage of temporal data representations and access paths such as the TIME index.)

Recall that Query 2 used an existential quantifier on time, and was processed using an *apply_append* and a *dupelim_project*. In fact, these could have been replaced by a *semi_join*. The following query uses a universal quantifier, and can be processed using an *anti_join* operator (*anti_join* is the complement of *semi_join*). Query 5 retrieves employees whose salaries exceeded 35K during the entire interval [2..5], i.e., employees whose salaries were never less than or equal to 35K, or *null*, during this period.

OODAPLEX Query 5:

```
for each e in extent(employee) where
    for all t in [2 .. 5] (salary(e)(t) > 35K)
        [N: name(e)]
end
```

OOAlgebra Query 5:

```
T1 := {[2,5]} [T:time]
T2 := a_a(T1 [T:time],
          salary [E:employee, T:time, S:numeric])
T3 := select(T2 [E:employee, T:time, S:numeric],
             less_eq_or_null(S, 35K) )
T4 := anti_join(extent(employee) [E:employee],
                T3[E:employee, T:time, S:numeric])
T5 := a_a(T4[E:employee], name[E:employee, N:string])
T6 := project(T5, [N])
```

Query 6 retrieves the running count of the employees in the company over the history of the company.

OODAPLEX Query 6:

```
for each t in lifespan(employee)
    [t, count(t_extent(employee)(t))]
end
```

OOAlgebra Query 6:

```
T1 := a_a(lifespan(employee)[T:time],
          t_extent[T:time, E:employee])
T2 := G_Agg(T1, [T], [count(E)])
```

As defined in [DAYA87], the Generalized Aggregation operator, *G_Agg*, takes three parameters: a relation, a list of grouping attributes, and a list of aggregate functions. It groups the relation by the grouping attributes, and applies the aggregate functions on each group.

Since $lifespan(employee)$ is the union of lifespans of all employee objects, and since (C4) and (C5) hold, it can be shown that $lifespan(employee) = project(t\_extent(employee)[T, E]; [T])$. Therefore, the *apply_append* operation does not produce more information than the *t_extent* relation itself. Thus, the two steps can be combined into one.

```
T1 := G_Agg(t_extent(employee) [T:time, E:employee],
            [T], [count(E)])
```

If a TIME index exists for the $lifespan(employee)$ table, there is no need to materialize the *t_extent* table. The G_Agg operation can be efficiently computed using the TIME index, as described below.

Starting from the earliest change point pt_s in the TIME index, retrieve all tuples in the bucket $B(pt_s)$. From these tuples, compute the total number of *distinct* employee *oid*'s. The result number represents the employee count at point pt_s. Since the employee counts remain the same until the next change point, there is no need to repeatedly compute the employee counts for those points between two consecutive change points. To return the entire running count, the same counting procedure proceeds at all subsequent change points in the TIME index.

7 Conclusion

We have presented a uniform approach to temporal query processing. Rather than modifying a non-temporal data model and query language to work for temporal databases, we rely on a rich object-oriented type system for modeling temporal information. We use the same query language, OODAPLEX, to manipulate non-temporal and temporal data. To incorporate additional temporal semantics into the system, we define several temporal functions and constraints. We achieve this without modifying the base object-oriented model or language. A major benefit to this uniform approach is that the existing, non-temporal, query processing framework can be smoothly extended for temporal databases.

We presented an object algebra, OOAlgebra, to be used in query optimization, and showed the translation between temporal OODAPLEX queries and their OOAlgebra forms. We showed how to use the additional temporal semantics defined by the constraints can be used in query optimization. We described a physical representation of temporal data and the TIME index for efficiently accessing temporal data. The new data representation and index methods call for new algorithms for implementing the OOAlgebra operators (e.g., *apply_append*) involving time attributes, which are also described in the paper. We believe that the techniques developed here can be applied to other models and query languages.

Future work includes detailed study of these new temporal algorithms and their cost analysis. Also, more work can be done to incorporate our techniques for temporal query processing into a fully extensible, object-oriented query optimization framework.

References

[ATKI89] M.P. Atkinson, et al. The Object-Oriented Database System Manifesto.

[AHN86] I. Ahn. Towards an Implementation of Database Management Systems with Temporal Support. *Proc., Second IEEE Data Engineering Conf.*, 1986.

[BULT87] G. von Bultzingsloewen. Translating and Optimizing SQL Queries Having Aggregates. *Proc. Thirteenth VLDB Conf.*, 1987.

[CEGO85] S. Ceri, G. Gottlob. Translating SQL into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries. *IEEE Trans. SE-11*, 4, April 1985.

[CLTA85] J. Clifford, A. Tansel. On an Algebra for Historical Relational Databases: Two Views. *Proc., ACM SIGMOD Conf.*, 1985.

[CODD79] E.F.Codd. Extending the Database Relational Model to Capture More Meaning. *ACM Trans. on Database Systems*, 4(4), December 1979.

[DAYA82] U. Dayal, N. Goodman, R.H. Katz. An Extended Relational Algebra with Control over Duplicate Elimination. *Proc., ACM Symp. on Principles of Database Systems*, 1982.

[DAYA83] U. Dayal. Processing Queries with Quantifiers: A Horticultural Approach. *Proc., ACM Symp. on Principles of Database Systems*, 1983.

[DAYA87] U. Dayal. Of Nests and Trees: A Unified Approach to Processing Queries that Contain Nested Subqueries, Aggregates, and Quantifiers. *Proc., Thirteenth VLDB Conf.*, 1987.

[DAYA89] U. Dayal. Queries and Views in an Object-Oriented Data Model. In *Proc., Second International Workshop on Database Programming Languages*, 1989.

[ELMA90] R. Elmasri, G. Wuu, Y. Kim. *The Time Index: An Access Structure for Temporal Data*, In *Proc., Sixteenth VLDB Conf.*, August 1990.

[ELMA91] R. Elmasri, Y. Kim, G. Wuu. Efficient Implementation Techniques for the Time Index, *Proc., Seventh IEEE Data Engineering Conf.*, April 1991.

[ELMA92] R. Elmasri, M. Jaseemuddin, V. Kouramajian. Partitioning of Time Index for Optical Disks *Proc., Eighth IEEE Data Engineering Conf.*, February 1992.

[ELWU90] R. Elmasri and G. Wuu. A Temporal Model and Language for ER Databases. *Proc., Sixth IEEE Data Engineering Conf.*, February 1990.

[GAMC91] D. Gabbay, P. McBrien. Temporal Logic and Historical Databases. *Proc., Seventeenth VLDB Conf.*, 1991.

[GAYE88] S. Gadia, C. Yeung. A Generalized Model for a Temporal Relational Database. *Proc., ACM SIGMOD Conf.*, 1988.

[GUSE91] H. Gunadhi, A. Segev. Efficient Indexing Methods for Temporal Relations. Lawrence Berkeley Lab Technical Report LBL-28798

[KAFE90] W. Kafer, N. Ritter, H. Schoning. Support for Temporal Data by Complex Objects. *Proc., VLDB Conf.*, 1990.

[KLGO85] A. Klausner, N. Goodman. Multirelations — Semantics and Languages. *Proc., Eleventh VLDB Conf.*, 1985.

[KOST89] C. Kolovson, M. Stonebraker. Indexing Techniques for Historical Databases. *Proc., Fifth IEEE Data Engineering Conf.*, February 1989.

[LOSA89] D. Lomet, B. Salzberg. Access Methods for Multiversion Data. In *Proc., ACM SIGMOD Conf.*, 1989.

[LUM84] V. Lum, et al. Designing DBMS Support for the Temporal Dimension. In *Proc., ACM SIGMOD Conf.*, 1984.

[MADA86] F. Manola, U. Dayal. PDM: An Object-Oriented Data Model. *Proc., International Workshop on Object-Oriented Database Systems*, 1986.

[MITC91] G. Mitchell, S.B. Zdonik, U. Dayal. *Object-Oriented Query Optimization: What's the Problem?* Tech. Report CS-91-41, Brown University, June 1991.

[NAAH89] S. B. Navathe, R. Ahmed. A Temporal Relational Model and a Query Language. In *Information Sciences*, North Holland, 49(1,2,3), 1989.

[ROSE87] D. Rotem, A. Segev. Physical Organization of Temporal Data, *proceedings, IEEE Data Engineering Conference*, 1987.

[ROSE91] E. Rose, A. Segev. TOODM - A Temporal Object-Oriented Data Model with Temporal Constraints. *10th International E-R Conf.*, October, 1991.

[SEGU89] A. Segev, H. Gunadhi. Event-Join Optimization in Temporal Relational Databases *Proc., Fifteenth VLDB Conf.*, August 1989.

[SHIP81] D. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1), March 1981.

[SMIT83] J.M. Smith, S.A. Fox, T.A. Landers. *ADAPLEX: Rationale and Reference Manual.* Technical Report CCA-83-08, Computer Corporation of America May 1983.

[SNOD87] R. Snodgrass. *The Temporal Query Language TQuel. ACM Trans. on Database Systems*, 12(2) 1987.

[STSN88] Stam, R., and R. Snodgrass. A Bibliography on Temporal Databases. *IEEE Bulletin on Data Engineering*, Vol. 11, No. 4, Dec. 1988.

[SUCH91] Su, S.Y.W., H.H.M Chen. A Temporal Knowledge Representation Model OSAM*/T and Its Query Language OQL/T. *Proc., Seventeenth VLDB Conf.*, 1991.

[TAGA89] A. U. Tansel, L. Garnett. Nested Historical Relations *Proc., ACM SIGMOD Conf.*, 1989.

[TUCL90] A. Tuzhilin, J. Clifford. A Temporal Relational Algebra as a Basis for Temporal Relational Completeness *Proc., Sixteenth VLDB Conf.*, 1990.

[WUDA91] G. Wuu, U. Dayal. A Uniform Model for Temporal and Versioned Object-Oriented Databases. *Digital Equipment Corp., Cambridge Research Lab., CRL-91/11*, November, 1991.

[WUDA92] G. Wuu, U. Dayal. A Uniform Model for temporal Object-Oriented databases. *Proc., Eighth IEEE Data Engineering Conf.*, February 1992.