A Query Calculus for Spatio-Temporal Object Databases

Tony Griffiths, Alvaro A. A. Fernandes, Nassima Djafri, Norman W. Paton
Department of Computer Science, University of Manchester
Oxford Road, Manchester M13 9PL, United Kingdom
{griffitt|a.fernandes|ndjafri|norm}@cs.man.ac.uk

Abstract

The development of any comprehensive proposal for spatio-temporal databases involves significant extensions to many aspects of a non-spatio-temporal architecture. One aspect that has received less attention than most is the development of a query calculus that can be used to provide a semantics for spatio-temporal queries and underpin an effective query optimization and evaluation framework. In this paper, we show how a query calculus for spatio-temporal object databases that builds upon the monoid calculus proposed by Fegaras and Maier for ODMG-compliant database systems can be developed. The paper shows how an extension of the ODMG type system with spatial and temporal types can be accommodated into the monoid approach. It uses several queries over historical (possibly spatial) data to illustrate how, by mapping them into monoid comprehensions, the way is open for the application of a logical optimizer based on the normalization algorithm proposed by Fegaras and Maier.

1. Introduction

Spatio-temporal databases have been the focus of considerable research activity over a significant period. However, there still exist very few prototypes of complete systems (see [4] for a mid-1990s survey), and far less products that provide effective support for applications tracking changes to spatial and aspatial data over time. This is probably because the design and implementation of a complete spatio-temporal database is a challenging undertaking, involving extensions to all aspects of a non-spatio-temporal architecture – data model, query language, query optimizer, query evaluator, programming environment, storage manager, indexes, etc. The diversity of open issues relating to such an undertaking has led most researchers to focus on specific aspects of the problem (e.g., indexing, or join algorithms), rather than addressing the development of a complete spatio-temporal DBMS. This has given rise to a substantial collection of results that can be built upon by developers of complete systems, although such an endeavour has been pursued only rarely. The Tripod project, from which this paper emerges, is seeking to design and prototype a complete spatio-temporal database system that extends the ODMG standard for object databases [6] with facilities for managing vector spatial data, and for the description of past states of both spatial and aspatial data.

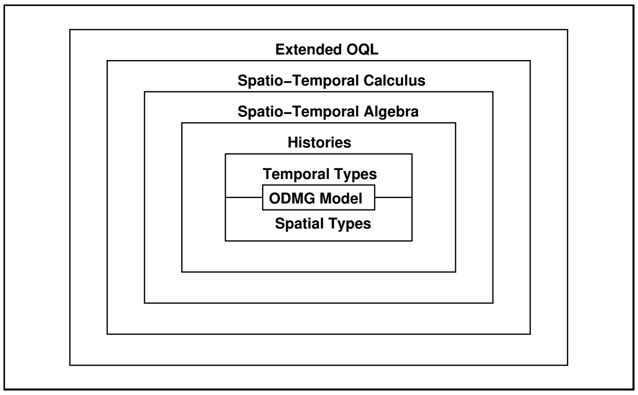

Figure 1. Tripod Components

Figure 1 illustrates the relationships between the different components in the Tripod design. At the core is the ODMG object model. The ODMG model is extended in Tripod with two new categories of primitive type: spatial types and temporal types. The spatial types used in Tripod are those of the ROSE algebra [15], which supports the vector types `Points`, `Lines` and `Regions`. The temporal types supported in Tripod are one-dimensional versions of the ROSE algebra types `Points` and `Lines`, and are known as `Instants` and `TimeIntervals`, respectively. The close relationship between the spatial and the temporal types increases consistency in the representation of the different kinds of data. Past states of all ODMG types, including the spatial and temporal types, can be recorded using histories. A *history* is a set of *timestamp-value* pairs, where the timestamp is of a temporal type and the value is of any of the types in the extended ODMG model. Figure 1, from `Histories` inwards represents a spatio-historical object model.

This, however, leaves open the question as to how the resulting model can be queried, and how these queries can both be given a precise semantics and be effectively optimized. In this regard, the monoid comprehension calculus of [8] is extended in Tripod to accommodate the querying of spatial and temporal data. In Figure 1, this is illustrated by the layers representing the *query algebra* and the *query calculus*, the former being derived from the latter using mappings described in [8]. The focus of this paper is on the query calculus layer in Figure 1. This layer is important, in that it provides a semantics for historical queries in object databases, and also because its incorporation within the monoid comprehension calculus of [8] allows the reuse of the associated query optimization framework, as supported by the OPTGEN query optimizer generator. The changes to the calculus for OQL provided in [8] to support spatial, temporal and historical features are pure extensions (i.e., the existing calculus is retained as a subset), and the extensions can be used together or separately in queries.

Figure 2 presents an example of a class definition that illustrates the sort of database that can be described using the model. The class `City` is itself historical, which means that the duration for which a `City` exists is recorded. Of the three properties of `City`, the `name` attribute is not historical (and thus only the last value assigned to the `name` is stored), the `elections` attribute is not historical but does store values of a temporal type, the attribute `wards` is collection-valued, the `mayor` attribute is historical, and the `landmass` attribute is both historical (so previous as well as current values for the landmass are stored) and of a spatial type. For reasons of space, the definition of the class `Ward` is omitted.

```
historical<TimeIntervals,year>
   class City
   (extent cities) {
      attribute string name;
      attribute <Instants,year> elections;
      attribute set<Ward> wards;
      historical<TimeIntervals,month>
         attribute string mayor;
      historical<TimeIntervals,year>
         attribute Regions landmass; }
```

Figure 2. Example Spatio-Historical ODL

The remainder of the paper is structured as follows. Section 2 provides an overview of the structure of Tripod timestamps, snapshots and histories (as well as their behaviour) by construing them as instances of abstract data types (ADT). Section 3 provides an overview of the monoid comprehension approach to query processing in object databases. Section 4 shows how the constructs in Section 2 can be incorporated into the framework described in Section 3 to give rise to a query calculus for historical object databases. Section 5 provides examples of use, including how queries requiring aggregation over historical attributes can be expressed. Section 6 discusses related work. Section 7 draws some of the conclusions stemming from the contributions of the paper.

2. A Historical Object Model: Construction, Structure and Behaviour

Tripod supports the storage, management and querying of entities that change over time through the notion of a *history*. A history models the changes that an entity (or its attributes, or the relationships it participates in) undergoes as the result of assignments made to it. In the Tripod object model, a request for a history to be maintained can be made for any construct to which a value can be assigned, i.e., a history is a history of changes in value and it records episodes of change by identifying these with a timestamp. Such timestamps are instances of set-based temporal types for which a rich collection of predicates and operations is available. This section provides an overview of the structure of Tripod timestamps, snapshots and histories (as well as their behaviour) by construing them as instances of ADTs. The overview focuses on what is needed for the development of the notion of histories as monoids introduced in Section 4 that opens the way for the definition of the query calculus contributed by the paper. For reasons of space, complete formal descriptions of the Tripod constructs referred to cannot be given here but have been made available elsewhere [10, 11].

2.1. Timestamps as Instances of Temporal Types

Tripod extends the set of ODMG primitive types with two temporal types, called `Instants` and `TimeIntervals`, over which a number of operations and predicates (i.e., Boolean-valued operations) are defined. The underlying domain of interpretation is a structure which we refer to as a *temporal realm* because it is defined to be a one-dimensional specialization of the two-dimensional (spatial) realms defined by Güting and Schneider [15]. Roughly, a temporal realm is a finite set of integers (whereas a spatial realm is a finite integer grid). Reasons why we adopt this viewpoint and terminology include:

- Realm values are set-based, which we find more suitable than individual ones for the kind of set-at-a-time strategies that are prevalent in query processing architectures.

- Realm operations are well-defined and have a rich set of predicates and constructors with nice closure properties.
- Tripod is a spatio-temporal database system and we find it useful (for users, developers and researchers) to have realms as a unifying notion for the interpretation of operations on spatial *and* temporal values.
- This unification at the level of interpretations propagates upwards in the sense that the predicates and operation on realms are defined once and used (possibly after renaming) over both spatial and temporal values. This also facilitates the reuse of implemented software components, such as those which the authors developed and described in [18].

In a temporal realm, we may think of a time-point as an integer. Then, an `Instants` value is a set of time-points and a `TimeIntervals` value is a set of pairs of time-points where the first element is the start, and the second the end, of a contiguous, closed time-interval. A *timestamp* is either an `Instants` value or a `TimeIntervals` value. Figure 3 illustrates timestamps in graphical form. In Figure 3, timestamp ***A*** is a `TimeIntervals` value, and timestamps ***B*** and ***C*** are `Instants` values. ***B*** happens to be a singleton.

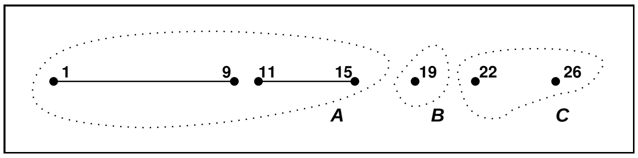

Figure 3. Example Tripod Timestamps

Let $\mathbb{T}$ denote the set of all timestamps. Given $\tau \in \mathbb{T}$, some representative predicates and operations defined on $\mathbb{T}$ are shown in Figure 4. The operation names should give readers an intuitive understanding of their meaning based on operations on sets of integers (and integer pairs) and on classical definitions for temporal predicates (such as Allen's [2]). For full details, see the formal semantics in [11] (which, for the reasons alluded to above, follows [15] closely).

$$\begin{array}{rl} \underline{=}: & \tau \times \tau \rightarrow \texttt{boolean} \\ \underline{\texttt{after}}: & \tau \times \tau \rightarrow \texttt{boolean} \\ \underline{\texttt{common\_points}}: & \tau \times \tau \rightarrow \texttt{boolean} \\ \underline{\texttt{intersection}}: & \tau \times \tau \rightarrow \tau \\ \underline{\texttt{minus}}: & \tau \times \tau \rightarrow \tau \end{array}$$

Figure 4. Example Operations on Timestamps

Although timestamps can be used by application designers to complement related primitive types in the ODMG standard (e.g., `Interval`, or `Time`), their main purpose is to allow histories to be constructed and operated upon, as described below. This requires the introduction of two other primitive types, called `Instant` and `TimeInterval`. Their values are not timestamps, instead they can be thought of as the building blocks for timestamps. Consistently with this view, the operations defined on `Instant` and `TimeInterval` (and their inverses) are shown in Figure 5.

$$\begin{array}{rl} \underline{\texttt{assemble}}^{I}: & 2^{\texttt{Instant}} \rightarrow \texttt{Instants} \\ \underline{\texttt{assemble}}^{T}: & 2^{\texttt{TimeInterval}} \rightarrow \texttt{TimeIntervals} \\ \underline{\texttt{disassemble}}^{I}: & \texttt{Instants} \rightarrow 2^{\texttt{Instant}} \\ \underline{\texttt{disassemble}}^{T}: & \texttt{TimeIntervals} \rightarrow 2^{\texttt{TimeInterval}} \end{array}$$

Figure 5. Conversions over Temporal Values

2.2. Snapshots as Values from the Object Model

As mentioned above, a request for a history to be maintained can be made for any construct to which a value can be assigned. Each such value is a *snapshot* of the construct at the times denoted by the associated timestamp. As a consequence of the possible value assignments that are defined in the ODMG object model, a history can be kept in the Tripod object model for all of object identifiers, attribute values, and relationship instances. Thus, a history associates timestamps and snapshots drawn from the domain of (one among) object identifiers (if the history is of an object), attribute values (whose type can be any valid Tripod, and hence ODMG, type) or relationship instances (of any ODMG-supported cardinality).

2.3. The Structure of Histories

A *history* is a quadruple $H = \langle V, \theta, \gamma, \Sigma \rangle$, where V denotes the domain of values whose changes H records, θ is either `Instants` or `TimeIntervals`, γ is the granularity of θ, and Σ is a set of pairs, called *states*, of the form $\langle \tau, \sigma \rangle$, where τ is a Tripod timestamp and σ is a snapshot. In the rest of the paper, let $\mathbb{T}$ denote, as before, the set of all timestamps; $\mathbb{V}$, the set of all snapshots; $\mathbb{S}$, the set of all states; and $\mathbb{H}$, the set of all histories.

In a history, a set Σ of states is constrained to be an injective function from the set $\mathbb{T}_H$ of all timestamps occurring in H to the set $\mathbb{V}_H$ of all snapshots occurring in H, i.e., for any history H, $states_H : \tau \in \mathbb{T}_H \rightarrow \sigma \in \mathbb{V}_H$. Therefore, the following invariants hold, for any history $H = \langle V, \theta, \gamma, \Sigma \rangle$:

1. Every timestamp occurring in Σ is of type $\theta \in \{\texttt{Instants}, \texttt{TimeIntervals}\}$ and has granularity γ.

2. For every snapshot σ occurring in Σ, $\sigma \in V$.

3. A particular timestamp is associated with at most one snapshot, i.e., a history does not record different values as valid at the same time.

4. A particular snapshot is associated with at most one timestamp, i.e., if a value is assigned more than once, in the corresponding history the new occurrence causes the timestamp of the previous occurrence to adjust appropriately (in other words, coalescing takes place by default, and, if so, explicit mention of earlier values may not be maintained).

For example, $H = \langle$ `int`, `TimeIntervals`, `year`, $\{ \langle [1981 - 1986], 22 \rangle, \langle [1989 - 1991], 24 \rangle \} \rangle$ is a history of integer snapshots (say, a property like `salary`) timestamped with time-intervals whose elements denote years. Thus, from year `1981` to year `1986`, the salary was `22`, and from year `1989` to year `1991`, the salary was `24`.

2.4. The Behaviour of Histories

This subsection provides an overview of the operations available in Tripod to operate on histories construed as instances of an ADT, which leads to their behaviour being categorized into constructor (not discussed in this paper), query, merge and update operations. The overview is quite focussed. Full details are available elsewhere [10]. Note that, if needed, we distinguish operations defined on histories from operations defined on timestamps by <u>`underlining`</u> the latter.

2.4.1 Query Operations on Histories

Representative query operations on histories are shown in Figure 6. Note that the first expression in Figure 6 is in fact a template for a set of signatures parameterized on any element of the set of predicates on timestamps. For example, given that <u>`before`</u> is a member of that set, letting ω = <u>`before`</u> in the template yields the following signature `ContainsTimestamp_`<u>`before`</u> : $\mathbb{H} \times \mathbb{T} \rightarrow$ `boolean`. Other such parameterized templates include `FilterByTimestamp_`ω.

`ContainsTimestamp_`ω :	$\mathbb{H} \times \mathbb{T}$	$\rightarrow$	`boolean`
`FilterBySnapshot` :	$\mathbb{H} \times \mathbb{V}$	$\rightarrow$	$\mathbb{H}$

Figure 6. Querying Histories

Given `boolean` = $\{$`true`, `false`$\}$, `ContainsTimestamp_`ω (H, τ) = `true` if $\exists \tau' \in dom(states_H) \wedge \omega(\tau, \tau')$, otherwise `ContainsTimestamp_`$\omega$ (H, τ) = `false`. For example, if the state sets of two histories H_1 and H_2 both with V = `int`, θ = `TimeIntervals` and identical γ, are $\Sigma_1 = \{\langle [1-6], 12 \rangle, \langle [9-11], 14 \rangle\}$ and $\Sigma_2 = \{\langle [5-10], 13 \rangle, \langle [13-20], 15 \rangle\}$ then `ContainsTimestamp_`<u>`before`</u>$(H_1, [9-10])$ = `true` and `ContainsTimestamp_`<u>`after`</u>$(H_2, [21-22])$ = `false`. In contrast to `ContainsTimestamp_`ω which queries a history for a true/false reply, `FilterBySnapshot` exemplifies operations that query histories for a reply that is itself a history. Given $H = \langle V, \gamma, \theta, \Sigma \rangle$ and $H' = \langle V, \gamma, \theta, \Sigma' \rangle$, `FilterBySnapshot`$(H, \sigma)$ = H' where $\Sigma' = \{\langle t, \sigma \rangle | states_H(t) = \sigma\}$. For example, if H_1 and H_2 are as above, then `FilterBySnapshot`$(H_1, 12) = H'_1 = \langle V, \gamma, \theta, \{\langle [1-6], 12 \rangle\} \rangle$ and `FilterBySnapshot`$(H_2, 12) = H'_2 = \langle V, \gamma, \theta, \{\} \rangle$.

2.4.2 Merge and Update Operations on Histories

Representative merge and update operations on histories are shown in Figure 7.

$\uplus$:	$\mathbb{H} \times \mathbb{H}$	$\rightarrow$	$\mathbb{H}$
`DeleteTimestamp` :	$\mathbb{H} \times \mathbb{T}$	$\rightarrow$	$\mathbb{H}$
`InsertState` :	$\mathbb{H} \times \mathbb{S}$	$\rightarrow$	$\mathbb{H}$

Figure 7. Merging and Updating Histories

Given $H_1 = \langle V, \theta, \gamma, \Sigma_1 \rangle$ and $H_2 = \langle V, \theta, \gamma, \Sigma_2 \rangle$, $\uplus(H_1, H_2) = H_3 = \langle V, \theta, \gamma, \Sigma_3 \rangle$, where $\Sigma_3 = (state_{H_1} | (dom(state_{H_1}) \setminus dom(state_{H_2})) \cup state_{H_2}$[1]. In other words, taking the union of two histories is equivalent to taking the union of their state sets but choosing the state in the second argument whenever there is a state in the first argument with the same timestamp but different snapshot. This is to satisfy the invariants that characterize histories. For example, using infix notation, if the state sets of two histories H_1 and H_2 are as exemplified above, then the state set of $H = H_1 \uplus H_2$ is $\Sigma = \{\langle [1-5], 12 \rangle, \langle [5-10], 13 \rangle, \langle [10-11], 14 \rangle, \langle [13-20], 15 \rangle\}$. `DeleteTimestamp` takes a history $H = \langle V, \theta, \gamma, \Sigma \rangle$ and a timestamp τ of type θ and yields a new history $H' = \langle V, \theta, \gamma, \Sigma' \rangle$. The operation maps Σ into a state set Σ' in which all states in Σ whose timestamp τ' is such that <u>`common_points`</u>(τ, τ') is true, have been recomputed so that τ does not occur in Σ'. For example, if $\tau = [3-4]$ and $\Sigma = \{\langle [1-6], 12 \rangle\}$, then $\Sigma' = \{\langle [1-3, 4-6], 12 \rangle\}$. Note that the semantics of `DeleteTimestamp` relies on the definitions of <u>`common_points`</u> (and, as can be seen in [10],

[1] Recall that, given a function $f(x)$, its *restriction to the set* E, denoted by $f|E$, is the set of pairs $\langle x, y \rangle$ such that $y = f(x)$ and $x \in E$.

minus). `InsertState` takes a history $H = \langle V, \theta, \gamma, \Sigma \rangle$ and a state $\langle \tau', \sigma' \rangle$, where τ' is of type θ and $\sigma' \in V$, and yields a new history $H' = \langle V, \theta, \gamma, \Sigma' \rangle$. If σ' is equal to some σ occurring in Σ then the timestamp τ associated with it is recomputed into a timestamp τ_+ that includes τ', and $\Sigma' = \Sigma \setminus \{\langle \tau, \sigma \rangle\} \cup \{\langle \tau_+, \sigma \rangle\}$. If, on the other hand, σ' does not occur in Σ, then Σ is recomputed into a state set Σ_+ that is everywhere equal to Σ except that every state in Σ whose timestamp has common points with τ' has been recomputed so as to make that no longer the case in Σ_+, and $\Sigma' = \Sigma_+ \cup \{\langle \tau', \sigma' \rangle\}$. For example, if $\langle \tau', \sigma' \rangle = \langle [5-8], 13 \rangle$ and $\Sigma = \{\langle [1-6], 12 \rangle\}$, then $\Sigma' = \{\langle [1-4], 12 \rangle, \langle [5-8], 13 \rangle\}$ and if $\langle \tau', \sigma' \rangle = \langle [5-8], 12 \rangle$ and $\Sigma = \{\langle [1-6], 12 \rangle\}$, then $\Sigma' = \{\langle [1-8], 12 \rangle\}$.

3. The Monoid-Based Approach to Query Processing

3.1. Monoids

In abstract algebra, a *monoid* is a triple $(T, \oplus, Z_\oplus)$ consisting of a set T together with a binary associative operation $\oplus : T \times T \rightarrow T$, called the *merge* function for the monoid, and an identity element $Z_\oplus$, called the *zero element* of the monoid. Note that, by definition, for all $x, y, z \in T$, $(x \oplus y) \oplus z = x \oplus (y \oplus z)$, and, for all $x \in T$, $Z_\oplus \oplus x = x \oplus Z_\oplus = x$. In database contexts, we take T to be a database type. Examples of monoids are $(\texttt{int}, +, 0)$, $(\texttt{int}, *, 1)$, $(\texttt{int}, max, 0)$, $(\texttt{boolean}, \wedge, \texttt{true})$ and $(\texttt{boolean}, \vee, \texttt{false})$. By consistent and careful naming, it is possible (and customary) to let the merge function identify the type of the monoid, thus, e.g., $(+, 0)$ abbreviates $(\texttt{int}, +, 0)$. When, as in the examples above, T is a scalar type, monoids on T are called *primitive monoids*. If T is a collection (also commonly referred to as bulk) type, monoids on T are called *collection monoids* and require an additional function, called the *unit* function for the monoid and denoted $U_\oplus$, so that it is possible to construct all possible instances of T. An example of a collection monoid is $(\texttt{set}, \cup, \{\}, \lambda x.\{x\})$, where $U_\oplus = \lambda x.\{x\}$ is the function that given any x returns the singleton $\{x\}$. For example, to construct the set of integers $\{2, 7, 5\}$ one could write $\oplus(U_\oplus(2), \oplus(U_\oplus(7), U_\oplus(5)))$ which gives, for $\oplus = \cup$ and $U_\oplus = \lambda x.\{x\}$, the set $\{2\} \cup (\{7\} \cup \{5\}) = \{2, 7, 5\}$. It is customary to omit the unit function when denoting a collection monoid because its specification is easily inferable from the merge function that identifies the monoid. Collection monoids for sets, bags and $list$s are, therefore, written as $(\cup, \{\,\})$, $(\uplus, \{\!\{\,\}\!\})$, and $(++, [\,])$, respectively, where $\uplus$ denotes union without duplicate removal and $++$ denotes list append, and $\{\,\}$, $\{\!\{\,\}\!\}$ and $[\,]$ denote the empty set, the empty bag and the empty list, respectively.

3.2. Monoid Comprehensions

The monoid comprehension calculus [7, 8] is now briefly described. A *comprehension over the monoid* $\oplus$ is an expression of the form $\oplus\{e | \overrightarrow{r}\}$, where the expression e is called the *head* of the comprehension and $\overrightarrow{r} = r_1, \ldots, r_n$, $n \geq 0$, is a sequence of qualifiers each one of which is either a *generator* of the form $v \leftarrow e'$ or a *filter* in the form of a predicate p over the terms in the comprehension. In a generator, v is called the *range variable* and the expression e' is the *domain* from which bindings for v are drawn.

A well-formedness requirement (statically verifiable) stipulates that if any of the generators in a monoid comprehension is over an idempotent or commutative monoid, then the comprehension must be itself over an idempotent or commutative monoid. This means, e.g., that a comprehension that maps a set monoid (which is idempotent and commutative) into a list monoid (which is neither) is ill-formed, whereas one that maps a bag monoid into $(+, 0)$ is well-formed, since both are commutative. Monoid comprehensions also have a simple procedural interpretation. For example, a monoid comprehension of the form $\oplus\{h | x \leftarrow E, p\}$ denotes a result R computed as shown in Figure 8. Generalizing to n generators gives rise to n nested loops. It is also straightforward to generalize for multiple filters.

$$
\begin{array}{l}
\texttt{R} := Z_\oplus; \\
\textbf{foreach}\ x\ \textbf{in}\ E: \\
\quad \textbf{if}\ p(x) = \texttt{true} \\
\quad\quad \textbf{then}\ \texttt{R} := \oplus(\texttt{R}, U_\oplus(x)); \\
\textbf{return}\ \texttt{R};
\end{array}
$$

Figure 8. Procedural Interpretation

A monoid comprehension characterizes a set of instances of the type over which it is defined. By applying the reductions given in [7] that set can be obtained. In this sense, these reductions characterize a calculus. Since, as shown below, monoid comprehensions serve as the target of a mapping from OQL expressions, the monoid comprehension calculus qualifies as a query calculus for object databases and the reductions characterize a formal declarative semantics for OQL (thereby allowing the result of an OQL query to be inferred).

We give some examples of OQL queries alongside their corresponding monoid comprehensions. Figure 9 shows, in the general case, the monoid comprehension that a select-from-where clause maps into. The OQL query in Figure 10 returns the set of names of cities that have more than ten wards, and maps into the monoid comprehension given below in the same figure. A comprehensive description of a mapping of OQL 1.2 into monoid comprehensions is given in [13]. An algebraic optimizer over monoid comprehen-

```
OQL: select   e
     from     x_1 in e_1, ..., x_n in e_n
     where    p
```

Monoid comprehension:

$$\uplus\{e|x_1 \leftarrow e_1,\ldots,x_n \leftarrow e_n, p\}$$

Figure 9. Mapping Select-From-Where

```
select   distinct c.name
  from   c in Cities
 where   count(c.wards) > 10
```

$$\cup\{\mathsf{c.name}|\mathsf{c} \leftarrow \mathsf{Cities}, +\{1|\mathsf{w} \leftarrow \mathsf{c.wards}\} > 10\}$$

Figure 10. OQL to Monoid Comprehension

sions is described in [8]. Our practical experience with the monoid comprehension approach in implemented systems has been reported for parallel databases in [21] and for incremental view maintenance in [1].

4. Extending the Monoid-Based Query Calculus

This section shows how the constructs in Section 2 can be incorporated into the framework described in Section 3 to give rise to a query calculus for historical object databases.

4.1. Timestamps and Snapshots as Monoids

Realm-based types in Tripod are implemented as primitive types with the same status as other ODMG primitive types such as `boolean`, `int` and `string`. Since `Instants` and `TimeIntervals` are not scalar types, but set-based ones, certain aspects of their behaviour, as formalized in [11], can be captured using collection monoids. The $\underline{\texttt{assemble}}^I$ function, introduced in Section 2.1, that maps a set of values of type `Instant` into a value of type `Instants` is membership and cardinality preserving, and so is $\underline{\texttt{disassemble}}^I$, its inverse. This means that $\underline{\texttt{assemble}}^I(\underline{\texttt{disassemble}}^I(X)) = \underline{\texttt{disassemble}}^I(\underline{\texttt{assemble}}^I(X))$, for all X. In contrast, the $\underline{\texttt{assemble}}^T$ function, which maps a set of values of type `TimeInterval` into a value of type `TimeIntervals`, coalesces, to the smallest convex (i.e., gap-free) interval containing them, any two intervals which either meet or overlap. Therefore, for some X, $\underline{\texttt{assemble}}^T(\underline{\texttt{disassemble}}^T(X)) \neq \underline{\texttt{disassemble}}^T(\underline{\texttt{assemble}}^T(X))$. We can now define collection monoids over `Instants` and `TimeIntervals` where the zero elements are the respective empty instances (denoted $\{\}^I$ and $\{\}^T$, respectively), the unit functions are $\lambda x.\texttt{assemble}^I(\{x\})$ and $\lambda x.\texttt{assemble}^T(\{x\})$, respectively, and, finally, the merge functions are the assembly of the union of the disassembly of the arguments, i.e., $\underline{\texttt{union}}^I(X, X') = \underline{\texttt{assemble}}^I(\underline{\texttt{disassemble}}^I(X) \cup \underline{\texttt{disassemble}}^I(X')) = X \cup X'$ and $\underline{\texttt{union}}^T(X, X') = \underline{\texttt{assemble}}^T(\underline{\texttt{disassemble}}^T(X) \cup \underline{\texttt{disassemble}}^T(X'))$, respectively. Both monoids inherit from the $(\cup, \{\ \})$ monoid the properties of being idempotent and commutative. We can now write comprehensions over the $(\underline{\texttt{union}}^I, \{\}^I)$ monoid (over `Instants`) and the $(\underline{\texttt{union}}^T, \{\}^T)$ monoid (over `TimeIntervals`). For these comprehensions, the lexicon for filters is extended with the set of Boolean-valued Tripod predicates (e.g., $\underline{\texttt{overlap}}$, $\underline{\texttt{meet}}$) over these primitive Tripod types and the set of domain generators is extended with expressions that denote the latter (e.g., attributes, relationships, path expressions, etc., whose domain is a Tripod temporal type, as well as Tripod operations that return instances of such types).

For example, assume the `TimeIntervals` value $TI = [1-6, 8-10, 12-14, 15-20]$, then the monoid comprehension in Figure 11 counts the intervals that contain the given `Instants` literal.

$$+\{1|\mathsf{v} \leftarrow \underline{\mathsf{union}}^T\{\mathsf{x}|\mathsf{x} \leftarrow \mathsf{TI},\ \underline{\mathsf{contains}}(\mathsf{x}, [9])\}\}$$

Figure 11. Comprehension Over Temporal Values

Note that coalescing is implicit in the fact that $\underline{\texttt{union}}^T$ is defined in terms of $\underline{\texttt{assemble}}^T$, which, as discussed above, coalesces by definition.

In summary, we have indicated how to extend the monoid comprehension calculus with two collection monoids, one for each of the possible forms for a timestamp. More collection monoids can, of course, be defined over temporal Tripod types which we do not cover due to lack of space. Note also that although the realm-based spatial types which Tripod also supports as primitive types are not discussed in this paper, the definition of monoids (and monoid comprehensions) over them follows a similar strategy as described above for realm-based temporal types.

As previously mentioned, Tripod extends the set of primitive ODMG types with realm-based temporal (and spatial) types. In practice, this means that application designers have new expressive domains over which to define attributes and relationships and thereby model temporal (and spatial) aspects of applications. Section 2.2 characterized snapshots simply as values of valid types in the extended type system. This implies that they have associated monoids (or not) exactly as described in [8], for a (non-extended) ODMG setting, and above, for the realm-based temporal (and spatial) types in the extended type system. The monoid comprehensions that can be written are not changed. It is histories that add more possibilities, as follows.

4.2. Histories as Monoids

For the purposes of query processing, it is reasonable to assume that a history has been checked for well-formedness (regarding granularity and type-correctness of both timestamps and snapshots), in which case we simplify our notation and equate a history $H = \langle V, \theta, \gamma, \Sigma \rangle$ with its state set Σ. Note that, in Tripod, there is no type whose instances are histories that designers can use to model an application (as there are, in contrast, for each of the realm-based temporal (and spatial) types). We signal this distinction by not using the `typewriterfont` we use for the latter, and instead retain from Section 2.3 the denotation for the set of all histories (respectively, timestamps, snapshots, states), i.e., $\mathbb{H}$ (respectively, $\mathbb{T}$, $\mathbb{V}$, $\mathbb{S}$). We can define collection monoids over $\mathbb{H}$. For example, one where the zero element is the empty history, denoted by $\{\langle\ \rangle\}$, the unit function is $\lambda x.\{\langle x \rangle\}$ for any $x \in \mathbb{S}$, and the merge function is $\Cup$, as defined in Section 2.4.2. Note that since the definition of $\Cup$ picks the first argument to restrict on, it follows that the corresponding monoid, $(\Cup, \{\langle\ \rangle\})^2$, is not commutative but it is idempotent. As done above for `Instants` and `TimeIntervals` monoids, we can write monoid comprehensions over $\mathbb{H}$ in which the lexicon for filters stems from the set of Boolean-valued operations over histories outlined in Section 2.4.1, and the set of domain generators is extended by history-denoting expressions (including monoid comprehensions other than $(\Cup, \{\langle\ \rangle\})$).

$$\mathsf{max}\{\mathsf{d} \mid \mathsf{d} \leftarrow \cup\{\underline{\mathsf{duration}}(\mathsf{s.timestamp}) \mid \mathsf{s} \leftarrow \Cup\{(\mathsf{t},\mathsf{v}) \mid (\mathsf{t},\mathsf{v}) \leftarrow \mathsf{H}, \mathsf{v} < 20\}\}\}$$

Figure 12. Comprehension Over History

For example, given the <u>`duration`</u> operation on `TimeIntervals` (defined, for a `TimeIntervals` t, as <u>`end`</u>(<u>`last`</u>(t)) $-$ <u>`start`</u>(<u>`first`</u>(t))), and a history $H = \{\langle [1-6], 12\rangle, \langle [9-11], 14\rangle, \langle [12-25], 30\rangle\}$, then the monoid comprehension in Figure 12 computes the longest duration among the snapshots in H that are smaller than 20, i.e., the longest period for which there are records for a value less than 20, and returns 5 granules (which is the lifespan of the snapshot 12).

We have indicated how Tripod's spatial and temporal extensions to the ODMG as well as histories give rise to an extended set of monoid comprehensions. The remainder of this section discusses in what sense this gives rise to an extended query calculus. Later, Section 5 shows how aggregations over historical and spatio-historical data can be expressed using the extended set of monoid comprehensions.

4.3. A Query Calculus Founded on History Monoid Comprehensions

Work on the monoid approach to query calculi for object databases has yielded a wealth of results on typing, equivalence-based rewriting and mapping into physical algebras that, together, comprise as well defined a query processing framework as relational databases have enjoyed for decades now. By showing how timestamps and histories can give rise to monoids and then monoid comprehensions we have effectively indicated how one can extend an ODMG setting with realm-based temporal (and, by analogy, spatial) types and the notion of a history.

In [8], the core of a logical query optimizer is characterized by a set of transformations (i.e., rewriting rules) and a confluent, terminating normalization algorithm to apply the latter. In the usual fashion, the rewriting rules can be used to specify heuristics (e.g., push selections inside as far as they will go) that are likely to result in equivalent expressions which are less costly to evaluate than the original ones. Each transformation in [8] is applicable or not depending on the properties of the monoid in its antecedent.

Given the monoids that one can define over the additional primitive types and over histories, the transformations defined in [8] provide us with the basic components of a query calculus for historical object databases. Thus, since the timestamp monoids suggested above are idempotent and commutative, all six transformations in [8] that apply to collection monoids apply to the timestamp monoids as well, whereas, in contrast, for the history monoid suggested above, the two such transformations that require the argument monoid to be commutative are ruled out. The overall result of the approach we have described is to endow Tripod, a historical object database extending ODMG-compliant ones, with a well-defined query calculus.

[2] Both $\{\langle$ and $\rangle\}$ are tokens (indivisible, therefore) used as left and right bracket symbols, respectively.

5. Examples of Queries over Temporal and Spatial Data

This section shows how the query calculus for historical object databases contributed by this paper can express queries that are representative for the class of applications that Tripod has been designed to support.

(Q1) *How many elections have there ever been across all cities?*

$$+\{1 \mid \quad e \leftarrow \underline{\mathsf{disassemble}}^{I}(\mathsf{c.elections}), \mathsf{c} \leftarrow \mathsf{Cities}\}$$

$$+\{e \mid \quad e \equiv \underline{\mathsf{number\_of\_components}}(\mathsf{c.elections}), \mathsf{c} \leftarrow \mathsf{Cities}\}$$

(Q2) *How many elections has each city had?*

$$\cup\{ \quad \langle \mathsf{N} : \mathsf{c.name},\ \mathsf{E} : +\{\, e \mid e \equiv \underline{\mathsf{number\_of\_components}}(\mathsf{c.elections})\}\rangle \mid \mathsf{c} \leftarrow \mathsf{Cities}\}$$

Figure 13. Temporal Queries

Given the schema fragment in Figure 2, consider first the queries over a temporal (as opposed to historical) attribute shown in Figure 13. In the case of aggregation queries over temporal values, sometimes a compiler has two options when it comes to mapping them into a monoid comprehension. The choice stems from the fact that the Tripod realm-based temporal values are set-based. Given this fact, the obvious course of action is to disassemble the temporal value into the set consisting of the individual components of the temporal value and then apply the standard aggregation monoids over them. However, for certain aggregations, a more efficient option is available that relies on the fact that certain operations defined over temporal values effectively return an aggregate. For example, `number_of_components` effectively computes a count over a value of type `Instants` or `TimeIntervals`. **(Q1)** above can, therefore, be mapped into either of the monoid comprehensions shown in Figure 13. Since temporal types are primitive types in Tripod, the compiler knows both that `number_of_components` is available and what semantics it has, in which case it is best to compile **(Q1)** into the second of the above monoid comprehensions. Given that **(Q2)** presents a similar choice, we assume it to be mapped into the monoid comprehension shown in Figure 13. In the comprehension corresponding to **(Q1)** the equivalence symbol denotes the introduction of a local name e for the bindings produced. Note that the first version (with `1` in the head) is a count, while the second is (with the name e in the head) is a sum.

Queries **(Q1)** and **(Q2)** do not range over any history, insofar as they simply aggregate over elements of a snapshot value whose type is a set of time points. In contrast, in order for queries **(Q3)** to **(Q7)** shown in Figure 14 to be evaluated, a history of changes has to be scrutinized. The history is that of the mayors of a city, denoted, for a city c, by c.mayor. Since a history monoid is available (as defined in Section 4.2), one can use histories in the right hand side of generators in the body of a monoid comprehension to bind each of its states to a pair (t,v) of variables, with t denoting the timestamp, and v, the snapshot at t. Notice also that operations such as FilterBySnapshot can be used over a history in an analogous way to that in which `number_of_components` was used over a temporal value in queries **(Q1)** and **(Q2)** above, i.e., denoting a scan carried out by the kernel rather than by the query evaluator.

(Q3) *How many mayors has Manchester ever had?*

$$+\{\, 1 \mid m \leftarrow \quad \cup\{v \mid (t, v) \leftarrow \mathsf{c.mayor},\ \mathsf{c} \leftarrow \mathsf{Cities}, \mathsf{c.name} = \text{“Manchester”}\,\}\}$$

(Q4) *How long has Ken Livingstone been the mayor of London for?*

$$+\{\underline{\mathsf{duration}}(t) \mid \quad (t, v) \leftarrow \mathsf{FilterBySnapshot}(\mathsf{c.mayor}, \text{“Ken Livingstone”}),\ \mathsf{c} \leftarrow \mathsf{Cities}, \mathsf{c.name} = \text{“London”}\,\}$$

(Q5) *Retrieve the history of Paris mayors between 1960 and 1970.*

$$\cup\{ \quad \langle \mathsf{c}, \uplus\,\{(t, v) \mid (t, v) \leftarrow \mathsf{FilterByTimestamp}_{\mathsf{overlaps}}(\mathsf{c.mayor}, [1960 - 1970])\}\rangle \mid \mathsf{c} \leftarrow \mathsf{Cities}, \mathsf{c.name} = \text{“Paris”}\,\}$$

(Q6) *Which cities had a landmass with an area larger than 200 square miles on 1st June 1970?*

$$\cup\{\mathsf{c.name} \mid \quad \mathsf{c} \leftarrow \mathsf{Cities},\ (t, v) \leftarrow \mathsf{c.landmass},\ \underline{\mathsf{area}}(v) > 200,\ \underline{\mathsf{contains}}(t, [01/\mathsf{JUN}/1970])\,\}$$

(Q7) *What was the maximum landmass of a city on 1st June 1970?*

$$\max\{x \mid \quad x \leftarrow \cup\{\underline{\mathsf{area}}(v) \mid \mathsf{c} \leftarrow \mathsf{Cities},\ (t, v) \leftarrow \mathsf{c.landmass},\ \underline{\mathsf{contains}}(t, [01/\mathsf{JUN}/1970])\,\}\}$$

Figure 14. Historical and Spatio-Temporal Queries

Aggregations over spatial values is analogous to that for temporal values discussed in the context of queries **(Q1)** and **(Q2)** above, i.e., there may be an opportunity to apply a primitive operation that computes an aggregate over the individual components of a spatial (realm-based, and hence set-based) value, otherwise the compiler uses the spatial equivalents of `disassemble` and `assemble` for each of the spatial types, i.e., `Points`, `Lines`, and `Regions`. The monoid comprehensions for queries **(Q6)** and **(Q7)** suggests the benefits of the method described in this paper to integrate spatial and historical features into an object database query calculus. It is immediately apparent that combining spatial and temporal aggregations requires no extra machinery. Consider queries **(Q6)** and **(Q7)** in Figure 14 and the monoid comprehensions they are mapped into.

In queries **(Q6)** and **(Q7)**, notation is abused when a date (such as 01/JUN/1970) is used instead of the integer values from which the domain of `Instants` and `TimeIntervals` values are constructed. For reasons of space, this paper discusses the details of neither calendrical mechanisms nor the handling of granularities. Current time and pseudo-open intervals are supported as described in [10].

6. Related Work

The contributions reported in this paper aim to lay the foundations for spatio-temporal query optimization and processing in ODMG-compliant databases. In particular, the paper shows that the monoid comprehension approach to the problem can be adopted in the context of a very rich collection of temporal and spatial types that extend the standard ODMG type system. In this section, the contributions of the paper are set against work on temporal and spatio-temporal data models and languages in both the relational and the object-oriented settings. Much recent work, in a wide range of topics, is reported in [23].

Relational Temporal Models Current research into temporal object models and query languages stems from similar work in the relational setting during the 1980s. Such models typically either timestamped individual tuples or attributes with an interval-based valid-timestamp. The timestamping approach adopted in this paper can be seen to be an adaptation to an object-oriented setting of methods employed by, e.g., Gadia [9], who modelled attribute values as functions from temporal elements onto the attribute domain, where a temporal element is a union of disjoint time intervals. The operations defined over Tripod temporal types exploit constructs first proposed by Güting and Schneider in the early 1990s in a spatial setting. Our temporal predicate operations have their foundations in the interval algebra defined by Ladkin [17], which in turn extends that of Allen [2] to unions of convex intervals. There are many proposals of temporal relational models but few have been implemented, and this has had the consequence that comprehensive discussions of temporal query optimization and processing are scarce (see, e.g., the excellent survey in [16] – and an exceptionally detailed proposal in [5] – which focus on query algebras, leaving query calculi in the background by and large).

Object-Oriented Temporal Models There are also relatively few proposals for temporal object models (see the survey in [22]), and even fewer for spatio-temporal ones (see the survey in [19]). Of these, the historical object model proposed by Bertino et al. [3] most closely relates to our work. They propose a temporal extension to the ODMG object model, called *T*_ODMG, that uses property timestamping. The value for each *T*_ODMG object property is a function of time, utilizing interval-based timestamps that closely resemble the primitive Tripod `TimeInterval` type. Their model addresses concerns that we have not addressed, e.g., object properties can be made *immutable* (i.e., their value cannot be modified during the object's lifetime), and particular attention is paid to modelling objects that migrate to another type during their lifetimes. While the structural component of *T*_ODMG is well documented, the behavioural aspects of temporal domains is less so. The details of such behaviour are a necessary precursor to the definition of a query language and corresponding optimizer, and without that it is difficult to compare the contributions of this paper with those of [3] in greater detail.

Models for Moving Objects Both our data model and that of [3] utilize a *discrete* model of time. Other proposals exist for data models that capture objects whose properties (spatial and aspatial) are continuously changing. These models are typified by the *moving object* approach adopted in [14]. Such models allow the state of each spatial and aspatial property to be expressed as a continuous function of time. Queries about the position of spatial data can then be inferred by the interpolation of spatial values between known bounds. This provides an expressive mechanism for the representation of moving points, lines and polygons. Querying moving object databases is achieved by extending an existing database algebra through a process called *lifting*. This allows non-temporal kernel algebra operations to be applied to temporal types. The Tripod data model and calculus does not address such domains, as we explicitly target applications in which objects change in discrete steps.

Spatio-Temporal Query Optimization Most research into this topic is focusing on the development of efficient strategies for processing spatial and spatio-temporal data, e.g., on join [24] and nearest-neighbour [20] queries. Another line of investigation is the work on DEDALE [12], which uses constraint database ideas to underpin query optimization. This paper, in contrast, concentrates on describing a calculus that can underpin a logical optimizer. The focus, therefore, is on providing the basis for rewriting logical expressions from a calculus whose properties are well-studied rather than on algebraic-level strategies for optimization that are concerned with the cost models underlying the physical execution of queries. The mapping from the spatio-temporal monad comprehensions characterized here into algebraic expressions made of executable operators is outside the scope of the paper and the subject of future work we intend to carry out, as indicated by Figure 1.

7. Conclusions

The design and implementation of a comprehensive spatio-temporal database system presents many challenges, not least those relating to the design, formalization and optimization of declarative queries. This paper has:

1. Introduced a collection of primitive temporal types based on the spatial ROSE algebra [15], thereby providing consistent and complementary facilities for describing time and space.

2. Described how the new temporal types can be used to underpin the notion of a history, whereby changes to values can be recorded over time.

3. Indicated how (1) and (2) above can be incorporated into the monoid comprehension calculus framework of [8], thereby providing both a semantics for spatio-temporal queries over object databases and a practical framework for the development of optimizers for such queries.

4. Illustrated the resulting query calculus in use with a collection of example queries.

Overall, the proposal provides concise but powerful facilities for modelling and querying spatio-historical object databases. Ongoing work is developing an extension to OQL to provide user-level expression of queries supportable by the calculus, and is investigating efficient algorithms for query evaluation.

Acknowledgements Discussions with Michael Worboys, John Stell, Chris Johnson, Keith Mason and Bo Huang, our partners in the Tripod project, have contributed significantly to shape the contributions of this paper. We are pleased to acknowledge our debt to them. This work has been funded by the UK Engineering and Physical Sciences Research Council and their support is gratefully acknowledged.

References

[1] M. A. Ali, A. A. A. Fernandes, and N. W. Paton. Incremental Maintenance of Materialized OQL Views. In *Proc. DOLAP 2000*, pp. 41–48.

[2] J. F. Allen. Maintaining Knowledge about Temporal Intervals. *CACM*, 26(11):832–843, November 1983.

[3] E. Bertino, E. Ferrari, G. Guerrini, and I. Merlo. Extending the ODMG Object Model with Time. In *Proc. ECOOP'98*, pp. 41–66, 1998.

[4] M. H. Böhlen. Temporal Database System Implementations. *SIGMOD Record*, 24(4), 1995.

[5] M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Temporal Statement Modifiers. *ACM TODS*, 25(4), December 2000.

[6] R. G. G. Cattell, editor. *The Object Database Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[7] L. Fegaras. Optimizing Queries with Object Updates. *JIIS*, 12(2-3):219–242, March-June 1999.

[8] L. Fegaras and D. Maier. Optimizing Object Queries Using an Effective Calculus. *ACM TODS*, 25(4), December 2000.

[9] S. K. Gadia. A Homogeneous Relational Model and Query Languages for Temporal Databases. *ACM TODS*, 13(4):418–448, 1988.

[10] T. Griffiths, N. W. Paton, and A. A. A. Fernandes. An ODMG-Compliant Spatio-Temporal Data Model, 2000. `http://pevepc13.cs.man.ac.uk/PrePrints/index.htm`.

[11] T. Griffiths, N. W. Paton, and A. A. A. Fernandes. Realm-Based Temporal Data Types, 2000. `http://pevepc13.cs.man.ac.uk/PrePrints/index.htm`.

[12] S. Grumbach, P. Rigaux, and L. Segoufin. The DEDALE System for Complex Spatial Queries. In *Proc. SIGMOD*, pp. 213–224, 1998.

[13] T. Grust and M. H. Scholl. Translating OQL into Monoid Comprehensions—Stuck with Nested Loops? Technical Report 3a/1996 (Revised), Department of Mathematics and Computer Science, Konstanz, Germany, September 1996.

[14] R. Güting, M. Böhlen, M. Erwig, C. Jensen, N. Lorentzos, M. Schneider, and M. Vazirgiannis. A Foundation for Representing and Querying Moving Objects. *ACM TODS*, 25(1):1–42, March 2000.

[15] R. H. Güting and M. Schneider. Realm-Based Spatial Data Types: The ROSE Algebra. *VLDB Journal*, 4:243–286, 1995.

[16] J. L. E. McKenzie and R. T. Snodgrass. Evaluation of Relational Algebras Incorporating the Time Dimension in Databases. *ACM Computing Surveys*, 23(4):501–543, 1991.

[17] P. B. Ladkin. *The Logic of Time Representation*. PhD thesis, University of California at Berkeley, November 1987.

[18] V. Müller, N. W. Paton, A. A. A. Fernandes, A. Dinn, and M. H. Williams. Virtual Realms: An Efficient Implementation Strategy for Finite Resolution Spatial Data Types. In *SDH'96*, pp. 11B.1–11B.13, August 1996.

[19] A. Pavlopoulos and C.Theodoulidis. Review of Spatio-Temporal Data Models. Technical report, Department of Computation, UMIST, United Kingdom, October 1998. Available from `http://www.crim.org.uk/`.

[20] G. Proietti and C. Faloutsos. Selectivity Estimation of Spatial Queries on Region Data Stored Using an R-tree. In *Proc. 15th ICDE*, 1999.

[21] J. Smith, P. Watson, S. de F. M. Sampaio, and N. W. Paton. Polar: An Architecture for a Parallel ODMG Compliant Object Database. In *Proc. CIKM 2000*, pp. 352–359, November 2000.

[22] R. Snodgrass. Temporal Object Oriented Databases: A Critical Comparison. In *Modern Database Systems: The Object Model, Interoperability and Beyond*, chapter 19, pp. 386–408. Addison-Wesley/ACM Press, 1995.

[23] The CHOROCHRONOS Participants. CHOROCHRONOS, A Research Network for Spatiotemporal Database Systems. *SIGMOD Record*, 28(3):12–21, 1999.

[24] Y. Theodoridis, T. Sellis, A. Papadopoulos, and Y. Monolopoulos. Cost Models for Join Queries in Spatial Databases. In *Proc. 14th ICDE*, 1998.