Chapter 1

TRIPOD: A SPATIO-HISTORICAL OBJECT DATABASE SYSTEM

Tony Griffiths, Alvaro A.A. Fernandes, Norman W. Paton, Seung-Hyun Jeong
Nassima Djafri

Department of Computer Science
University of Manchester, Manchester M13 9PL, UK

griffitt | alvaro | norm | jeong | ndjafri @cs.man.ac.uk

Keith T. Mason[1], Bo Huang[2], Mike Worboys[2]

[1]*Department of Earth Sciences,* [2]*Department of Computer Science*
University of Keele, Staffordshire ST5 5BG, UK

k.t.mason@esci.keele.ac.uk, b.huang | michael@cs.keele.ac.uk

Abstract The storage and analysis of large amounts of time varying spatial and aspatial data is becoming an important feature of many application domains. This requirement has fueled the need for spatio-temporal extensions to data models and their associated querying facilities. To date, much of this work has focused on the relational data model, with object data models receiving far less consideration. Where descriptions of such object models do exist, there is currently a lack of systems which build upon these models to produce database architectures that address the broad spectrum of issues related to the delivery of a fully functional spatio-temporal DBMS. This chapter presents an overview of such a system by describing a spatio-historical object model that utilizes a specialized mechanism, called a history, for maintaining knowledge about entities that change over time, and a tour through the query processing architecture of the system. Key features of the resulting proposal include: (i) consistent representations of primitive spatial and timestamp types; (ii) a component-based design in which spatial, timestamp and historical extensions are formalized incrementally, for subsequent use together or separately; (iii) compatibility with mainstream query processing frameworks for object databases; and (iv) the integration of the spatio-temporal proposal with the ODMG standard.

Keywords: Spatio-temporal database, object database, GIS

1. Introduction

Many applications, for example in planning or transport, must store and analyse large amounts of time varying data. However, such data can be difficult to model effectively in existing database systems, which has led to spatio-temporal databases becoming an active area of research. We contend that due to the considerable design and implementation challenges involved in developing complete systems, much of the research activity in spatio-temporal databases has focused on specific parts of the problem, at the expense of a more holistic view of database systems design and development [Paton et al., 2000]. This has resulted in a lack of prototype systems that can track changes to spatial and aspatial data over time. 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, 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 endeavor has been pursued only rarely. This chapter provides an overview of the Tripod project, which is developing a spatio-temporal object database system that extends the ODMG standard for object databases [Cattell, 2000]. Figure 1.1 illustrates the relationships between the different components in Tripod. At the core is the ODMG object model, which is extended with primitive spatial and timestamp types.

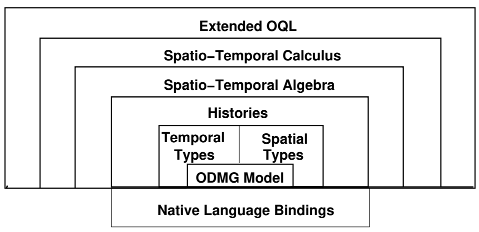

Figure 1.1. Tripod components.

The spatial types are those of the ROSE algebra [Güting and Schneider, 1995], and the timestamp types are one dimensional versions of the two dimensional ROSE algebra types `Points` and `Lines`. Past states of all ODMG types, including the spatial and timestamp types, can be recorded using a specialised mechanism called a history. Figure 1.1 from *Histories* inwards represents a spatio-historical object model.

Outside *Histories* in Figure 1.1, the upper half of the figure represents the declarative query interface, while the lower half of the figure represents the imperative programming interface. Tripod databases can be queried using ODMG OQL, which is currently used without extensions to its original constructs. This is possible because: (i) histories, points, lines, regions, instants and timeIntervals are queried in the same way as other ODMG collections; and (ii) because point, line, region, instant and timeInterval are queried in the same way as ODMG structured literals. We anticipate that future developments will involve extensions to the OQL language to support, for example, temporal aggregation. Tripod's OQL is given a semantics and an optimization infrastructure through a mapping onto an extension of the monoid comprehension calculus of [Fegaras and Maier, 2000], as described in [Griffiths et al., 2001a]. The programming interface follows the ODMG approach by mapping object model constructs into programming language objects within an existing object-oriented programming language.

This chapter is structured as follows. Section 2 presents an example case study used throughout the remainder of this chapter. Section 3 presents the Tripod object model, focusing on its spatial and timestamp literals, and its historical construct. Through examples drawn from the case study presented in Section 2, Section 4 describes an architecture implementing this model, and illustrates how Tripod's programming language bindings and query processing architecture can be exploited to support the case study. The case study is representative of many in which the application tracks discrete changes to both spatial and aspatial data over time.

2. Case Study: UK National Land Use Database

In the UK, a project is underway to create a National Land Use Database (NLUD) (`http://www.nlud.org.uk/`). The NLUD aims to provide a complete, consistent and detailed geographical record of land use in England. Land use parcels (the basic spatial units of the system) will likely be formed from Ordnance Survey (the UK mapping agency) Digital National Framework (DNF) 'atomic polygons', which are themselves defined by topographic features and uniquely referenced by a system of topographic identifiers. The NLUD will be delivered by specific projects that respond to particular user requirements. The Tripod investigation, although not officially linked with the NLUD project, builds on one such initiative, the NLUD Previously Developed Land (PDL) project, as a basis for testing the applicability of the Tripod model and languages on a land use change scenario. The PDL project has been set up by the NLUD partnership to monitor the supply and re-use of vacant, derelict, or previously developed sites, that might be available for further development [Ordnance Survey, 2001b].

Under the PDL proposals, sites are categorised as belonging to one of six possible classes, including land and buildings which are now vacant, derelict land and buildings, and land and buildings going through the various stages of planning permission or construction. A key objective of the NLUD PDL project is to maintain the life histories of PDL sites and to support the update and maintenance of site records as the user records change [Ordnance Survey, 2001a]. Potential changes to individual land use parcels might record changes to one or more of the site attributes, for instance an alteration of PLD classification, or might come from one of seven possible categories of geometric change, including: creation, destruction, alteration, reincarnation, fusion, fission and reallocation.

3. The Tripod Object Model

The ODMG Object Model provides a set of object and literal types – including collection types, (e.g., `Set`, `Bag` and `List`) and atomic types (e.g., `long`, `float` and `string`) – with which a designer can specify their own object types, and construct a particular database schema. Each user-defined type has a structure (a collection of attributes and binary relationships with other user-defined types) and a behaviour (a collection of methods whose implementation is specified using the language binding).

Tripod supports the storage, management and querying of spatial and aspatial 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 that 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. For example, the `lu_parcel` type shown in Figure 1.2 declares historical attributes (`owner` and `land_type`), a spatio-historical attribute (`gext`), and two historical relationships (`has_tpfea` and `in_admin`). In addition, the `lu_parcel` type is itself declared to be historical, indicating that the database should maintain a history (called *lifespan*) recording when instances of this type are active or inactive (i.e., logically deleted) in the database. In contrast, the `admin_region` class is not declared as historical, and therefore its instances will not have their lifespan maintained.

The remainder of this section provides an overview of the Tripod object model [Griffiths et al., 2001b] by presenting its constructs as instances of abstract data

```
class admin_region
( extent admin_regions key name)
{ attribute string name;
  attribute Instant founded;
  historical(timeIntervals, MONTH) attribute regions gext;
  historical(timeIntervals, MONTH) relationship set<lu_parcel>
    has_parcel inverse lu_parcel::in_admin;
};

class council extends admin_region ( extent councils ) { ...  };

class county extends admin_region ( extent counties ) { ...  };

historical(timeIntervals,MONTH) class lu_parcel
( extent lu_parcels key site_reference )
{ attribute string site_reference;
  historical(timeIntervals, YEAR) attribute list<string> owner;
  historical(timeIntervals, MONTH) attribute string land_type;
  historical(timeIntervals, MONTH) attribute regions gext;
  historical(timeIntervals, MONTH) relationship set<topo_feature>
    has_tpfea inverse topo_feature::lup;
  historical(timeIntervals, MONTH) relationship admin_region in_admin
    inverse admin_region::has_parcel;
};

historical(timeIntervals,MONTH) class topo_feature
( extent topo_features key toid )
{ attribute string toid;
  historical(timeIntervals,MONTH) attribute string feature_type;
  historical(timeIntervals,YEAR) attribute regions gext;
  historical(timeIntervals,MONTH) relationship lu_parcel lup
    inverse lu_parcel::has_tpfea;
};
```

Figure 1.2. Land Use Schema Definition

types (ADTs), and commences by overviewing the structure of Tripod spatial values, showing how these provide a foundation for Tripod timestamps.

3.1 Spatial Literals

Tripod's spatial data types (SDTs) are based on the ROSE (RObust Spatial Extensions) approach described in [Güting and Schneider, 1995]. Underlying the ROSE approach is the notion of a *realm*. A realm is essentially a finite set of points and non-intersecting line segments defined over a discrete grid that forms the ROSE algebra's underlying geometric domain.

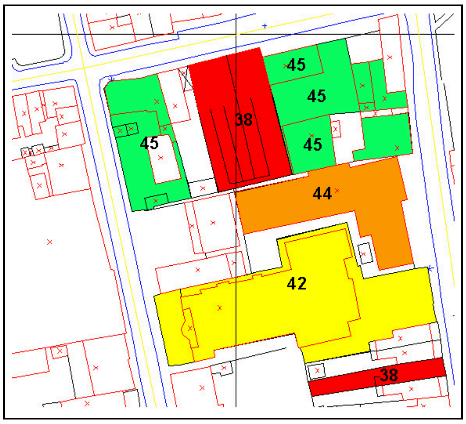

Figure 1.3. Example of `lu_parcel` objects in a realm

The ROSE approach defines an algebra over three SDTs, namely `Points`, `Lines` and `Regions`, and an extensive collection of spatial predicates and operations (including set operations) over these types [Güting and Schneider, 1995]. Every spatial value in the ROSE algebra is set-based, thus facilitating set-at-a-time processing. Roughly speaking, each element of a `Points` value is a pair of coordinates in the underlying geometry, each element of a `Lines` value is a set of connected line segments, and each element in a `Regions` value is a polygon containing a (potentially empty) set of holes.

Some examples of spatial objects taken from the NLUD are shown in Figure 1.3. The polygonal objects **38**, **42**, **44** and **45** are `Regions` values (note that **45** is a set of four polygons that contain holes), representing land use parcels. Other objects of interest (marked with an $\times$) are represented by `Points` values denoting their centroid. This NLUD example does not use `Lines` values, although they could be used to represent geographical objects such as roads. An example of an operation to find the area of a proposed fusion of the Regions values representing objects **38** and **45** (denoted by $\mathtt{r}_1$ and $\mathtt{r}_2$ respectively), if they share a common border, in pseudo code could be:

```
if(r1.border_in_common(r2)) {
  Regions fused := r1.plus(r2);
  float fused_area := fused.area();
}
```

where `plus` is the ROSE algebraic operation that computes the union of two spatial values.

3.2 Timestamp Literals

Tripod extends the set of ODMG primitive types with two timestamp types, called `Instants` and `TimeIntervals`. The underlying domain of interpretation is a structure that we refer to as a *temporal realm* because it is defined to be a one-dimensional specialization of the two-dimensional (spatial) realms. In general terms, a temporal realm can be thought of as a finite set of integers (whereas a spatial realm is a finite integer grid). Reasons why we adopt this viewpoint and terminology include (amongst others):

- Tripod is a spatio-historical database system and we find it useful (for developers as well as users) to have realms as a unifying notion for the interpretation of operations on spatial *and* temporal values.
- Realm operations are well-defined and have a rich set of predicates and constructors with nice closure properties.
- Realm values are collections, which we find more suitable than singletons for the kind of set-at-a-time strategies that are prevalent in query processing architectures.
- This unification at the level of interpretations propagates upwards in the sense that the predicates and operations on realms are defined once and used (possibly after renaming) over both spatial and temporal values.

In a temporal realm, we may think of a time-point as an integer. Then, an `Instants` value is a collection of time-points and a `TimeIntervals` value is a collection of pairs of time-points where the first element is the start, and the second the end, of a contiguous time-interval. A *timestamp* is either an `Instants` value or a `TimeIntervals` value. Figure 1.4 illustrates timestamps in graphical form, where timestamp ***A*** is a `TimeIntervals` value, and timestamps ***B*** and ***C*** are `Instants` values. Notice that ***B*** happens to be a singleton.

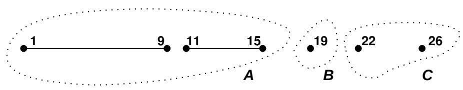

Figure 1.4. Example Tripod Timestamps

In the ROSE algebra, there is no predefined notion of one `Points` value being spatially ordered with respect to another `Points` value; any such notion of ordering must be defined within application programs that use the algebra. The Tripod temporal algebra, therefore, extends the ROSE algebra with ordering predicates based on the underlying order of the temporal realm's integer domain. These predicates take into consideration the collection-based nature of the timestamp types. Therefore, in addition to what might be considered the 'standard' temporal predicates (e.g., those defined by Allen's algebra [Allen, 1983]), our temporal predicates are extended to take into account quantification over the individual elements of the timestamp. For example, whether every element of a timestamp `A` must be contained by an element from timestamp `B`, or just some. In addition, the temporal realm utilises a calendar that maps from the underlying integer domain to one more suited to human cognition.

Although Tripod timestamps can be used by application designers to complement the 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.

3.3 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 chapter, let $\mathbb{T}$ denote 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 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), and 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).

The remainder of this section provides an overview of the operations available to operate on histories construed as instances of an ADT, which leads to their behaviour being categorised into constructor, query, merge and update operations. These operations require precise definition so that the semantics of operations and structures in the higher level layers of the Tripod OM can be appropriately specified. For example, the language bindings utilise operations to create and manipulate historical data, and the query language and its associated calculus utilise operations that filter and traverse histories to retrieve appropriate results. For space reasons, the semantics of these operations are provided by exemplars; for full details see [Griffiths et al., 2001b] [Griffiths et al., 2000].

Representative retrieval operations on histories are shown in Figure 1.5. Note that the first expression in Figure 1.5 is in fact a template for a set of signatures parameterised on any element of the set of predicates on timestamps. For example, given that $\underline{\texttt{starts\_before}}$ is a member of that set, letting $\omega = \underline{\texttt{starts\_before}}$ in the template yields the following signature $\texttt{ContainsTimestamp}\_\underline{\texttt{starts\_before}} : \mathbb{H} \times \mathbb{T} \rightarrow \texttt{boolean}$.

$$\begin{aligned} \texttt{ContainsTimestamp\_}\omega : &\quad \mathbb{H} \times \mathbb{T} \quad \rightarrow \quad \texttt{boolean} \\ \texttt{FilterByTimestamp\_}\omega : &\quad \mathbb{H} \times \mathbb{T} \quad \rightarrow \quad \mathbb{H} \\ \texttt{FilterBySnapshot} : &\quad \mathbb{H} \times \mathbb{V} \quad \rightarrow \quad \mathbb{H} \end{aligned}$$

Figure 1.5. Example Retrieval Operations

For example, if the state set of two histories representing the history of change to instances H_1 and H_2 of the $\texttt{lu\_parcel}$ type's $\texttt{gext}$ attribute (both with $V = \texttt{Regions}$, $\theta = \texttt{TimeIntervals}$ and identical γ) are $\Sigma_1 = \{\langle[1-6], \texttt{r}_1\rangle, \langle[9-11], \texttt{r}_2\rangle\}$ and $\Sigma_2 = \{\langle[5-10], \texttt{r}_3\rangle, \langle[13-20], \texttt{r}_1\rangle\}$ (where $\texttt{r}_1$, $\texttt{r}_2$ and $\texttt{r}_3$ are $\texttt{Regions}$ values) then: $\texttt{ContainsTimestamp}\_\underline{\texttt{starts\_before}}(H_1, [5-7]) = \texttt{true}$ and $\texttt{ContainsTimestamp}\_\underline{\texttt{ends\_after}}(H_2, [15-22]) = \texttt{false}$.

$$\begin{aligned} (union)\ \Cup : &\quad \mathbb{H} \times \mathbb{H} \quad \rightarrow \quad \mathbb{H} \\ \texttt{DeleteTimestamp} : &\quad \mathbb{H} \times \mathbb{T} \quad \rightarrow \quad \mathbb{H} \\ \texttt{InsertState} : &\quad \mathbb{H} \times \mathbb{S} \quad \rightarrow \quad \mathbb{H} \end{aligned}$$

Figure 1.6. Example Update Operations

Representative update operations on histories are shown in Figure 1.6. As an example, the union of two histories (obtained through the $\Cup$ operator) 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 constraint that a history does not record different values as valid at the same time. 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 $H = H_1 \Cup H_2$ is $\Sigma = \{\langle[1-5, 13-20], \texttt{r}_1\rangle, \langle[5-10], \texttt{r}_3\rangle, \langle[10-11], \texttt{r}_2\rangle\}$. The definitions of $\Cap$ and $\backslash\backslash$, for intersecting and subtracting histories, are analogous.

$\texttt{DeleteTimestamp}$ takes a history $H = \langle V, \theta, \gamma, \Sigma\rangle$ and a timestamp τ of type θ and yields a new history $H' = \langle V, \theta, \gamma, \Sigma'\rangle$ in which all states in Σ whose timestamp τ' is such that $\underline{\texttt{common\_instants}}(\tau, \tau')$ is true, have been recomputed so that τ does not occur in Σ'. For example, if $\tau = [3-4]$ and $\Sigma = \{\langle[1-6], \texttt{r}_4\rangle\}$, then $\Sigma' = \{\langle[1-3, 4-6], \texttt{r}_4\rangle\}$.

$\texttt{InsertState}$ 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 instants 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 $\langle \tau', \sigma' \rangle = \langle [5-8], \mathtt{r}_1 \rangle$ and $\Sigma = \{\langle [1-6], \mathtt{r}_2 \rangle\}$, then $\Sigma' = \{\langle [1-5], \mathtt{r}_2 \rangle, \langle [5-8], \mathtt{r}_1 \rangle\}$ and if $\langle \tau', \sigma' \rangle = \langle [5-8], \mathtt{r}_4 \rangle$ and $\Sigma = \{\langle [1-6], \mathtt{r}_4 \rangle\}$, then $\Sigma' = \{\langle [1-8], \mathtt{r}_4 \rangle\}$.

4. Architecture

This section describes in more detail the various components of the Tripod architecture shown in Figure 1.1, and in particular (as shown in Figure 1.7) how these components interact with each other in the specification of spatio-historical database applications. There are three main components in the Tripod architecture: a Persistent Store that is responsible for loading and saving persistent objects to and from the database and also for maintaining metadata about a particular database schema; a Query Processor that is responsible for optimizing and executing database queries; and a Programming Language Binding that is responsible for providing programming language access to the database.

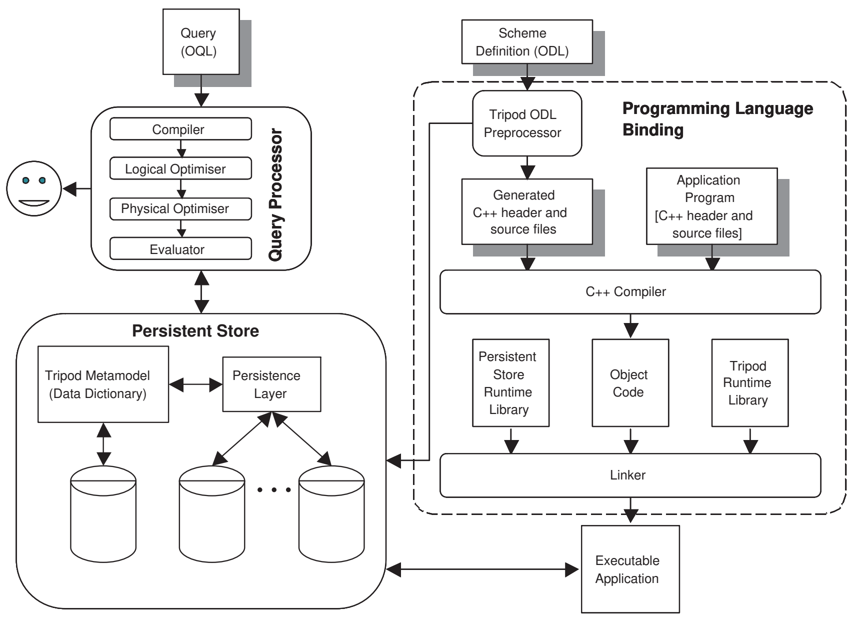

Figure 1.7. Detailed Tripod Architecture.

The definition of a Tripod database consists of two parts: a schema (defined using a declarative object definition language (ODL)) to specify the structure of user-defined types and their behaviour, and an implementation of each of these behaviours specified using a programming language binding – in our case this is C++. Since the ODMG model does not define an object manipulation language (OML), developers must use the programming language binding to create, update and delete objects. The Tripod ODL preprocessor lies at the core of the process of producing a database specification. It is responsible for analysing an ODL schema specification to produce: a set of C++ header files whose structure corresponds to that of the types expressed in the ODL schema definition; an instance of the Tripod metamodel (which is a superset of the ODMG metamodel) containing high-level information about the structure of the database schema that is used by (amongst others) the query processor; and methods to load and save persistent objects to and from the Tripod persistent store.

Once the application program and type information is compiled into object code, it is linked with libraries that implement the Tripod runtime system, and the persistent store. The library implements the core ODMG object model types as well as the Tripod spatial, timestamp and historical types. The persistent store runtime library contains the functionality needed to create and manage database connections, transactions and queries. The output of this process is an executable application that interacts with the underlying spatio-historical OODBMS.

When the state of a database needs to be queried, developers can either write native language application programs or issue declarative OQL queries.

The following sections illustrate how Tripod's programming language bindings and query processing architecture are utilised to support the population and querying of a database corresponding to the schema of Figure 1.2. In particular we will focus on the transformations that a spatio-historical query undergoes during its life-cycle from a declarative OQL query to a physical execution plan.

4.1 The Language Bindings

The Tripod language bindings provide developers with a programming language (C++) interface that allows them to create, update and delete objects (i.e., an OML). The language bindings are also used to specify a user-defined type's operations. When the state of a database needs to be queried, developers can either issue declarative OQL queries or write native language application programs. The language bindings extend those of the ODMG standard by mapping the Tripod OM types into C++ classes that can persist in the database, and provide implementations of the Tripod spatial, timestamp and historical types.

For each type in a Tripod schema, the Tripod ODL preprocessor generates a corresponding C++ class. For example, Figure 1.8 is the class definition automatically generated for the `lu_parcel` type of Figure 1.2 (note that all operations have been omitted). Line 1 illustrates that each persistence capable class inherits from the built-in `d_Object` type. Each type's historical properties are mapped to a history template type which requires the type of the property, its timestamp type, and its granularity. For example, the `owner` attribute is mapped to a history type (line 6) whose snapshots are each a list of strings. The `has_tpfea` relationship on the other hand is mapped to a history type (line 9) whose snapshots are each a set of topological features.

```
1  class lu_parcel : public d_Object {
2  private:
3   history<d_TimeIntervals,Status,MONTH> lifespan;
4  public:
5   d_String site_reference;
6   history<d_TimeIntervals,d_List<d_String>,YEAR> owner;
7   history<d_TimeIntervals,d_String,MONTH> land_type;
8   history<d_TimeIntervals,d_Regions,MONTH> gext;
9   history<d_TimeIntervals,d_Rel_Set<topo_feature,lup>,MONTH> has_tpfea;
10  history<d_TimeIntervals,d_Rel_Ref<admin_region,has_parcel>,MONTH>
11       in_admin;
12 };
```

Figure 1.8. `lu_parcel` class definition

Figure 1.9 shows how instances of user-defined types are created using an extended version of the C++ `new` operator. Line 8 creates a new `lu_parcel` object that is stored in the `ludb` database, with an appropriate lifespan. Lines 10 and 11 utilise the History type's `InsertState` function to populate this object's `gext` spatio-historical attribute with two states whose snapshots are previously created regions values (not shown). Line 12 deletes a portion of this history. Lines 14 to 19 illustrate how a query can be issued within a program, whose results are then available to be manipulated within the program. Lines 14 to 19 are equivalent to the query: "*Name all land parcels that ever enclosed landfill sites*". For further example of the language bindings the reader is directed to [Griffiths et al., 2001c].

4.2 Query Processing

Tripod's OQL employs and extends the facilities of OQL to retrieve spatial, timestamp and historical information. States in histories are extracted through iteration in the OQL *from-clause*. Constraints in the *where-clause* can then be applied to the snapshot value, timestamp or index number of a state, and

```
d_TimeIntervals t1("[1/1990 - until_changed]");
d_TimeIntervals t2("[1/1990 - 4/1995]");
d_TimeIntervals t3("[4/1995 - 5/1999]");
d_State<d_Regions,d_TimeIntervals> s1(regions1,t1);
d_State<d_Regions,d_TimeIntervals> s2(regions2,t3);
d_Bag<d_Ref<struct<lu_parcel, topo_feature, d_TimeIntervals> > > res;

d_Ref<lu_parcel> lup8601 = new(ludb,"lu_parcel", t1) lu_parcel;

lup8601->gext.InsertState(s1);
lup8601->gext.InsertState(s2);
lup8601->gext.DeleteTimestamp(t2);

d_oql_execute(res, "SELECT l.site_reference, tp.toid,
                      INTERSECTION(l.gext.vt, tp.gext.vt)
                    FROM  lu_parcels l, topo_features tp
                    WHERE l.gext.value ENCLOSES tp.gext.value  AND
                          l.gext.vt COMMON_INSTANTS tp.gext.vt AND
                          tp.feature_type = \'landfill\'");

print_results(res);
```

Figure 1.9. Language binding example

the granularity of the timestamp, through operations that are defined in the Tripod OM. Finally, the result is obtained through the projection operation in the *select-clause*. Figure 1.9 has illustrated one spatio-historical query, and further examples are given in Figure 1.10 using the NLUD application.

4.2.1 Logical Optimization. Using information contained in the Tripod metamodel, OQL queries are mapped to a spatio-historical calculus and then to a spatio-historical (logical) algebra, each of which is subject to logical optimization. The calculus provides opportunities for logical optimization using rewrite rules based on the *normalisation algorithm* of Fegaras and Maier [Fegaras and Maier, 2000]. Figure 1.11 shows, in the general case, how an OQL select-from-where clause maps into a monoid comprehension calculus query.

In the specific case of the query issued in Figure 1.9, this mapping results in the comprehension shown in Figure 1.12.

After the normalization stage, the next step is to translate the comprehension into a (logical) algebraic form consisting of joins, selections, unnests, and reductions. The following are two basic rules for translating a monoid comprehension into the join and selection algebraic operators respectively.

- $X \bowtie_p Y = \{(v,w) \mid v \leftarrow X, w \leftarrow Y, p(v,w)\}$

What is the area of the parcels that at some point in time intersected with land parcel 2312? (spatio-historical query)

```
select  area(lupgext2.value), lup2.site_reference
from    lup1 in lu_parcels, lup2 in lu_parcels,
        lupgext1 in lup1.gext, lupgext2 in lup2.gext
where   lup1.site_reference = '2312' and
        lupgext1.value.intersects (lupgext2.value) and
        lupgext1.validTime.common_instants (lupgext2.validTime) and
        lup1 != lup2
```

What were the neighbouring parcels of parcel 2801 that ever enclosed landfill site? (spatio-historical join)

```
select  distinct lup2.site_reference
from    lup1 in lu_parcels, lup2 in lu_parcels,
        lupgext1 in lup1.gext, lupgext2 in lup2.gext
        tpfeas in lup2.has_tpfea, topo_feature in tpfeas.value,
        tpfea_type in topo_feature.feature_type
where   lup1 != lup2 and
        lup1.site_reference = '2810' and
        lup2gext.value.border_in_common(lup1gext.value) and
        tpfea_type.value = 'landfill' and
        lup2gext.validTime.common_instants (topo_feature.validTime)
```

Figure 1.10. Spatio-Historical Queries

OQL:
```
select  e
  from  x1 in e1, ..., xn in en
 where  p
```

Monoid comprehension: $\uplus\{e \mid x_1 \leftarrow e_1, \ldots, x_n \leftarrow e_n, p\}$

Figure 1.11. Mapping Select-From-Where

$$\cup\{\langle l.site\_reference,\ tp.toid,\ \underline{\texttt{intersection}}(t_1,t_2)\rangle \mid l \leftarrow lu\_parcels,\ tp \leftarrow topo\_features,\ (t_1,v_1) \leftarrow l.gextent,\ (t_2,v_2) \leftarrow tp.gextent,\ \underline{\texttt{encloses}}(v_1,v_2),\ \underline{\texttt{common\_instants}}(t_1,t_2),\ tp.feature\_type = "landfill"\}$$

Figure 1.12. Monoid Comprehension

- σ_p(X) = {v | v ← X, p(v)}

The first of these two rules characterizes the join X $\bowtie_p$ Y of the collections X and Y using the predicate p. The second rule characterizes the selection σ_p(X) of the elements in X that satisfy the predicate p.

4.2.2 Physical Optimization and Query Evaluation. Once the Tripod optimizer has generated an appropriate calculus expression, this expression can be transformed into a collection of equivalent logical algebraic expressions. Based on database statistics, it is the responsibility of the physical optimizer to select the most appropriate expression one. The calculus expression for our example query (shown in Figure 1.12) can be expressed by combinations of join, selection, and unnest algebraic operators such as those illustrated in Figure 1.13.

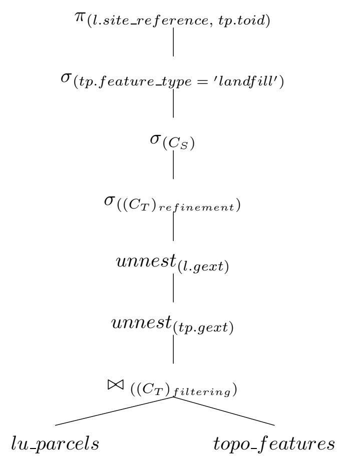

Figure 1.13a.

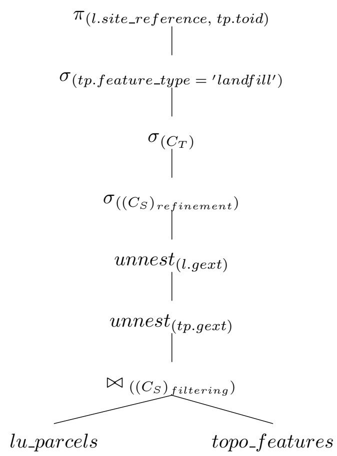

Figure 1.13b.

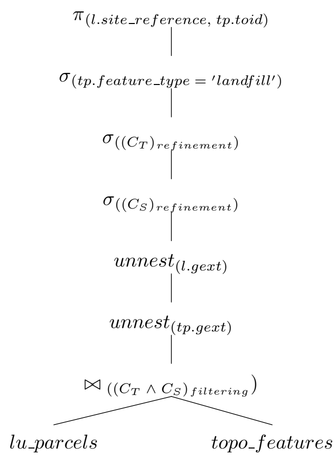

Figure 1.13c.

Figure 1.13. Logical Algebraic Expressions

In terms of spatio-temporal joins over geometric histories, Figure 1.13 describes three possible strategies: a) uses a temporal join over histories followed by a spatial selection, b) uses a spatial join over histories followed by a temporal selection, and c) uses a specialized spatio-temporal join over histories. In Figure 1.13, C_T and C_S stand for the temporal and spatial portion of a given spatio-temporal join condition, respectively. The subscript *filtering*s associated with join conditions imply that those conditions are applied to approximations of *spatial histories* such as *MBRs* (*Minimum Bounding Rectangles*) and *MBCs* (*Minimum Bounding Cuboids*) over histories. In terms of the query results (i.e., the set of pairs of states), join processing over histories consists of a *filtering step* for the results, followed by the spatial/temporal selections required to refine the intermediate results based on the value of each snapshot. Those selections that are involved in the spatial/temporal portion of the given spatio-temporal join condition are associated with the subscript *refinement*s.

In this example (for each binding of the states l and tp from the extents $land\_parcels$ and $topo\_features$):

$$(C_T)_{filtering} \equiv \underline{\texttt{common\_instants}}(\mathbf{TMBR}(l.gext), \mathbf{TMBR}(tp.gext))$$

$$(C_S)_{filtering} \equiv \underline{\texttt{encloses}}(\mathbf{SMBR}(l.gext), \mathbf{SMBR}(tp.gext))$$

$$(C_T)_{refinement} \equiv \underline{\texttt{common\_instants}}(l.gext.vt, tp.gext.vt)$$

$$(C_S)_{refinement} \equiv \underline{\texttt{encloses}}(l.gext.val, tp.gext.val)$$

where the functions $\mathbf{TMBR}$ and $\mathbf{SMBR}$ take as argument a spatial history and return approximated rectangles minimally enclosing the spatial history in temporal(with zero for y-axis values) and spatial domains, respectively.

In the expression in Figure 1.13a the intermediate result of a temporal join over historical attributes of instances l of $lu\_parcels$ and tp of $topo\_features$ are nested collections (i.e. *spatial histories*). Each of these collections must therefore be unnested to allow the construction of the result (a collection of pairs of individual *states* of spatial values) of the required spatio-temporal join. Next, a temporal selection refines the unnested join result which is filtered under the temporal part of the given spatio-temporal condition for *histories*. Lastly, a spatial selection is executed to apply the spatial part of the spatio-temporal join condition to the temporally filtered result. The expression in (b) is similar to the expression (a), but the spatial part of the spatio-temporal join condition is applied first. However, in expression (c), a join is performed with the whole spatio-temporal join condition. Two selections for spatial and temporal refinements are thus needed.

Once generated, the logical algebraic expressions have to be finally translated into a physical query plan comprising physical operators. Focusing on spatial and temporal joins, various algorithms for physical operators have been proposed [Zhang et al., 2000], [Arge et al., 2000], however, the algorithms basically use a limited number of fundamental techniques, such as sort-merge, hashing, and indexing.

For example, Figure 1.14 illustrates a physical query plan corresponding to the join processing of algebraic expression (c) in Figure 1.13 (note that the latter stages of the plan have been omitted for space). The query plan adopts an index-based spatio-temporal join algorithm employing a *synchronized R-tree traversal* [Brinkhoff et al., 1993].

$$
\begin{array}{c}
Filter_{((C_T)_{refinement})} \\
| \\
Filter_{((C_S)_{refinement})} \\
| \\
Unnest_{(l.gext)} \\
| \\
Unnest_{(tp.gext)} \\
| \\
Sync\_Traversal\_Join_{(L,\, TP,\, R_L,\, R_{TP},\, (C_T \wedge C_S)_{filtering})}
\end{array}
$$

Figure 1.14. A Physical Query Plan With A Spatio-Temporal Physical Join Operator

A specialised physical operator, *Sync_Traversal_Join* takes two sets of spatio-temporal objects (i.e., objects that maintain a history of a spatial property), two R-trees [Guttman, 1984] over 3-dimensional cuboid approximations of spatial histories [Theodoridis et al., 1996], and a spatio-temporal join condition as arguments. The operator outputs a subset of the Cartesian product of the two input collections whose elements satisfy the given join condition by traversing the two R-trees using a depth-first synchronized R-tree traversal. Central to the algorithm is the exploitation of the property that MBCs in each internal node of an R-tree cover the MBCs within its subtree. Hence, if two internal nodes have MBCs that do not intersect, then there is no intersection within their subtrees either. To compute all pairs of entries in leaf nodes which satisfy a given spatial join condition, the algorithm performs a synchronous depth-first traversal of the two R-trees. *Filter* is a physical selection operator that takes as arguments a set of objects and a selection condition, and outputs the objects that satisfy the condition. In Figure 1.15, the details of the spatio-temporal join algorithm are described.

In summary, the mapping from OQL to the calculus and then onto an equivaluent algebra is achieved in four major phases in the Tripod framework:

1 *Translation into an intermediate calculus*: During this phase, a Tripod OQL query is translated into a monoid comprehension.

Sync_Traversal_Join$(\mathbf{Q}, \mathbf{R}, \mathbf{T_Q}, \mathbf{T_R}, \mathbf{C_{ST}}) \rightarrow \mathbf{S}$
input: $\mathbf{Q}$ and $\mathbf{R}$, each of which is a collection of spatio-temporal objects.
$\mathbf{T_Q}$ and $\mathbf{T_R}$, which are the descriptors of the 3D R-trees for Q and R, respectively.
$\mathbf{C_{ST}}$ is a conjunction of spatial and temporal join predicates
output: The result of the join is a collection $\mathbf{S}$ of pairs of historical objects from $\mathbf{Q}$ and $\mathbf{R}$.
begin
$S := \emptyset$;
$P := \emptyset$; /* P is an intermediate result from ***3DRTreeTraversal*** */
$P :=$ ***3DRTreeTraversal***$(\mathbf{T_Q}.\mathbf{root}, \mathbf{T_R}.\mathbf{root}, \mathbf{C_{ST}})$;
for each pair $\langle \mathbf{e_q}.\mathbf{oid}, \mathbf{e_r}.\mathbf{oid} \rangle \in \mathbf{P}$ *do*
fetch the historical object $\mathbf{H_q}$ from $\mathbf{Q}$ referenced by $\mathbf{e_q}.\mathbf{oid}$;
fetch the historical object $\mathbf{H_r}$ from $\mathbf{R}$ referenced by $\mathbf{e_r}.\mathbf{oid}$;
$S := \mathbf{S} \cup \langle \mathbf{H_q}, \mathbf{H_r} \rangle$;
end for
end

/* synchronized depth-first 3D R-tree traversing */
3DRTreeTraversal$(\mathbf{Ref_q}, \mathbf{Ref_r}, \mathbf{C_{ST}}) \rightarrow \mathbf{P}$, respectively
input: $\mathbf{Ref_q}$ and $\mathbf{Ref_r}$, which point to a node in the R-trees $\mathbf{T_Q}$ and $\mathbf{T_R}$.
output: The result of the tree traversal is a collection $\mathbf{P}$ of pairs of object identifiers stored in leaf nodes of R-trees $\mathbf{T_Q}$ and $\mathbf{T_R}$.
begin
/*$\mathbf{e_q}$ and $\mathbf{e_r}$ are entries in nodes $\mathbf{N_q}$ of $\mathbf{T_Q}$ and $\mathbf{N_r}$ of $\mathbf{T_R}$ referenced by $\mathbf{Ref_q}$ and $\mathbf{Ref_r}$, and are in the form of $\langle \mathbf{ref}, \mathbf{cuboid} \rangle$ in internal nodes, or $\langle \mathbf{oid}, \mathbf{cuboid} \rangle$ in leaf nodes, where $\mathbf{ref}$ is a pointer to a child node, $\mathbf{oid}$ is a pointer to an object stored, and $\mathbf{cuboid}$ is an MBC */
for each $\mathbf{e_q} \in \mathbf{N_q}$ *do*
for each $\mathbf{e_r} \in \mathbf{N_r}$ *where* $\mathbf{e_q}.\mathbf{cuboid}$ and $\mathbf{e_r}.\mathbf{cuboid}$ satisfy $\mathbf{C_{ST}}$ *do*
if ($\mathbf{N_q}$ is a leaf node)
then $P := P \cup \langle \mathbf{e_q}.\mathbf{oid}, \mathbf{e_r}.\mathbf{oid} \rangle$;
else $P :=$ ***3DRTreeTraversal***$(\mathbf{e_q}.\mathbf{ref}, \mathbf{e_r}.\mathbf{ref}, \mathbf{C_{ST}})$;
end for
end for
end

Figure 1.15. A Spatio-Temporal Join Algorithm Based on Synchronized 3D R-tree Traversal Technique

2 *Normalization*: During this phase, some forms of the nested queries are unnested using the normalization algorithm of [Fegaras and Maier, 2000] and the generator domains are reduced to simple paths.

3 *Translation into an intermediate algebraic form*: After the normalised form of the comprehen is obtanied, it is translated into an intermediate algebra.

4 *Physical plan generation*: During the last phase, all the available access paths and physical algorithms are considered in order to generate different plans. Finally, the best plan is selected and executed.

5. Related Work

Despite progress in certain aspects of spatio-temporal modelling and implementation (e.g., indexing, join algorithms, etc.), there are few examples of spatio-temporal database systems, and most lack support for changes to aspatial data. Langran [Langran, 1992] developed a spatial vector model in which line segments are used as primitives to produce polygons. Each of these polygons is then timestamped with its own attribute history using discrete semantics. The TRIAD model [Peuquet and Qian, 1997] takes a different approach by using events as the basic notion in their raster-based model. More recently, MADS [Parent et al., 1999] reflects many of the concerns addressed in the Tripod object model, including the orthogonal treatment of spatial and aspatial data. However, MADS does not address manipulation and querying issues.

Recently, much work on spatio-temporal databases has centered on objects whose properties (spatial and aspatial) are continuously changing (so-called *moving-objects*). Such models (e.g., [Güting et al., 2000]) 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. However, such models do not provide comprehensive support for temporally changing aspatial data and object model constructs such as relationships, which are supported in a uniform way in Tripod. In contrast, the Tripod data model and calculus do not model continuous change, as we explicitly target the large body of applications in which objects change in discrete steps, as exemplified by the NLUD case study.

One of the aims of the Tripod design has been to provide effective support for spatial (but possibly not historical) and historical (but possibly not spatial) applications, as well as those that must manage spatio-historical data. As a "pure" spatial database, Tripod provides spatial types as primitive types in the object model, and thus the extension to the ODMG model in Tripod can be seen as analogous to the extension of object-relational products with spatial Data Blades or Cartridges. As a "pure" temporal database, Tripod provides a concise collection of valid-time modelling facilities. Probably the most closely related work in this particular respect is that described in [Bertino et al., 1998], which also presents an extension to the ODMG model. The principal contributions of Tripod relative to [Bertino et al., 1998] have been to address programming and querying of the temporal ODMG extension. Another closely related proposal is that of TOQL [Fegaras and Elmasri, 1998]. Tripod has taken a similar approach to TOQL in the addition of histories into the ODMG object model, although the Tripod temporal types are richer than those in TOQL, and a wider range of query and manipulation operations are supported for histories.

6. Conclusions

This chapter has provided an overview of the Tripod project, which is developing a spatio-temporal object database system. Key features of Tripod are: (i) Modelling, querying and programming facilities are extensions of those provided by the ODMG standard. The extensions to the standard in Tripod are orthogonal, in that the spatial and historical facilities can be used together or separately. (ii) The spatial and temporal types have shared origins, and both provide comprehensive, consistent, collection-based structures and operations to higher levels in the architecture. (iii) The historical modelling facilities can be applied consistently to spatial and to aspatial aspects of an application. In most applications in which historical spatial data is important, historical aspatial data is important as well. (iv) The proposal is targeted at discrete changes to spatial and aspatial data; although there has been considerable attention directed in recent years at moving object databases, we note that few prototypes have been developed that support histories of spatial and aspatial data.

Acknowledgments

Discussions with John Stell, Chris Johnson and Bob Barr, our partners in the Tripod project, have contributed significantly to shape the contributions of this chapter. We are pleased to acknowledge our debt to them. This research is funded by the UK Engineering and Physical Sciences Research Council (EPSRC), whose support is gratefully acknowledged.

References

Allen, J. (1983). Maintaining knowledge about temporal intervals. *CACM*, 26(11):832–843.

Arge, L., Procopiuc, O., Ramaswamy, S., Suel, T., Vahrenhold, J., and Vitter, J. S. (2000). A unified approach for indexed and non-indexed spatial joins. In Zaniolo, et al. (eds.), *Advances in Database Technology - EDBT 2000, Proceedings*, volume 1777 of *Lecture Notes in Computer Science*, pp. 413–429. Springer.

Bertino, E., Ferrari, E., Guerrini, G., and Merlo, I. (1998). Extending the ODMG Object Model with Time. In *Proceedings ECOOP'98*, pp. 41–66.

Brinkhoff, T., Kriegel, H.-P., and Seeger, B. (1993). Efficient processing of spatial joins using r-trees. In Buneman, P. and Jajodia, S., editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, 1993*, pp. 237–246. ACM Press.

Cattell, R. G. G., editor (2000). *The Object Database Standard: ODMG 3.0*. Morgan Kaufmann.

Fegaras, L. and Elmasri, R. (1998). A Temporal Object Query Language. In *Proc. TIME*, pp. 51–59. IEEE Press.

Fegaras, L. and Maier, D. (2000). Optimizing Object Queries Using an Effective Calculus. *ACM TODS*, 25(4):457–516.

Griffiths, T., A.A.A. Fernandes, Djafri, N., and N.W. Paton (2001a). A Query Calculus for Spatio-Temporal Object Databases. In *Proc. TIME*, pp. 101–110. IEEE Press.

Griffiths, T., Fernandes, A., Paton, N., Mason, K., Huang, B., and Worboys, M. (2001b). Tripod: A Comprehensive Model for Spatial and Aspatial Historical Objects. In *Proceedings of ER01*, pp. 84–102. Springer-Verlag.

Griffiths, T., Fernandes, A.A.A., Paton, N.W., Mason, T., Huang, B., Worboys, M., Johnson, C., and Stell, J. (2001c). Tripod: A Comprehensive System for the Management of Spatial and Aspatial Historical Objects. In Aref, W., editor, *Proc. 9th ACM Int. Symposium on Advances in Geographic Information Systems (ACM-GIS)*, pp. 118–123. ACM Press.

Griffiths, T., N.W. Paton, and A.A.A. Fernandes (2000). An ODMG-Compliant Spatio-Temporal Data Model. Preprint series, Dept of Computer Science, University of Manchester. http://pevepc13.cs.man.ac.uk/PrePrints/index.htm.

Güting, R. et al. (2000). A Foundation for Representing and Querying Moving Objects. *ACM Transactions on Database Systems*, 25(1):1–42.

Güting, R. and Schneider, M. (1995). Realm-Based Spatial Data Types: The ROSE Algebra. *VLDB Journal*, 4(2):243–286.

Guttman, A. (1984). R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD*, pp. 47–57.

Langran, G. (1992). *Time in Geographical Information Systems*. Taylor and Francis.

Ordnance Survey (2001a). Previously Developed Land Technical Specification. Web page. http://www.nlud.org.uk/.

Ordnance Survey (2001b). The National Land Use Database. Web page. http://www.nlud.org.uk/.

Parent, C., Spaccapietra, S., and Zimanyi, E. (1999). Spatio-Temporal Conceptual Models: Data Structures + Space + Time. In *Proc. ACM GIS*, pp. 26–33.

Paton, N. W., Fernandes, A. A., and Griffiths, T. (2000). Spatio-Temporal Databases: Contentions, Components and Consolidation. In *Proc. 11th In-*

ternational Workshop on Database and Expert Systems Applications (ASDM 2000 -International Workshop on Advanced Spatial Data Management), pp. 851–855, London. IEEE Press.

Peuquet, D. and Qian, L. (1997). An Integrated Database Design for Temporal GIS. In *Proc. 7th SDH*, pp. 21–31. Taylor and Francis.

Theodoridis, Y., Vazirgiannis, M., and Sellis, T. (1996). Spatio-Temporal Indexing for Large Multimedia Applications. In *Proc. 3rd IEEE Conference on Multimedia Computing and Systems*, pp. 441–448, Hiroshima, Japan.

Zhang, D., Tsotras, V., and Seeger, B. (2000). A Comparison of Indexed Temporal Joins. Technical report, Time Center.