Model-oriented Configuration Management for Relational Database Applications

Tien N. Nguyen
Electrical and Computer Engineering Department
Iowa State University
tien@iastate.edu

Abstract

The ability to manage the evolution of a database application is crucial for a successful database development process. However, existing software configuration management (SCM) approaches for databases in database management systems (DBMSs) are still limited to providing version control functionality for the rollback to a previous state of the database itself. Modifications made to design diagrams such as entity-relationship diagrams (ERDs) or to specifications and documentation are not well managed over time by an DBMS. Key limitations of existing versioning approaches for databases include their inadequacy in efficiently representing, storing, and connecting fine-grained, structural *changes to both design artifacts and data records. In this paper, we describe* McmDB*, an SCM-centered relational DBMS, that distinguishes itself from existing systems by its ability to manage design changes in an ERD and its schema constructs, and to maintain consistent configurations among those constructs and versions of data records. The logical connections among design specifications and elements of an ERD are versioned as well.*

1 Introduction

Database technology has been having a major impact on the growing use of software and playing a critical role in many areas of our daily life including business, medicine, education, law, engineering, etc. A computerized database is often created and maintained by a database management system (DBMS). DBMS is considered as a single or collection of software programs that facilitates the processes of defining, constructing, and manipulating databases for applications. To support developers in their process of designing a database application, an DBMS provides them with a conceptual representation of data, called a *data model*, that hides many of the details of physical storage in computer.

Entity-Relationship (ER) model is one of the most popular high-level conceptual data models. The ER model describes a database in term of entities, relationships, and attributes. The description of database is often referred to as *database schema*. During database design, developers use a set of graphical notations to specify database schemas. In the ER model, the most commonly used scheme of diagramming to aid in the visual representation of designs is *ER diagram* (ERD). Each object in an ER diagram is called a *schema construct*.

Conceptual database design is a dynamic process where design diagrams and their schemas are constantly in the evolution. The development of a large-scale database always produces a large number of different versions of designs. For example, an ER diagram could be changed due to the normalization procedure, transforming a design from one version to another that conforms to a certain normal form. However, the version and software configuration management supports (SCM) in many existing DBMSs are still limited to providing version control functionality for the rollback to a previous state of entire database or to a previous version of a schema. *Design changes* made to the structure of a design schema, to the attributes of its entities or relations, or to the number of properties and their data types in an entity are not well recorded over time by an DBMS. Developers often have to use *ad hoc* method such as memorization or paper notes to keep track of design evolution in a database. This leads to the reduction in their efficiency and causes errors in the development process.

In SCM tools, version control approaches for ER diagrams are too generic, thus, fail to capture changes at the logical level. Those existing SCM systems are heavily file-oriented and depend on a text line-oriented model of internal changes that disregards the logical structure of an ER diagram and schema constructs. Developers revise their software designs by modifying diagram entities. However, those existing file-oriented SCM systems are only able to display changes between versions of the diagram at the *textual representation* level.

Proceedings of The Sixth IEEE International Conference on Computer and Information Technology (CIT'06)
0-7695-2687-X/06 $20.00 © 2006 IEEE

The development process for a large database involves several developers. The development history may also be branched by different developers in a parallel development process. Later on, one may want that all changes performed on different branches of development will be incorporated to produce a common successor version. Recognizing who has made specific modifications to the common design diagram and what is the nature of those changes would greatly improve the development process. Unfortunately, DBMSs are lacking of those SCM collaborative supports such as branching versions of an ER design diagram, merging or differencing two versions at the model level, etc.

In this paper, we presents a *model-oriented* approach to managing design evolution in an ERD and its schema constructs, and to maintain their consistent configurations with versions of data records. The approach is based on a versioned data model with attribute tables that is capable of supporting multiple development paths with its tree-structured discrete time version model. To handle version control and SCM for complex artifacts in design diagrams of a database application, a structure versioning layer was built. The layer uses attributed, directed graphs as its internal representation for database artifacts.

We have realized this approach in a prototype of an SCM-centered relational DBMS, called *McmDB*, helping developers to manage changes at both design and data levels in a relational database application. A set of difference tools was built to show logical changes in ER diagram designs and their elements, and to show differences in the database itself at different versions. Requirement and design specifications are also versioned in a structural and fine-grained manner. Logical connections between specifications and design schema constructs in an ER diagram are represented as hyperlinks and recorded over time as well.

Section 2 describes our versioned data model. To support database objects in design schemas, a structure versioning mechanism based on attributed, directed graphs was developed (Section 3). Section 4 explains how database artifacts are represented and versioned in McmDB. Section 5 describes McmDB's SCM functionality. Section 6 discusses related work. Conclusions appear in the last section.

2 Versioned Data Model

Our goal is not only to provide version control for data records, but also to add SCM supports for the design artifacts produced during the development process of a database application. Since the database model is based on the tables of data records, McmDB's data model also uses a special data structure, called *attribute table*, to facilitate version control for data.

Figure 1 summarizes three main concepts in an attribute table: *node*, *slot*, and *attribute*. A node is the basic unit of

Figure 1. Data Model

identity. A node has no values of its own — it has only its unique identity. A slot is a location that can store a value of any data type, possibly a reference to a node, or to a sequence of nodes. A slot can exist in isolation but typically slots are attached to nodes, using an attribute. An attribute is a mapping from nodes to slots. An attribute may have particular slots for some nodes and map all other nodes to a default slot. All the slots of an attribute hold values of the same data type. Thus, in an attribute table, rows correspond to nodes and columns correspond to attributes (see Figure 1). The cells of an attribute table are slots. A slot may also exist in a *container*, an entity with identity and ordered slots. A container may be heterogeneous (a *record* in which each slot has its own type) or homogeneous (a *sequence* of slots of the same type). A sequence has identity and may be fixed or variable in size.

To accommodate the version control aspect of a database, we need to add version control to attribute tables. Once we add versioning, a table gets a third dimension: the version (see Figure 1). The version model used in McmDB is called *discrete time-based product versioning*. It is extended from the *project versioning* model used in Molhado [14], which was applied for source code version control in a software project. In McmDB, a *version* is global across entire database and is a point in a *tree-structured discrete time* abstraction. That is, the third dimension in the attribute table in Figure 1 is tree-structured to support variants and versions move discretely from one point to another. The version model is state-based, where each version is a first class entity that represents a database state. In our previous work [14], versioning algorithms for different primitive data types were developed.

The *current version* is the version designating the current state of a database application. Any version may be made *current*. Every time a versioned slot is assigned a (different) value, we get a new *temporary* version, branching off the current version. To the users, subsequent changes to a temporary version do not create a new temporary version.

That temporary version will only be recorded if a user explicitly requests that it be captured. In contrast, no version is created if a non-versioned slot is modified. To record the history of an individual data record or design object, the state of entire database is captured. Doing that is quite efficient because only changes at the slot level are recorded. In brief, using Molhado, McmDB knows how to correctly store and retrieve versioned slots that belong to a particular version point in a tree-structured discrete time line.

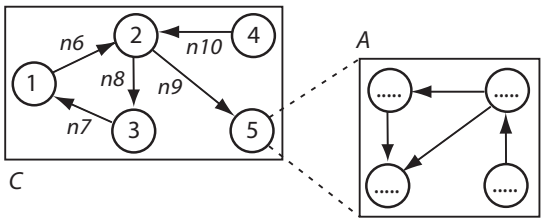
3 Versioning for Structures

The versioned data model that was presented in the previous section accommodates version control for *data records* in a database. However, each node in an attribute table is an individual, therefore, version and configuration management for an ERD and its schema objects, which have complex structures, could not be directly supported via that version model. A structure-oriented representation is designed to address this issue.

The structure-oriented representation model is based upon a special data structure, called *attributed, typed, nested*, and *directed graphs*. In this type of graph, each edge has exactly one source and one target node. A node or an edge has a unique identifier and can be associated with multiple attribute-value pairs. The domain of a value can be any data type. These typed attributes accommodate multiple properties associated with objects and relationships, depending on the interpretation given to nodes and edges. Our model also allows a directed graph to be nested within another in order to support composition among objects.

A structure-oriented versioning algorithm for this type of graph was developed to provide the fine-grained content change and version management. The algorithm takes advantage of Molhado's storage and versioning capabilities for versioned attribute tables. That is, an attributed, directed graph is represented as an attribute table. Graph nodes/edges are represented as rows, and attribute values as slots in the attribute table. To support nesting graphs, our *versioned reference* mechanism is used [14]. Common structures are shared among versions and fine-grained versioning is achieved for any object represented by a node. Using a directed graph for object's internal structure, any structure among direct sub-components of a composite can be modeled. This is a distinguished feature from existing SCM models, which support the composition of multiple sub-components but no relations between them can be defined. This type of graph also supports *document-centric* artifacts such as documentation since those artifacts can be regarded as DOM tree-structured documents [5].

Figure 2 shows an example of our graph-based representation. There are two graphs in Figure 2: the directed graph corresponding to a schema object A is nested within the directed graph corresponding to object C via node 5. The attribute table representing for object C is shown. Nodes "n1" to "n5" are "node" nodes (i.e. representing for a graph node) while nodes "n6" to "n10" are "edge" nodes (i.e. representing for an edge). Each "edge" node has "source" and "sink" slots. For example, "edge" node "n6" "connects" nodes "n1" and "n2". Each "node" node has a children slot. For example, "n2" has two outgoing edges ("n8" and "n9"). Node 5 has no outgoing edge, thus, the "children" slot of "n5" contains *null*. However, since it contains object A, the "ref" slot of node 5 refers to the object A. Note that a graph node inside A might have its "ref" slot referring to another atomic or composite object. The attribute table for object A is similar (not shown).



Attribute table for C

node	"type"	"source"	"sink"	"children"	"ref"	"attr1"	....
n1	*node*	*undef*	*undef*	[n6]	null	....	
n2	*node*	*undef*	*undef*	[n8,n9]	null	....	
n3	*node*	*undef*	*undef*	[n7]	null	....	
n4	*node*	*undef*	*undef*	[n10]	null	....	
n5	*node*	*undef*	*undef*	null	A	....	
n6	*edge*	n1	n2	*undef*	null	....	
n7	*edge*	n3	n1	*undef*	null	....	
n8	*edge*	n2	n3	*undef*	null	....	
n9	*edge*	n2	n5	*undef*	null	....	
n10	*edge*	n4	n2	*undef*	null	....	

Figure 2. Graph-based Representation

4 Representation for Artifacts

4.1 Entity and Relationship

To represent an entity in an ERD, a special type of object, named "Entity", is defined. Each "Entity" can be saved, loaded, and exists within the version space defined by the tree-structured discrete time-based version model described in Section 2. Each "Entity" carries an identifier that serves to identify it uniquely in entire database application.

Figure 3 illustrates the representation for the internal structure of an "Entity". Its internal structure is modeled as an attributed tree (an extended type of an attributed graph). The root of the tree has one or multiple children nodes, each of them represents an associated attribute of the entity (i.e. a field in the corresponding database *table*). Each field node

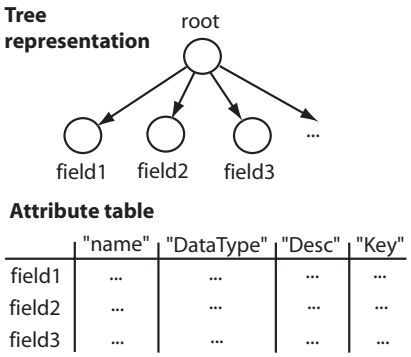


Figure 3. Representation for an Entity

is associated with additional attributes for the field such as "Name", "DataType", "Description", and "Key". The representation for a relationship and its attributes in an ERD is similarly. In addition, it also has an array of versioned slots, representing the cardinalities of the relationship in an ERD. Similar to an "Entity", a "Relationship" has its unique identifier and it can be stored and retrieved from disk. To support *nested relations* in the *nested relational model* [6], the nesting mechanism of directed graphs presented earlier is used, allowing composite attributes and complex tuples with a hierarchical structure. In this case, nested tree is sufficient for nesting structure representation.

Figure 4 partially shows the ER design diagram of a department store. There are four entities and three relationships. An attributed graph is used to represent the structure of an ER diagram. In this graph, each entity and relationship is represented by a node. The "ref" attribute defines for each node a reference to the corresponding "Entity" or "Relationship" object. Edges connect entities together to reflect the connections in the diagram. Attribute "type" is used to distinguish between an entity or a relationship node.

The database records are directly represented via attribute tables whose rows correspond to records, attributes correspond to fields, and slots correspond to data values. The system automatically maintains the consistency between the schemas and database tables.

4.2 Documentation

During the development and maintenance process of a database, it is very useful for developers to record documentation on the requirements and designs. McmDB has two mechanisms to support documentation: embedded (i.e. stored within the system) and hyperlinked documentation (i.e. might be outside of the system). First of all, McmDB supports embedded documentation in XML format. In McmDB, each XML document is considered to have a hierarchical internal structure, called *document tree*. Each tree node represents a logical unit at a structural level. A child

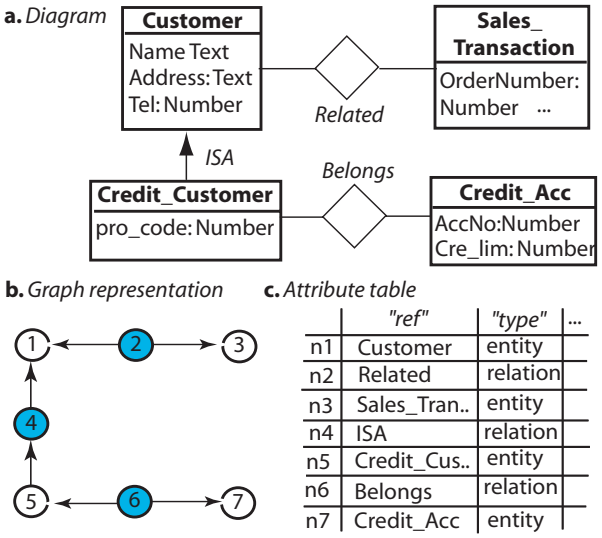


Figure 4. Representation for an ER diagram

node represents a sub unit of the parent unit. An attributed tree is sufficient to represent an XML document tree.

To support hyperlinked documentation, an additional attribute, "href" attribute, is defined for all nodes in any representation graph. The "href" values are assumed to be URLs for the target of a link, just as they are in an "HREF" attribute of an HTML anchor (i.e. an "A" element). They are realized as *versioned* slots, therefore, facilitating the management of dangling hyperlinks. In the case that a hyperlink linking to an internal object within McmDB, an "href" slot contains a reference to the object's representation node of the representation graph/tree. For example, users could create a hyperlink from a requirement or design item to an entity in an ERD, or from a data record to another.

5 User-level SCM Operations

This section describes user-level SCM functions in McmDB. A typical operational model is as follows. A user opens a version of an existing database project or create a new project. The selected version is now the current version. The ER diagram at the selected version is displayed in the ER diagram editor. Using the editor, the user can manipulate the ER diagram to create a new version. To edit the fields of a table, the user clicks on an entity, and a window is displayed to show the table's structure. The user can view the list of database tables at the selected version. No form, page, or query mechanisms are supported yet.

If any modification is made to the versioned objects at the current version, a new version would be temporarily created, branching off the current version. Subsequent modifications will not change the temporary status of that version until a *capture* command is issued. If the user does

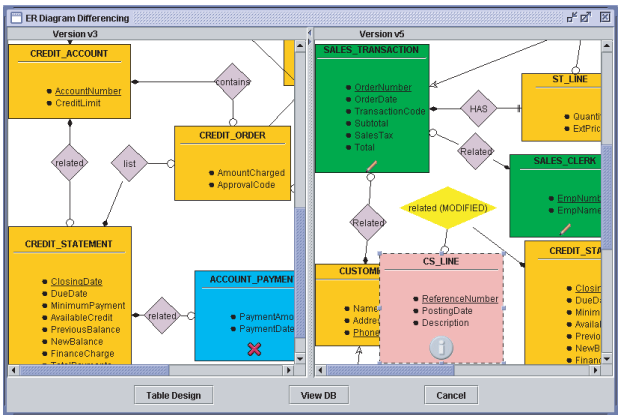


Figure 5. Differencing for an ER diagram

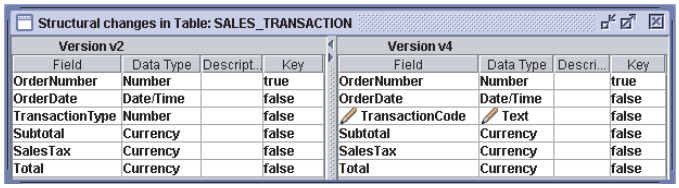


Figure 6. Changes in Table Schema

not want to keep the temporary version, he/she can *discard* it. Otherwise, the user can *capture* the state of the database and related artifacts including ER diagram objects and documentation at a version. The *capture* command changes a temporary version into a captured one. A unique name as well as date, authors, and descriptions can be attached to the newly captured version for later retrieval.

A set of difference tools was built, allowing developers to compare different versions of an ER diagram, a table schema, or a table of data records. Figure 5 shows structural changes between two versions of an ER design diagram. The icons attached to elements show the nature of the changes. For example, between *v3* and *v5*, entity "ACCOUNT_PAYMENT" was deleted, entity "CS_LINE" was inserted, entities "SALE_TRANSACTION" and "SALE_CLERK" have been modified in their attributes, and the relationship "related" was also changed. If the user clicks on a modified entity, he/she could see the differences in the corresponding table schema (see Figure 6). For example, in Figure 6, between *v2* and *v4*, the name and the type of the field "TransactionType" were changed into "TransactionCode" and "Text", respectively. Furthermore, since McmDB provides fine-grained version control for data, it can also display modifications at the data level (see Figure 7). For example, from *v5* to *v6*, two records were deleted (having "X" icons), one record was inserted (having "i" icons), and some data values were modified (having "pencil" icons).

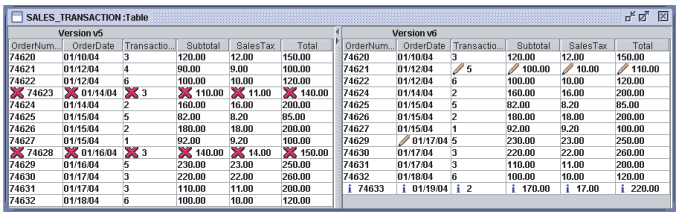


Figure 7. Changes at Data Level

6 Related Work

One of the areas that draw substantial research in database is *temporal database* [15]. A temporal database is considered to contain time-varying data. Research in temporal databases aims at providing application-independent DBMS support for time-varying information [8]. This line of research has been focusing on providing better temporal data semantics and better temporal query capabilities. Time has been added into many data models to create temporal data models such as temporal entity-relationship model [1], semantic data model [7], and knowledge-based data model [19]. Support for time in databases could be at the level of user-defined time (i.e. attribute values drawn from the temporal domain), valid time, or transaction time (e.g. event timestamps or interval timestamps). Timestamp-based data models used in DBMSs include timestamped attribute values, timestamped groups of attributes, timestamped tuples, etc. To operate on those temporal data models, several temporal, relational query languages [4, 9] have been proposed such as HSQL [17], TSQL [13], etc.

Our data model described in Section 2 is similar in spirit to a temporal data model. It is based on *versioned* attribute tables. The time dimension in our data model is logical, discrete, and at the user-defined level. Moreover, our version model supports not only time-dimensional versions as in existing temporal databases, but also different *variants* of design artifacts that may be simultaneously updated by multiple developers during a development process. Also, existing temporal database models lack of structural versioning and SCM supports for complex design diagrams.

Schema versioning is also a related research area dealing with the use of multiple heterogeneous schemata for various database related tasks [16]. *Schema modification* is accommodated when an DBMS allows changes to the schema definition of a populated database. *Schema evolution* is accommodated when a database system facilitates the modification of the database schema without loss of existing data.

A number of temporal algebras have been proposed to extend the static relational model to support schema evolution [11, 20]. Several other valid-time and temporal algebras were extended to deal explicitly with schema evolution [9, 12]. To accommodate schema versioning at the

relation or entity level, several researchers suggested methods to establish a set of invariants to ensure the semantic integrity of the schema and a set of rules or primitives for effecting the schema changes [2, 3, 10]. Some researchers suggested a set of atomic operations within the relational model to guarantee a consistent and possibly reversible database structure [18]. Other researchers use a transaction-time meta-relation approach, in which previous schemas may be constructed through temporal rollback of the meta-relations using a taxonomy for schema evolution [16].

Comparing to our approach, schema evolution does not imply full historical support for the schema; only the ability to change the schema definition without loss of data. Schema modifications will not necessarily result in a new version. Typically schema changes will be of a finer grain than the definable versions. Schema versioning requires that a history of changes be maintained to enable the retention of past schema definitions. However, schema versioning capability is often limited to the version control support for the structure of database schema constructs. Existing schema versioning supports in an DBMS did not include structure versioning for the structure of ER diagrams, consistent configuration management for design objects and specifications, and collaborative/concurrent supports such as workspace management, logical differencing and merging functionality for different versions of design diagrams.

7 Conclusions

Managing the evolution of a database application is crucial for a successful database development process. Version and configuration management for a database in DBMSs should not be limited to providing version control for database itself or for database schemas as in existing approaches. Instead, a complete development history of a database should be captured in order to give developers full understanding of the development process. The research work presented in this paper addresses this issue.

Our main contribution is the *model-oriented* SCM approach in *McmDB*, an SCM-centered relational DBMS, which manages the evolution of a database system including design evolution in an ERD and its schema constructs, and consistent configurations at both design and data levels. The approach is based on a versioned data model with attribute tables that is capable of supporting multiple development paths with its tree-structured discrete time version model. To handle SCM for complex artifacts in design diagrams of a database application, a graph-based structure versioning mechanism was developed. McmDB is not only able to retrieve the right values of data records at a particular version, but also to correctly determine the proper contents of design artifacts such as entities/relationships in an ERD and associated documentation.

References

[1] A. Ait-Braham, B. Theodoulidis, and G. Karvelis. Conceptual modelling and manipulation of temporal databases. In *Proceedings of the 13th Int. Conference on Entity Relationship Approach*, pages 296–313. Springer Verlag, 1994.

[2] J. Banerjee, H. Chou, J. Kim, and H. Korth. Schema evolution in object-oriented persistent databases. In *Proc. of the 6th Advanced Database Symposium*, pages 23–31, 1986.

[3] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. Williams, and M. Williams. The gemstone data management system. *Object-oriented Concepts, Databases and Applications*, pages 283–308, 1989.

[4] J. Chomicki. Temporal query languages: a survey. In *Temporal Logic: ICTL'94*, volume 827, pages 506–534. Springer-Verlag, 1994.

[5] Document Object Model. http://www.w3.org/dom/.

[6] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 1994.

[7] M. Hammer and D. McLeod. The semantic data model: a modeling mechanism for database applications. In *SIGMOD'78: Proceedings of the 1978 ACM SIGMOD International Conference on Management of Data*, pages 26–36. ACM Press, 1978.

[8] C. S. Jensen and R. T. Snodgrass. The TSQL2 data model. *TLQ2 Commentary*, September 1994.

[9] J. L. Edwin McKenzie and R. T. Snodgrass. Evaluation of relational algebras incorporating the time dimension in databases. *ACM Computing Surveys*, 23(4):501–543, 1991.

[10] B. Lerner and A. Habermann. Beyond schema evolution to database reorganisation. *SIGPLAN Notice*, 25(10):67–76, 1990.

[11] N. Lorentzos and R. Johnson. Extending relational algebra to manipulate temporal data. *Information Science*, 13(3):289–296, 1988.

[12] L. McKenzie and R. Snodgrass. Schema evolution and relational algebra. *Information System*, 15(2):207–232, 1990.

[13] S. Navathe and R. Ahmed. A temporal relational and query language. *Information Science*, 49(2):147–175, 1989.

[14] T. N. Nguyen, E. V. Munson, J. T. Boyland, and C. Thao. An Infrastructure for Development of Object-Oriented, Multi-level Configuration Management Services. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 215–224. ACM Press, 2005.

[15] G. Ozsoyoglu and R. T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, 1995.

[16] J. Roddick. A Survey of Schema Versioning Issues for Database Systems. *Information and Software Technology*, 37(7):383–393, 1996.

[17] N. L. Sarda. Extensions to SQL for historical databases. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):220–230, 1990.

[18] B. Shneiderman and G. Thomas. An architecture for automatic relational database system conversion. *ACM Transactions on Database Systems*, 7(2):235–257, 1982.

[19] R. T. Snodgrass. Temporal databases. In *Theories and Methods in Spatio-Temporal in Geographic Space (639)*. Springer Verlag, 1992.

[20] A. Tuzhilin and J. Clifford. Temporal relational algebra as a basis for temporal relational completeness. In *Proceedings of the 16th International Conference on Very Large Databases*, pages 13–23. Morgan Kaufmann, 1990.