시간 지원 데이타 베이스 관리 시험대

김 동 호† 전 근 환† 정 경 자† 김 기 중† 류 근 호††

요 약

시간 지원 데이타베이스 관리 시험대는 유효 시간과 수록 시간을 지원한다. 본 논문에서는 시간 지원 데이타 베이스 관리 시스템 시험대의 설계와 구현을 연구한다. 시험대는 구문 분석기, 의미 분석기, 코드 생성기 및 실행기로서 구성된다. 구문 분석기는 시간 지원 질의어로부터 파스 트리를 생성한다. 그리고 의미 분석기는 시스템 카탈로그를 이용하여 시간 지원 질의어의 의미와 정당성을 점검한다. 코드 생성기는 Update network와 같은 실행 트리를 생성하는데 실행 트리를 위하여 뷰 증진 형성 방법을 사용하였다. 마지막으로 인덱싱 구조와 동시성 제어에 대하여 토의하였다.

Temporal Database Management Testbed

Dong Ho KIM†, Keun Whan JEON†, Kyung Ja JEONG†,
Kee Jung KIM† and Keun Ho RYU††

ABSTRACT

The Temporal Database Management Testbed supports valid and transaction time. In this paper, we discuss the design and implementation of a testbed of a temporal database management system in main memory. The testbed consists of a syntactic analyzer, a semantic analyzer, a code generator, and an interpreter.

The syntactic analyzer builds a parse tree from a temporal query. The semantic analyzer then checks it for correctness against the system catalog. The code generator builds an execution tree termed an update network.We employ an incremental view materialization for the execution tree. After building the execution tree, the interpreter activates each node of the execution tree. Also, the indexing structure and the concurrency control are discussed in the testbed.

1. Introduction

Conventional database management systems (DBMS) cannot provide for store and access of time-varying data, because conventional DBMS only support up-to-date information. In some applications it is not appropriate to discard old information. Therefore, temporal database management systems (TDBMS) support time to override those drawbacks.

Time can be classified as *valid time*, *transaction time*, and *user defined time*[SA86]. The valid time is a time at which an event happens in the real world, while the transaction time is a time at which it is recorded in the database. Depending on supported time, four different types of relations may result: *snapshot*, *rollback*, *historical*, and *temporal*. Each one of these can be associated with a class of time. A snapshot relation only supports current time. A rollback relation supports

· This work was supported in part by the Ministry of Trade Industry & Energy, in part by the Korea Science and Engineering Foundation, KOSEF 931-0900-067-2 and in part by NSF grant IRI-8902707.

†준 회 원 : 충북대학교 컴퓨터과학과
††정 회 원 : 충북대학교 컴퓨터과학과 부교수

논문접수 : 1994년 2월 21일, 심사완료 : 1994년 4월 25일

transaction time and a historical relation supports valid time without rollback. Lastly, a temporal relation supports all of the above, valid time and transaction time. Therefore, a TDBMS should support all four types of database relations and two types of times.

In this paper, we discuss the design and implementation of a TDBMS testbed in main memory (termed simply the testbed). The testbed supports the temporal query language TQuel[Snod87]. The basic idea of the temporal database system was from McKenzie [McK88]. The testbed employs incremental query evaluation to implement the main memory resident data. To focus on this aspect we have not yet implemented support for secondary storage; all data resides in main memory. The availability of inexpensive, large main memories coupled with the need for faster response time bring a new perspective to database technology[Bit86]. The study of main memory databases has been active for several years [AHK85,DKO+84,Eic88,Hag86, LC87, LS89], but they have been studied in conventional databases. We design and implement a temporal database management system in main memory, which is named testbed.

The testbed consists of a syntactic analyzer, a semantic analyzer, a code generator, and an interpreter. The syntactic analyzer parses a temporal query and builds a parse tree including time operator. The semantic analyzer then checks it for correctness against the system catalog. The code generator builds an execution tree for time operator. Finally, the interpreter activates each node of the execution tree and then each operators with time factor are executed, and produces a result for the temporal query.

In Section 2, an overview of temporal queries and their examples are given.
Then, in Section 3, the testbed is described. Section 4 discusses the syntactic analyzer and the semantic analyzer. Sections 5 and 6 discuss the code generator as well as the interpreter. Section 7 discusses the index structure and the concurrency control for the testbed. Finally, we discuss future work in the last section.

2. Temporal Queries: an Overview

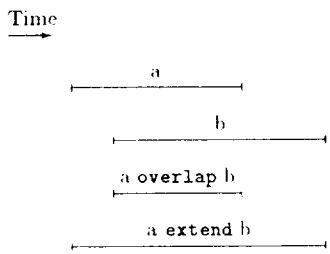
2.1 An Overview

We first review the primary constructs from TQuel[Sno87] which is a strict superset of Quel, the query language for Ingres [SWKH76].

TQuel queries may contain valid and when clauses. The valid clause specifies the value of the implicit valid time attribute. The valid at clause indicates a single time in the temporal attribute while the valid interval clause defines an interval of time as *valid from ... to ...* in the temporal attribute.

The when clause is the temporal predicate analogue to Quel's where clause. The temporal predicate that follows the keyword defines a boolean value. As the where clause defines constraints in terms of the contents of the database relation, the when clause defines time constraints that should be met.

The temporal operators, such as overlap, precede, extend, and equal, can determine two time values defining an interval while the operators begin of and end of each determines a single time. For example, the predicates **(a overlap b)** and **(a extend b)** are shown in Figure 1.



(Fig. 1) Example of the overlap and the extend

The query, "list the associate professors in 1988 and their year of promotion" may be expressed as follows.

```
range of f is Faculty
  retrieve (name=f.name, rank=f.
  rank, salary=f.salary)
  valid from begin of f to end of f
  where f.rank = 'associate'
  when f overlap '1988'
```

This example illustrates the valid clause and the overlap operation in the when clause.

2.2 Temporal Relation

To extract information out of the temporal database, the retrieve, append, and delete query commands are used. Queries in the testbed have time predicates as well as the conventional predicates in the where clause.

The examples are executed in the testbed. The examples in this section use a temporal relation *Faculty* shown below. The relation *Faculty* has 3 attributes of which are explicit; *name, rank, and salary.*

<Table 1> Faculty(name, rank, salary)

name	*rank*	*salary*	*vf*	*vt*	*ts*	*te*
Jane	assistant	25,000	1971	1976	2	∞
Jane	associate	35,000	1976	1981	3	∞
Jane	full	55,000	1981	∞	4	∞
Fransisco	assistant	30,000	1983	1987	5	∞
Fransisco	associate	40,000	1987	∞	7	∞
Daniel	associate	40,000	1988	∞	6	∞

The attributes *vf* and *vt* express the start and end of the valid time. Similarly, the attributes *ts* and *te* express the start and end of the transaction time. In 1971, Jane joined the department as an assistant professor. After that, she was promoted from assistant professor to associate professor in 1976 and from associate professor to full professor in 1981. This example uses transaction number to represent the transaction time.

We show examples to generate the above relation, **Faculty.**

2.3 Creating a Database

The create command is used to create new temporal relations in the testbed. For example, the create command,

```
create persistent interval Faculty
  (name is char, rank is char, salary
  is int)
```

creates a relation with the following schema.

<Table 2> Faculty Schema

name	*rank*	*salary*	*vf*	*vt*	*ts*	*te*

2.4 Adding Tuples

To add tuples to a relation, the append command is used. The append statement can be used in two different formats, depending on if a new relation is being created or not. The append statement uses the valid and when clauses except for adding new tuples. For example, let's assume that a new tuple is added to a new relation as the following query,

```
append to Faculty
  (name = 'Jane', rank = 'assistant',
  salary = 25000)
  valid from begin of '1971' to end
  of '1975'
```

then, a new tuple in the new relation, **Faculty,** is added above.

<Table 3> Adding a tuple

name	*rank*	*salary*	*vf*	*vt*	*ts*	*te*
Jane	assistant	25,000	1971	1976	2	∞

The append command is able to be used with valid and when clauses as shown in the following example. Note that the valid time of the referenced tuple is specified. The query appends new tuples where existing tuples have "Jane" as the value of their name attribute.

```
range of f is Faculty
append to Faculty
  (name = f.name, rank = 'associ-
  ate', salary = 35000)
  valid from begin of '1976' to end
  of '1980'
  where f.name = 'Jane'
  when f overlap '1975'
```

The result is

<Table 4> Result from appending a tuples

name	*rank*	*salary*	*vf*	*vt*	*ts*	*te*
Jane	assistant	25,000	1971	1976	2	∞
Jane	associate	35,000	1976	1981	3	∞

This statement uses the valid and when clauses, when there is only one tuple variable, *f* which is associated with an interval relation. The valid time is effective depending on the valid clause in the temporal query. I. e., the beginning of the year is 1971 and the end of the year is 1976.

The when clause defines the set of tuples that are valid at the time the year is 1975. The relation *Faculty* of Section 2.2 is created using similar append statements.

2.5 Retrieving Tuples

To retrieve tuples from a relation, the retrieve command is used. This command only displays the tuples retrieved. The example query in Section 2.1 generates the following relation.

<Table 5> Result from retrieving tuples

name	*rank*	*salary*	*vf*	*vt*
Fransisco	associate	40,000	1987	∞
Daniel	associate	40,000	1988	∞

This example uses the valid and when clauses. The valid time clause uses the tuple variable **f** that ranges over each of the tuples of the relation **Faculty**. The when clause defines the set of tuples that are valid at the time the year is **1988**.

2.6 Deleting Tuples

To delete tuples in a relation, the delete command is used. This command does not delete the tuples. Instead adds a new tuple to the existing relation because of the *non-deletion* semantics of transaction time. For example, let's delete the tuple which have name attribute "Fransisco".

```
range of f is Faculty
delete f
  valid from begin of '1987' to end
  of '1987'
  where f.name = 'Fransisco'
```

when f overlap '1987'

Then, the result from the above query is shown below.

<Table 6> Result from deleting tuple

name	*rank*	*salary*	*vf*	*vt*	*ts*	*te*
Fransisco	assistant	30,000	1983	1987	5	∞
Fransisco	associate	40,000	1987	∞	7	8
Fransisco	associate	40,000	1988	∞	8	∞

In this example, a new tuple which has a valid time **1988** is appended with the number of transaction start, **8** and the number of transaction end, **8** is inserted in the deleting tuple instead of deleting the tuple which have name attribute "Fransisco".

2.7 Saving the Results

The retrieve statement normally displays its result. In order to save them for later use, for example, the following retrieve statement is used. It uses the **into** clause.

```
range of f is Faculty
retrieve into newFaculty (Name = f.
  name, Rank = f.rank, Salary = f.
  salary) valid from begin of '1990'
  to end of '1990'
  where f.name = 'Daniel'
  when f overlap '1990'
```

The relation, **newFaculty,** is created, and the result of the above query is stored in the relation.

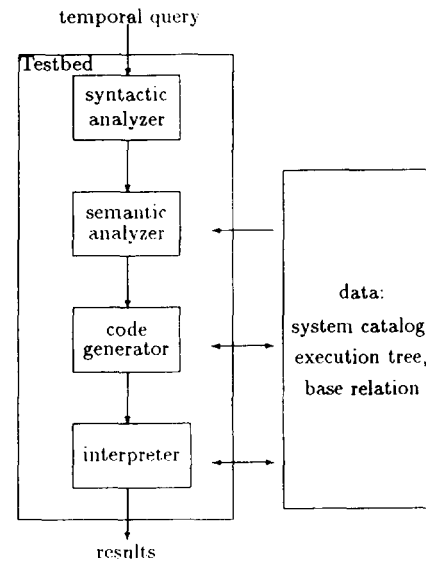
<Table 7> Result from saving tuples newFaculty(Name,Rank,Sarary)

Name	*Rank*	*Salary*	*vf*	*vt*	*ts*	*te*
Daniel	assistant	40,000	1990	1991	9	∞

3. The Testbed

3.1 Overview of the Structure

The testbed consists of a syntactic analyzer, a semantic analyzer, a code generator, and an interpreter, as shown in Figure 2.



(Fig. 2) The Testbed Architecture

The syntactic analyzer builds a parse tree for a temporal query, and the parse tree is then checked for correctness against the system catalog by the semantic analyzer, as in conventional query processing except for time operators. If the output data from the semantic analyzer is semantically correct, the parse tree is then passed to the code generator.

We use the parse tree for the view definitions to be mapped onto an acyclic graph of processing nodes[HT86, Rou82b, RK86, Sno82], which we refer to as an execution tree, that is built by the code generator.

Finally, the interpreter evaluates the query execution plans in batch fashion.

3.2 Incremental View Materialization

A base relation is an autonomous, named

relation stored in the database, which is not defined in terms of other relations[Dat86]. In contrast, a view is a named relation that is defined in terms of other relations, either base relations or other views. Therefore base relations are stored in the database; views may, but need not, be stored in the database.

We choose incremental view materialization, because incremental view materialization is more efficient than either query modification or recomputed view materialization if the following conditions are satisfied simultaneously: (1) the number of queries against a view is sufficiently larger than the number of updates to its underlying relations, (2) the sizes of the underlying relations are sufficiently large, (3) the selectivity factor of the view predicate is sufficiently low, (4) the percentage of the view retrieved by queries is sufficiently high, and (5) the volatility of the underlying relations, defined as the percentage of tuples that change between accesses to the view, is sufficiently low. Much work has been done on view materialization policies [BLT86, Han87a,Han87b, Hor85,HT86,Rou82a,Rou82b, Sno82] .

Horwitz[Hor85], Roussopoulos[Rou82b], and Snodgrass[Sno82] have both proposed that a view definition be mapped onto an acyclic graph of processing nodes, which Snodgrass refers to as the view's update network. The update network has the form of a parse tree.

Similarly, we have a concept of an execution tree with temporal function as an update network. This incremental expression evaluation by using the execution tree differs fundamentally from that of non-incremental expression evaluation. First, the execution tree, unlike the parse tree, is persistent. It is built when a view is defined, activated each time one of the view's underlying relations is changed, and destroyed only when the view itself is deleted from the database. Second, operator nodes may have their own local memory and procedures. For example, immediate results from one activation of the tree may be cached in operator nodes for use in the next activation of the tree. Third, the input to, and the output from, the execution tree is defined by differentials rather than relation states.

3.3 Operator Nodes

The following nodes may appear in an execution tree: cartesian, select, derivation, project, store, and display. The cartesian node, the select node, and the project node are similar to conventional nodes, but the derivation node, the store node, and the display node are different. The cartesian node computes the cartesian product from existing which is relations or differential of view. The select node selects tuples satisfying specified predicate from its ancestor node, but if there is a single relation, the select is performed on existing relation. The derivation node computes the valid clause and the when clause. The project node projects from its ancestor node depending on target elements. Finally, either the store node or the display node is built. The store node stores final result and the display node displays final results.

3.4 Execution Tree

The execution tree enables incremental view materialization. An execution tree is

built by the code generator. After building the execution tree, its nodes are activated by the interpreter. These procedures are applied to all nodes: cartesian product, select, derivation, project, store, and display.

4. Syntactic Analyzer and Semantic Analyzer

The syntactic analyzer first reads a temporal query and then separates it into tokens such as temporal constants, variable names, keywords. Finally, the parse tree is built from the temporal query.

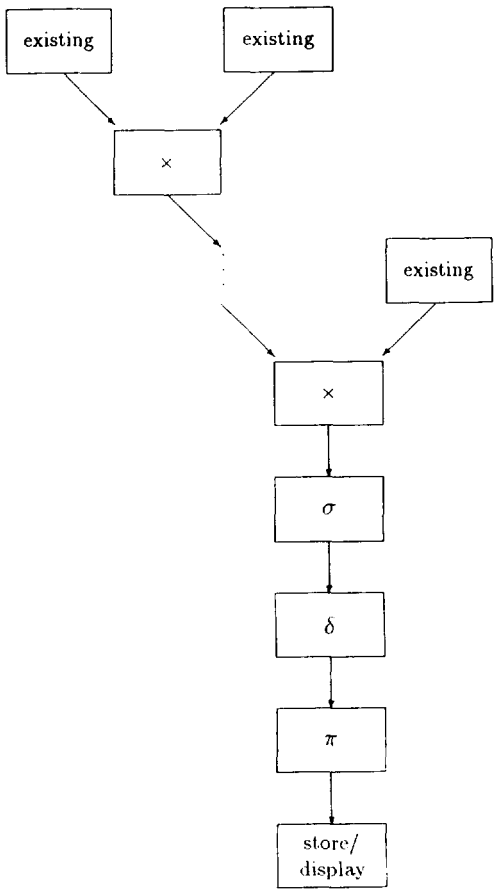
The semantic analyzer not only determines the semantics of the temporal query, but also passes it to the code generator if the parse tree is semantically correct. The semantic analyzer is similar to a conventional one except for the processing of temporal expressions. The semantic analyzer checks the correctness of the query with the system catalog.

The syntactic analyzer is implemented using the lex and yacc. We choose the data specification language IDL (Interface Description Language)[Sno89] for the data structure of the testbed, because IDL not only provides adequate functionality for describing complex data structures, but also can be automatically mapped into C code by the IDL translator. The data structure of the parse tree constructed by IDL is described elsewhere [Ryu91].

5. Code Generator

The code generator builds an execution tree which consists of nodes depending on the query command, as shown Fig. 3.

The terminal nodes of the execution tree, which may be shared among several trees,



(Fig. 3) The execution tree of the testbed

represent the existing relations. Each root node of these trees represents a derived relation, the result of a specific query. All nodes represent a single algebraic operator, such as cartesian, select, derivation, project, performed to support temporal query.

Fig. 4 is a summary of the code generator.

All nodes of the execution tree except for the store and display node are built for all query commands. The display node is built for retrieve statements with no *into* or *to* clause, but the store node is not. Therefore, the display node is only for displaying attributes. Each node contains pointers to their

descendents and information about each of their descendents in the execution tree. Also, operator nodes in the execution tree contain data structures for holding their input relation state(s). The relationships between nodes with pointers to descendents and information in terms of these descendents are represented by an acyclic graph.

```
codegeneration
{
    /* Add all tuple variables reference in the current
    retrieval statement being processed to the list of
    active tuples. */
    find_tuple_variables(*currentStmt.thetupleVars);
    /* Determine the number of active tuples in this
    retrieval statement. If only one tuple variable is
    active, no cartesian operator nodes are needed;
    otherwise, a cartesian operator binary sub_tree
    must be constructed. */
    totaltupleVars = 0;
    foreachinSETtupleVarRef((*thetupleVars)->
        active_tupleVars, SAtupleVarRef, AtupleVarRef)
    {
        totaltupleVars = totaltupleVars + 1;
    }
    if (totaltupleVars == 1)
    {
        build_existing
    }
    else
    {
        build_cartesian_subtree   /* build cartesian node
                                  for two or more existing
                                  relation */
    }
    build_select_node
    build_derivation_node         /* build derivation node
                                  for a valid and when
                                  clause */
    build_project_node            /* build project node for
                                  extracting attributes */
    if ((typeof(*currentStmt) == KvoidRetrieve)
        || (typeof(*currentStmt) == KuniqueRetrieve))
    {
        build_display_node
    }
    else
    {
        build_store_node
    }
}
```

(Fig. 4) The structure of the code generator

An existing node which is a relation or a differential has not only a relation extracted from the database, but also information in terms of descendents. If there is a single relation, a descendent points to select node, otherwise it points to cartesian node.

If two or more relations exist, a cartesian node is created. The cartesian node will have information in terms of the tuples and a pointer to its descendent node. A descendent of the cartesian node contains a select node. Similarly, the select node, derivation node, and project node each has information in terms of its descendent. If a single relation exists, a select node is built for executing select operator from existing relations because a cartesian node does not need to be created. Whereas, if existing relations are two or more, one or more cartesian subtrees are constructed for all existing relations, and the select node is built for extracting results from cartesian node. Other nodes, i.e., select, derivation, project, are built for all commands.

6. Interpreter

The interpreter processes tuples from one or more relations from a restricted subset of temporal queries. A forest of execution trees for representing the temporal query is a data structure built by the code generator. Input to the interpreter is the set of tuples to be processed and the execution tree representing one or more queries.

The testbed supports the temporal retrieve statements such as void retrieve, retrieve with an *unique* clause, and retrieve with an *into* and *to* clause, without aggregates. The retrieve statement does not change contents of tuples but only display query results. The testbed also supports versions of the append

and delete statements. On each change to a base relation, the interpreter activates the execution tree, where ancestor nodes in the execution tree cause the activation to be propagated to its descendent nodes. A command which be argumented by nodes consist of update specification and relation. The update specification has a before image and an after image of tuples. These form differentials of the existing relation.

For each operator node in a path of execution tree, the interpreter processes as follows.

- Visit each algebraic operator node that is a descendent of the existing relation nodes, and then add to each visited node the addresses of the procedures that will be used for algebraic operator processing.
- Find the existing relation node associated with the tuple just input.
- Pass the tuple just created to all descendent nodes of the existing relation node associated with the node just executed.

After this procedure, the result produced by the interpreter may be saved into the new relation or displayed.

The existing relation is extracted from a database relation. One or more existing relations can exist depending on the number of tuple variables, but the same existing relations are not reconstructed. Therefore, the interpreter may make a differential from the existing relation. Cartesian product operations are computed for two or more existing relations, and then new tuples are produced. The select operation then selects tuples from the new relation by using where clause predicates, and then the derivation operation selects according to when and valid clause temporal predicates. Extracting one or more targets are evaluated by the project operation, and at this point, all operations are completed, except for the store and display operation for storing and displaying the information.

7. Index Structure and Concurrency Control

Now we describe the index structure and the concurrency control working on going job. The testbed should manage memory resident data, and the different access method, commit processing, data structure, and concurrency control and recovery should be considered.

The temporal database have been designed segment index and SR tree[TCG+93], the time index and the monotoic B+ tree [EWK90], and proposed several different storage structure[Ahn86]. A wide variety of index structure has been proposed for main memory database [LC86,DLO+84,WK90]. However, all they did not give any answer for the main memory of the temporal database. They have only given the one side of the answer.

We are designing two different index structures in the testbed. One is for current data, which is that data values on which the index is built need not be stored in the index itself, as is done in like B trees. Another is for historical data, which is that data values on which the index is built from historical data. The reason is that the current data should be accessed frequently, but the historical data should be not.

The concurrency control should be considered non two phase locking protocol because

the temporal databases have non-deletion policy which means that all transactions can not delete their record, and the concurrency control should be simplified from conventional one, although they are main memory database.

8. Summary and Future Work

The TDBMS testbed in main memory supports valid and transaction time. The testbed consisted of a syntactic analyzer, a semantic analyzer, a code generator, and an interpreter. The syntactic analyzer builds a parse tree for a temporal query and the semantic analyzer then checks correctness of the parse tree against the system catalog. The code generator builds an execution tree termed an update network as an acyclic inverted tree. We defined the following operators: a cartesian node, a select node, a derivation node, a project node, a store node, and a display node. The execution tree contains these nodes and uses for incremental view materialization. After building the execution tree, the interpreter activates each node of the execution tree according to the select, valid, and when clause in the temporal query. The procedure for executing the interpreter is: first, read the execution tree which is a forest of inverted algebraic operators and second, visit each algebraic operator node that is a descendent of the existing relation nodes, and then add to each visited node the addresses of the procedures that is used for algebraic operator processing. Third, find the existing relation node associated with the tuple input and finally pass the tuple just created to all descendent nodes of the existing relation node associated with the node just executed.

According to the command, the results evaluated are stored in the store or display node. Also, example queries were given to show the execution of the testbed.

The interpreter should be argumented to activate the execution tree of the database whenever a base relation should be changed. Also, intermediate relation states for nodes in the tree are stored between activations of the tree. If the definition of an incrementally maintained materialized view is mapped onto execution trees, these view execution trees are integrated into a single execution tree for database. These features with the view definition command will be included in future work.

We did not consider an archival manager using high capacity WORM technology, for the TDBMS testbed. In the future, we will condider WORM storage like optical disk for historical data. The interaction between a cache manager and an asynchronous archive manager in new storage structures will be also studied in the future.

9. Acknowledgements

The initial system was implemented by Edwin McKenzie and Vikram Debashish. The author wishes to thank Richard Snodgrass for his valuable suggestions and encouragement. Dr. Richard Snodgrass is a leader of TempIS project. The part of this paper was written while the author was working as a member of TempIS project at Department of Computer Science of the University of Arizona.

References

[AHK85] A. Ammann, M. Hanrahan, and R.

Krishnamurty. Design of a main resident DBMS. In *Compcon*, pages 54-57, 1985.

[Ahn86] I.Ahn. Performance modeling and access methods for temporal database management systems. Ph.D Thesis, Computer Science Dept., Univ. of North Carolina at Chapel Hill, Aug. 1986.

[Bit86] D. Bitton. The effect of large main memory on database systems. In Proc. *of ACM SIGMOD International Conference on Management of Data*, May 1986.

[BLT86] J. A. Blakeley, P. A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proc. of ACM SIGMOD International Conference on Management of Data*, May 1986.

[Dat86] C. J. Date. *Relational Database: Selected Writings*. MA: Addison-Wesley, 1986.

[DKO+84] D.J. DeWitt, R.H. Katz, F. Olken, L.D. Shapiro, M.R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 1-8, Jun. 1984.

[Eic88] Margaret H Eich. Main memory database research directions. Technical Report TR 88-CSE-35, Computer Science and Eng. Dept., Southern Methodist Univ., Nov. 1988.

[EWK90] R. Elmasri, G. Wuu, and Y. Kim. The time index:an access structure for temporal data. In *Proc. of the Conference on Very Large Database*, Aug. 1990.

[Hag86] R.B. Hagmann. A crash recovery scheme for a memory-resident database system. *IEEE Trans. on Computer*, 35 (9):839-843, Sep. 1986.

[Han87a] E. N. Hanson. Efficient support for rules and derived objects in relational database systems. Ph.D Thesis, Computer Science Dept., Univ. of California, Berkeley, 1987.

[Han87b] E. N. Hanson. A performance analysis of view materialization strategies. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 440-453, May 1987.

[Hor85] S. B. Horwitz. Generating language-based editors: A relationally-attributed approach. Ph.D Thesis, Computer Science Dept., Cornell Univ., Aug. 1985.

[HT86] S. B. Horwitz and T. Teitelbaum. Generating editing environment based on relations and attributes. *ACM TOPLAS*, 8(4):577-608, Oct. 1986.

[Jen90] C.S. Jensen. Towards the realization of transaction time database systems. Ph.D Thesis, Computer Science Dept., Univ. of Maryland at College Park, Dec. 1990.

[LC86] T. J. Lehman and M.J. Carey. A study of index structures for main memory database management systems. In *Proc. of the Conference on Very Large Database*, pages 294-303, Aug. 1986.

[LC87] T. J. Lehman and M.J. Carey. A recovery algorithm for a high-performance memory-resident database system. In *Proc. of ACM SIGMOD International Conferenceon Management of Data*, pages 104-117, May 1987.

[LHM+86] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A snapshot differential refresh algorithm. *In Proc. of ACM SIGMOD International Conference on Management of Data*, pages 50-60, May 1986.

[LS89] E. Levy and A. Silberschatz. Log-driven backups: A recovery scheme for large memory database system. Research Report TR-89-24, Computer Science Dept., Univ. of Texas at Austin, Sep. 1989.

[McK88] E. McKenzie. An algebraic language for query and update of temporal databases. Ph.D Thesis, Computer Science Dept., Univ. of North Carolina at Chapel Hill, Sep. 1988.

[MS89] E. McKenzie and R. Snodgrass. An evolution of algebras incorporating time. Technical Report TR-89-22, Computer Science Dept., University of Arizona, Sep. 1989.

[RK86] N. Roussopoulos and H. Kang. Principles and techniques in the design of ADMS. *IEEE Computer*, 19(12):19-25, Dec. 1986.

[Rou82a] N. Roussopoulos. The logical access path schema of a database. *IEEE Tran. on Soft. Eng.*, 8(6):563-573, Nov. 1982.

[Rou82b] N. Roussopoulos. View indexing in relational databases. *ACM TODS*, 7(2): 258-290, Jun. 1982.

[Ryu91] K. Ryu. A temporal database management system main memory prototype. TempIS Technical report, No. 26, University of Arizona, Jul. 1991.

[SA86] R. Snodgrass and I. Ahn. Temporal databases. *IEEE Computer*, 19(9):35-42, Sep. 1986.

[SG89] A. Segev and H. Gunadhi. Event-join optimization in temporal relational databases. In *Proc. of the Workshop on Query Optimization*, pages 205-215, Aug. 1989.

[Sno82] R. Snodgrass. Monitoring distributed systems: A relational approach. Ph.D Thesis, Computer Science Dept., Carnegie-Mellon Univ., Dec. 1982.

[Sno87] R. Snodgrass. The temporal query language TQuel. *ACM TODS*, 12(2):247 -298, Jun.1987.

[Sno89] R. Snodgrass. *The Interface Description Language: Definition and use. Rockville*, MD: Computer Science Press, 1989.

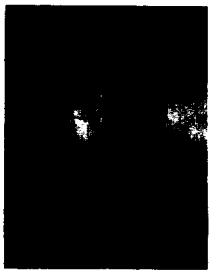
[Soo91] M.D. Soo. Bibliography on temporal databases. *ACM SIGMOD Record*, 20(1): 14-23, 1991.

[Sto87] M. Stonebraker. The design of Postgres storage system. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 289-300, Sep. 1987.

[SWKH76] M. Stonebraker, E. Wong, P. Kreps, and G. Held. The design and implementation of Ingres. *ACM TODS*, 1 (3):189-222, Sep. 1976.

[TCG+93] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Databases*. The Benjamin/Cummings Publishing Company, Inc, 1993.

[WK90] K. Y. Whang and R. Krishnamurthy. Query optimization in a memory-resident domain relational calculus system. *ACM TODS*, 15(1):67-95, 1990.



김 동 호

1993년 충북대학교 전자계산학과 졸업

1993년 충북대학교 대학원 현재 전자계산학전공 석사과정

관심분야 : 시간지원 데이타베이스, 시공간 데이타베이스, 정보검색, 멀티미디어

전 근 환

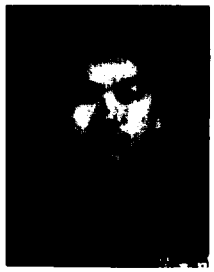
1993년 군산대학교 전자계산학과 졸업

1993년 충북대학교 대학원 현재 전자계산학전공 석사과정

관심분야 : 시간지원 데이타베이스, 시공간 데이타베이스

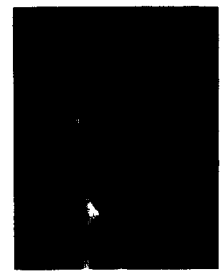
정 경 자

1988년 충북대학교 전산통계학과 졸업
1993년 충북대학교 대학원 전자계산학과(이학석사)
1993년 충북대학교 대학원 현재 전자계산학과 박사과정
관심분야 : 시간지원 데이타베이스, 시공간 데이타베이스, 정보검색



김 기 중

1983년 공군사관학교 졸업
1987년 서울대학교 계산통계학 졸업
1992년 충북대학교 대학원 현재 전자계산학과 석사과정
관심분야 : 시간지원 데이타베이스, 시공간 데이타베이스



류 근 호

1976년 숭실대 전산학과 졸업
1980년 연세대 산업대학원 전산전공(공학석사)
1988년 연세대 대학원 전산전공(공학박사)
1976년~1986년 육군 군수 지원사 전산실(ROTC장교), 한국 전자 통신 연구소(연구원), 한국 방송 통신대 전산학과(조교수) 근무
1989년~1991년 Univ. of Arizona 연구원
1986년~현재 충북대학교 컴퓨터과학과 부교수
관심분야 : 시간지원 데이타베이스, 시공간 데이타베이스, DBMS 및 OS, 객체 및 지식베이스 시스템 등임