

Linköping Electronic Articles in
Computer and Information Science
Vol. 3 (1998):nr 4

Study of Supporting Sequences
in DBMSs --
Data Model, Query Language,
and Storage Management

Ling Lin

Linköping University Electronic Press
Linköping, Sweden

<http://www.ep.liu.se/ea/cis/1998/004>

*Published on Feb. 17, 1998 by
Linköping University Electronic Press
581 83 Linköping, Sweden*

**Linköping Electronic Articles in
Computer and Information Science**

ISSN 1401-9841

Series Editor: Erik Sandewall

© 1998 Ling Lin

Typeset by the author using FrameMaker

Recommended citation

*<Author>. <Title>. Linköping Electronic Articles in
Computer and Information Science, Vol 3(1998): nr 4.
<http://www.ep.liu.se/ea/cis/1998/004/>. Feb. 17, 1998.*

This URL will also contain a link to the author's home page.

The publishers will keep this article on-line on the Internet (or its possible replacement network in the future) for a period of 25 years from the date of publication, barring exceptional circumstances as described separately.

The on-line availability of the article implies a permanent permission for anyone to read the article on-line, to print out single copies of it, and to use it unchanged for any non-commercial research and educational purpose, including making copies for classroom use.

This permission can not be revoked by subsequent transfers of copyright. All other uses of the article are conditional on the consent of the copyright owner.

The publication of the article on the date stated above included also the production of a limited number of copies on paper, which were archived in Swedish university libraries like all other written works published in Sweden.

The publisher has taken technical and administrative measures to assure that the on-line version of the article will be permanently accessible using the URL stated above, unchanged, and permanently equal to the archived printed copies at least until the expiration of the publication period.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: <http://www.ep.liu.se/> or by conventional mail to the address stated above.

Abstract

Many real life applications requires data that are inherently sequential. Sequential data exist in many domains such as temporal databases, execution monitors, trigger mechanisms, and list processing.

Traditional database systems did not pay special attention to sequence data which results in tedious query expression and bad performance. This report summarizes recent research on supporting sequence data in DBMSs, covering issues such as data model, query language, query optimization, and storage management. The sequence database system SEQ is described.

Ling Lin

Engineering Database and System Laboratory
Department of Computer and Information Science
Linköping Universitet
Linköping, Sweden

1 Introduction¹

Many real life applications requires data that is inherently sequential. Sequential data exist in many domains such as temporal databases, execution monitors, trigger mechanisms, and list processing. Examples of sequence data include: stock prices in business applications, temperature reading in scientific measurements, and event sequences in automatic control.

Traditional database systems are based on relational model which treats tables as sets, not sequences. Consequently, expressing sequence queries is tedious and execution is very inefficient [13]. Here is an example:

A weather monitoring system records information about various meteorological phenomena, such as volcano eruptions and earthquakes. These event sequences are ordered by time. Now we ask the query:

- For which volcano eruptions was the strength of the most recent earthquake greater than 7.0?

It is difficult to express this innocuous query in a relational query language like SQL and inefficient to execute. The SQL expression could be like this:

```
SELECT V.name
FROM Volcanos V, Earthquakes E
WHERE E.strength > 7.0 AND
      E.time = (SELECT max(E1.time)
                FROM Earthquakes E1
                WHERE E1.time < V.time)
```

A conventional query optimizer would probably generate the following query evaluation plan. For every Volcano tuple in the outer query, the subquery would be invoked to find the time of the most recent earthquake. Each such access to the subquery would involves an aggregate over the entire Earthquake relation. The time of the most recent earthquake is used as a join condition to probe the Earthquake relation in the outer query. Finally, the selection condition to check that the strength is greater than 7.0 would be applied. A more efficient evaluation strategy does however exists: the two sequences can be scanned in lock step (similar to a sort merge join). The most recent earthquake record scanned can be stored in a temporary buffer. Whenever a volcano record is processed, the value of the most recent earthquake record stored in the buffer is checked to see if

1.A seminar based on this report was given to EDSLAB members on Dec. 17th, 1997.

the strength was greater than 7.0, possibly generating an answer. This query can therefore be processed with a single scan of the two sequences, and using very little memory. The key to such optimization is the sequentiality of the data and the query.

[13] points out that sequence data need to be modelled as an abstract data type. Special operators such as *sub-sequence selection*, aggregate functions such as *sum*, *max*, *min*, and *moving average*, should be associated with the data type. More importantly, the ordered semantics of sequences should be utilized in query optimization (e.g., stream processing) and storage management (e.g., clustering).

2 The *SEQ* Model for Sequence Databases

[14] describes a general model for sequence data named *SEQ*. A sequence data model consists of an *ordering domain*, a *record domain*, and the relationships between them. See FIGURE 1.

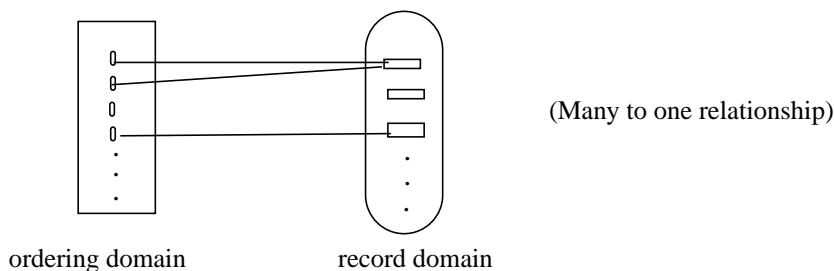


FIGURE 1. *The SEQ data model*

The ordering domain can be composed of any kind of ordered data such as integer, time stamps, etc. Each element in the ordering domain is called a *position*. Records can be of any data type such as floating values, strings, or even relational tables. Different positions in the ordering domain can be mapped to the same record, but every record can only be mapped to one position (i.e., many to one relationship).

Notice that there can be “holes” in the ordering domain, which results in sparse sequence. Sparse sequences correspond to real life sequences where there are missing values in the measurements.

Operations over sequences include: 1) transform operators (apply a function fn on each record in the sequence); 2) binary operators (e.g., two sequences join); 3) offset operators (e.g., shift in position or record domain); and 4) aggregate operators (e.g., *moving average*, *max*, *min*).

An interesting operator in [14] is sequence zooming, i.e., transform sequences according to different granularity on the ordering domain. Consider a sequence of stock data, with high and low values recorded every trading days. However, users might want to view the data at a weekly or monthly granularity, thus effectively modifying the ordering domain to weeks or months. Since the mapping from days to weeks (or months) is well defined, it is possible to allow such “zoom” operations. [14] defines a *collapse* as transformation from a coarse domain to a finer domain (e.g., from weeks to days), and *expansion* as the inverse. Certain well-known collapses and expansions on the ordering domain are pre-defined, and can be used by the query operators.

3 The SEQ Database System

Continuing the work of [14], [15] describes the design and implementation of a sequence database system named SEQ, including its query language *SEQUIN* and its query optimization engine. SEQ is a component of PREDATOR [20], a database system which supports enhanced abstract data types (such as image, video, audio, etc.). SEQ is built on top of the SHORE Storage Manager. It contained 35,000 lines of C++ code (excluding SHORE) at the time when [15] was published. This section summarizes the main points in [15].

3.1 The State of the Art

According to [15], currently general-purpose database systems provide very limited support for sequence data:

- The Order-by clause in SQL only specifies the order in which answers are presented to the users.
- In temporal databases there are few constructs that can exploit sequentiality. Many temporal queries expressed in SQL-92 using features like correlated subqueries and aggregation are typically very inefficient to execute.
- Most commercial database systems allow a sequence to be represented as a ‘BLOB’ which is managed by the system, but interpreted solely by the application program.
- Some object-oriented systems like O2 [4] provide array and list constructs but do not support query languages over them.

- The object-relational database system Illustra [8] provides database support for time-series data along with relational data. A time-series is an ADT (Abstract Data Type) value implemented as a large array on disk. A number of ADT methods are implemented to provide primitive query functionality on a time-series. The methods may be composed to form meaningful queries.

3.2 ADT v.s. E-ADT

There has been much research related to Abstract Data Type (ADT) technology beginning with [18]. Object-relational systems like Illustra [8] and Paradise [7] allow an attribute of a relational record to belong to an ADT. Each ADT defines methods that may be invoked on values of that type. An ADT can itself be a structured complex type like a sequence, with other ADTs nested inside it. Relations are *top-level* type and all queries are posed in the relational query language.

On the contrast, the PREDATOR design [20] enhances the ADT notion by supporting “Enhanced Abstract Data Types” (E-ADTs). Both sequences and relations are modelled as E-ADTs. Each E-ADT supports one or more of the following:

- Storage Management
- Catalogue Management (such as meta-data)
- Query Language (such as SQL and *SEQUIN*)
- Query Operators and Optimization
- Query Evaluation.

The E-ADT paradigm is a novel contribution that differentiates PREDATOR from the traditional ADT-method. The ability to name objects belonging to different E-ADTs allow *any* E-ADT to be the top-level type. For example, both relations and sequences are of top-level types in E-ADT paradigm. A detailed discussion on E-ADT can be found in [17].

3.3 The Sequence E-ADT

This section discusses how to support sequence E-ADT in SEQ [15].

Methods defined for the ordering domain are: *LessThan(Pos1, Pos2)*, *Equal(Pos1, Pos2)*, and *GreaterThan(Pos1, Pos2)*, which allow comparisons to be made among positions. *Collapse* is an operator used to transform sequences between different granularity in the ordering domain.

Methods on sequences are: *OpenScan(Cursor)*, *GetNext(Cursor)*, and *CloseScan(Cursor)*, which provide a scan of the sequence in the forward order of the ordering domain. Any positions in the domain which are not mapped to a record are ignored in the scan. *GetElem(Pos)* is used to find the record at the specified position in the sequence (or fails if none exists at that position).

Section 3.3.1 will discuss the storage implementation of sequences in SEQ. Section 3.3.2 will discuss the query language *SEQUIN* and its optimization.

There are three important properties of each sequence:

- *Cardinality* -- the number of records in the sequence.
- *Record width* -- the number of bytes in each record.
- *Density* -- the percentage of the positions in the underlying ordering domain that are non-empty.

Several sequences with different properties were generated to choose the best choice of storage implementation of sequences (Section 3.3.1). In all the measurements the SHORE storage manager buffer pool was set at 200 8K pages. Logging and recovery was turned off to mimic a query-only environment.

3.3.1 Storage Implementation

According to [15], there are three alternatives to implement a sequence using the SHORE storage manager:

- File -- stores the records of a sequence in a SHORE file.
- Ldlist -- stores a sequence as an array of record-id, with each record-id points to the actual record.
- Array -- stores a sequence as a compressed array of records (i.e., does not store the empty positions).

[15] used the efficiency of the “scan” operator over a sequence as the criteria for a good physical implementation and concluded that “compressed array” was the best storage choice. The rest of the experiments are based on the compressed array storage implementation.

3.3.2 *SEQUIN* Query Language

SEQUIN is a declarative language for querying sequence, similar in flavour to SQL. The result of a *SEQUIN* query is always a sequence. Following are some examples of *SEQUIN*, including operations such as selection,

moving average, and zooming. Two stock price sequences Stock1 and Stock2 are used in the examples. Both sequences have the same schema: {time: Hour, high: Double, low: Double, volume: Integer} and are both ordered by time.

- Estimate the monetary value of Stock1 traded in each hour when the low price fell below 50.

```
PROJECT ((A.high + A.low) / 2) * A.volumn
From Stock1 A
Where A.low < 50 .
```

- Finding the 24-hour moving average of the difference between the prices of the two stocks.

```
PROJECT avg(A.high - B.high)
From Stock1 a, Stock2 B
OVER $P - 23 TO $P
```

- Zoom:

```
PROJECT min(A.volume)
FROM Stock1 A
ZOOM days
```

The first example selects part of the sequence based on the condition that low price was less than 50. The second example applies a 24-hour moving average over the whole sequence. The third example demonstrates the zooming operation (zooms from hours to days).

Here I would like to point out something which is important and also related to my research. Notice that the first example could be much more efficient to execute if the IP-index [10] is available. Suppose that the IP-index is built on the “low” value of the stock sequence, then the sub-sequences which satisfy “low < 50” will be constructed quickly, then the projection can be applied to the returned sub-sequences instead of to the whole sequence. Currently in SEQ system the only way to process the first query is to scan the whole sequence and apply the projection to the positions whenever “low < 50” is satisfied. This is yet another example to show that an inverse index [10] is very important in sequence query processing.

For the forward queries [10] they use weighted binary search to find the record of position i , there is no index on the inverse direction as the IP-index does.

3.3.3 Query Optimization

There are several query optimization techniques discussed in [15], including selection push-down, incrementally computation of aggregate operators, and common sub-expression detection.

In selection push-down, they use a weighted binary search to access the position i (by using the meta-info such as the density and range of the sequence). Again as mentioned before, there is no index for selecting subsequences when the records satisfy some conditions. So selection push-down is efficient only in the forward query [10] case. To support selection push-down for inverse queries efficiently, some kind of index similar to the IP-index [10] has to be built.

Incremental computing of aggregate operators can be illustrated by this simple example: Consider the 3-position moving average of a sequence 1, 2, 3, 4, 5. Once the sum $1 + 2 + 3$ has been computed, then $2 + 3 + 4$ can be computed as $(1 + 2 + 3) - 1 + 4$. This is accomplished by storing the first sum $(1 + 2 + 3)$ in some buffer pool.

Another important optimization method in SEQ is stream processing. An example of stream processing was already introduced in Section 1, where two sequences can be scanned in parallel without materialize intermediate results. This technique is extremely important when sequences become large. It saves both query execution time and the storage space.

3.4 Combining Sequences with Relations

SQL and *SEQUIN* can be nested within each other to express queries concerning both relations and sequences. Consider a relation *Stocks* with the schema(*name:string*, *stock_history:Sequence*). The *stock_history* is a sequence of hourly information on the high and low prices, and the volume of the stock traded in each hour. An example query would be to find for each stock, the number of hours after hour 3500 when the 24-hour moving average of the high price was greater than 100.

```
SELECT S.name, SEQUIN (
  "PROJECT count (*)
  FROM (PROJECT avg(H.high) as avghigh
        From $1 H
        OVER $P -23 TO $P) A
  WHERE A.avghigh > 100 and $P > 3500
  ZOOM ALL"
  S.Stock_history)
From Stocks S;
```

SQL parser passes the *SEQUIN* sub-query to the *SEQUIN* parser. The SQL optimizer is called on the outer query block, and the *SEQUIN* optimizer is called on the nested query block. There is currently no optimization performed across query blocks belonging to different E-ADTs.

Notice again that this query would be more efficient to execute if the IP-index [10] is available. Suppose that the IP-index is built on the 24-hour moving average of the high price, then the number of hours when the 24-hour moving average of the high price was greater than 100 can be computed very fast since the IP-index stores cardinality information [11]. This counting can be accomplished efficiently without even going through all the positions which satisfy the condition. This is extremely important when the resulting sequences are large.

It is interesting to compare the system of SEQ with Illustra [8]. Illustra supports sequences (more specifically, time-series) as ADTs with a collection of methods. The above example would be expressed in Illustra as the following:

```
SELECT S.name
  count(filter("time > 3500",
    filter("high > 100",
      mov_avg(-23, 0,
        project("time,high", S.stock_history))))))
From Stocks S;
```

[15] claims that: 1) A query language based on function composition can be more awkward to use than a high-level language like *SEQUIN*; 2) Queries based on functional composition have a procedural semantics (not declarative semantics), little or no inter-function optimization is performed. [15] made a performance comparison with Illustra but was not allowed to publish the results. Anyway, they claim that Illustra can't perform:

- pipeline operations on two functions
- identifying common sub-expressions in queries (when queries are expressed as functions)
- selection push down...

[15] claims that their approaches result in overall performance improvements of approximately two orders of magnitude compared to Illustra. they claim that the differences are symptoms of a basic design difference between SEQ and ADT-method systems.

Until now I have covered the main points in [13], [14], and [15] about supporting sequences in database systems. In the next section I will discuss storage management for large objects in disk-based database systems. The reason why we discuss storage management for large objects is that sequences usually grow very large in real life applications, which leads to the question of how to store these large sequences in disk in the way that 1) random access of any position will be fast and 2) the dynamic growing property of sequences will be supported well.

4 Storage Management for Large Objects

4.1 History Overview

Traditional DBMSs require attributes in a table to be the size of less than 255 bytes. Long fields are treated separately by BLOBs -- Binary Large Object Boxes, which is a long binary string interpreted and maintained by the application program, not the DBMSs.

System R [1] long field manager divided long fields into a linked list of small manageable segment, each 255 bytes in length. Only read/write on the entire long field was supported, partial access or update was not supported.

Later, an extension to SQL introduced the long field cursor, for partial read/write. So the long field was stored as a sequence of 4K data pages. The maximum length of a long field in extended SQL was about 2 gigabytes [12].

At the same time, the Wisconsin Storage System [6] developed a similar mechanism for storing long fields. A long field was split into 4K pages (called slices) and a partially filled slice (called crumb). A directory of slices plus a crumb decides how to access the long field. The maximum size was 1.6 megabytes.

EXODUS [4][5] stores *all* objects in a general-purpose storage mechanism that can handle objects of any size -- the limit is imposed by the amount of physical storage available. EXODUS uses a data structure that was inspired by the ordered relation data structure in INGRES [19]. It is a B+-tree indexed on byte position within the object, with the B+-tree leaves as the data blocks. Object smaller than a page are stored as single records. Random read/write of large objects is very convenient.

The Starburst long field manager [9] deals with fast movement of very large objects (100 megabytes) between main memory and disk. It aims at

moving the whole object fast (in fewer disk block access) instead of random access of a partial object. It supports fast sequential read/write. For updates, only appending and trimming at the end of the object is supported. The approach used was “buddy segments” and bitmap encoding. For details see [9].

It can be seen from the above discussion that EXODUS storage management for large objects is the best choice for sequence implementation. This is because 1) it supports objects that are dynamically growing (due to the B+-tree implementation); and 2) access or update in the middle of the object is as efficient as access or update at the end of the object. The second property is important for sequence data since operations on sequences need random access quite often. Starburst long field manager is more suitable for managing other kind of large objects such as image, audio, video, where operators over those objects require more sequential access than random access.

In SHORE [3] there is a persistent data type named “sequence”. The rest of the section will discuss the implementation of this data type.

4.2 The SHORE Implementation of Sequences

SHORE (Scalable Heterogeneous Object REpository) [3] is a distributed persistent object system evolved from EXODUS. There is a persistent data type named “sequence” (a persistent variable-length array) which can grow arbitrary large (only limited by the physical storage available) and supports efficient search, insert, and delete in any position of the sequence. SHORE “sequence” is special in the sense that it has the following two features at the same time: 1) allows dynamic growing to any size; 2) allows efficient random access. Among other systems I have studied, either they allow largely growing size but only sequential access, or they support random access but only for predefined (fixed) size. I have not seen any other data structure which combines these two features together like SHORE does.

4.2.1 Implementation

This section describes the implementation of *sequence* in SHORE. When an application program creates a *sequence* object, SHORE allocates an arbitrary memory size (an array with 8 elements) for the initial sequence (since the size of the sequence is unknown and can grow dynamically). When the application program extends the sequence to the size which can no longer fit in the allocated space (by insert or append operations),

SHORE allocates another memory space which is double size as the old one and copies the old sequence to the newly allocated space. In this way the sequence grows in size*2 to any size as required.

When it is time to write the sequence from the main memory to disk, SHORE storage manager checks if the sequence fits in one disk page or not. If so, then the sequence is represented as (page #, slot #) which is the same as a record stored in disk. If the sequence occupy multiple pages, then the sequence is represented as a B+-tree index on byte positions within the sequence, plus a collection of leaf (data) blocks. The size of a leaf block can be set to 1 to 4 continuous disk pages. An example of a large sequence stored in disk is in FIGURE 2.

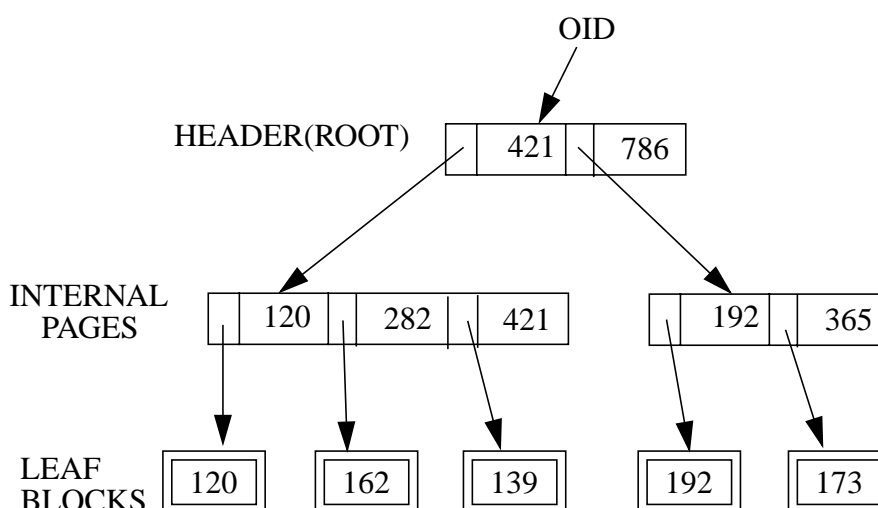


FIGURE 2. An Example of a Large Sequence in disk

When the application program requires a sub-sequence starting from position i to position $i+l$ (length l), the bytes ranging from i to $i+l$ in the leaf node(s) are transferred to the buffer pool in a continuous space. A *scan descriptor* records where these bytes come from (i.e., disk addresses), see FIGURE 3. The application program uses the scan descriptor to access the sub-sequence. (No copying from buffer pool and application address space is done in EXODUS in order to speed up access.)

4.2.2 Performance:

According to [4][5], storage utilization in EXODUS for large objects is 80% or higher, which means the leaf blocks will be at least 80% full. Time spent in search a sequence is pretty stable (nearly constant to 100 ms), this

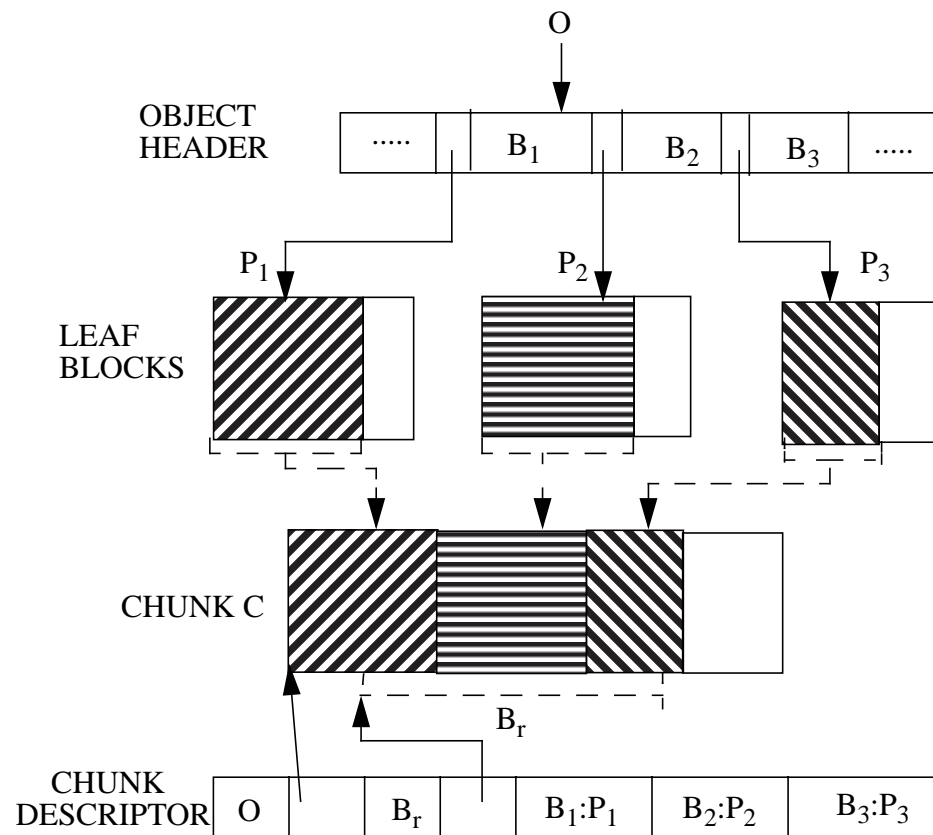


FIGURE 3. *Continuous Buffer in SHORE*

is because the level of the B+-tree is very low, no more than 3 levels are needed to hold up to 8MB-4GB objects.

5 Conclusions

This report represents what I have studied lately on supporting large sequences in DBMSs, covering issues such as data model, query language, and storage implementation. The sequence database system SEQ was introduced and the storage management of EXODUS (SHORE) for large dynamic objects was presented. I also pointed out some related issues to my research work.

References

- [1] M. Astrahan *et al.*, “System R: Relational Approach to Database Management”, *ACM TODS*, Vol. 1, No. 2 (June 1976).
- [2] F. Bancilhon, C. Delobel, and P. Kanellakis (eds): “*Building an Object-Oriented Database System: The Story of O2*”. Morgan Kaufmann Publishers, 1992.
- [3] Michael J. Carey, David J. Dewitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, Michael J. Zwilling, “Shoring Up Persistent Applications”, in *Proceeding of the 1994 ACM-SIGMOD Conf. on the Management of Data*, Minneapolis, MN, May 1994.
- [4] M. J. Carey, D. J. DeWitt, J. e. Richardson, and E. J. Shekita: “Storage Management for Objects in EXODUS”, in “*Object-Oriented Concepts, Databases, and Applications*”, by W. Kim and F. Lochovsky, eds., Addison-Wesley Publishing Co., 1989.
- [5] M. J. Carey, D. J. DeWitt, J. e. Richardson, and E. J. Shekita: “Object and File Management in the EXODUS Extensible Database System”, in *Proc. of the 12th VLDB Conf.*, Kyoto, Japan, 1986.
- [6] H-T Chou et al: “Design and Implementation of the Wisconsin Storage System,” in *Software Practice and Experience*, Vol. 15, No. 10, Oct., 1985.
- [7] D. J. Dewitt, N. Kabra, J. Luo, J. M. Patel and J. Yu: “Client-Server paradise”, in *Proc. of VLDB Conf.*, Santiago, Chile, 1994.
- [8] Illustra Information Technologies, Inc. *Ullustra User’s Guide*, June 1994.
- [9] T. J. Lehman and B. G. Lindsay: “The Starburst Long Field Manager”, in *Proc. of the 15th VLDB Conf.*, Amsterdam, 1989.
- [10] L. Lin, T. Risch, M. Sköld, D. Badal, “Indexing Values of Time Sequences”, in *Proceedings of 5th International Conference on Information and Knowledge Management*, pp. 223-232, Rockville, Maryland, Nov. 1996.
- [11] L. Lin, “*A Value-Based Indexing Technique For Time Sequences*”, Lic. Thesis No 597, Linköping University, Jan., 1997, ISBN 91-7871-888-0.

- [12] R. Lorie and J. Daudenarde: “Design System Extensions User’s Guide”, April, 1985.
- [13] P. Seshadri, M. Livny, and R. Ramakrishnan: “Sequence Query Processing”, in *Proc. of ACM SIGMOD’94*, Minneapolis, MN, May 1994.
- [14] P. Seshadri, M. Livny, and R. Ramakrishnan: “SEQ: A Model for Sequence Database”, in *Proc. of the 11th Data Engineering Conf.*, Taipei, Taiwan, March, 1995.
- [15] P. Seshadri, M. Livny, and R. Ramakrishnan: “The Design and Implementation of a Sequence Database System”, in *Proc. of the 22nd VLDB Conf.*, Mumbai, India, 1996.
- [16] P. Seshadri: “*Management of Sequence Data*”, Ph.D Thesis, University of Wisconsin-Madison, 1996.
- [17] P. Seshadri, M. Livny, and R. Ramakrishnan: “The case for Enhanced Abstract Data Types”, in *Proc. of the 23rd VLDB Conf.*, Athens, Greece, 1997.
- [18] M. Stonebraker: “Inclusion of New Types in Relational Data Base Systems”, in *Proc. of Data Engineering*, pp. 262-269, 1986.
- [19] M. Stonebraker, H. Stettner, N. Lynn, J. Kalash, and A. Guttman, “Document Processing in a Relational Database System”. In *ACM Tran. on Office Information Systems*, vol. 1, No. 2, April 1983.
- [20] PREDATOR project web page: “<http://simon.cs.cornell.edu/Info/Projects/PREDATOR/>”.