Querying Continuous Time Sequences

Ling Lin Tore Risch
Department of Computer Science
Linköping University
Linköping, Sweden
{linli, torri}@ida.liu.se

Abstract

Time sequences appear in various application domains. Many applications require time sequences to be seen as *continuous* where implicit values can be derived from explicit values by arbitrary user-defined interpolation functions. This paper describes the implementation of an extended SELECT operator, σ*, that retrieves implicit values from a discrete time sequence under various user-defined interpolation assumptions. The σ* operator is efficiently supported by an indexing technique termed the IP-index. Possible optimizations of the σ* operator are investigated and verified by experiments on SHORE. The σ* operator is applicable to any 1-D sequence data.

1 Introduction

Modern Database applications involve large amounts of time sequences. Examples of time sequences appear in various application domains: 1) Scientific experiments such as temperature reading generated by sensors; 2) business applications such as stock price indexes or bank account histories; 3) medical data such as patients' temperature readings or cardiology data; 4) event sequences in automatic control and process supervision. In concept, a time sequence (TS) can be modelled as a sequence of states S_i*. Each state has a time stamp and a value, i.e., $S_i=(t_i, v_i)$.

To meet the requirements of these applications, considerable research effort has been dedicated to querying time sequences. Most of the work deals with similarity search, i.e., finding sub-sequences that match a given pattern in some error distance [1][2][3][12][22]. Various approaches have been suggested, such as using the Discrete Fourier Transform, interpolation approximation, or defining some shape querying languages. The reason why nearly all research on time sequences has been dedicated to examine shapes is as [22] pointed out, "individual values are usually not important but the relationships between them are". However, we argue that in many applications individual values are at least as important as shapes of time sequences. Two examples are given below:

Example 1

[22] gives the example of finding the pattern of "goalpost fever" (Fig. 1.1) in a patient's temperature reading (a time sequence). "Goalpost fever" is one of the symptoms of Hodgkin's disease, behaving as two consecutive fevers during 24 hours. This query was formulated as a shape query in [22] as "finding those sub-sequences with exactly two peaks". However, since a "fever" means the body temperature is higher than 38°C, this query can also be formulated as "finding the two time intervals when the *values* inside the intervals are greater than 38 and the distance between them is less than 24 hours".

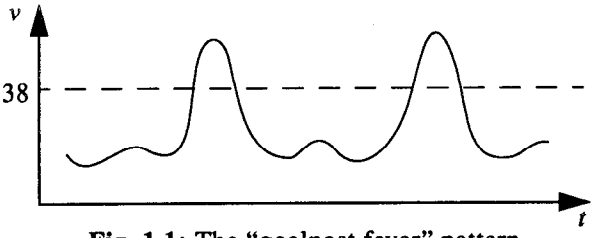

Fig. 1.1: The "goalpost fever" pattern

Example 2

Fig. 1.2 shows a periodic time sequence representing the pressure of a cylinder inside an engine. The data was collected by a sensor in a real-life application [9]. The pressure of the cylinder changes with its angle periodically (360°) and reaches a peak once in every period. On monitoring the behaviour of the engine, an interesting query would be "when did the pressure reach its

Proceedings of the 24th VLDB Conference
New York, USA, 1998

peak in every period?" [9]. It can be seen from Fig. 1.2 that all peaks have the property that v > 1.5. So this query could be reformulated as "when was the *value* greater than 1.5?".

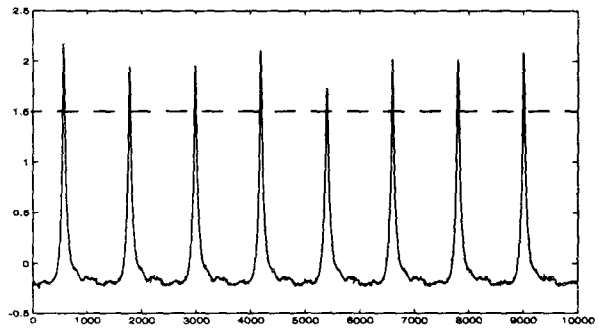

Fig. 1.2: The real-life pressure sequence

Thus, we argue that queries concerning *values* of time sequences are as important as queries concerning *shapes* of time sequences. We term queries concerning values as *value queries* in contrast to shape queries. To support value queries on a time sequence is not trivial because most applications require time sequences to be seen as *continuous* where implicit values can be derived from explicit (stored) values.

In this paper we present an extended SELECT operator, σ*, and its implementation. The σ* operator retrieves implicit values from time sequences under various user-defined interpolation assumptions. A new indexing technique, the IP-index [13], has been developed to support the σ* operator. The preliminary work on the IP-index, [13], concerns only main-memory implementation and investigated only some aspects of the insertion and search time of the index. In this paper we analyze the behaviour of the IP-index with respect to the properties of time sequences and typical query patterns, and the analysis is based on an implementation on the disk-resident database system SHORE [6]. The efficiency of the σ* operator is demonstrated by experiments on SHORE using both synthetic and real-life time sequences. Possible optimizations of the σ* operator are investigated and verified. We also investigate space usage of the IP-index with regard to the cardinality of the time sequences to show that it is practical to build IP-indexes for large time sequences. An interesting comparison of the IP-index with conventional secondary indexes is also given.

The σ* operator and its corresponding optimization rules can be plugged into extensible database systems such as Illustra [11] or PREDATOR [21] where extensible cost-based or rule-based optimizations are supported. For example, Illustra's "time series" data type [25] could benefit from the σ* operator.

This paper is organized as following: Section 2 discusses related work. Section 3 shows how the σ* operator works for different selection conditions and discusses possible optimizations. A comparison of the IP-index with conventional secondary indexes is given in Section 4. Section 5 shows experimental results made on SHORE. Conclusions and future work are given in Section 6.

2 Related Work

The importance of associating interpolation methods with temporal data was pointed out by Clifford [8] as the "Comprehension Principle", i.e. "under any reasonable interpretation a historical database defined over a sequence of states $<S_1, S_2,..., S_n>$ should be considered as modelling an enterprise completely over the entire closed interval $[S_1, S_n]$". It was also mentioned that the mapping from a finite set of moments $< S_1, S_2,..., S_n >$ into the densed interval $[S_1, S_n]$, termed the "Continuous Assumption", could be a accomplished by various interpolation methods.

Segev [16][17] proposed a temporal data model based on time sequences. Four *types* of time sequences are defined according to what interpolation assumption is applied, a) Step-wise constant (all values in $[S_i, S_{i+1})$ are assumed to be equal to v_i), b) Continuous (a curve-fitting function is applied over $[S_i, S_j)$), c) Discrete (missing values cannot be interpolated) d) User-defined (a user-defined interpolation function is applied).

Although it was pointed out in the early 80's that it is important to support interpolation assumptions on time sequences, very few *implementation* issues have been addressed. For example: how to support queries based on arbitrary user-defined interpolation assumptions, and how to process these queries *efficiently*, especially when the time sequences are very long. These are the motivations for this paper.

In [8] an extended SELECT operator (σ*) was mentioned that denotes selecting implicit states from $<S_1, S_2,..., S_n>$ based on the "step-wise constant" assumption. But no implementation was discussed. A recent paper [5] points out that by the "step-wise constant" assumption, a database DB can be seen as a larger database $\overline{DB}$ that contains both explicit and implicit information. It suggests that a user query Q can be transformed into a "system query" Q'. Q' contains the "step-wise constant" assumption so that applying Q' to DB will yield the same result as applying Q to $\overline{DB}$. In this paper we take a different approach. We associates the interpolation assumption with the SELECT operator σ instead. In this way there is no need to transform the database DB to $\overline{DB}$ or the user query Q to Q'. Also we support more sophisticated interpolation functions such as linear interpolation or moving average.

In [22], in order to find the "goalpost fever" pattern, the temperature sequence has to be transformed into some "feature preserved" representation. By contrast, in our solution, we view the problem as a value query so that there is no need to transform the original time sequence. This example also shows that it is not true that the amplitude of a time sequence can always be ignored, named "amplitude independence" in [22]. In fact, the amplitude of a sequence in an x-y axis is always sensitive to the unit of the y-axis, just as the fre-

quency of the sequence is sensitive to the unit of the x-axis. In this sense value queries are as important as shape queries.

Time sequences can be seen as a special case of 1-D sequence data [20] where the ordering domain [20] is "time". Other ordering domains are integers, space positions, etc. The σ* operator and its optimization techniques are applicable to any 1-D sequence data, although this paper only concerns time sequences. [19] motivated the importance of sequence query processing and addressed efficiency issues. It pointed out that the *ordered* semantics of sequences should be utilized in query optimization for sequence data. Based on [19], [20] presents a sequence database system named SEQ. In Section 3.3 we will show that the IP-index can be utilized to improve the efficiency of selection push down in SEQ [20]. We found that the sequence database system SEQ is currently the most relevant related work to the σ* operator and our IP-index.

3 The σ* Operator and the IP-index

To make our discussion independent of any data model and physical implementation, a time sequence is denoted as a sequence of states TS =<S_1, S_2,...S_n> where S_i = (t_i, v_i) (i = 1, 2...n). By associating a user-defined interpolation function *ifn* with it, TS will be transformed into $\overline{TS}$ (following the same notation as [5]). $\overline{TS}$ is a *continuous* time sequence defined over the time interval [t_1, t_n] by applying *ifn* on the discrete TS. A *SELECT operator* σ on $\overline{TS}$ returns *sub-sequences* (time intervals) where the values or time stamps inside those intervals satisfy some conditions, i.e., σ($\overline{TS}$) = (t', t'')*. In the extreme case, the σ operator returns "points", i.e., implicit or explicit states S'=(t', v'). Since $\overline{TS}$ is continuous, it is impossible to generate all S's and store them in the database. Instead, we associate the interpolation assumption *ifn* with the σ operator, resulting in the σ* *operator*. Applying σ* to a TS will generate the same result as applying σ to the corresponding $\overline{TS}$, i.e., σ*(TS, *ifn*) = σ($\overline{TS}$). Let us take a look at how the σ* operator works for different selection conditions.

3.1 $\sigma^*_{t=t'}$(TS)

The operator $\sigma^*_{t=t'}$(TS)[1] returns the state S'=(t', v') in the continuous $\overline{TS}$ whose time stamp is t'. The value v' is calculated by:

1. Find the state S_i in TS where

   S_i.time ≤ t'< S_{i+1}.time --- (step 1)

2. Apply the interpolation function *ifn* to the neighbour states of S_i.

   v' = *ifn*(t', *surrounding_states*(S_i)) --- (step 2)

1. A precise notation should be $\sigma^*_{t=t'}$(TS, *ifn*) where *ifn* is the user-defined interpolation function. We omit the argument *ifn* assuming that a system-defined (default) interpolation function (e.g., linear interpolation) is used.

In step 2 above, the definition of *surrounding_states*(S_i) is determined by the interpolation function *ifn*. For example: a) If *ifn* is "linear interpolation", then *surrounding_states*(S_i) = {S_i, S_{i+1}}; b) If *ifn* is moving-average over three states, then *surrounding_states*(S_i) = {S_{i-1}, S_i, S_{i+1}} (or perhaps {S_i, S_{i+1}, S_{i+2}}). In the simplest case of the "step-wise constant" assumption, we have *surrounding_states*(S_i) = {S_i}.

IP Operator

The key to support the interpolation assumption on TS is in step 1 -- to *locate* the position in TS where to apply *ifn*. We define step 1 as the *IP operator*. It returns the *state_id* where *ifn* can be applied. Therefore, $\sigma^*_{t=t'}$(TS) is implemented by the execution of the IP operator and *ifn*, as illustrated in Fig. 3.1:

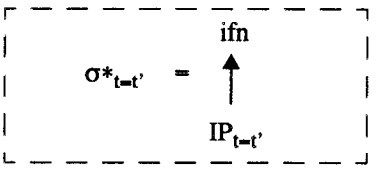

Fig. 3.1: The relationship between the σ* operator and the IP operator

A naive way to implement $IP_{t=t'}$(TS) is to linearly scan TS to find the state S_i where S_i.time ≤ t'< S_{i+1}.time. More efficient implementations of the IP operator can take advantage of the physical organization of TS and available indexes. For example, If TS is implemented by an array [15], then a binary search will do.

Many applications assume the "step-wise constant" interpolation assumption on time sequences. In this case there is no need to apply *ifn*. We have in step 2 v'= S_i.value where S_i is returned by step 1 -- $IP_{t=t'}$(TS).

The advantage of the $IP_{t=t'}$(TS) operator is that it is *independent* of the interpolation assumption *ifn*. Intuitively, it returns the *nearest* neighbour states for the time point t'.

3.2 $\sigma^*_{v=v'}$(TS)

The operator $\sigma^*_{v=v'}$(TS) returns a sequence of states (t', v')* in $\overline{TS}$ whose values are equal to v'. The time points t's can be calculated similarly as in Section 3.1:

1. Locate the states S_i in TS where

   S_i.value ≤ v'< S_{i+1}.value --- (step 1)

2. Apply ifn^{-1} [2] on the neighbor states of S_i.

   t' = ifn^{-1}(v', *surrounding_states*(S_i)) --- (step 2)

The above step 1 is defined as the $IP_{v=v'}$(TS) operator. It returns the positions in the time sequence where to apply ifn^{-1}. For example, in Fig. 3.2, $IP_{v=v'}$(TS) will return <S_1, S_6, S_{10}>. This state sequence is termed *anchor-state sequence* and is stored in the IP-index [13]. In the

2. ifn^{-1} is the inverse function of *ifn*.

next section we briefly recall the idea of the IP-index.

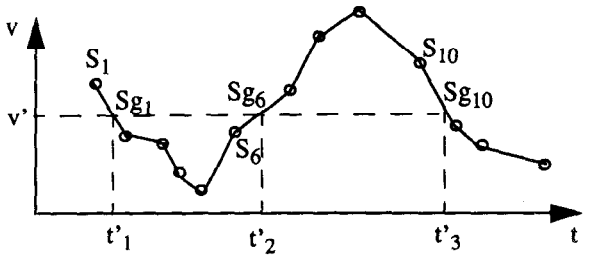

Fig. 3.2: Illustration of a value query on a TS

3.2.1 IP-index

The IP-index stores the anchor-state sequence of v' by recording all segments that intersect with the line v = v'. For example, in Fig. 3.2, the segments that intersect with the line v = v' are Sg_1, Sg_6, Sg_{10}. Thus the anchor-state sequence of v' is $<S_1, S_6, S_{10}>$. The anchor-state sequence of v' is denoted as *A(v')*. The cardinality of A(v') is denoted as *card(A(v'))* and is also stored in the IP-index.

The structure of the IP-index is the following: The keys in the index are the ordered, distinct values of the v_is in TS, and each key k_i is associated with a pointer to A(v') for those v' where $k_i \le v' < k_{i+1}$. Thus, the anchor-state sequence of any value v' can be retrieved easily by performing a range search (i.e., $k_i \le v' < k_{i+1}$) on the IP-index and retrieve $A(k_i)$. Therefore, the $IP_{v=v'}(TS)$ is efficiently supported by the IP-index.

Notice that a naive way to execute $IP_{v=v'}(TS)$ (without the IP-index) is to linearly scan the whole time sequence to find those S_is where S_i.value $\le v' < S_{i+1}$.value. We will demonstrate the performance improvement in the experiments in Section 5.3.

We shall point out that a limitation of the IP-index is that if the interpolation function *ifn* introduces new *extreme* points (thus introduce new segments) to the original time sequence, then the IP-index needs to be modified to include the extra segments as well, as mentioned in [13].

3.2.2 First Few Answers

Since A(v') is an ordered sequence of states, $IP_{v=v'}(TS)$ can be implemented as a *stream* where the *next* element of $IP_{v=v'}(TS)$ is the *next* state in A(v'). Therefore, $\sigma^*_{v=v'}(TS)$ can be implemented as a stream as well: the *next* state of $\sigma^*_{v=v'}(TS)$ is generated by applying ifn^{-1} over the neighbor states of the *next* state returned from $IP_{v=v'}(TS)$ ("step 2" above).

By implementing $\sigma^*_{v=v'}(TS)$ as a stream the *first few answers* [4] can be generated quickly. This is especially important when card(A(v')) is large. To generate the first few answers, the interpolation function ifn^{-1} is applied to only the first few states in A(v'). Notice that the stream of $\sigma^*_{v=v'}(TS)$ and $IP_{v=v'}(TS)$ can be generated in the *reverse* order as well, i.e., the states with newer time stamps come out first. This is useful in many applications since newer states are usually more interesting than older ones.

By contrast, linearly scanning TS will take very long time to get the first answer when the first answer appears late in the TS. This will be shown in experiments in Section 5.4.

3.2.3 New Functions Proposed

As Silberschatz et. al. [24] point out, the new generation of object-relational database systems will allow complex types, nested relations, and object-oriented features. SQL-3 is under way to standardize queries on complex types. Stonebraker [25] points out that TSs should be modelled as a new abstract data type in object-relational databases (instead of as tables in relational databases). Operations on TSs can be defined as functions (methods) such as "moving_avg(TS, 5, '1995-07-15')" [25] (five-day moving average on July 15th, 1995). To query continuous TSs, we propose the functions:

- `get_time_stamps(TS, '=', v')`
  // assume default interpolation assumption
- `get_time_stamps(TS, '=', v', ifn)`
  // assume user-defined interpolation assumption *ifn*

to return the time points when the values are equal to v' for a continuous TS. These time points t's can be extracted from the pairs (t', v')* that are returned by the $\sigma^*_{v=v'}(TS)$ operator.

3.3 Range Queries

Range queries are essential for time sequences. It should be possible to extract sub-sequences by:

1. A time interval (t1, t2);
2. A value range (v1, v2).

Since we view time sequences as continuous, the result of a range query is a sequence of time intervals. For example, in Fig. 3.2, $\sigma^*_{v>v'}(TS)$ will return the sequence of time intervals: $<(t_1, t'_1), (t'_2, t'_3)>$.

Range queries based on time conditions, i.e., $\sigma^*_{t>t'}(TS)$ (or $\sigma^*_{t<t'}(TS)$) are relatively easy to support because binary search on the time sequence array can find the position of t1 and t2 even when t1 and t2 are implicit time stamps. Range queries on value conditions, i.e., $\sigma^*_{v>v'}(TS)$ (or $\sigma^*_{v<v'}(TS)$) are difficult to support when v1 and v2 are implicit. Without a suitable index the whole sequence has to be scanned.

By using the IP-index, the $\sigma^*_{v>v'}(TS)$ operator can be supported surprisingly easy. For example, in Fig. 3.2, $\sigma^*_{v>v'}(TS)$ returns (t_1, t'_1) and (t'_2, t'_3). The time points t'_1, t'_2 and t'_3 can be extracted from the results of $\sigma^*_{v=v'}(TS)$. In this way there is no need to visit those states inside the range v>v' (or inside the time intervals (t_1, t'_1) and (t'_2, t'_3)). This indicates that the *cost of the range query $\sigma^*_{v>v'}(TS)$ is nearly the same as the cost of $\sigma^*_{v=v'}(TS)$.*

Furthermore, we will show in next section that the $\sigma^*_{v>v'}(TS)$ operator is also useful in processing discrete time sequences.

3.3.1 Discrete Range Queries

The operator $\sigma^*_{v>v'}$(TS) improves the efficiency of query processing even when the time sequence is *discrete*. [20] gives an example query that asks for the monetary value of Stock1 traded in each hour when the low price fell below 50.

```
SELECT[1] ((A.high+A.low)/2)*A.volume
FROM Stock1 A
WHERE A.low < 50
```

[20] claimed that selection push-down (A.low<50) should be applied here to optimize the query so that the calculation of "((A.high+A.low)/2)*A.volume" only needs to be done for those states whose low values are below 50. But, without an index, the whole time sequence has to be *scanned* to find these states. One may argue that a conventional secondary index on the "low" value will help. Unfortunately it does not. We will explain the reasons in Section 4.

By applying the $\sigma^*_{v<50}$(TS) operator, we can retrieve the time points t' and t'' (see Fig. 3.3) directly and then apply the calculation to only those S_is in the range (t', t'').

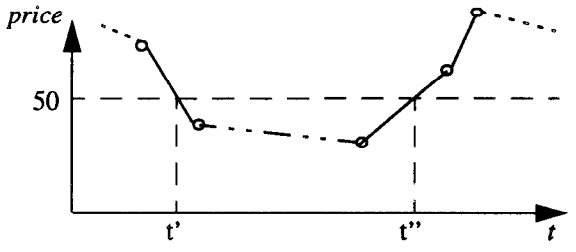

Fig. 3.3: A stock price sequence

Selections on the value dimension appear very often in real-life applications, but we have not seen any other indexes similar to the IP-index that is designed particularly for the value dimension of time sequences.

3.3.2 New Functions Proposed

To support range queries on continuous time sequences, we propose the functions:

- `get_time_intervals(TS, '>', v')`
  // assume default interpolation assumption
- `get_time_intervals(TS, '>', v', ifn))`
  // assume user-defined interpolation assumption *ifn*

to return those time intervals when the values are greater than v'. This function is translated to the $\sigma^*_{v>v'}$(TS) operator and is efficiently supported by the IP-index.

3.4 Time Window Queries

Some value queries only concern a *part* of the time sequence, i.e., a time window. An example of a time window query could be: When did the patient have a fever *in the last few days* (denoted as t > t')? Using the new functions defined in Section 3.2.3, this query can be expressed as the following:

```
SELECT t
FROM Temperature_seq TS
WHERE t IN get_time_stamps(TS, '='[2], 38)
      AND t > t'
```

The answer of this query is marked by the two crosses in Fig. 3.4. This query can be processed in two steps: 1) $\sigma^*_{v=38}$ (TS); 2) $\sigma^*_{\triangleright t'}$ (TS). The $\sigma^*_{v=38}$ (TS) generates all (explicit or implicit) states S' where S'.value = 38. Every state S' from $\sigma^*_{v=38}$ can be checked to see if it is in the time interval (t', now) to get the resulting states.

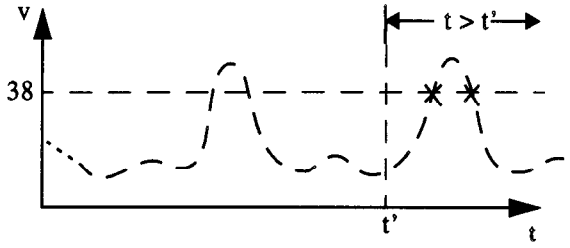

Fig. 3.4: A time window query

Optimization of Time Window Queries

When there are many states returned from $\sigma^*_{v=38}$ and the resulting states are very few (the time window is small), it might be a waste to calculate all S's and check the condition later. Recall the operator $\sigma^*_{v=38}$ (TS) is accomplished by $IP_{v=38}$ (TS) and *ifn* (illustrated as (a) in Fig. 3.5), the selection $\sigma^*_{\triangleright t'}$ (TS) can be "pushed down" to the $IP_{v=38}$ (TS) operator, resulting in the operator $IP_{v=38 \text{ AND } \triangleright t'}$(see (b) in Fig. 3.5). The operator $IP_{v=38 \text{ AND } \triangleright t'}$ can be accomplished by binary searching A(38) to find the first state S_i where S_i.time > t'. In this way only a part of the anchor-state sequence A(38) (the part that is inside the time window) is involved in query processing.

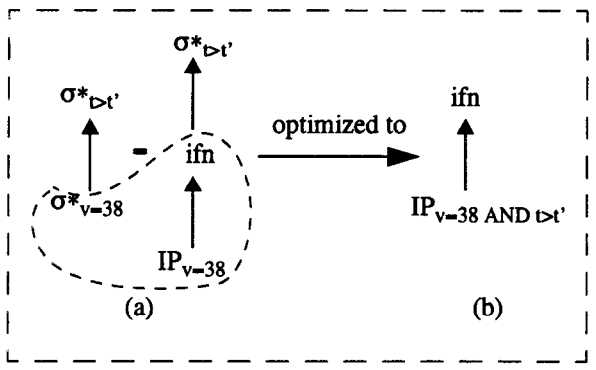

Fig. 3.5: Optimization of time window queries

Another possible optimization strategy is to generate $\sigma^*_{v=38}$ as a reverse stream (as described in Section

1. In [20] 'PROJECT' was used instead of the SQL keyword SELECT.

2. In reality we should use ">" instead of "=" since a fever means body temperature > 38°C. All the discussions will hold.

3.2.2) and terminate when t > t' does not hold. This strategy, compared to the one in Fig. 3.5, has the limitation that the output stream is not in the same (time) order as the input stream so that it cannot be used in methods such as sort-merge joins of time sequences. Also notice that a *general* time window query t' < t < t'' requires binary search on A(v') to efficiently find the positions of t' and t''.

Performance comparison of these different strategies for time window queries are given in Section 5.5 by experiments on SHORE.

Time window queries for the condition t < t' can be optimized similarly by pushing the condition t < t' down to the IP operator as in Fig. 3.5. However, no binary search is needed here since the starting position is the *first* state in A(v'). The stream output of the IP operator is terminated when the condition S_i.time < v' does not hold any more.

4 Comparison of the IP-index with Conventional Secondary indexes

This section explains why the IP-index is needed even when there are conventional secondary indexes available. The reason why the IP-index is compared with conventional secondary indexes is that the IP-index is essentially a secondary index as well. A secondary index is a "nonclustering index", as defined in [24]. TSs are normally clustered by time stamps t_is, not by values v_is. Therefore, all indexes on the value domain of a TS are considered to be secondary indexes.

Suppose that Fig. 4.1 represents a patient's temperature reading sequence TS = S_i* where S_i=(t_i, v_i), and linear interpolation is assumed to transform the temperature sequence TS into a continuous function $\overline{TS}$. A conventional secondary index on the value v_is will use the distinct values of v_is as keys k_j and record all the (t_i, v_i) pairs where v_i equals to the key k_j. By contrast, the IP-index associates the keys k_js with their anchor-state sequences (Section 3.2.1). Let us compare the IP-index with the conventional secondary index in dealing with the following value queries:

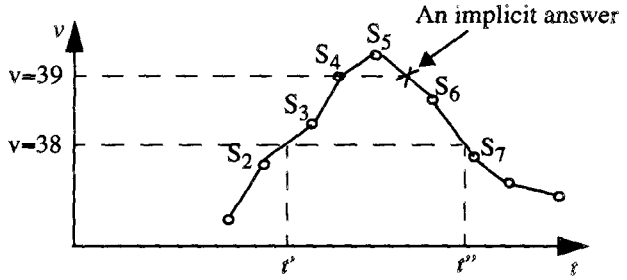

Fig. 4.1: Comparing the IP-index with a conventional secondary index

1. When did the patient have the temperature 38°C?

   A conventional secondary index will return *nil* since there are no *explicit* values equal to 38. By contrast, by using the IP-index we will get <t', t''>.

2. When did the patient have the temperature 39°C?

   A conventional index will only return t_4 (suppose v_4 = 39), while the correct answer (if we want to support the interpolation assumption) should include an implicit point as well that is between S_5 and S_6 (marketed in Fig. 4.1).

3. During what time period did the patient have the temperature higher than 38°C (i.e., have a fever)?

   By using the IP-index, this query will return the time interval (t', t'') (Section 3.3). There is no way to return this interval by using conventional indexes since t' and t'' are implicit.

   Now let us drop the "continuous" assumption and assume that the time sequence is discrete. Then the answer of this query would be [t_3, t_6], where *no implicit time points* are involved any more. It seems that the conventional secondary index would work now. Well, it returns a set of discrete states (S_3, S_4, S_5, S_6) (since these states have values greater than 38). To group these states into the time interval [t_3, t_6] is not a trivial task, especially when the answer is *several* intervals (Fig. 3.2) for large time sequences.

To conclude, the IP-index has the following advantages over conventional secondary indexes:

1. The IP-index supports not only explicit values but also implicit values. This is achieved by the concept of the anchor-state sequences, A(v').
2. The IP-index keeps the *ordering* semantics of the original time sequence. The S_is in the A(v') are ordered by time as they are in the original time sequence. A conventional secondary index destroys the ordering of the original TS.
3. For range queries (v > v') on a TS, the IP-index is needed for efficiency *regardless of whether interpolation is required or not.*

5 Experiments

To investigate the behaviour of the IP-index with respect to the properties of the time sequence, and measure the performance of the σ* operator, we implemented the IP-index in the object-oriented database system SHORE [6]. The reason why we did not use a relational database system is that, as pointed out by Stonebraker [25], it is not a good choice to implement a time sequence as a relational table due to space and efficiency reason.

5.1 Implementations Notes

The reason why we chose SHORE is that a recent paper by Seshadri [20] demonstrates that a SHORE array of records is a good choice of physical implementation of sequential data. Therefore, we chose to implement the time sequence TS as an array of records (t_i, v_i) in

SHORE. For simplicity (without affecting the performance) we use integers *i* (4-bytes) to store the time stamp t_i (instead of using the SQL time stamp value such as "1997/20/01"). The v_is are stored as 4-bytes floating point numbers.

The IP-index is implemented as a B+-tree in SHORE. The keys in the B+-tree are the floating numbers v_is and each key is associated with a pointer to its anchor-state sequence. The anchor-state sequences are implemented as arrays of integers (not arrays of records (t_i, v_i)). For example, if A(v') = $<S_1, S_6, S_{10}>$, then <1, 6, 10> (an array of integers) is stored. There are two reasons for this: 1) We only store (t_i, v_i) in the original time sequence array. It will be redundant to store (t_i, v_i) in every A(v'). 2) The anchor-states only indicate the positions in the TS where to apply *ifn*. To apply *ifn*, all neighbour states (see *surrounding_states*(S_i) in Section 3.1) need to be retrieved from TS (so it does not help if (t_i, v_i) is stored duplicated in A(v')).

Since anchor-state sequences are expected to be of dynamic length, these arrays are implemented as SHORE large objects which can grow arbitrary large. For further details of implementations, please refer to the report [15]. All measurements were done on a SPARC 20 machine with 64M main memory. The SHORE buffer pool size was set to 40 8K pages.

Both synthetic and real-life time sequences were used in the measurements. The reason for using synthetic time sequences is that we need to control several parameters of the time sequences in order to understand the behaviour of the IP-index with respect to their properties. The reason for using real-life time sequences was to evaluate how the IP-index behaves in reality. We used the real data in most measurements. Synthetic sequences were only used when it was necessary to control the parameters of the TS.

5.2 The Size of the Index tree Versus the Cardinality of TS

The first experiment was to answer the critical question: Since most time sequences are very large, is it *practical* to build IP-indexes for large time sequences with regard to space usage and efficiency issue? Recall the IP-index contains an index tree and many anchor-state sequences. We investigated how the size of the IP-index tree (the number of index entries) and the lengths of the anchor-state sequences, i.e., card(A(v'))s, grow with the cardinality of the TS.

5.2.1 The Time Sequence Used in the Experiments

The time sequence used in this experiment was the real-life pressure sequence in Fig. 1.2 with cardinality 100K and the value range (-0.5, 2.5).

The first 1K, 10K and 100K of the pressure sequence was used to vary the cardinality of TS. The precision of values (v_is) was varied from 0.1, 0.01 to 0.001. An IP-index was built for every combination of the above variations (e.g., the first 1K sequence with precision 0.1, the first 10K sequence with precision 0.1, etc.).

5.2.2 Experimental Results

The sizes of the IP-index trees with respect to the cardinality of TS and the precision of values are plotted in Fig. 5.1. The lengths of A(v')s with respect to the cardinality of TS and the precision of values are plotted in Fig. 5.2.

Fig. 5.1 show that: 1) the lower the precision is, the smaller the index tree will be; 2) for a specific value precision, the size of the IP-index tree (the number of index entries) does not grow much with the cardinality of the TS. (For the precision 0.1 and 0.01 the index tree size stays constantly small regardless of the growth of the time sequence.) The reason for the slow growing of the index tree is that there are repeated values in a non-monotonic time sequence. For a specific precision and value range of v_is, there are a limited number of possible keys in the index tree (Section 3.2.1). This investigation shows that *it is practical to build IP-indexes for large time sequences with regard to space usage*. Meanwhile, since the index tree will generally be small, searching the IP-index to find A(v') will be very fast.

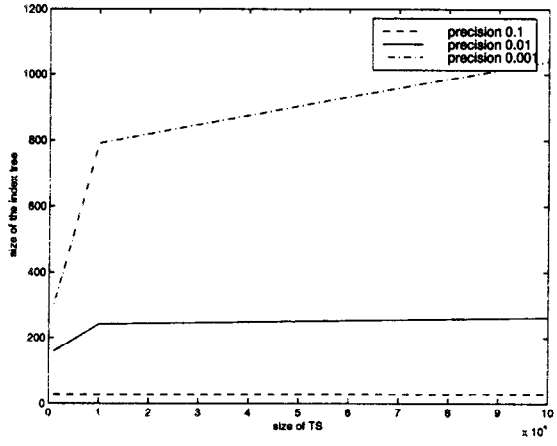

Fig. 5.1: How the size of the index tree grows with the cardinality of TS

Fig. 5.2 shows how card(A(v')) grows with the cardinality of TS. For every precision the *maximum* card(A(v')) was plotted as the worst case behaviour. Maximum card(A(v')) happens when v' = -0.25 where the values are very noisy, as can be seen from Fig. 1.2. The card(A(-0.25)) is 4945 for the 100K pressure sequence, resulting in the ratio of 4945/100K = 5% (worst case). This only happens when the values are very noisy around v'. In most applications the time sequence will generally have much shorter A(v')s, especially in the case of monotonic trend time sequences such as stock prices.

Fig. 5.2 shows that: 1) the lower the precision is, the smaller the maximum card(A(v')) will be; 2) the maximum card(A(v')) grows linearly with the cardinality of the pressure sequence. This is again because of the periodic property of the pressure sequence. The longer the

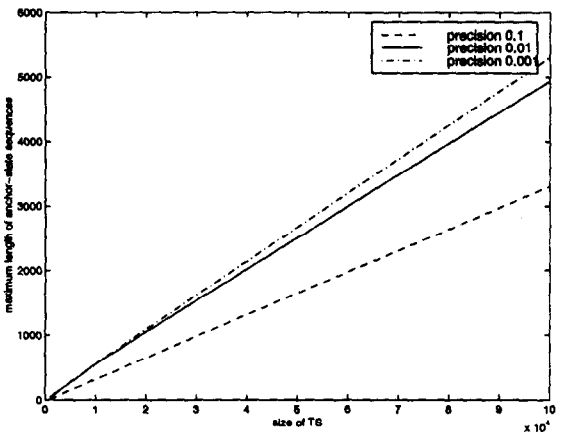

Fig. 5.2: How the maximum cardinality of A(v')s grows with the cardinality of TS

TS is, the more number of segments will probably cross the line v = v' (Section 3.2.1). This indicates that A(v') will normally grow with the size of TS for any value v'.

For the case of long growing TS and A(v')s, the older part of TS (i.e., the part of TS that has time stamps t<t') can be archived (or vacuumed [26]) to tape storage. The corresponding IP-index can be archived easily by copying the B+-tree and archiving the parts of the A(v')s that are inside the time window t<t'.

The Case of Stock Price Sequences

For monotonic trend time sequences such as stock prices, the size of the IP-index tree will be relatively large compared to a periodic time sequence due to the less number of repeated values. By contrast, all anchor-state sequences will then be much shorter than those of periodic time sequences. The overall effect, i.e., the total space usage (the index tree plus the anchor-state sequences) will be generally smaller than that of periodic time sequences.

5.3 $\sigma^*_{v=v'}$ (TS) -- Using the IP-index or Scanning TS

As pointed out in Section 3.2.1, the only way to process $\sigma^*_{v=v'}$ (TS) without the IP-index is to linearly scan the TS. To demonstrate the importance of the IP-index, we compared the time difference between using the IP-index and linear scanning. Recall the operator $\sigma^*_{v=v'}$ (TS) is accomplished by $IP_{v=v'}$ (TS) and *ifn* (Section 3.2). To exclude the time spent in *ifn*, we assume t' = S_i.time (step 2 in Section 3.2) where S_i is returned by the IP operator in step 1. In this case the execution time of $\sigma^*_{v=v'}$ (TS) will exclude the time spent in interpolation, both for using the IP-index and for linear scanning. A detail is that S_i.time is not stored in A(v'); it has to be read from the time sequence array by using the state_id S_i, which is stored in A(v') (see Section 5.1).

5.3.1 Constructing the Synthetic Time Sequence

In order to control the properties of the time sequence used in the experiments, we generated a synthetic time sequence v(i) = m(i) * $sin(k$*i) (i = 1, 2...10K), which is periodic time sequence with *growing* amplitude, see Fig. 5.3. The function m(i) is used to control the v_is so that 1) all v_is are inside a limited value range (it was [-10, 10] in the measurement) and 2) value ranges behave in the "step-wise constant" pattern as shown in Fig. 5.3. The reason for a limited value range is to make the B-tree size limited since we showed in the last section that most real time sequences result in limited size of the IP-index tree. The reason for the "step-wise constant" pattern of value ranges is that it makes it easy to construct different cardinalities of A(v')s by specifying the value of v'. For example, in Fig. 5.3 we have A(1.25) = 2*11 since there are 11 periods of sine data intersect with the line v = 1.25. The smaller the value v' is (v' > 0), the longer the A(v') will be. The maximum card(A(v')) happens when v' = 0. The card(A(0)) was tuned to 2000 in the experiments by the parameter k (by tuning the frequency of the TS). Compared to the cardinality of the whole sequence, 10K, it results in the ratio of 2K/10K = 20%, which is sufficient to model the worst case behaviour since we showed in last section that the worst case of card(A(v')) for the pressure sequence was only 5% of the cardinality of TS, although values are very noisy around v' = -0.25.

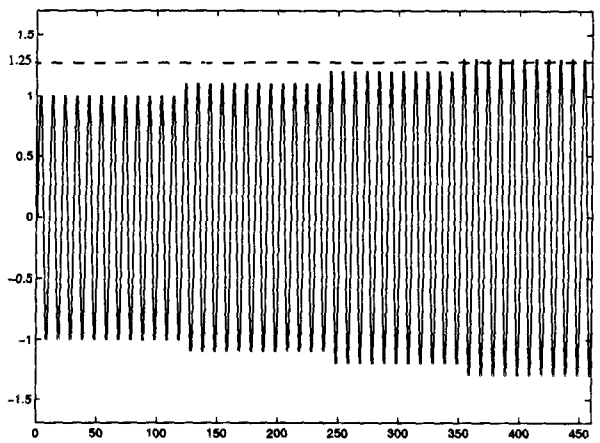

Fig. 5.3: The synthetic sine sequence

5.3.2 Experimental Results -- The Linear Case

We expect that the execution time of $\sigma^*_{v=v'}$ (TS) using the IP-index will be linear to card(A(v')) since the card(A(v')) is the number of states needed to be visited to get the results. By contrast, the execution time of $\sigma^*_{v=v'}$ (TS) using linearly scanning TS will be linear to the cardinality of the whole TS since every state in the TS needs to be visited.

The selected v's and their corresponding cardinalities used in the measurements are listed in Table 1. The execution times of $\sigma^*_{v=v'}$ (TS) with regard to card(A(v'))s are shown in Fig. 5.4. It verifies our "linear" speculation (above). It shows that the execution time of $\sigma^*_{v=v'}$ (TS) by linearly scanning TS is the same for any value v', no matter how long the A(v') is. By contrast, the execution time of $\sigma^*_{v=v'}$

Table 1: Selected v's and the cardinalities of A(v')s

v'	9.4	9.2	9	8.4	7.3	4.9	3.0	0
cardinality	14	60	106	246	504	1064	1508	2000

Table 2: Selected v's and the positions where they first appear in the TS

v's	1.0	2.9	5.1	7.3	8.5
first appears in position	122	2342	4912	7482	8882

by using the IP-index is linear to card(A(v')). Thus, *the smaller the card(A(v')) is, the more we gain by using the IP-index compared to linearly scanning TS.* Notice that in most real life applications the submitted queries $\sigma^*_{v=v'}$(TS) are normally for *short* A(v')s. For example, in Fig. 1.2 we are interested in those peaks where v > 1.5. Since $\sigma^*_{v>1.5}$(TS) is processed by $\sigma^*_{v=1.5}$(TS) (Section 3.3), the execution time is determined by the cardinality of A(1.5), which is then only 80 for the 100K time sequence, resulting in the factor of 80/100K = 0.08%. In this case the time difference between using the IP-index or not is dramatic.

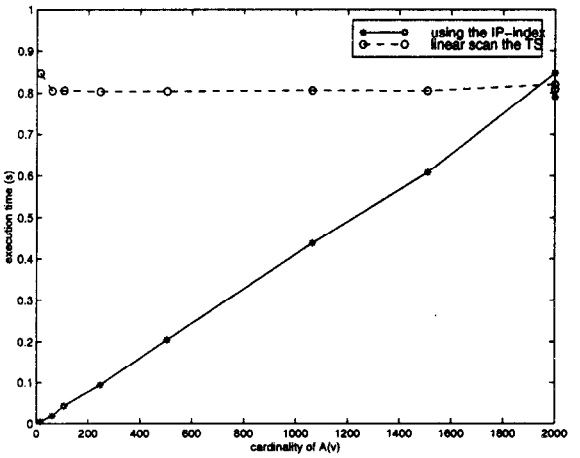

Fig. 5.4: The execution times of $\sigma^*_{v=v'}$ (TS)

Another interesting observation is that for the card(A(v')) = 2000 (i.e., v' = 0), the query processing time of $\sigma^*_{v=v'}$ (TS) by using the IP-index is approximately the same as linearly scanning TS -- we do not gain anything any more. The reason is that to retrieve those S_is whose state_ids are in A(0), all disk pages storing the TS have to be visited since those S_is are evenly distributed in the disk pages that store the TS (page divisions for the TS are illustrated in Fig. 5.5). The cardinality of the anchor-state sequence is then 20% (2000/10K) of the cardinality of the original TS. The threshold of 20% is dependent on the page size, of course. The bigger the page size is, the smaller the threshold will be.

5.3.3 Experimental Results -- The Non-Linear Case

More investigations show that the nice "linear" property of the IP-index in Fig. 5.4 is only valid when the states in A(v') tend to reside in the same page, as the sine sequence does. Fig. 5.5 illustrates this. Suppose that the portion of the sine sequence in Fig. 5.5 (defined over the time interval [0, 460]) occupies 4 pages, then all the states in A(1.25) will reside in the same page (the last page). And all states of A(1.20) will reside in two pages. In this case the number of pages visited is linear to the cardinality of A(v'). In reality most time sequences do not have this nice property. States in A(v') are "scattered" in different pages instead of clustered together. For example, states in A(1.5) in the pressure sequence (Fig. 1.2) are scattered instead of clustered. In this case the execution time of $\sigma^*_{v=v'}$ (TS) using the IP-index will not be linear to the cardinality of A(v'), instead it will be linear to the number of disk pages visited. We tested the $\sigma^*_{v=v'}$ (TS) on the pressure sequence in Fig. 1.2 with cardinality 100K and precision 0.01. The results are shown in Fig. 5.6. What is surprising is that the execution time of $\sigma^*_{v=v'}$ (TS) for shorter A(v')s can be bigger than the execution time of $\sigma^*_{v=v'}$ (TS) for longer A(v')s. This indicates that to estimate the cost of $\sigma^*_{v=v'}$ (TS) using the IP-index, we need to have knowledge of the distribution of those S_is in A(v') in addition to the cardinality of A(v'). In the worst case we have to assume every S_i in A(v') resides in a different disk page.

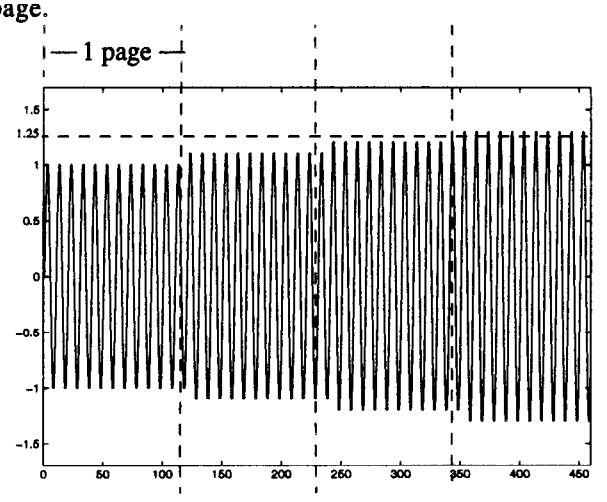

Fig. 5.5: The page division of a portion of the sine sequence

5.4 Getting the First Answer

We also measured the time to get the first answer of

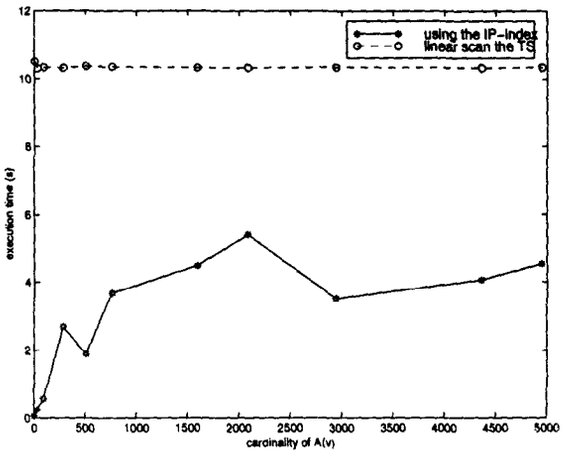

Fig. 5.6: The execution times of $\sigma^*_{v=v'}$ (TS) for the pressure sequence

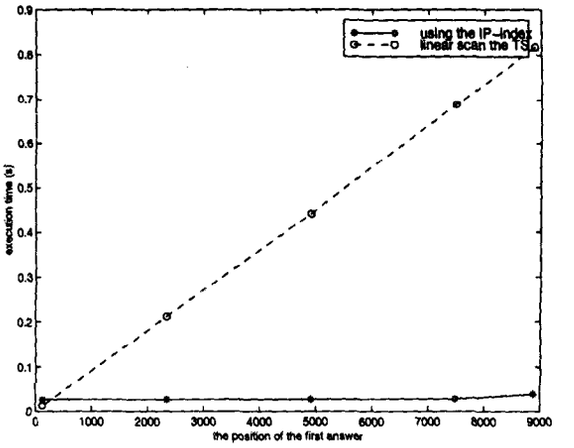

Fig. 5.7: The execution times of the first answer of $\sigma^*_{v=v'}$ (TS)

$\sigma^*_{v=v'}$ (TS) by using the IP-index, compared to linearly scanning TS. As mentioned in Section 3.2, it is important to get the first answer *quickly* in real-time query processing.

5.4.1 Constructing the Experimental Data

By using the synthetic sine sequence it is easy to simulate the situation when the first answer appears in different positions in the time sequence. The selected v's and the positions where they first appear in the TS (i.e., the state_id of the first state in A(v')) are listed in Table 2.

5.4.2 Experimental Results

The execution times of getting the first answer of $\sigma^*_{v=v'}$ (TS) with regard to the position of the first answer appears in the TS are shown in Fig. 5.7. It shows that by using the IP-index the time to get the first answer is constant regardless of the position of the first anchor-state (because the first state_id in A(v') indicates where to retrieve the state S_i in TS). By contrast, the time for linear scanning to get the first answer can be very slow when the first anchor-state appears late in the TS.

The conclusion is that it is essential to have the IP-index in real-time query processing.

To conclude, in this section we have measured the performance of $\sigma^*_{v=v'}$ (TS) with respect to the properties of time sequences. Measurements on range queries $\sigma^*_{v>v'}$ (TS) are not included since, as we pointed out in Section 3.3, the cost of $\sigma^*_{v>v'}$ (TS) is the nearly the same as the cost of $\sigma^*_{v=v'}$ (TS). In next section we will look at time window queries.

5.5 Time Window Queries

We also measured three different strategies for time window queries that were discussed in Section 3.4. The three strategies are: 1) Scanning A(v') to calculate all t's and check the condition later; 2) binary searching A(v'); 3) reversely scanning A(v').

5.5.1 Constructing the Experimental Data

The time sequences used in the measurements were the sine sequence (Fig. 5.3) and the pressure sequence (Fig. 1.2).

The time window was defined as t>t'. The window size was varied from 100, 500, 1K, 5K to 10K for the sine sequence and 1K, 5K, 10K, 50K to 100K for the pressure sequence. Every window size results in a different number of anchor-states visited (Section 3.4). These numbers are plotted as x-axis in Fig. 5.8 and Fig. 5.9.

5.5.2 Experimental Results

The measurements show that: 1) reverse scanning of A(v') is the most efficient strategy since no extra overhead is needed; 2) binary searching A(v') (to get close to the position of t>t') performs almost as efficient as reverse scanning; 3) when the time window is small, the difference between not searching A(v') and binary search A(v') is dramatic.

The conclusion is that it is very important to optimize time window queries by pushing the condition t>t' into the IP operator (Section 3.4) when the window is small.

Notice that binay search can also be performed on the original TS to process time window queries by finding the position of t' to start scanning TS. But this will always be slower than the strategy 2 above, i.e., binary searching A(v') (or strategy 3 above, i.e., reverse scanning A(v')) since A(v') is normally much shorter than the whole TS.

In summary, we have analysed the behaviour of the IP-index and the performance of the σ^* operator with respect to the properties of time sequences and typical query patterns.

6 Conclusions and Future Work

Time sequences appear in various domains in modern database applications. Research work on time

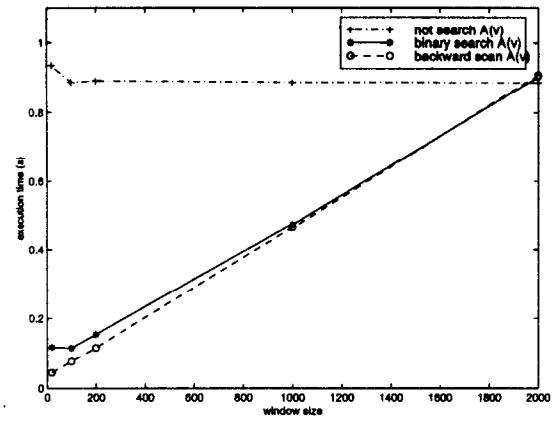

Fig. 5.8: The time window query for the sine sequence (10K)

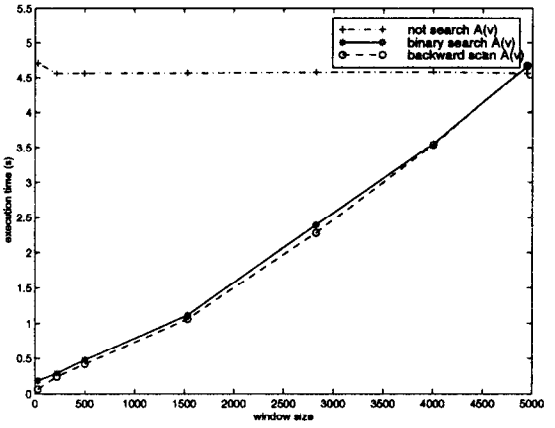

Fig. 5.9: The time window query for the pressure sequence (100K)

sequences has mainly dealt with similarity search, which concerns shapes of time sequences. This paper presents the extended σ* operator to retrieve implicit values from time sequences under various user-defined interpolation assumptions. We have developed the IP-index [13] to efficiently support the σ* operator. The efficiency of the σ* operator for getting all or the first few answers was demonstrated by experiments made on SHORE. The relationship between the behaviour of the IP-index and the performance of the σ* operator with regard to the properties of time sequences were investigated and verified. Possible optimizations of the σ* operator were discussed and verified by experiments. Space usage of the IP-index with regard to the size of the time sequence was analysed to show that it is practical to build IP-indexes for large time sequences.

In a survey by Chomicki on temporal query languages [7], it is argued that the densed temporal domain is very useful in many applications but is difficult to implement efficiently since the set of time instances is very large. The IP-index provides the ability to *derive* the densed instances from the original discrete sequence, saving both storage and query processing time. The actual number of time instances (termed "states" in this paper) needed to be stored are determined by the range and precision of the values in the sequence. Also the sampling frequency can change during different periods, higher frequency can be used for interesting value ranges and lower frequency can be used for uninteresting ranges. Different interpolation functions can also be applied to different subsequences.

We have found several research papers where the σ* operator is needed for sequence data: 1) in [10], for case-based reasoning on event sequences where a "location method" based on some value condition is needed; 2) in [14], for finding the grids in a map whose terrain elevation are inside some value range; 3) in [20], for retrieving sub-sequences in a stock price sequence where the prices are in some range. Also the example query on the pressure sequence [9] used in this paper shows the importance of the σ* operator.

In future work we would like to develop a good data structure for dynamic time sequences [23]. As pointed out by Shoshani [23], a time sequence that is both *dynamic* and *irregular* is the most difficult to be supported physically. The data structure has to be variable length *and* support fast random access in the time domain (fast random access on the value domain is supported by the IP-index). Our current plan is to partition the large TS into arrays (each array fitting in one page) and use a B+-tree to index these arrays. It is also interesting to investigate a good data structure for the anchor-state sequences since they are dynamic and vary much in length. The design goal is not to waste space for small A(v)s and to support fast random access for large A(v)s, which is needed in time window queries.

We will then develop the cost model for the σ* operator based on the new data structure, so that query optimization concerning the σ* operator can be carried out by the database system. Since many extensible database systems (such as Illustra [11] and PREDATOR [21]) support "plug in" of new abstract data types together with their storage, manipulation methods and their indexes, the IP-index, the σ* operator and its cost model can be plugged into those systems to support queries on time sequences or any 1-D sequence data [20]. New functions for time sequences or other 1-D sequence data such as "get_time_stamps(seq_name, '=', v')" and "get_time_intervals(seq_name, '>', v') can be defined and supported efficiently by the IP-index.

We would also like to investigate how to extend the IP-index to two-dimensional time sequences. An example of a two-dimensional application can be found in [14] where an IP-index is needed for a two-dimensional terrain map.

Acknowledgements

The authors wish to thank Martin Sköld for inspiring discussions and valuable comments. Special thanks to Richard Snodgrass in University of Arizona who sug-

gested to implement the IP-index in disk-based database systems and investigate the behaviour of anchor-state sequences. The authors are grateful to Zebo Peng in ESLAB of our department for providing the SPARC 20 machine for measurements. Marvin Solomon in shore_support@cs.wisc.edu answered the authors numerous questions on installing and running SHORE.

References

1. R. Agrawal, C. Faloutsos and A. Swami: "Efficient Similarity Search in Sequence Databases". In *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms*, pp. 69-84, Chicago, Oct. 1993.
2. R. Agrawal, K. Lin, H. S. Sawhney and K. Shim: "Fast Similarity Search in the Presence of Noise, Scaling, and Translation in Time-Series Databases". In *Proceedings of 21st VLDB Conference*, 1995.
3. R. Agrawal, G. Psaila, D. L. Wimmers and M. Zaït: "Querying Shapes of Histories". In *Proceedings of 21st VLDB Confe rence*, 1995.
4. R. J. Bayardo Jr. and D. P. Miranker: "Processing Queries for First-Few Answers", in *Proceedings of 5th International Conference on Information and Knowledge Management*, Rockville, Maryland, 1996.
5. C. Bettini, X. S. Wang, E. Bertino and S. Jajoda: "Semantic Assumptions and Query Evaluation in Temporal Databases", *Proceedings of SIGMOD Conference*, May 1995.
6. M. J. Carey, D. J. Dewitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White and M. J. Zwilling: "Shoring Up Persistent Applications", in *Proceeding of the 1994 ACM-SIGMOD Conference on the Management of Data*, Minneapolis, MN, 1994.
7. J. Chomicki: "Temporal Query Languages: a Survey". In *Proceedings of the International Conference on Temporal Logic*, Bonn, Germany, July 1994.
8. J. Clifford and D. S. Warren: "Formal Semantics for Time in Databases". In *ACM Transactions on Database System*, Vol 8, No. 2, June 1983.
9. Lars Eriksson and Lars Nielsen: "Ionization Current Interpretation for Ignition Control in Internal Combustion Engines". In *IFAC Control Engineering Practice*, Vol. 5, No. 8, August. 1997.
10. M. Jaczynski: "A Framework for the Management of Past Experiences with Time-Extended Situations". In *Proceedings of 6th International Conference on Information and Knowledge Management*, Las Vegas, Nevada, Nov. 1997.
11. Illustra Information Technologies, *Illustra User's Guide*, June 1994.
12. C. S. Li, P. S. Yu and V. Castelli: "HierachyScan: A Hierachical Similarity Search Algorithm for Databases of Long Sequences". In *Proceedings of Data Engineering Conference*, pp. 546-553, Feb. 1996.
13. L. Lin, T. Risch, M. Sköld and D. Badal: "Indexing Values of Time Sequences", in *Proceedings of 5th International Conference on Information and Knowledge Management*, Maryland, Nov. 1996.
14. L. Lin and T. Risch: "Using a Sequential Index in Terrain-aided Navigation", in *Proceedings of 6th International Conference on Information and Knowledge Management*, Las Vegas, Nevada, Nov. 1997.
15. L. Lin: "Implementing the IP-index in SHORE", in *Linköping Electronic Press*, "http://www.ep.liu.se/ea/cis/1997/017/", 1997.
16. A.Segev and A.Shoshani: "Logical Modeling of Temporal Data", In *Proceedings of SIGMOD Conference* May 1987.
17. A. Segev and A. Shoshani: "A Temporal Data Model Based on Time Sequences", in [27], pp. 248-269.
18. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price: "Access Path Selection in a Relational Database Management System", in *Proceedings of ACM SIGMOD'79*, Boston, Massachusetts, 1979.
19. P. Seshadri, M. Livny and R. Ramakrishnan: "Sequence Query Processing", in *Proceedings of ACM SIGMOD'94*, Minneapolis, MN, May 1994.
20. P. Seshadri, M. Livny, and R. Ramakrishnan: "The Design and Implementation of a Sequence Database System", in *Proceedings of the 22nd VLDB Conference*, Mumbai, India, 1996.
21. P. Seshadri, M. Livny, and R. Ramakrishnan: "The case for Enhanced Abstract Data Types", in *Proceedings of the 23rd VLDB Conference*, Athens, Greece, 1997.
22. H. Shatkay and S. B. Zdonik: "Approximate Queries and Representations for Large Data Sequences". In *Proceedings of Data Engineering Conference*, pp.536-545, Feb. 1996.
23. A. Shoshani and K. Kawagoe: "Temporal Data Management". In *Proceedings of the 12th VLDB Conference*, Kyoto, Japan, Aug. 1986.
24. A. Silberschatz, H. F. Korth and S. Sudarshan: "*Database System Concepts*". The McGraw-Hill Companies, Inc. ISBN 0-07-044756-X, 1996.
25. M. Stonebraker: "*Object-Relational DBMSs*". The Morgan Kaufmann Publishers, Inc., ISBN 1-55860-397-2, 1996.
26. M. Stonebraker: "The Design of the POSTGRES Storage System." In *Proceedings of the 13rd VLDB Conference*, pp. 289-300, Sep., 1987.
27. A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev and R. Snodgrass (editors). "*Temporal Databases, Theory Design and Implementation*". The Benjamin/Cummings Publishing Company, ISBN 0-8053-2413-5, 1993.