Indexing Values of Time Sequences

Presented at the 5th International Conference on Information and Knowledge Management (CIKM'96), Rockville, Maryland, November 12-16, 1996

Ling Lin
Department of Computer Science
Linkoping University, Sweden
linli@ida.liu.se

Tore Risch
Department of Computer Science
Linkoping University, Sweden
torri@ida.liu.se

Martin Sköld
Department of Computer Science
Linkoping University, Sweden
marsk@ida.liu.se

Dushan Badal
Department of Computer Science
University of Colorado at Colorado Springs, USA
badal@sunshine.uccs.edu

Abstract

A time sequence is a discrete sequence of values, e.g. temperature measurements, varying over time. Conventional indexes for time sequences are built on the time domain and cannot deal with *inverse queries* on a time sequence (i.e. computing the times when the values satisfy some conditions). To process an inverse query the entire time sequence has to be scanned. This paper presents a dynamic indexing technique on the value domain for large time sequences which can be implemented using regular ordered indexing techniques (e.g. B-trees). Our index (termed *IP-index*) dramatically improves the query processing time of inverse queries compared to linear scanning. For periodic time sequences that have a limited range and precision on their value domain (most time sequences have this property), the IP-index has an upper bound for insertion time and search time.

1 Introduction

In many real-time and temporal database applications the state of a data object *o*, varies over discrete time points, forming a *time sequence* (*TS*). A time sequence can be viewed as a state sequence S_i with $S_i=(t_i, v_i)$, where v_i is the value of the data object at time t_i.

There are three basic characteristics of such time sequences:

1. Time sequences are *ordered*, i.e. $\forall\ i, j: i > j \rightarrow t_i > t_j$.
2. Each value v_i is functionally dependent on the time t_i, but the inverse does not hold.
3. The value v_i can be 1-dimensional (e.g. for temperatures or voltages), 2-dimensional (e.g. for positions in a plane), or of higher dimensionality. In this paper we will concentrate on 1-dimensional data. Our ideas can be extended to multi-dimensional data as well.

Two basic classes of queries on time sequences can be identified:

1. Forward queries, e.g.
   - What was the value at time point *t'?*
   - What was the value range in the time interval [*t'*, *t''*]?
2. Inverse queries, e.g.
   - At what time point(s) t was the value equal to *v'*?
   - In what time interval(s) [*t'*, *t''*] was the value larger (smaller) than *v'*?

Complex queries can be composed by combining these basic queries.

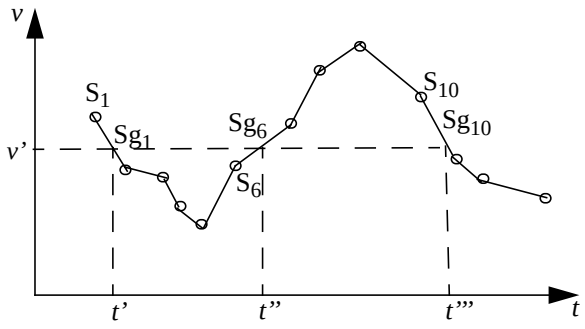

Fig. 1.1: Illustration of inverse queries

For example, if the data in Fig. 1.1 represents a patient's temperature reading over a time period, a forward query could be "What was the patient's temperature at 11:00 yesterday?", and an inverse query could be "At what time period did the patient have a temperature higher than 38°C?". Note that the result of the inverse query is a sequence of time intervals.

Forward queries can be supported by B+-trees[9], AP-trees[12], I-trees[23], or by computational methods[10]. Inverse queries, however, are difficult to support since there can be more than one time point (time interval) where the value is equal to (larger than, smaller than) *v'*. This paper provides an indexing method to efficiently answer inverse queries on *TS*s. The index supports efficient insertions of new states at the end of *TS*s.

The intuition behind our index is illustrated in Fig. 1.1. The *TS* is viewed continuously as a sequence of segments $Sg_i=[S_i, S_{i+1}]$. The time points when the value is equal to *v'* in Fig. 1.1 are <*t'*, *t"*, *t"'*>. These time points can be computed (by interpolation) if we get all the segments Sg_i that intersect the line *v=v'* (i.e., the segments Sg_1, Sg_6, Sg_{10} in Fig. 1.1). We propose an index method that retrieves all the intersecting segments for a value *v'*. This index performs especially well for periodic *TS*s with a limited range and precision on the value domain. We have measured its performance in a main-memory database.

2 Related Work

There has been much research work done on time sequences. Most of it deals with similarity matches [2][17][20], i.e., finding all similar time sequences (or subsequences) that match a given pattern within some error distance. There have been indexes[3][4] and query languages [4] developed to achieve this goal.

Several indexing methods have been proposed for temporal relations[9][12][21][22]. Most of them are intended to support operations like temporal join, temporal selection, etc., and they mostly assume interval time stamps rather than time points.

By contrast our goal is to develop an indexing technique to support inverse queries on *TS*s. This index can be seen as an index on the value domain rather than on the time domain. To the best of our knowledge no work has been done in this area.

Our idea is to transform the problem of inverse queries into k-dimensional spatial search problems, i.e. finding all intervals intersecting a given line. There have been several indexing methods proposed for k-dimensional spatial search, e.g. k-d trees[18], R-trees[13] and SR-Tree[16]. There are also some index trees proposed in computational geometry to deal with interval problems, e.g., Interval Trees[8], and Segment Trees[5]. However, none of the above methods are suitable for inverse queries on *TS*s. The reasons are: 1) *TS*s consist of large sets of intervals $[S_i, S_{i+1}]$ which are dynamically growing, while most spatial data structures assume a fixed search space. 2) The intervals in *TS*s have a special property that the end point of Sg_i is the starting point of Sg_{i+1} (i.e. S_{i+1}). We will show that this property makes our index algorithm much more simple compared to R-trees. Our index method can be built upon a regular ordered one-dimensional index such as B-trees, while R-trees require a complicated algorithm for handling boundary conditions between regions.

Related work can be found in [9] where temporal operations are viewed as interval intersection problems and where B^+-trees are used to index interval time stamps. Another related work[15] views temporal aggregation problems as an interval overlapping problem and then uses the Segment Tree[5] to build an index for computing temporal aggregates.

[19] proposes a temporal data model for *TS*s. It defines four *types* of *TS*s according to what interpolation assumptions are applied, a) Step-wise constant (all values between $[S_i, S_{i+1}]$ are assumed to be equal to v_i), b) Continuous (a curve-fitting function is applied between $[S_i, S_j]$), c) Discrete (missing values cannot be interpolated) d) User-defined (a user-defined interpolation function is applied). Our index can be used to answer inverse queries covering all the above cases.

3 The IP-index

We start with the simplest inverse queries on *TS*s, i.e. "At what time point was the value equal to *v*?", denoted as $F^{-1}(v)$.

A naive way of answering inverse queries is to do curve fitting on the *TS* to generate the function *v=F(t)*, and then solve the equation $t=F^{-1}(v)$. This method is not practical when the *TS*s are long and dynamically growing.

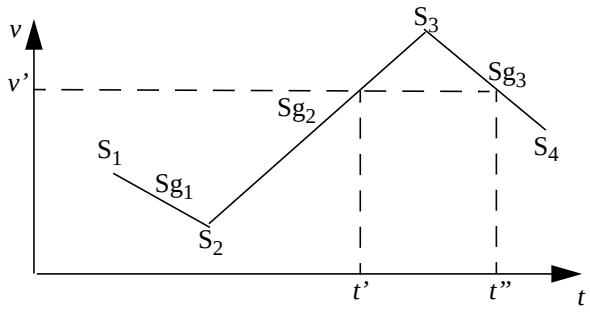

Fig. 3.1: An example *TS* and an inverse query

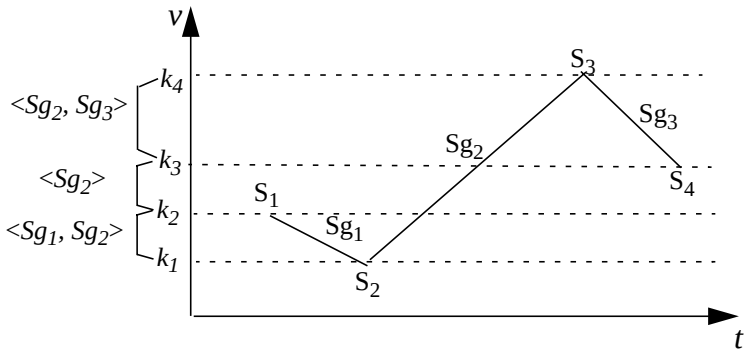

Fig. 3.2: Illustration of the IP-index

We propose a better solution. Each state S_i in the *TS* is viewed as points in the two-dimensional plane *t*-*v* as shown in Fig. 3.1. Then each consecutive states S_i, S_{i+1} constitute a line segment Sg_i. Then, if we can find all segments Sg_i that intersect the line $v=v'$, we can answer inverse queries. For example, in Fig. 3.1, the segments which intersect the line $v=v'$ are $<Sg_2, Sg_3>$. The answer of the inverse query $F^{-1}(v')$ will then be:

- If the "step-wise" constant or "discrete" assumption is applied, then $F^{-1}(v')=nil$, since there is no value defined between S_2, S_3 and S_3, S_4 respectively.
- If the "continuous" or "user-defined" assumption is applied, then $F^{-1}(v')=<t', t''>$, where t' and t'' are calculated by applying some interpolation function (e.g. linear interpolation, least square, etc.) on the states around the segments Sg_2 and Sg_3 respectively.

So, the problem of inverse queries is transformed into the problem of finding all the intersecting segments for the line $v=v'$. A naive way to solve the problem is to scan the entire *TS* to check if any two consecutive states S_i, S_{i+1} "contain" v', i.e. if $v_i < v' < v_{i+1}$, or $v_{i+1} < v' < v_i$. Such an algorithm, however, has the complexity of $\Theta(N)$, where N is the size of the *TS*. Below we propose an indexing technique to perform inverse queries more efficiently.

3.1 The IP-index Definition

If we project each line segment Sg_i on the *v*-axis, we get non-overlapping intervals $I_j=[k_j, k_{j+1})$, where each k_j is a distinct value of v_i (see $k_1 ... k_4$ in Fig. 3.2). We can see that all values that belong to one interval have the same sequence of intersecting segments (marked to the left in Fig. 3.2). Our index associates with each interval $[k_j, k_{j+1})$ all segments Sg_i that span[1] it. It is termed the *IP-index*. A simple illustration of the IP-index is shown in Fig. 3.2, where we associate each interval $[k_j, k_{j+1})$ with the sequence of spanning segments Sg_i.

Since the segments are consecutive, each segment Sg_i is uniquely identified by its starting state S_i. We use S_i to represent the segment Sg_i in the IP-index. We term the starting states of each segments that intersect the line $v=v'$ as the *anchor-states* of v'. Then, the sequence of intersecting segments can be represented as the sequence of anchor-states, which is termed the *anchor-state sequence*. The anchor-state sequence is a state sequence ordered by time.

Since each interval $[k_j, k_{j+1})$ is uniquely identified by its starting point k_j, we use k_j to represent the interval $[k_j, k_{j+1})$ in the IP-index.

Suppose that $k_1<k_2<...<k_j<...$ are the ordered distinct values of v_i in the *TS*. Then each index entry N_i in the IP-index has the form [*key, anchors*] where

- $N_i.key=k_j$.
- $N_i.anchors$ is the anchor-state sequence for all v' such that $v' \geq k_j$ and $v' < k_{j+1}$. It is also denoted as $anchors(k_j)$.

For example, the IP-index for the simple *TS* in Fig. 3.2 is:

$anchors(k_1)=<S_1, S_2>$
$anchors(k_2)=<S_2>$
$anchors(k_3)=<S_2, S_3>$
$anchors(k_4)=nil$

1. We say a segment Sg_i spans an interval I_i when the projection of Sg_i on the *v*-axis spans the interval I_i, i.e. if $Sg_i=((t_s, v_s), (t_e, v_e))$ and $I_i=(v_a, v_b)$, then $v_s \leq v_a$ and $v_e \geq v_b$.

We should point out that if the interpolation method introduces new extreme points (and thus introduces extra segments) to the original time sequence, the IP-index needs to be modified to include the extra segments as well.

3.2 The IP-index Insertion Algorithm

As we mentioned in the introduction, the IP-index supports efficient insertion of new states at the end of *TS*s. Each new state forms a new segment, and this section shows how to efficiently insert a new segment Sg_i into the IP-index.

Suppose that in Fig. 3.2 we have inserted states S_1, S_2 and S_3, and then we want to insert a new state S_4. This means that we already have three index entries in the IP-index with keys v_1 (=k_2), v_2 (=k_1), v_3 (=k_4) respectively, and we also have *anchors*(k_1)=<S_1, S_2>, *anchors*(k_2)=<S_2>, *anchors*(k_4) =*nil*. To insert the state S_4=(t_4, v_4) we need to do the following:

1. The new state S_4 creates a new index entry with the key v_4 (=k_3), which divides the existing interval [k_2, k_4) into two intervals, [k_2, k_3) and [k_3, k_4) (see Fig. 3.2).

   The segments that span the new interval [k_2, k_3) are the same as the segments that spanned the old interval [k_2, k_4) (which are already present in the IP-index), i.e., *anchors*(k_2)=<S_2> stays unchanged.

   The segments that span the new interval [k_3, k_4) are the segments that spanned the old interval [k_2, k_4) plus the new segment Sg_3, i.e.,

   anchors(k_3)= *anchors*(k_2)+[1]S_3 = <S_2>+S_3 = <S_2, S_3>.

2. For all the entries in the IP-index with keys inside the interval [k_3, k_4) (in Fig. 3.2 there happens to be no such key), append S_3 to the end of their associated anchor-state sequences. This is because Sg_3 spans all the sub-intervals inside the interval [k_3, k_4).

The result of the insertion conforms with Fig. 3.2.

So, the insertion of a new state S_i=(t_i, v_i) (i=4 in the above example) into the IP-index has the following steps:

1. If v_i is an existing key in the IP-index, then go to step 4.
2. Search the index entries N_i in the IP-index to find the index entry N_L where
   $N_L.key = max\{(N_i.key) \mid ((N_i.key) \le v_i)\}$.
   This step finds the existing interval (in the example of Fig. 3.2 we have $N_L.key=k_2$ thus the interval is [k_2, k_4)) which the new key (v_4 in the example) lies within.
3. Insert a new index entry with $key=v_i$ and *anchors* $=N_L.anchors$.
4. For all index entries N_j where $N_j.key$ lies inside the interval [min(v_{i-1}, v_i), max(v_{i-1}, v_i)), append the starting state (S_{i-1}) of the new segment to $N_j.anchors$.

3.3 Implementation

Note that the above insertion algorithm is about how to associate the intersecting segments with each inserted value v_i. It does not assume any specific implementation. Actually the IP-index can be implemented by any ordered indexing

1. We use '+' to denote adding a new element to the end of a sequence.

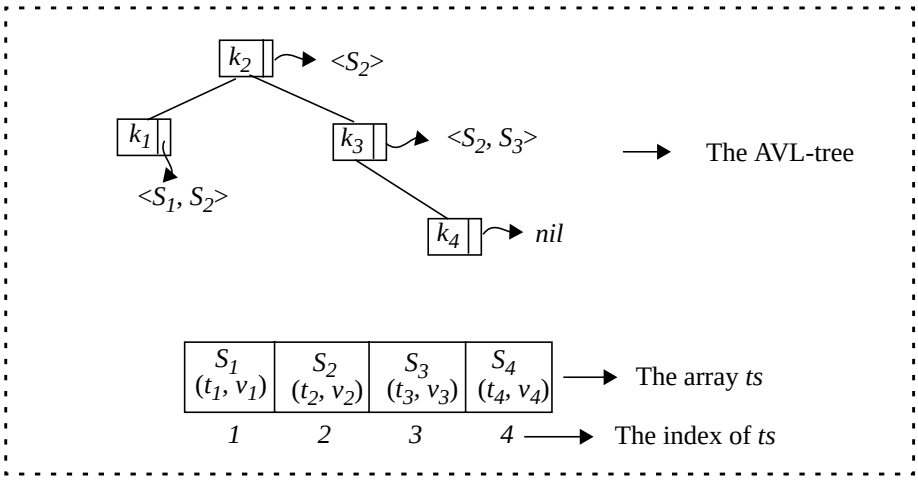

Fig. 3.3: The AVL-tree implementation of the IP-index in Fig. 3.2

technique, e.g., B-Trees, AVL-Trees[1] or 2-3 Trees[7], and the anchor-state sequence can be implemented as a sequential data structure (list or array) which supports fast appending.

To verify our ideas we implemented the IP-index in a main-memory database[11]. The time sequence was stored in an array *ts*, where *ts*[i]=(t_i, v_i). We used an AVL-tree as indexing data structure since it has small re-balancing time. (Notice that the keys v_i do not arrive in order, which means that the tree needs to be re-balanced during insertion.) Each *index entry* in the above algorithm corresponds to a *node* in the AVL-tree. The anchor-state sequence was implemented as a dynamic array of integers, where each integer is an index of the array *ts*.

Fig. 3.3 illustrates the AVL-tree implementation of the IP-index in Fig. 3.2.

Before we give the insertion algorithm for the AVL-tree implementation of the IP-index, we explain the notation and functions that will be used in the algorithm:

- *tree* -- the AVL-tree implementing the IP-index.
- *ts* -- the array storing the time sequence
- S_i=(t_i, v_i) is the arriving state (to be inserted into the IP-index).
- *insert_ts(ts, i, S_i)* -- inserts the state S_i into array *ts* where *ts*[*i*]=(t_i, v_i).
- *insert_node*(*tree*, v_i, *anc)* -- inserts into the AVL-tree a new node with key=v_i, *anchors=anc.*
- *get_lower*(*tree*, v_i): searches the AVL-tree to find the node N_L where

  N_L.*key*= *max*{N_j.*key* | N_j.*key* ≤ v_i, 1≤*j*≤*size(tree)*}

  This function is used to find the existing interval which needs to be split into two parts; e.g. in Fig. 3.2, *get_lower*(*tree*, v_4).*key*=k_2. The function returns *nil* if no node is found.
- *exist_key(tree, v_i)*: returns *true* if there already exists a node in the IP-index whose key is equal to v_i.

The code for the IP-index implementation using an AVL-tree is as follows:

```
Insert_ip(tree,ts,t_i,v_i):
    S_i=<t_i,v_i>
    insert_ts(ts,i,S_i)
      /* insert the state into the array which stores the
         time sequence */
    if not exist_key(tree,v_i)
      N_L=get_lower(tree,v_i)                    (part 1)
      if  N_L=nil
        insert_node(tree,v_i,nil)
      else
        insert_node(tree,v_i,N_L.anchors)
          /* insert a new index entry, and copy the
             anchor state sequence from the
             "lower" index entry */
      endif
    endif
    if i>1
      /* if not the first state in the time sequence */
      for each node N_j                          (part 2)
              (1≤j≤size(tree))
        where N_j.key lies inside the
           interval [min(v_i-1,v_i),
           max(v_i-1,v_i))
        N_j.anchors=N_j.anchors+(i-1)
          /* add the new anchor state to the anchor
             state sequences of all the intervals
             spanned by the new segment */
      end for each
    endif
```

Fig. 3.4: The IP-index insertion algorithm

3.4 Performance vs. Precision of Values

This section discusses the relationship between the performance of the IP-index and the precision of the values in the time sequence.

The algorithm in Fig. 3.4 contains two parts. Algorithm analysis shows that (Part 1) takes $\Theta(LogM)$ time where *M* is the total number of index entries in the IP-index, since they are actually AVL-tree search operations. Furthermore, (Part 2) takes *m*append_time* where *m* is the number of intervals which are spanned by the new segment (*append_time* is the time taken to add the new state to the end of an anchor-state sequence, which is assumed to be constant since we use a sequential data structure which supports fast appending).

So, if we limit the parameters *M* and *m*, we can reduce the insertion time of the IP-index. This can be achieved by limiting the precision of the measured values. The reason is: for a time sequence with range=*R* and precision=*P* in the value domain, the number of index_entries will be less than *R /P.* So, the lower the precision (the larger the value of P) is, the smaller the value of *M* and *m* will be. Thus, we can

reduce the insertion time by limiting the precision of the values, which will be shown in the performance measurement section.

The above observation is practical since 1) all measured time sequences have a limited range on value domain, 2) the original precision of the measured data is always limited due to errors and uncertainty in measurements. For example, when measuring the temperature of a patient the value range is the temperatures that the human body can possibly be alive at and at a precision that can represent changes that affect the well being of the patients. Therefore, even if the thermometer used for measuring the temperature of a patient has the precision of 0.001°C, we can still limit the precision to 0.1°C, which will both improve the performance of the IP-index and still be reasonable for the application.

From the above discussions we can see that the IP-index is not suitable for some unusual time sequences, e.g. periodic time sequences with unlimited precision, or signals which oscillate with an increasing amplitude over time (these two kinds of time sequences have unlimited range or precision, which makes the *M* parameter large). It is also not suitable for those time sequences with "big jumps" in the values all the time (this kind of time sequence makes the parameter *m* large). Fortunately, most time sequences from real applications do not have these properties.

3.5 The IP-index Search Algorithm

To search the IP-index is to find the index entry which contains the anchor-state sequence of the value *v'*, i.e., to search the index entries N_i in the IP-index to find the index entry N_L where

$$N_L.key = max\{(N_i.key) \mid ((N_i.key) \le v')\} \qquad (1)$$

Then $N_L.anchors$ contains the anchor-state sequence for the value *v'*.

In the example *TS* in Fig. 3.1, the index entry N_L for the value *v'* is [k_3, *anchors*] where *anchors*=<S_2, S_3>. The reason is that k_3 (=v_4) is the first key which is "below" or equal to *v'*.

The search algorithm is dependent on the implementation of the IP-index. In the AVL-tree implementation the search algorithm is to search for the node in the AVL-tree whose key satisfies the above condition. It is well known that the complexity is $\Theta(LogM)$ where *M* is the total number of nodes in the tree.

As we discussed in the last section, the value of the parameter *M* is determined by the precision of the values. The lower the precision is, the smaller the value of *M* will be. So, limiting the precision of values will reduce the search time of the IP-index as well.

4 Queries

There are several kinds of queries that can be answered efficiently using the IP-index. Using an example of a patient's temperature reading, we can answer queries like:

- Query 1: When did the patient have the temperature 37°C?

  This query is expressed as $F^{-1}(37)$ and it represents the simplest form of inverse queries. It only requires searching the IP-index to find the anchor-state sequence for the value 37 (plus some post-processing if interpolation is needed).

- Query 2: During what time period did the patient have the temperature higher than 37°C?

  This query can be expressed as $F^{-1}(v>37)$. We refer this kind of query as *inequality inverse queries*. To answer this query we first calculate all time points equal to $F^{-1}(37)$: These time points form a sequence of time intervals. Then for each time interval we check if the values inside the interval are greater than *37* or not. If so, then this time interval is returned.

- Query 3: When did the patient have a temperature *around* 37°C?

  This query can be expressed as $F^{-1}((37-e, 37+e))$, while *e* is a value which is application dependent. This kind of query is useful since many applications need to know "When was the value *approximately* equal to *v'*?" rather than "When was the value *exactly* equal to *v'*?".

  Query 3 can be easily computed given that we can compute Query 2. This is because
  $F^{-1}((v', v'')) = F^{-1}(v>v') \cap F^{-1}(v<v'')$, where '$\cap$' means 'interval intersection' and $v'=37-e$ and $v''=37+e$.

- Query 4: When did the patient have two consecutive fevers during 24 hours?

  This is used for analysing the symptoms of disease[20]. It is an example of shape queries on *TS*s. It can be computed as follows: 1) compute $F^{-1}(v>37)$ (which are the periods of "fever"), 2) check if there exist two time intervals in the "fever" periods that have the distance *d* of *24 hours*. (The distance between two time intervals can be defined either as the distance between the starting points of both intervals or as the distance between the midpoints of both intervals.)

5 Performance Measurements

We tested the performance of the IP-index using the AVL-tree implementation in a main-memory database[11][1].

We measured the insertion time and search time of the IP-index for different kinds of *TS*s. The size (number of states) of each *TS* was 10000.

1. A simulated periodic sequence, sin(t/100) (t=1...10000) with very high precison, plotted in Fig. 5.1.
2. An application time sequence[14] (which is the measurement of pressure in a fluidized bed) plotted in Fig. 5.2.
3. A simulated time sequence with a largely monotonic trend (but not strictly monotonic) plotted in Fig. 5.3.

Fig. 5.1: Sinus Data

Fig. 5.2: Pressure Data

Fig. 5.3: Monotonic Trend Data

5.1 Insertion for Periodic Time Sequences

In Fig. 5.4 and Fig. 5.5 we show the measured insertion times of the IP-index for the time sequences shown in Fig. 5.1 and Fig. 5.2 respectively. The insertion times are measured as the sequences grow.

- The curves labelled “Original Value Insert” show the insertion times of the IP-index. For the pressure data the range=[-6, 10] and the precision=0.001. For the sinus data the range=[-1, 1] and the precision= 0.000001. We can see that the insertion time increases linearly with the size of the sequence. This is because the precision is very high which makes the parameters *M* and *m* (see section 3.4) large.

1. All measurements were made on a HP9000/710 with 64 Mbyte of main memory and running HP/UX.

Fig. 5.4: Sinus Data Insertion

Fig. 5.5: Pressure Data Insertion

- For the curves labelled "Limited Precision Insert" the precision of the values is limited to 0.1 for the pressure data and 0.001 for sinus data respectively. We notice that the insertion time become constant after the total number of index entries has been inserted into the IP-index. This is because a) the limited precision makes the number of nodes of the AVL-tree does not grow any more; only the anchor-state sequence associated with each node grows with the time sequence and b) the limited precision makes the *m* parameter (number of intervals spanned by the new segment as discussed at the end of section 3.4) have an upper limit, which causes the insertion time to have an upper limit (See Fig. 3.4 (Part 2)).

Our measurements verify that for a periodic time sequence with a limited range and precision on the value domain, there will be an upper bound on the IP-index insertion time.

5.2 Search for Periodic Time Sequences

In Fig. 5.6 the IP-index search time is compared with linear scanning of the time sequence to find the anchor-state sequence for some randomly generated value *v'*. The test was done on the simulated periodic sequence with very high precison as plotted in Fig. 5.1. The comparison was measured as the sequence grows. The results show that the IP-index dramatically improves the performance of inverse queries. Note that the results are displayed in logarithmic scale since the difference between the IP-index search time and the linear scanning time is too great to display with linear scaled axes. (Note that the curve labelled "IP-index Search" in Fig. 5.6 is the same as the one labelled "Original Value Search" in Fig. 5.7; they do not look the same because they are displayed in different scaled axes.)

Fig. 5.6: Compare the IP-index with Linear Scanning

Fig. 5.7 and Fig. 5.8 show the performance of the IP-index search for two periodic *TS*s. After every 1000 insertions, we measured the average time to search for the anchor-state sequence for some randomly generated value *v'*. The results show that the search time is logarithmic due to the AVL-tree implementation (see the curves labelled "Original Value Search"). However, when the assumption of "limited range and precision" is satisfied, the IP-index search time has an upper bound regardless of the time sequence size (see the curve labelled "Limited Precision Search"). The reason is the same as in the insert case, i.e., the number of nodes (*M*) of the AVL-tree does not increase after all index entries are inserted, (only the anchor-state sequences associated with each node grow) so the search time stays constant to $\Theta(LogM)$.

Fig. 5.7: Sinus Data Search

Fig. 5.8: Pressure Data Search

5.3 Time Sequences with Monotonic Trends

In Fig. 5.9 we measured the performance of the IP-index for a simulated time sequence with a largely monotonic trend. We see that both the insertion time and the search time are approximately logarithmic due to the AVL-tree implementation. Since in this case we do not have a limited range on the value domain, the "upper limit" on insertion and search time cannot be achieved.

We also notice that a strictly monotonic time sequence does not need any IP-index. The reason is that the value domain is then monotonic just as the time domain is, which means that conventional indexes on the time domain can be applied to the value domain.

Fig. 5.9: Monotonic Trend Data

6 Conclusions and Future Work

This paper presented the IP-index which is an index on the value domain for time sequences. We showed how to use the IP-index to support *inverse queries*, such as finding all the time points when the temperature was equal to a given value *v* (computing $F^{-1}(v)$), or to find the time intervals where the values are greater (smaller) than a given value *v'* (computing $F^{-1}(v>v')$ or $F^{-1}(v<v')$).

The IP-index can be implemented using any ordered index structures, such as B-trees. The performance measurements showed that the IP-index radically improves the processing time of inverse queries on time sequences, compared to linearly scanning the sequence (the only alternative without the IP-index). For a periodic time sequence with a limited range and precision on the value domain, the IP-index insertion and search time have an upper bound regardless of the size of the sequence. Furthermore, by limiting the precision of the values the IP-index insertion and search times can be dramatically improved.

In future work we will investigate how to use the IP-index in query optimization. For example, we can define the cost models for the IP-index and store the cardinality (the lengths of the anchor-state sequences) in each index entry in order to estimate the cost of $F^{-1}(v')$ when the *TS* is very long. We can also set a "moving window" on the anchor-state sequence to discard or archive the old values of the *TS* when they are not required any more.

An interesting improvement is to extend the IP-index for indexing collections of *TS*s [19] based on the composite key *o+v* (*o* is the identifier of each *TS*).

Another future work will be to generalize the IP-index to n-

dimensional *TS*s, e.g. to query the past positions of an aircraft given that the *TS* is a spatial-temporal trajectory of the aircraft.

We also need to explore the IP-index for very large time sequences stored on disk.

7 Acknowledgements

The authors would like to acknowledge Olof Johansson for the valuable discussions which led to the IP-index concept.

References

[1] G. M. Adelson-Velskii and E. M. Landis: *Doklady Akademia Nauk SSSR*, 146, (1962), pp. 263-266; English translation in *Soviet Math*, 3, pp. 1259-1263.

[2] R. Agrawal, C. Faloutsos and A. Swami: Efficient Similarity Search in Sequence Databases, in *Proc. of the Fourth International Conference on Foundations of Data Organization and Algorithms*, Chicago, Oct. 1993, pp. 69-84.

[3] R. Agrawal, K. Lin, H. S. Sawhney, K. Shim: Fast Similarity Search in the Presence of Noise, Scaling, and Translation in Time-Series Databases, in *Proc. VLDB, Conf.,* 1995, pp. 490-501.

[4] R. Agrawal, G. Psaila, D. L. Wimmers and M. Zaït: Querying Shapes of Histories, in *Proc. 21st VLDB Conf.,* 1995, pp. 502-514.

[5] J. L. Bently: *Algorithms for Klee's Rectangle Problems*, Computer Science Department, Camegie-Mellon University, Pittsburgh, 1972.

[6] C. Bettini, X. S. Wang, E. Bertino, S. Jajodia: Semantic Assumptions and Query Evaluation in Temporal Databases, *Proc. SIGMOD Conf.,* May 1995, pp. 257-268.

[7] D. Comer: The Ubiquitous B-Tree, *ACM Comp. Surveys*, 11, 2, June 1979, pp. 121-137.

[8] H. Edelsbrunner: *Dynamic Rectangle Intersection Searching*, Institute for Information Processing, Rept. 47, Technical University of Graz, Graz, Austria.

[9] R Elmasri, G. T. J. Wuu and V. Kouramaijian: The Time Index and the Monotonic B^+-tree, in [22], pp. 433-455.

[10] E. T. Falkenroth: Computational Indexes for Time Series, *Proc. of 8th Intl. Conf. on Scientific and Statistical Database Management*, June 1996, Stockholm, Sweden, pp. 18-23.

[11] G. Fahl, T. Risch and M. Sköld: An architecture for Active Mediators, *Proc. Intl. Workshop on Next Generation Information Technologies and Systems*, Haifa, Israel, 1993, pp. 47-53.

[12] H. Gunadhi and A. Segev: Efficient Indexing Methods for Temporal Relations", *Trans. Knowledge and Data Engineering*, Vol. 5, No. 3, June 1993, pp. 496-509.

[13] A. Guttman: R-Trees: A Dynamic Index Structure for Spatial Searching, *Proc. ACM SIGMOD Conf.,* June 1984, Boston, MA.

[14] F. Johnsson, R. C. Zijerveld, C. M. van den Bleek, J. C. Schouten and B. Leckner: *Characterization of Fluidization Regimes in Circulating Fluidized Beds - time series analysis of pressure fluctuations*, Technical Reports, Chalmers Institute of Technology, Sweden, 1996 (submitted for publication).

[15] N. Kline and R. Snodgrass: Computing Temporal Aggregates, *Proc. Data Engineering Conf.*, 1995, pp. 222-231.

[16] C. P. Kolovson and M. Stonebraker: Segment Indexes: Dynamic Indexing Techniques for Multi-Dimensional Interval Data, *Proc. ACM SIGMOD Conf.,* 1991, pp. 138-148.

[17] C. S. Li, P. S. Yu and V. Castelli, "HierachyScan: A Hierachical Similarity Search Algorithm for Databases of Long Sequences", in *Proc. Data Engineering Conf.*, Feb. 1996, pp. 546-553.

[18] K. Ooi, B. McDonell and R. Sacks-Davis: Spatial kd-tree: Indexing mechanism for spatial database, in *IEEE COMPSAC 87*, 1987.

[19] A. Segev and A. Shoshani: A Temporal Data Model Based on Time Sequences, in [22], pp. 248-269.

[20] H. Shatkay, S. B. Zdonik: Approximate Queries and Representations for Large Data Sequences, in *Proc. Data Engineering Conf.,* Feb. 1996, pp.536-545.

[21] H. Shen, B. C. Ooi, and H. Lu: The TP-Index: A Dynamic and Efficient Indexing Mechanism for Temporal Databases, in *Proc. Data Engineering Conf.,* 1994, pp. 274-281.

[22] A. U. Tansel et al.(editors): *Temporal Databases, Theory Design and Implementation*, The Benjamin/Cummings Publishing Company, Inc. ISBN 0-8053-2413-5, 1993.

[23] K. Torp, L. Mark and C. S. Jensen: *Efficient Differential Timeslice Computation*, Technical Report, College of Computing, Georgia Institute of Technology, Georgia, USA, Sept. 1994 (submitted for publication).