Department of Informatics, University of Zürich

BSc Thesis

Implementing an Index Structure for Streaming Time Series Data

Melina Mast

Matrikelnummer: 13-762-588

Email: melina.mast@uzh.ch

August, 2016

supervised by Prof. Dr. Michael Böhlen and Kevin Wellenzohn

University of Zurich^{UZH}

Department of Informatics

Acknowledgements

Abstract

...

Zusammenfassung

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

The thesis presents a way to implement the described data structures after discussing the requirements. Furthermore, it documents the out coming experimental results. In the end of the thesis, in Chapter 8, the findings will be summarized and concluded.

1.1 Thesis Outline

2 Background

A streaming time series s is a unbounded sequence of data points that is continuously extended, potentially forever. Streaming time series are relevant to applications in diverse domains for example in finance, meteorology or sensor networks. All domains have applications that need to be fed continuously with the latest data e.g. the financial stock market or the weather information. But the processing of large volumes of time series data is impractical. Therefore, a system can only keep a limited size of data in main memory.
The data that is kept in main memory needs to be limited to just a portion of the streaming time series. Besides, in order to be practical for a application like the financial stock market, the data that arrives in a defined time interval (e.g. every 2 minutes) needs to be completely processed until the succeeding data arises.

2.1 TKCM

A streaming time series is not always gapless. E.g due to sensor failures or transmission error, values can get missing. Wellenzohn et al.[1] presents a two-dimensional query pattern over the most recent values of a set of time series to efficiently impute missing values. The two-dimensional query pattern $P_{\bar{t}}$ is defined with *l* reference time series on the spatial dimension and a time window of length *p* on the time dimension. The idea is to derive the missing value from the *k* most similar past pattern. Therefore, it determines for each *time series* a set of highly correlated *reference time series* which represent similar situations in the past e.g. similar weather situations. The value $\hat{s}(t)$ that is calculated as the average of the values $\{s(t)|t \in T_k\}$ will be imputed. Hence, TKCM is able to calculate an estimation of a missing value in streaming time series data.

TKCM must not only insert missing values, but also process the newest arriving values efficiently. In order to do that, TKCM must provide an insertion method for new arriving values to insert the new value into the time window W. Since the time window has a limited, given size $|W|$, an old value has to be deleted for the new arriving value. Provided that, the oldest time *t* does no more fit into the time window because the window is already full.
Further, TKCM must be able to handle duplicate values. For example, if the time window contains 100 temperature values from the same weather station and every 5 minutes a new value arrives. It is likely that the same temperature value arrives multiple times. Besides, the most similar base time values for a given value *v* should be efficiently found and returned.
These assumptions can be made for the implementation of the index structure for streaming time series data.

2.2 Access Methods

TKCM initializes a set $T = \{\}$. The set is filled during execution with all time points *t* for which pattern P_t has already been compared to the query pattern $P_{\bar{t}}$. Besides, TKCM initializes a set $T^* = \{\}$ that contains the *k* time points $t \in T$ that minimize the error $\delta(P_t, P_{\bar{t}})$. Therefore, $T^* \subseteq T$ is always true during execution.
TKCM uses two methods for accessing any time series $r \in S$, *random* and *sorted* access. The two methods are defined as follows:

Definition 2.2.1 *Random Access. Random access returns value r(t), given time series r and time point t.*

Definition 2.2.2 *Sorted Access. Sorted access returns the next yet unseen time point $t_s \notin T$ such that the value $r(t_s - o)$ is most similar to a given pattern cell $P_{\bar{t}}^{r,o}$. $t(s)$ is defined as:*

$$t_s = \underset{t_s \in W \setminus T}{\operatorname{argmin}} |r(t_s - o) - P_{\bar{t}}^{r,o}|$$

After T and T^* is initialized, TKCM iterates until set T^* contains the k time points t that minimize the difference $\delta(P_t, P_{\bar{t}})$.
Using the sorted access mode, the algorithm loops through the cells $P_{\bar{t}}^{r,o}$, reading the next potential time point $t_s \notin T$. The time point $t_s \notin T$ is added to T. The time point t_s has a corresponding patter P_{t_s} which is at least for one pattern cell similar to the query pattern $P_{\bar{t}}$. The random access mode is used to look up the values that pattern P_{t_s} is composed of. After each iteration a threshold τ is computed. The threshold τ is a lower-bound on the error $\delta(P_{t'}, P_{\bar{t}})$ for any time point $t^{'}$ that is yet unseen. Therefore, during the execution of the algorithm $\forall t^{'} \in T : \tau \leq \delta(P_{t'}, P_{\bar{t}})$ is valid. Informally this significances that the lower-bound is always smaller or equal to the error between pattern $P_{t'}$ and query patter $P_{\bar{t}}$ for all time points $t^{'}$ that are elements of T. Once $\forall t \in T^* : \delta(P_t, P_{\bar{t}}) \leq \tau$ the algorithm terminates. At the end, $T^* = T_k$.

3 Problem Definition

The present thesis tries to introduce an efficient way to implement the $random$ and $sortedaccess$ methods described in Section 2.2 for a streaming time series s.
Let $W = [\underline{t}, \bar{t}]$ be a sliding window of length $|W|$. Time $\underline{t}$ stands for the oldest time point that fits into the time window and $\bar{t}$ stands for the current time point for which the stream produced a new value. Besides, consider a set $S = \{s_1, s_2, ...\}$ of streaming time series. The value of time series $s \in S$ at time *t* is denoted as $s(t)$. Only the values in the time window W are kept in main memory. However, we assume that all the time points $t < \bar{t}$ have a time series *s* that is complete. Hence, $\forall t < \bar{t} : s(t) \neq NIL$ since *s* contains imputed values if the real ones were missing.

3.1 Operations

The system presented in the present thesis needs to efficiently perform on the streaming time series s in a sliding window $|W|$:

- shift$(\bar{t}, v)$: add value *v* for the new current time point $\bar{t}$ and remove value *v'* for the time point $\underline{t} - 1$ that just dropped out of time window W.
- lookup(t): return the value of time series *s* at time *t*, denoted by $s(t)$.
- neighbor(v, T): given a value *v* and a set of time points T, return the time point $t \in T$ such that $|v - s(t)|$ is minimal.

The $lookup$ operation is a random access method, while the $neighbor$ operation is a sorted access method.
Wellenzohn et al.[1] suggests a combination of two data structures: a B^+ tree and a circular array. The lookup operation can be performed by the circular array, while the neighbor operation takes advantage of the fact that the leaves of a B^+ tree are sorted.
The approach presented in in Chapter 4 the implementation of the random and sorted access modes using the suggested data structures. Further, it proposes a solution to handle duplicate values.

4 Approach

The lookup operation can be efficiently performed by the circular array, while the neighbor operation takes advantage of the fact that the leaves of a B^+ tree are sorted.
Each time series $s \in S$ can be implemented as a circular array. The circular array is kept in main memory. It uses random access to look up value $s(t)$ for a given time t. Further, for each time series s a B^+ tree is maintained that is also kept in main memory. The B^+ tree is ideal for sorted access by value and therefore for range queries. The data structures are described in detail in Section 4.1

4.1 Data Structures

In the following the circular array and its methods is described. Further, the B^+ tree and the referring methods like the addition and deletion of a measurement is shown.

4.1.1 Circular Array

A circular array is used to store the time series data. The data is assorted by time. Further, the time interval is predefined e.g. every 5 minutes a new value arrives.
The value and time are directly stored in the circular array. The last update position is stored in a variable and updated with every insertion. The circular array is shown in Figure 4.1.

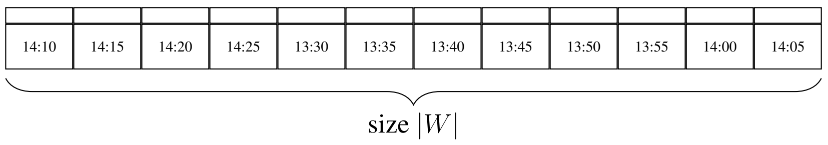

Figure 4.1: Circular array of size $|W|$.

The circular array stores the data, containing all measurement time stamps and values, the size, the last update Position and a counter, which counts the number of measurements added to the array. The addition of a new measurement is presented in Algorithm 4.1.1.

Add a new Value to the Circular Array If the counter of the array is equal or bigger than the size of the array, there is a measurement at the update position in the array that needs to be deleted. If not, there is no need to delete a value from the B^+ tree, since no value is overwritten in the circular array.

Algorithm 1 Update Circular Array

```
void update_CircularArray(BPlusTree *tree, CircularArray *array,
    timeStampT newTime, double newValue){

   int newUpdatePosition = 0;


   //array is not full yet
   if(array->count < array->size){
      if(array->count != 0){
         newUpdatePosition = (array->lastUpdatePosition + 1) %array->size;
      }
      array->count++;
   }

   //array is full -> one value is inserted and one is deleted
   else{
      newUpdatePosition = (array->lastUpdatePosition + 1) %array->size;
      //delete measurement from tree and circularArray
      delete(tree, array->data[newUpdatePosition].time, array->data[
          newUpdatePosition].value);

   }

   //update circularArray
   array->data[newUpdatePosition].time = newTime;
   array->data[newUpdatePosition].value = newValue;
   array->lastUpdatePosition = newUpdatePosition;

   addRecordToTree(tree, newTime, newValue);

}
```

Lookup a Value Due to the properties of a circular array the lookup of a value at time t is very efficient. Since the position can be directly calculated without looping through the array by using the $TIMESTAMP\_DIFF$ representing the interval between two consecutive measurements. The last update point can be used as reference time point for the calculation.

Algorithm 2 Lookup

```
bool lookup(CircularArray *array, timeStampT t, double *value){

   //array is empty
   if(array->count == 0){
      return false;
   }

   //steps from last timestamp to new timestamp
   int step = (int)(t - array->data[array->lastUpdatePosition].time)/
      TIMESTAMP_DIFF;

   //checks if array has enough values inserted for the necessary steps
   if(abs(step) < array->count){

      //chaining modulo for negative numbers
      int pos = (((array->lastUpdatePosition+step)%array->size)+array->
         size)%array->size;

      //value has been found
      if(array->data[pos].time == t){
         //set the pointer to the found value
         *value = array->data[pos].value;
         return true;
      }
   }
   //value was not found
   return false;
}
```

4.1.2 B^+ Tree

A B^+ tree is able to execute range queries very efficiently because the leaves of a B^+ tree are ordered and linked. To perform the *neighbor*(v, T) operation described in Section 3.1 even better, the B^+ tree we use has its leaves linked in both directions. The Section 4.1.2 describes the genaral structure of a B^+ tree. Further, the differences between the B^+ tree used in the present thesis and the B^+ tree presented in [3] are discussed.

The Structure of B^+ trees

A B^+ tree is organized in bocks, as implied by its name. All paths from the root to a leaf have the same length, hence it is *balanced*. There are three types of nodes that may exist in a B^+ tree: the root, interior nodes and leaves as you can see in Figure 4.1. The parameter *n* determines the size of the blocks in the B^+ tree. Hence, the blocks can have maximum *n* search-keys and *n* + 1 pointers, pointing to its child nodes. Every block is between half full and completely full. What can appear in blocks is listed in the following:

- The keys in the leaves are sorted from left to right.
- All pointers in a node point to the level below.
- The interior nodes use at least $[\frac{n+1}{2}]$ of its *n* + 1 pointers. In the root at least 2 pointers must be used.
- The first pointer in a node where the first key is *K* points to a node and hence a part of the tree where the keys are less than *K*. The second pointer points to a node where the keys are greater than or equal to *K*, as shown in Figure 4.2.

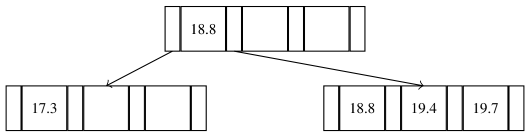

Figure 4.2: Left children keys < 18.8 and 18.8 $\leq$ right children keys.

The B^+ tree used for the implementation of the streaming time series data is slightly different to the B^+ tree in [3]. The required properties are the following:

- The search-keys of the B^+ tree are the temperature values in the sliding window W.
- The leaves are linked in both directions to efficiently perform the *neighbor*(v, T) operation.
- The leaves are sorted by the temperature values.
- The interior nodes have a temperature search-key.
- The leaves store the record of the time series data. A record consists of the temperature value and the associated time value.

Due to weather conditions the temperature values in the time window W can occur several times. Therefore, the keys are not unique. Since the temperature values are used as search-keys, the B^+ tree must be able to handle duplicate values. Section 4.3 proposes different possibilities that allow to use duplicate values in a B^+ tree.
Unlike the traditional B^+tree, where the leaves are just linked to their succeeding leaf, like shown in Figure **??**, the leaves are linked to the succeeding as well as the preceding leaf as

shown in Figure 4.3.

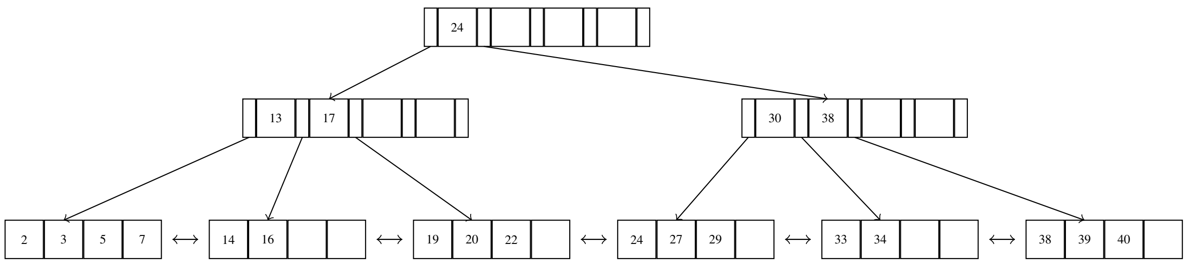

Figure 4.3: A B^+ tree with its leaves linked to the preceding and succeeding leaf.

4.2 Combination of the Data Structures

To efficiently implement the above mentioned operations the system combines two data structures: a circular array and a B^+ tree. In Figure 4.4 the originally proposed data structure in [1] is shown. The B^+ tree is connected with pointers to the circular array and vice versa. Further, the temperature values are used as keys in the B^+ tree. The size of the circular array is defined as $|W| + 1$ since an empty field is used to identify the current update position. The size $|W|$ and the order of the B^+ tree, which defines the size of the nodes, are parameters. Hence they can be changed.
The circular array and the B^+ tree are characterized in the following.

4.3 Duplicate Values in B^+ trees

This Section presents a solution to allow duplicate values in a B^+ tree. Further, the advantages and disadvantages are discussed and compared to other approaches.

4.3.1 Associated doubly, circular Linked List

The idea of this method is to associate a list to the each key that occurs multiple times. So instead of inserting the key again and using another block in the leaf, the new timestamp is just inserted to its associated list. The Figure 4.8 illustrates the needed modification to the originally presented B^+ tree in Figure fig:B+tree.
But it is still possible that the capacity of the array is reached and therefore memory needs to be reallocated. Besides, if a value is deleted from the array, the other values either have to be moved backwards to fill the empty place or the array has to stay just fragmentary filled. Either way is not ideal for a good performance.
Another way to meet the requirements is to use a linked list that contains the timestamps. The advantage against using an array is to add a new element to the linked list without an expensive reallocation of memory. A deletion can be done in $O(1)$ since the first element of the linked

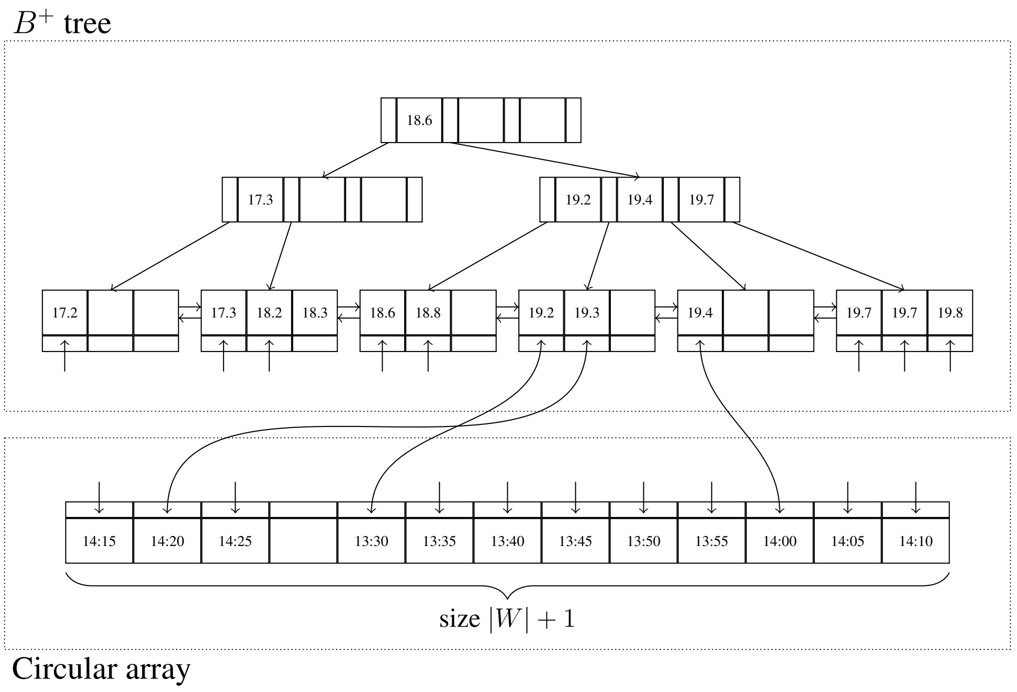

Figure 4.4: Proposed data structures in [1].

list would be the oldest element. But the insertion would cost $O(n)$ because one has to go through the entire list to find the last element.

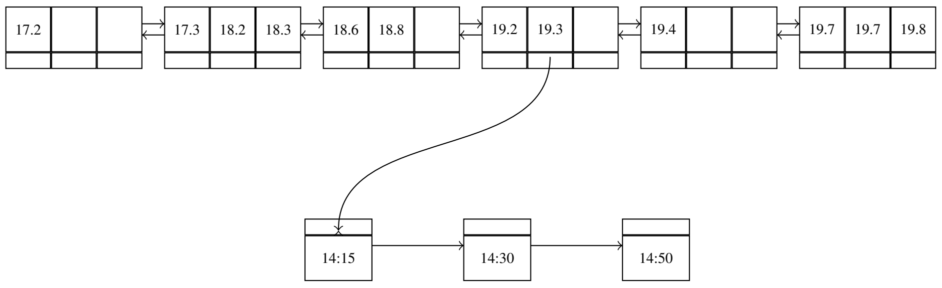

Figure 4.5: Singly linked list

Another idea is to use a circular linked list that is interconnected in both directions as you can see in Figure 4.6. The insertion and deletion would then just cost $O(1)$. But the additional pointers use more memory than a singly linked list. Hence, there is a trade-off between speed and memory allocation when using a linked list.

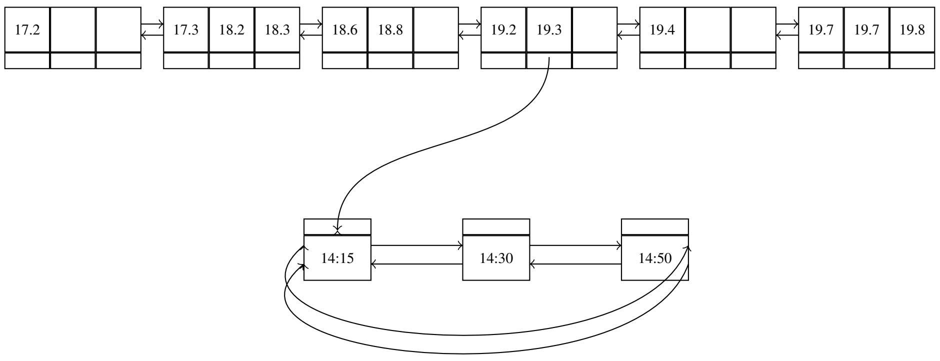

Figure 4.6: Doubly, circular linked list

4.3.2 Alternative Approaches

Create unique Keys

A record in the time series consists of a temperature value and a time value. For each time exists just one temperature value. Therefore the idea is to combine both values and hence always have a unique key.
There are two different actions if the case occurs where you have to add a record with a key that already exists in the B^+ tree. If the leaf is not already full, the record can be added to the existing leaf. But the keys should be reordered according to their temperature. If the leaf has to split the records with equal keys can be separated due to the associated timestamp. If the equal keys are split to different leaves the parent node must new include the same key and the associated timestamp of the first key of the right child leaf. Hence, the two leaves can now still be separated due to the additional ordering with timetamps. The Figure 4.7 illustrates how the above described tree might look.

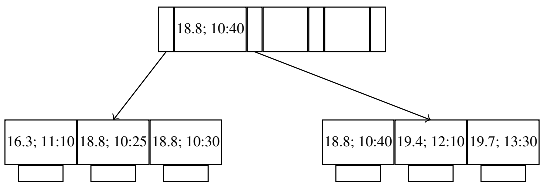

Figure 4.7: Example B^+ tree using a new unique key

Advantages and Disadvantages

(+)

- Implementation is similar to the B^+ tree with no duplicate search-keys.
- The *neighbor*(v, T) operation is equally fast as with no duplicate values.

(−)

- The requirement to allow duplicate values is not meet, since duplicates are just avoided by creating a new unique key.
- More memory needed since the interior nodes must be able to include a timestamp.
- Searching a key is slower than in a B^+ tree with no duplicates, since the timestamp value and the temperature value of a interior node has to be checked. This slows down a record insertion and deletion in the tree.

Associated List

The idea of this method is to associate a list to the each key that occurs multiple times. So instead of inserting the key again and using another block in the leaf, the new timestamp is just inserted to its associated list. The Figure 4.8 illustrates the needed modification to the originally presented B^+ tree in Figure fig:B+tree.

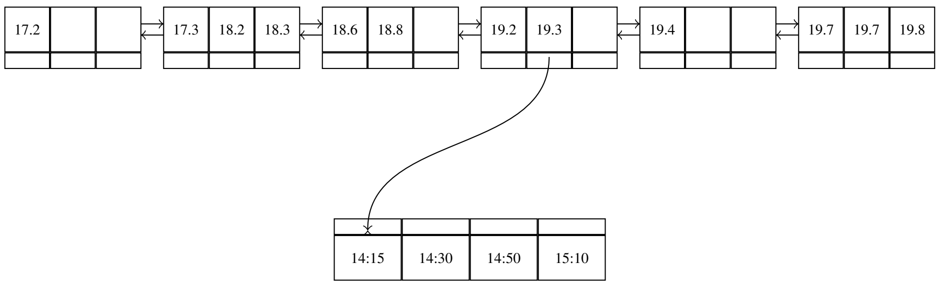

Figure 4.8: Temperature value 19.3 occurs four times in time window W.

Advantages and Disadvantages

(+)

- The *neighbor*(v, T) operation, after v has been found, is very fast if multiple temperatures associated to the same key must be added to T. Because the entire list can be added.
- Searching a key is equally fast as in a B^+ tree with no duplicate values. (Not valid for searching the associated record timestamp)
- A new timestamp can be inserted to the end of the list in $O(1)$ and since the time window W slides forward, the value that should be deleted first from the tree, normally, is at the first position in the list. Therefore, a value can be deleted in $O(1)$ from the list as well.

$(-)$

- The size and capacity of the list has to be stored, which uses more memory.
- Implementation difficulties because the size of an array cannot be dynamically allocated in the programming language C.
- If the array grows because a new timestamps needs to be added memory needs to be reallocated. But reallocating memory and copying the original array is expensive. TODO find better souce than https://en.wikipedia.org/wiki/Linkedlist

The implementation in the programming language C has the challenge that memory cannot be dynamically allocated for a growing array. But since we need a sort of array we can add a size and capacity variable to the referring record. The structure is shown in Listing 4.3.2.

Algorithm 3 Possible structure for a record

```
//Define the record
typedef struct {
      double searchKey;
      int size;          //slots used so far
      int capacity;   //total slotes
      long *data;   //array of timestamps
} Record;
```

Additional Leaves without a Parent Node

Another idea to handle duplicate values is to add additional leaves to the tree that do not have a parent node. As shown in Figure 4.9, the node containing the temperature value *18.3* had been split, since the values did no longer fit into one leaf. The value *18.4* would belong into the same leaf as *18.3* but there is no more space. Instead of splitting the leaf, the additional leaf without a parent is filled up. If e.g. a value *18.5* must be inserted the leaf without a parent must be split. The new leaf would again receive a parent and the old leaf including the duplicate values would stay parent-less.

Advantages and Disadvantages

$(+)$

- The *neighbor*(v, T) operation is equally fast, after v has been found, as with no duplicate values.
- The memory usage is even better if the tree contains leaves with no parent, since the pointer from parent to its child is not needed.
- The memory usage is equal or less than with no duplicate values.

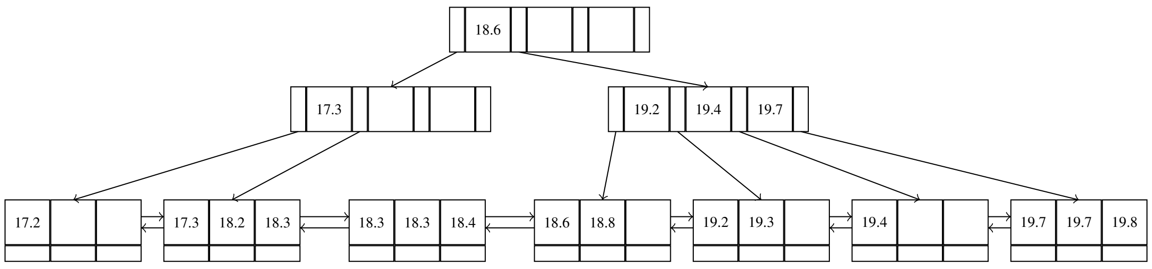

Figure 4.9: B^+ tree with an additional leaf without a parent.

- The deletion of a duplicate value should be equally fast as with no duplicate values in the tree, since the oldest value should be the leftmost duplicate value in a leaf.

$(-)$

- Searching a specific record may take long, depending on the number of duplicate temperature values to the left side of the record.

Null Values in Interior Nodes

The book *Database Systems - The Complete Book* [3] presents an additional approach to handle duplicate values. The definition of a key is slightly different when allowing duplicate search-keys. The keys the interior node $K_1, K_2, K_3, ..., K_n$ can be separated to *new* and old keys. K_i is the smallest new key that is part of the sub-tree linked with the $(i+1)$st pointer. If there is no new key associated with the $(i+1)$st pointer, K_i is set to null.

Example 4.3.1 *The example illustrated in Figure 4.10 illustrates the case, in which the interior node in the right sub-tree K_1 is set to null. The leaf node that pointer $(1+1)$ is associated with contains only duplicate key values which is indicated by setting the interior node K_1 to null. The search-key in the root node is set to 17 because it is the lowest new key in the right sub-tree. Since 13.2 is already in the left sub-tree, the duplicate search-key in the right sub-tree cannot be the key in the root.*
If e.g. 14.2 would have been added to the tree, the leaves must be reordered, since the B^+ tree property that all leaves are ordered from left to right would be hurt.

Advantages and Disadvantages

$(+)$

- The deletion of the oldest duplicate search-key in the tree would be equally fast as with no duplicate search-keys, since the oldest key is the leftmost of all its duplicates.

- The *neighbor*(v, T) operation is equally fast, after v has been found, as with no duplicate values.

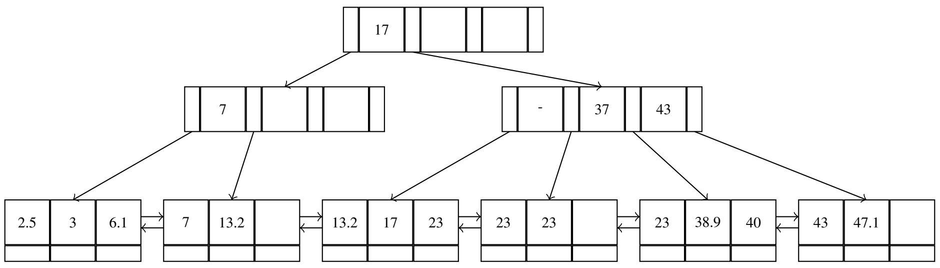

Figure 4.10: Duplicate handling proposed in [3].

$(-)$

- The right sub-tree may also contain keys that are lower than the root key. Therefore the neighbour leaves must be checked as well when searching for a particular key.
- In some cases the leaves have to be reordered.
- In case of duplicate values the neighbour leaves has to be checked as well to find the insertion point for a new key.

4.3.3 Comparison of the Methods for Duplicate Handling

The comparison of the four above described methods especially focuses on how they meet the requirements for the operations described in Section 3.1.
First of all, the *Create unique Key* approach can not be used for the implementation. It does not provide a solution for handling duplicate search-keys, it just avoids to handle duplicates by creating a new unique key.
The *Associated List* method described in Section 4.3.2 can be implemented differently, either using an array, a linked list or a doubly linked list. If it is implemented using an associated array the *neighbor*(v, T) operation is faster the more duplicate values are added to the associated lists. Further, the deletion and the insertion of a value to or from an existing list can be done in $O(1)$. But since the size of the array cannot be dynamically allocated, the memory needs to be reallocated if the array capacity has been achieved. This is very expensive. Using a linked list uses more memory since every temperature stored in the linked list needs more pointer and the insertion method is slower than if a array is used. The doubly linked list can provide a efficient insertion and deletion to the linked list but it uses even more pointer. Therefore, the *Associated List* method is not useful if memory is short. In this case, memory is an important issue, the more memory the B^+ tree needs the smaller the time window W can be to still be kept in main memory.
The other two methods: the *Additional Leaves without a Parent* described in 4.3.2 and the *Null Values in Interior Nodes* described in Section 4.3.2 are more space efficient. Both methods need more or less the same amount of memory if they allow duplicates in the B^+ tree than if they would not. The *neighbor*(v, T) operation is in both methods equally fast, since the leaves contain all records ordered from left to right, no matter if they have a duplicate search-key

or not.*neighbor*(v, T) operation. The deletion of a duplicate search-key, which must be done when the shift$(\bar{t}, v)$ method is called and the time window W is full, is more or less equally fast, except for some special cases discussed in the following. In both methods the oldest duplicate search-key is always the leftmost of all equal search-keys. But if a key k must be deleted that does not have a duplicate search-key but is still part of a node that contains duplicate search-keys, the deletion takes longer in both methods. Since all the duplicates in the left neighbour leaves that occur in the same leaf as k have to be checked before. Another special case is when the leaves need to be reordered after a deletion. This may just be necessary if the method *Null Values in Interior Nodes* is used. For example if 14 is added to the tree illustrated in Figure 4.10 the keys in the leaves would not be ordered from left to right any more. Since another 13.2 search key is part of the right sub-tree. Therefore, the records in the leaves must be reordered to get a right order again.
The main difference between the last both methods is the insertion and search of a record. A new record is inserted when the shift$(\bar{t}, v)$ method is called. In both methods the insertion of a record that has a key that already occurs in the tree, in this case it is a temperature value v that already exists in time window W, is depended on the number of duplicates that are already in the tree. The first potential leaf L_p is found by searching the insertion point for a new record, assuming that the tree can be searched using the B^+ properties listed in Section 4.1.2. After identifying L_p the *Null Values in Interior Nodes* method must check all the right neighbour leaves of L_p, until the last duplicate is found or until a bigger search-key is found. If a bigger search-key is found, it is clear that no other duplicate search-key can occur on the further right part of the tree. After the insertion point is identified the new record can be inserted. The same principle is used in the *Additional Leaves without a Parent* method, but if the right neighbour leaf is not parent-less it is directly clear that the insertion point must be before.
Both methods are at least equally space efficient as a B^+ tree with no duplicate values. In the case a new parent-less leaf has to be inserted, using the *Additional Leaves without a Parent* method, the tree uses even less space than it would with no duplicates, since the pointer from the parent to its child node does not exist.

To conclude, the first method *Create unique Key* is impractical, since it does not handle duplicate search-keys. The second method *Associated List* can have an efficient *neighbor*(v, T) operation but is altogether impractical in terms of space complexity. The last both methods have a equally fast *neighbor*(v, T) operation, after the value v has already been found, as with no duplicate values. But with both methods a deletion, a search or an insertion can take longer than with no duplicate values in some special cases. Any common and special case must be handled before a new value arrives in time window W. Therefore, the time window W can be just as large as the B^+ tree is able to handle a special case in time.
Since special cases e.g. a reordering of leaves is not necessary with the method *Additional Leaves without a Parent Node* this approach is implemented because it meets the requirements the best.

5 Complexity Analysis

5.0.1 Runtime Complexity

5.0.2 Space Complexity

6 Evaluation

memory und runtime evaluation: nodesize, verteilung der daten Datenset erstellen

6.1 Experimental Setup

6.2 Results

6.3 Discussion

7 Related Works

8 Summary and Conclusion

"The conclusion (10 to 12 per cent of the whole research thesis) does not only summarize the whole research thesis, but it also evaluates the results of the scientific inquiry. Do the results confirm or reject previously formulated hypotheses? The conclusion draws both theoretical and practical lessons that could be used in future analyses. These lessons are to be embedded as 2 recommendations for the research community and for policy-makers (note: policy relevance instead of policy prescriptive). In addition, the conclusion gives insights for further research."

Bibliography

[1] K. Wellenzohn, M. Böhlen, A. Dignos, J. Gamper, and H. Mitterer: *Continuous imputation of missing values in highly correlated streams of time series data*; Unpublished, 2016.

[2] Themistoklis Palapanas, Michail Vlachos, Eamonn Keogh, Dimitrios Gunopulos, Wagner Truppel: *Online Amnesic Approximation of Streaming Time Series*; University of California, Riverside, USA, 2004. `http://www.cs.ucr.edu/~eamonn/ICDM_2004.pdf`

[3] Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom: *Database Systems - The Complete Book*; ISBN 0-13-031995-3, 2002 by Prentice Hall