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

A streaming time series s is a unbounded sequence of data points that is continuously extended, potentially forever. They are relevant to applications in diverse domains like in finance, meteorology or sensor networks to name a few. All these applications need to be fed continuously with the latest data. But the processing of large volumes of time series data is impractical because a system can only keep a limited size of data in main memory.
Since streaming time series are unbounded, a system cannot hold the constantly generated data in his main memory endlessly. Therefore, 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.
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 6, the findings will be summarized and concluded.

1.1 Thesis Outline

...First...

2 Background and Related Work

Time series data is not always gapless. E.g. due to sensor failures or transmission errors values can get missing. But since many applications need complete data before further data processing is possible, the missing values need to be recovered first.
The paper *Continuous Imputation of Missing Values in Highly Correlated Streams of Time Series Data* [1] presents a two-dimensional query pattern over the most recent values of a set of time series. The algorithm was developed in the context of meteorology. The *Südtiroler Beratungsring (SBR)* operates a network of weather stations where each station collects continuously (every five minutes) new temperature data. If a value is missing due to sensor failures or other problems, it will be continuously imputed. The imputation needs to fulfill two main requirements:

- The algorithm needs to be efficient enough to complete the data imputation before the new measurement arrives.
- The imputation must rely on past measurements.

The algorithm provided in the paper is called Top-k Case Matching (TKCM) algorithm. It defines a two-dimensional query pattern that contains the measurements of the reference time series in a window of time. It seeks for the k patterns that match the query pattern best. The missing value is derived from the k past pattern. Therefore, it determines for each *time series* a small set of highly correlated *reference time series*. To scan the entire data set to identify the k best is not practical. Hence, only a small portion of the data is scanned. In the following, the data set of the *SBM* is described in Section 2.1. After, the TKCM algorithm presented in [1] is introduced. Finally, the connection to the present thesis is made in Section 2.3 and the implementation is introduced.

2.1 The Data Set

The *SBR* operates 115 weather stations in South Tyrol. Each of the weather stations records temperature every five minutes. 8% of the data set is missing. Missing values often occur in groups either due to transmission problems which lead to smaller missing groups of no more than 5 consecutive missing values or sensor failures which may lead to larger missing groups. 87% of the missing blocks are small and therefore not more than 5 blocks in length. The algorithm is described in Section 2.2.

2.2 TKCM

The implementation of the index structure described in the following thesis is a fundamental part of the TKCM algorithm.
The algorithm is a *k*-Nearest Neighbor Imputation algorithm that uses reference time series to find similar weather situations in the past.
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* e S at time *t* is denoted as $s(t)$. Only the values in the time window W are kept in main memory. If the current value of the time series is missing we write $s(\bar{t}) = NIL$. However, 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. To impute the missing values, reference time series are used. Let $R_s = [r_1, r_2, ...]$ be a sequence of *reference time series* $r_i \in S \setminus \{s\}$ for time series *s*. TKCM chooses for each base time series $s \in S$ the ranked sequence of reference time series R_s. The fundamental idea is to search for a co-occurrence for two time points t, $\bar{t}$ where the reference time series *r* has values $r(\bar{t}) \approx r(t)$, then also $s(\bar{t}) \approx s(t)$. Parameter *l* stands for the number of reference time series that TKCM should consider. R_s is computed using the Case Matching Algorithm further described in the paper. It consists of *l* reference values at time $\bar{t}$. Further, we define R_s^l as the first *l* time series of R_s that have a value $r_i(\bar{t}) \neq NIL$.
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.

Definition 2.2.1 *Query-Pattern. Let p and o denote the pattern length and pattern offset, respectively, where* $p > 1$ *and* $0 \leq o < p$. *Query pattern* $P_{\bar{t}}$ *contains all values of time series* $R_s^l \cup \{s\}$ *in the window of time* $[\bar{t} - p, \bar{t}]$, *except the missing value* $s(\bar{t}) = NIL$. $P_{\bar{t}}$ *is a set of triples* $(r, o, r(\bar{t} - o))$, *such that:*

$$P_{\bar{t}} = \{(r, o, r(\bar{t} - o)) | r \in \{s\} \cup R_s^l \wedge 0 \cup 0 \leq o < p\} \neq \{(s, 0, s(\bar{t}))\}$$

The value of a cell is given by $P_{\bar{t}}^{r,o}$, *hence* $(r, o, P_{\bar{t}}^{r,o}) \in P_{\bar{t}}$.

Example 2.2.1 *A data example for a triple is* $(r_1, 0, 20.5°C)$, *where* r_1 *represents the reference values,* 0 *is the pattern offset* o *and* $20.5°C$ *represents the degree at time* $\bar{t}$. *A pattern example for l = 2 and p = 1 would be:* $P_{\bar{t}} = \{(r_1, 0, 20.5°C), (r_2, 0, 19.5°C)\}$.

The TKCM algorithm searches for the *k* patterns P_t in the time window W that best match the query pattern $P_{\bar{t}}$. The pattern P_t matches the query patter $P_{\bar{t}}$ if for each cell $P_{\bar{t}}^{r,o} \in P_{\bar{t}}$ a corresponding cell $P_t^{r,o} \in P_t$ exists. The error between a pattern P_t and $P_{\bar{t}}$ is the sum of the cell-wise differences in the patterns:

$$\delta(P_t, P_{\bar{t}}) = \sum_{P_{\bar{t}}^{r,o} \in P_{\bar{t}}} |P_t^{r,o} - P_{\bar{t}}^{r,o}|$$

The *k* patterns that minimize $\delta(P_t, P_{\bar{t}})$ are matched by the TKCM. Further, their time points *t* is collected in a set T_k:

$$T_i = T_{i-1} \cup \{ \underset{\substack{t \in W \\ t \neq \bar{t} \wedge t \notin T_{i-1}}}{\operatorname{argmin}} \ \delta(P_t, P_{\bar{t}}) \}$$

The matched time points T_k are used for the imputation. The value $\hat{s}(t)$ that is calculated as the average of the values $\{s(t) | t \in T_k\}$ will be imputed. The calculation is as follows:

$$\hat{s}(t) = \frac{1}{|T_k|} \sum_{t \in T_k} s(t)$$

Example 2.2.2 *If the base time series s has a missing value at time* $\bar{t} = 20:20$ *the value needs to be imputed. The query pattern* $P_{\bar{t}}$ *is defined in this example with* $l = 2$ *reference time series and pattern length* $p = 2$*. Assumed that the* $k = 3$ *most similar matches are* $P_{20:15} = 19.5°C$*,* $P_{20:10} = 19.1°C$ *and* $P_{19:40} = 19.9°C$*, the computation for* $\hat{s}(t)$ *is as follows:* $\hat{s}(t) = \frac{1}{3}(19.5°C + 19.1°C + 19.9°C) = 19.5°C$*. Hence,* $19.5°C$ *is imputed for the missing value at time* $\bar{t} = 20:20$*.*

In conclusion, TKCM is able to calculate an estimation of a missing value in streaming time series data. To achieve this, TKCM uses three parameters, *k, l* and *p*. Parameter *k* represent the number of patterns TKCM matches and collects. In practice $k = [1, 40]$ most useful, since the impact of an exceptional outlier is less aggravated. The parameters *l* and *p* define the size of the two dimensional query pattern $P_{\bar{t}}$. The two dimensions are *l* for space and *p* for time. The accuracy of the imputation value increases as both parameters *l,p* increase, because more relevant query patterns can be matched. But the runtime of TKCM increases with large query patterns. Since the missing values often occur in blocks, as described in Section 2.1, the parameters *l* and *p* generally can be set moderately. E.g. $l \in [3, 5]$ and $p \in [2, 6]$ because most of the missing blocks are no more than 5 consecutive values. If $p \in [2, 6]$, the query pattern $P_{\bar{t}}$ contains at least one past value of the base time series *s* in 87% of the cases.

2.3 Implementing TKCM

The TKCM must not only insert missing values, but also process the newest arriving values efficiently. Therefore, 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 size $|W|$, a old value has to be deleted for the new arriving value. Provided that, the oldest time *t* does not fit in the time window any more if the window is already full. Besides, the most similar base time values for a given value *v* should be efficiently found and returned.
The implementation of these requirements is the main topic of the present thesis. Therefore, the implementation of the TKCM is introduced in the following.

2.3.1 Methods Definition

Retrieving the *k* most similar patterns to a query pattern $P_{\bar{t}}$ is the computationally most expensive part of TKCM.

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

Definition 2.3.2 *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 $P^{r,o}_{\bar{t}}$. $t(s)$ is defined as:*

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

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 $P^{r,o}_{\bar{t}}$, 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, $T^* = T_k$.

2.3.2 Data Structures

The above described access modes can be efficiently performed by combining two data structures. Namely, a B^+ tree and a circular array. 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, TKCM maintains for each time series s a B^+ tree that is also kept in main memory. The B^+ tree is ideal for sorted access by value and therefore for range queries. Unlike the well known B^+tree, where the leaves are just linked to their succeeding leaf, like shown in Figure 2.1, the leaves are linked to the succeeding as well as the preceding leaf as shown in Figure 2.2.

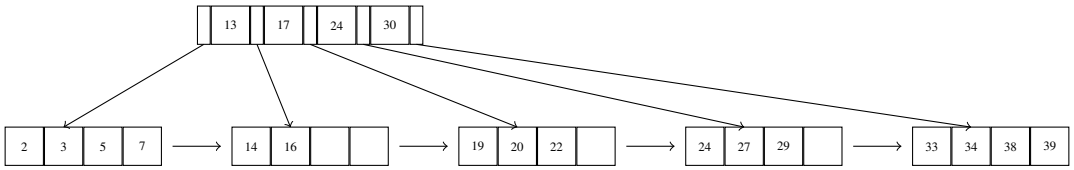

Figure 2.1: Traditional B^+ tree.

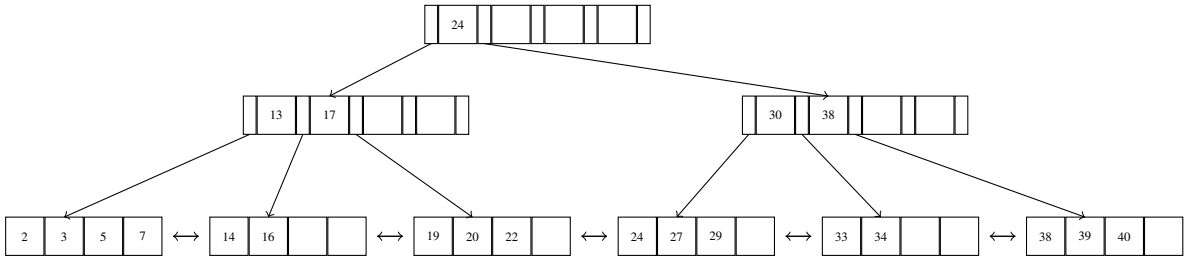

Figure 2.2: A B^+ tree with its leaves linked to the preceding and succeeding leaf.

The implementation of the data structures, as well as the implementation of sorted access and random access, is topic of the present thesis. The requirements and the used data structures are discussed in Chapter 3. In addition to that, the implementation is described in Chapter 4, and finally the implementation is evaluated and its experimental results are discussed in Chapter 5.

3 Requirements

This chapter the requirements that the later described implementation of the index structure must meet are described, in order to be useful for the TKCM algorithm. The operations are introduced in Section 3.1. Further, the data structures are described in Section 3.2 and finally the ...

3.1 Operations

A streaming time series s is a sequence of data points that is extended continuously for example every 5 minutes and eventually forever. Since streaming time series are unbounded, a system can only keep a portion of a time series in main memory. [1] proposes an index structure that meet the requirements for streaming time series data.
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 newest time point for which the stream produced a new value.

The system presented in the present thesis that uses the proposed index structure in [1], namely a circular array and a $B+$ tree, 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 teT such that $|v - s(t)|$ is minimal.

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 and, unlike in a traditional B^+ tree, interconnected in both directions.
Since the system should be integrable in the TKCM algorithm, it needs to be able to handle the Data Set presented in Section 2.1. Therefore, the B^+ tree must be capable to deal with duplicate values, because the same temperature value in a streaming time series may occur several times.

3.2 Data Structures

To efficiently implement the above mentioned operations the system combines two data structures: a circular array and a B^+ tree. In Figure 3.1 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.

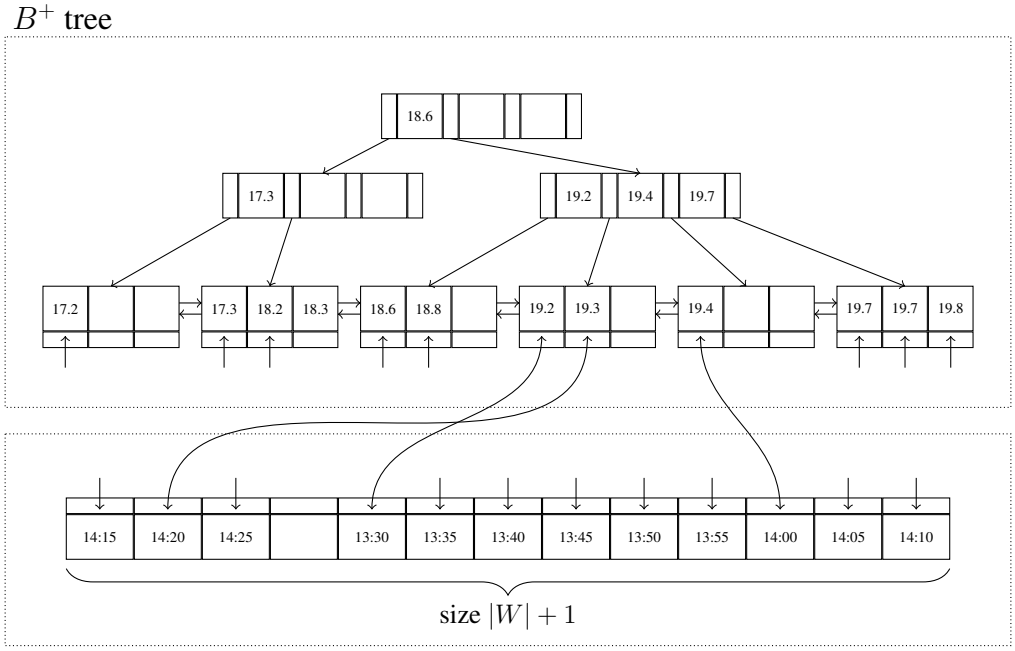

Figure 3.1: Proposed data structures in [1].

3.2.1 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 circular array presented in Figure 3.1 has a size of $|W| + 1$. The empty field is used to identify the next update position. Hence, the position where the next arriving value is inserted. In addition to the time, the associated temperature value is stored in the B^+ tree and linked with a pointer.
But the circular array presented in the present thesis is slightly different. The value must not be associated by a pointer, but can also be directly stored in the circular array. Besides, instead

of an empty field to identify the next update position, the position is stored in a variable and updated with every insertion. The circular array is shown in Figure 3.2

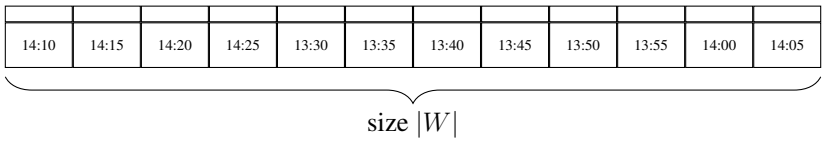

Figure 3.2: Circular array without pointers to the B^+ tree.

3.2.2 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 3.2.2 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 3.2. 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 3.3.

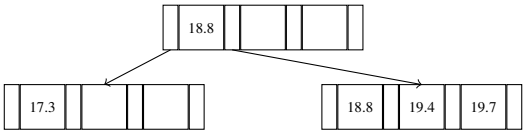

Figure 3.3: Left children keys < 18.8 and right children keys ≤ 18.8 .

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.

Due to the weather conditions the temperature 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 3.3 proposes different possibilities that allow to use duplicate values in a B^+ tree.

3.3 Allowing Duplicate Values

B+ trees are often

3.3.1 Methods Discussion

Add the time to the key

If you would add the timestamps to keys would be unique. So if the case occurs where you have the same keys but in different leaves because the leaves were full and had therefore be splitted you could include the timestamps which divides the two different values to the parent. –> every key with timestamp smaller than the parents timestamp is in the left leaf and every key with timestamp bigger or equal is in the right one –> Problems: more memory needed. always check needed if there is a timestamps in the leaf + neighbor method would still work because you have the value and the set of time stamps

Add several times to the key

The second was to have just one entry for each key, but the 'payload' associated with each key points to a different block of data, which is a linked list pointing to all instances of items that are having the same key

Not change insertion but the search algorithm

There are several different ways of handling duplicate keys. One way is use an unmodified insert algorithm which allows duplicate keys in blocks but is otherwise unchanged. The issue with a structure such as this is the search algorithm must be modified to take into account several corner cases which arise For instance one of the invariants of a B+Tree may be violated

in this structure. Specifically if there are many duplicate keys, a copy of one of the keys may be in a non-leaf block. However, the key may appear in blocks that which appear logically before the block which is pointed at by the key in the internal block. Thus the search algorithm must be modified to look in the previous blocks to the one suggested by the unmodified search algorithm. This will slow down the common case for search.There is another issue with this straight forward implementation, if there are many duplicate keys in the index, the index size may be taller than necessary. Consider a situation were for each unique key there are perhaps hundreds of duplicates, the index size will be proportional to the total number of keys in the main file, however, you only need to index an index on the unique keys. One of the files indexed in our program will be indexing has such characteristics to its data. It indexes strings (as the keys) with associated instances where those strings show up in our documents. There can be hundreds to thousands of instances of each unique string. Therefore the approach I took was to store only the unique keys in the index, and have the duplicates captured in overflow blocks in the main file. An example of such a tree can be seen in figure 2. Consider key 6; there are 5 instances of this key in the tree. The tree is order 3, indicating the keys cannot all fit in one block. To handle this situation an overflow block is chained to the block which is indexed by the tree structure. The overflow block then points to the next relevant block in the tree.To create a structure such as this, the insert algorithm had to be modified. Like the previous version these modifications do not come without a cost, in particular the invariant which states all block must be at least half full has been relaxed. This is not true in this B+Tree, some blocks like the one containing key number 7, are not half full. This problem could be partially solved by using key rotations to balance the tree better. However, there are still corner cases where there would be a block which is under-full. One such corner case includes when a key falls between two keys which have overflow blocks. It must then be in a block by itself, since this B+Tree has the invariant which state that if a block is overflowing it can only contain one unique key. In the future we would like to implement key rotations to help partially alleviate the problem of under-full blocks.

The advantage of this approach to B+Trees with duplicate keys is the index size is small no matter how the ratio of duplicate keys to the total number of keys in the file. This property allows our searches to be conducted quicker. Since the overflow blocks are chained into the B+Tree structure we still have the property of being able to fast sequential scans. One consequence is we have defined all queries on our B+Trees to be range queries. This is fine because all of our queries were already range queries. In conclusion we relax the condition that all blocks must be at least half full to gain higher performance during search.

http://hackthology.com/lessons-learned-while-implementing-a-btree.html

3.3.2 Book

4 Implementation

5 Evaluation

5.1 Experimental Setup

5.2 Results

5.3 Discussion

6 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