# Applying Bit-Wise Dynamic Interleaving in the RCAS Index

Master Basic Module

Institut für Informatik
der
Universität Zürich

Nicoletta Farabullini

Supervisors
Kevin Wellenzohn, Sven Helmer, Michael Böhlen

Zürich
02.06.2020

# Contents

# 1 Introduction

Many data sets within the business and engineering world are semi-structured and hierarchical. This data is commonly queried as a content-and-structure (CAS) type, where a path dimension directs to the location of an attribute and a value dimension contains its content. Commonly used CAS indexes often either prioritize one dimension or create separate indices with respect to each dimension. Both approaches lead to imbalances and large intermediate results. The Robust Content-And-Structure (RCAS) index is a robust index that applies a well-balanced dynamic interleaving to semi-structured hierarchical data with the use of composite keys. It combines path and values dimensions and dynamically interleaves their respective bytes. This approach was proven successful thanks to the application of the dimension-wise discriminative byte for the partitioning and consequent development of a tree-like structure by means of recursion.

The byte-wise RCAS index provides a solid new scheme. However, the partitioning of the path and value dimensions with respect to the discriminative bytes contains between 2 and 256 partitions. The experiments in Wellenzohn et al [**1**] have shown that typically few large partitions are present in the path dimension whereas for the value dimension there are many small partitions. Additionally each node has a variable number of partitions. These imbalances can potentially cause issues in the index performance. A possible solution to this problem is the development of a bit-wise dynamic interleaving in the RCAS index. The goal of this Basic Module was to implement such an approach.

The primary difference with the original schema is the conversion of the path and value dimensions to bits, allowing for only two partitions to be needed in each node: one for the 0 bit and one for the 1; the subsequent index becomes a binary tree. One limitation in this implementation is the assumption that the query descendant axis can only be only present at the end of the path query.

This report presents a step-by-step explanation of this process and a detailed comparison with the original approach. In the future, the query descendant will have to be implemented such that it is accounted for at any position within the path. Additionally the robustness of this new index will have to be tested by performing an experimental evaluation.

# 2 Byte-wise RCAS index

The Robust Content-And-Structure (RCAS) index is a robust index that applies a well-balanced dynamic interleaving to semi-structured hierarchical data and combines path and values dimensions. To support the qualitative explanation of the process, the data set in Table 1 is used:

Table 1: Example data set

| Key number | Path Dimension P | Value Dimension V |
|---|---|---|
| $k_1$ | /bom/item/canoe$ | 00 01 0E 50 |
| $k_2$ | /bom/item/carabinier$ | 00 00 00 F1 |
| $k_3$ | /bom/item/car/battery$ | 00 03 D3 5A |
| $k_4$ | /bom/item/car/battery$ | 00 03 D3 B0 |
| $k_5$ | /bom/item/car/belt$ | 00 00 0B 4A |
| $k_6$ | /bom/item/car/brake$ | 00 00 0C C2 |
| $k_7$ | /bom/item/car/bumper$ | 00 00 0A 8C |

For the purpose of analysis and testing, the path dimension is of type string and the value is a 32 bit unsigned integer shown in hexadecimal. However the latter could be of any other data type.

The first step in constructing the RCAS index is to find the discriminative byte with respect to a chosen dimension, aka the first byte at which the listed keys differ. Looking at Table 1, it is possible to qualitatively observe that the discriminative byte for the path dimension is the 13th character, where $k_1$ presents "n" and the remaining keys "r". For the value dimension, the discriminative byte is the second as $k_1$ has "01" where the remaining keys have "00". Once these two quantities are stored, one dimension is chosen and the keys are split into sub-lists belonging to a partition M. Each sub-list contains keys with the same byte at the position of the discriminative for the chosen dimension. Following up on the same example and choosing the value dimension, the partitioning will result as such:

M = {{k_1}, {k_3, k_4}, {k_2, k_5, k_6, k_7}}

The keys are further sub-divided by performing the discriminative byte operation in each sub-list with respect to the other dimension. Since $k_1$ is the only key present in one sub-list, the iteration for this key is terminated. For the remaining keys, since value was initially chosen, path is the dimension for the next iteration. The discriminative byte for the path for keys 2,5,6,7 is the 14th character, where $k_2$ presents "a" and the remaining keys "/". $k_3$ and $k_4$ have identical paths, hence the dimension is switched back to value and a new discriminative byte is computed. The partitions will result as such:

M = {{k_2}, {k_5, k_6, k_7}}
M = {{k_3}, {k_4}}

This process is performed recursively by alternating the dimension and further filtering the partitioning until all keys are exhausted. If the keys at a dimension are all equal, the discriminative byte is set to a value greater by 1 from the length of the keys and the dimension is switched. The derived structure is tree-like, where the root node contains the first set of discriminative bytes and it expands until the end of each key is in a leaf. A depiction of such tree is shown in the paper Wellenzohn et al [**1**] and imported here in Figure 1.

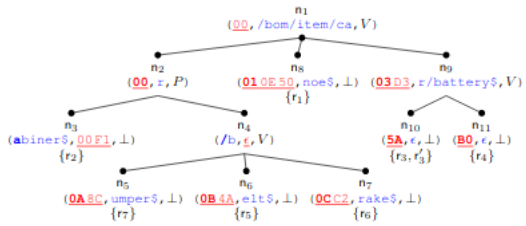


***Figure 1: Byte-wise RCAS index for the composite keys 1-7***

# 3 Bit-wise RCAS index

A bit-wise approach to the RCAS index was conceived as a possible solution to the issues caused by the use of bytes. The related analysis includes the first five keys belonging to the original data set in Table 1 with a simplified version of the paths.

Table 2: Data set for bit-wise implementation

| Key number | Path Dimension P | Value Dimension V |
|---|---|---|
| $k_1$ | "/b/i/canoe$ | 00 01 0E 50 |
| $k_2$ | "/b/i/carabinier$ | 00 00 00 F1 |
| $k_3$ | "/b/i/car/battery$" | 00 03 D3 5A |
| $k_4$ | "/b/i/car/battery$" | 00 03 D3 B0 |
| $k_5$ | "/b/i/car/belt$" | 00 00 0B 4A |

Each composite key points to an item in the database through a reference:

Table 3: References for keys

| Key number | Reference |
|---|---|
| $k_1$ | $r_1$ |
| $k_2$ | $r_2$ |
| $k_3$ | $r_3$ |
| $k_4$ | $r_4$ |
| $k_5$ | $r_5$ |

The initial difference with the original schema is the conversion of the path and value dimensions to bits. The new set of bit inputs leads to a partition M of only two sub-lists: one for the 0 bit and one for the 1. Consequently, the tree becomes of type binary and its structure mutates.

Adapted versions of the corresponding algorithms applied to dimensions and queries are described in the following subsections.

## 3.1 Algorithm 1: Find discriminative bit

The first step in constructing the RCAS index is to find the discriminative bit: the bit at which the path and value attributes differ and assign it to the variable $g$. In other words, this quantity corresponds to the first position within the keys attributes split by dimension where at least one of the bits of an attribute differs from the others.

Table 4 provides the bit-wise conversion of the value dimension.

Table 4: Bit-wise value conversion of Table 2

| Key number | Value Dimension V |
|---|---|
| $k_1$ | 00000000000000010000111010110000 |
| $k_2$ | 00000000000000000000000011110001 |
| $k_3$ | 00000000000000111101001101011010 |
| $k_4$ | 00000000000000111101001110110000 |
| $k_5$ | 00000000000000000000101101001010 |

It is possible to observe that the first discriminative bit is at position 14 (counting is zero based), where $k_3$ and $k_4$ have bit 1.

The algorithm is implemented only taking into account one dimension per time:

1. Import the list of keys, a chosen dimension (D), $g$ set at 0 at the root node
2. Set one key as reference (e.g. the first in the list of keys) and loop over the list of keys to perform a bit-wise comparison:
   (a) If the bits of the first key and the other keys at $g$ are equal, increase $g$ by 1 and proceed with the next bit
   (b) If at any point two bits differ, return the value corresponding to the position

The practical implementation is developed in the $dsc_{inc}$ function.

```
dsc_inc(list of keys, Dimension D, g) {
    while(g < size(first_key.D)) {
        for(another_key in list of keys) {
                if(first_key.D[g] != another_key.D[g]) {
                    return g;
                }
        }
        g++;
    }
    return g;
}
```

It is important to note that if all keys are equal, then $g$ is returned as one integer value higher than the total length of the keys.

## 3.2 Algorithm 2: Compute partitioning

Obtaining the discriminative bit allows to partition the list of keys into two different sub-lists, one for keys having bit value 1 and the other for bit value 0 at position $g$. These two lists are gathered into a partition M and respectively named $M_{bit1}$ and $M_{bit0}$. The related partitioning construction is straight-forward:

1. Select each key separately
2. If the value of the key for the specified dimension at $g$ is 0, place the key in $M_{bit0}$
3. If the value of the key for the specified dimension at $g$ is 1, place the key $M_{bit1}$

The bit-wise application of this algorithm presents a much simpler structure compared to the byte-wise version. In the byte-wise approach, M has to include all possible byte values for a total of 256.

## 3.3 Algorithm 3: Construct the RCAS index

The first two algorithms set the grounds to construct a new version of the RCAS index. Analogously to the M partitioning development, the tree will only present two children for each node as the choice of bit is binary. The function $ConstructRCAS$ recursively generates the nodes. Each node of the tree contains several parameters: the current node sub-strings $s_P$ and $s_V$ (respectively for the path and the value dimensions), a vector of pointers to be filled with the key reference whenever the node is of type leaf, the chosen dimension, the node's children oriented in the left and right directions.

These parameters as well as the $dsc_{inc}$ function are imported and manipulated into the $ConstructRCAS$ function:

1. The $dsc_{inc}$ function is called twice to obtain the discriminative bits for both dimensions, stored in the variables $g'_P$ and $g'_V$.
2. The discriminative bits from the previous iteration are stored in the variables $g_P$ and $g_V$. Note: these values are set to 0 at the root node.
3. $s_P$ is filled with bits located between $g_P$ and $g'_P$ in the path dimension
4. $s_V$ is filled with bits located between $g_V$ and $g'_V$ in the value dimension.

A couple of special situations may derive from the last two steps:

1. If both $g'_V$ and $g'_P$ are greater than the length of the key, the node is set as a leaf, the related reference vector is filled and the recursion is terminated.
2. If only one of the two variables is greater, there are no more bits to analyse in the corresponding dimension, which is switched to the other to process the remaining data.

In case of a regular node, the dimension also has to be switched, e.g. if the dimension of the current iteration is the value, now it has to be changed to the path. The ConstructRCAS function is called recursively until a leaf node is reached. The left and right children of the node are respectively assigned the partition of the 0 and 1 bit:

```
n -> left = ConstructRCAS(M.bit0, Dimension, g'_P, g'_V);
n -> right = ConstructRCAS(M.bit1, Dimension, g'_P, g'_V);
```

Figure 2 shows the tree obtained from this recursion with labeled nodes and the chosen dimensions.

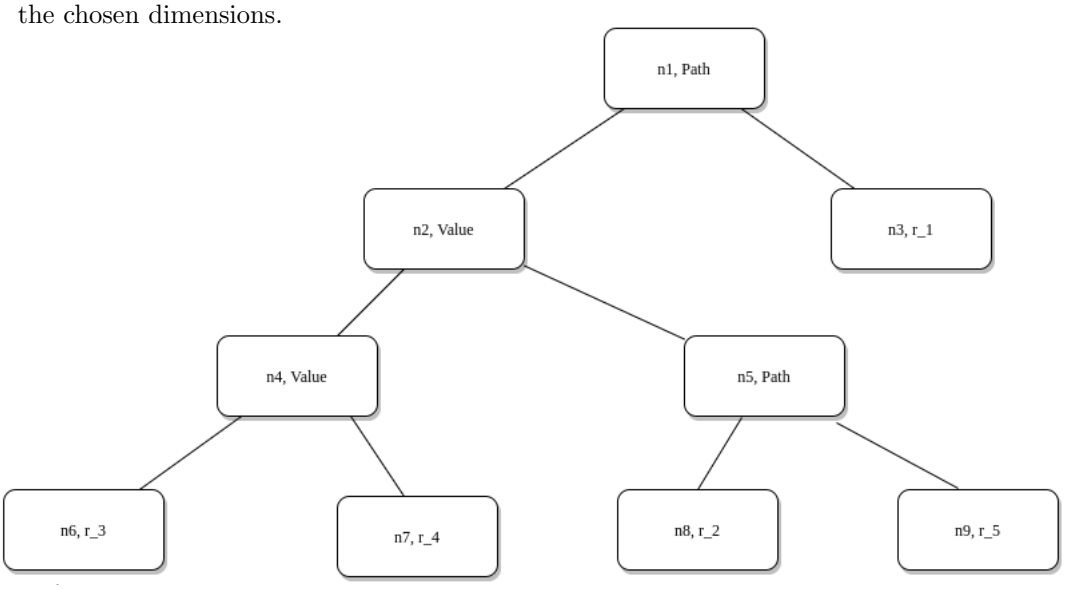


***Figure 2: Bit-wise RCAS index for the composite keys 1-5***

The bits contained in each node are listed in Table 4.

Table 5: Data set for bit-wise implementation

| Node | Path bits | Value bits |
|---|---|---|
| $n_1$ | 00101111011000100010111101101001001011110110001101100001011 | 00000000000000 |
| $n_2$ | 100100 | [] |
| $n_3$ | 011100110111101100101001001 00 | 010000111001010000 |
| $n_4$ | 0101111011000100110000101110100011101000110010101110010011110010010 0 | 1111010011 |
| $n_5$ | [] | 000000 |
| $n_6$ | [] | 01011010 |
| $n_7$ | [] | 10110000 |
| $n_8$ | 1100001011000100110100101101110011010010110010101110010001001 00 | 000011110001 |
| $n_9$ | 0101111011000100110010101101100011101000010010 0 | 101101001010 |

## 3.4 Algorithm 4: Querying the RCAS index

The creation of the binary tree permits the evaluation of a CAS query on the RCAS index. The query has to contain a query path ($q$), a low value ($v_l$) and a high value ($v_h$). The goal in this step is to verify whether the query is a match for the index. In order for a MATCH to be returned two conditions have to be fulfilled:

1. The set key path has to match the query path
2. The set key value has to fall between $v_l$ and $v_h$

The $CasQuery$ function traverses the tree and checks for these conditions in each node. Three are the possible outputs:

1. MATCH: the condition is fulfilled
2. MISMATCH: the condition is not fulfilled
3. INCOMPLETE: more information is needed

The dimensions are initially treated separately in the functions $MatchValue$ and $MatchPath$. The outputs are combined at the end. For each node explored by $CasQuery$, the first step is to append the current bits contained into $s_P$ and $s_V$ to two buffers $buff_P$ and $buff_V$. In other words, $s_P$ and $s_V$ respectively denote the path and value bits contained between the previous discriminative bit up to (but excluding) the current one. Note that for the first iteration, the previous discriminative bit is set to 0. After checking the buffers for each condition, further actions can be performed. If both return MATCH, the recursion is terminated and it is possible to proceed to Algorithm 5. If at least one returns MISMATCH, the recursion is stopped. Otherwise, the $CasQuery$ function is called recursively where at least one predicate returns INCOMPLETE.

The byte-wise approach presents two more variables in the $CasQuery$ function: s for the state information and W to collect results. These two quantities were deemed unnecessary in this implementation.

### 3.4.1 $MatchValue$ function

The $MatchValue$ function is used to query the value predicates and it can output MATCH, MISMATCH or INCOMPLETE. Additionally a low ($lo$) and a high ($hi$) integer variable are set, respectively containing the discriminative bits between $buff_V$ and $v_l$ and $buff_V$ and $v_h$. The method to fill these two variable is analogous to the $dsc_{inc}$ function in Algorithm 1: starting at position 0, if the two bits are equal, the integer variable is increased by 1, otherwise the loop is broken. This case also leads to the possibility of having values greater than the size of the buffers.

A MISMATCH is returned in two scenarios:

1. If $buff_V$ is less than $v_l$ at position $lo$

2. If $buff_V$ is greater than $v_h$ at position $hi$

It is important to check whether the sizes of $hi$ and $lo$ are less than the sizes of $buff_V$ and respectively $v_h$ and $v_l$. The function checks for a MATCH:

1. If the two size scenarios are evaluate to true, $buff_V$ is greater than $v_l$ at position $lo$ and $buff_V$ is less than $v_h$ at position $hi$

2. If we are in a leaf node. This is because the leaf denotes the end of the key, where the remaining entries for the value dimension have already been established not to be a MISMATCH.

In all other cases, $MatchValue$ returns INCOMPLETE.

#### 3.4.2 $MatchPath$ function

A similar logic is applied to the $MatchPath$ function with some differences in the related implementation. Before importing the query path $q$, the check for the query descendant has to be performed. In the general case, the query descendant can be found anywhere in the query path; however for the purpose of this project, only the case where it is placed at the end of the string is considered. In order to check for this feature, a for loop is applied over $q$ to check for the "//" pattern. If it returns TRUE, the last two members of the string (aka the //) are removed and the $query_{descendant}$ variable is set to TRUE, otherwise FALSE.

To check for the equivalence of $buff_P$ and $q$, an integer variable $i$ is set to 0 at the beginning of the function. If $buff_P$ is the same as $q$ at position $i$, it is increased by one (with the condition to not surpass the length of any of the two), otherwise MISMATCH is returned. If the latter is not the case, another set of check is performed. A MATCH is returned if:

1. The size of $i$ is greater or equal to both $q$ and $buff_P$ as it implies that the two strings are identical

2. $buff_P$ has a larger size than $q$ and the $query_{descendant}$ is TRUE

In the case that $buff_P$ has a larger size than $q$ and the $query_{descendant}$ is FALSE, a MISMATCH is returned. Finally, the case where $q$ is larger than $buff_P$ returns INCOMPLETE.

### 3.5 Algorithm 5: Collect references

If $CasQuery$ returns MATCH for both $MatchValue$ and $MatchPath$, the last step of the process is entered. The function supporting this algorithm ($Collect$) checks the nature of the current node and potentially assigns a reference. If the node is of type leaf, the related reference is output, otherwise the $Collect$ function is called recursively in both the left and right child.

# 4 Conclusions and future work

The Robust Content-And-Structure (RCAS) index is a robust index that applies a well-balanced dynamic interleaving to semi-structured hierarchical data with the use of composite keys. This approach applies the concept of dimension-wise discriminative byte and development of a tree-like structure. However the partitioning contains between 2 and 256 partitions, which can cause issues in the index performance. In fact, the path dimension can have large partitions and the value dimension can include a higher number of smaller partitions. A possible solution to this problem is the development of a bit-wise dynamic interleaving in the RCAS index.

The conversion of path and value to bits provides a partitioning with only two lists: one for the 0 bit and one for the 1; the subsequent type of the tree becomes binary. These facts indicate that this new approach could be beneficial to the index robustness. However this analysis assumes that the only possible location of the query descendant is located at the end of the path query.

Consequently, future work will have to be done to implement the query descendant such that it is accounted for at any position within the path and evaluate the robustness of this new index.

## References

[1] K. Wellenzohn, M. H. Böhlen, and S. Helmer. Dynamic Interleaving of Content and Structure for Robust Indexing of Semi-Structured Hierarchical Data. PVLDB, 13(10), 2020.