Applied Compression Techniques in RDF Stores

Marcus Pinnecke

This guy!
This lecture based in parts of the text book
Semantic Web
Semantic Web
Web 3.0
Semantic Web

"The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries"

World Wide Web Consortium (W3C)
Semantic Web, the Web of Data

- **Semantic Web**
  - Coined by Tim Berners-Lee, inventor of the WWW
  - Todays WWW: information space for humans (i.e., human readable documents)
    - Links documents together (hyper-text)
  - The WWW of tomorrow **is a global database understandable by machines**
    - Links documents and **data** together
    - Optimized for processing by machines

- **Enabling factors**
  - Already established technologies of the WWW (e.g., HTTP, or IRIs\(^1\))
  - **Semantic** (i.e., how to interpret data) **enabled by Resource Description Framework (RDF)**

---

\(^1\) think about IRIs as unicode variant of generalized URLs, e.g., [http://www.example.com/rdf/vocabulary#size](http://www.example.com/rdf/vocabulary#size)
RDF Concept and Abstract Syntax (I)

Resource Description Framework (RDF)

- RDF is a framework (meta data model) to represent information in the web
- Abstract syntax as data model to link RDF-based languages and specifications
- Foundation of abstract syntax is RDF graph

- Nodes are subjects, and objects
  - Are either IRIs, literals, and blank nodes
  - IRIs, and literals identify things (resources, or entities)
- Edges are properties (IRIs)

- A triple \((subject, predicate, object) = (s, p, o)\) forms an RDF statement
  - A binary relationship \(p\) between resources \(s\) and \(o\)
RDF Concept and Abstract Syntax (II) RDF Example

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>course:AdvDB</td>
<td>xmpl:fullName</td>
<td>„Advanced Topics in Databases“</td>
</tr>
<tr>
<td>course:AdvDB</td>
<td>xmpl:occurs</td>
<td>terms:summer</td>
</tr>
<tr>
<td>course:AdvDB</td>
<td>xmpl:lecturer</td>
<td>ovgu:Bronesk</td>
</tr>
<tr>
<td>ovgu:Bronesk</td>
<td>xmpl:name</td>
<td>„David Broneske“</td>
</tr>
<tr>
<td>ovgu:Bronesk</td>
<td>rdf:type</td>
<td>xmpl:human</td>
</tr>
</tbody>
</table>

*course, xmpl, ovgu, terms, and rdf* are *shortcuts* for some IRIs

Figure: Example RDF graph (visualization)
RDF Concept and Abstract Syntax (III)

RDF Vocabulary

Shortcuts: Distinguish between different sense, e.g., summer (seasons:summer and terms:summer)

- Collections of IRIs intended for use in RDF graphs

```
http://www.example.com/rdf/vocabulary#age
http://www.example.com/rdf/vocabulary#name
http://www.example.com/rdf/vocabulary#size
```

namespace IRI

vocabulary
RDF Concept and Abstract Syntax (IV)

RDF Vocabulary

Shortcuts: Distinguish between different sense, e.g., `summer` *(seasons:summer and terms:summer)*

- **Collections of IRIs** intended for use in RDF graphs

  vocabulary 1  vocabulary 2  ....  vocabulary n

- Possible huge set of vocabularies in one RDF graph
RDF Concept and Abstract Syntax (IV)

RDF Vocabulary

Shortcuts: Distinguish between different sense, e.g., summer (seasons:summer and terms:summer)

- Collections of IRIs intended for use in RDF graphs

  vocabulary 1  vocabulary 2  ....  vocabulary n

- Possible huge set of vocabularies in one RDF graph

  Many shared prefixes on strings in one vocabulary
  (e.g., http://www.example.com/rdf/vocabulary#)
RDF Dictionaries

Storage of Triple Values

-or-

How to store and access large collections of strings
## Motivation

RDF graphs contain **huge amount** of **IRIs** (unicode strings, e.g., UTF-32 w/ 32bit per character)

- One occurrence of built-in vocabulary *rdf* with namespace IRI
  
  ```
  http://www.w3.org/1999/02/22-rdf-syntax-ns#
  ```

  alone requires 172 Byte (!) in UTF-32

- **Compact representation of values is essential** to achieve better time/space tradeoff and/or modularity (graph + values separately)

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>ovgu:Bronesk</td>
<td>rdf:type</td>
<td>xmpl:human</td>
</tr>
<tr>
<td></td>
<td></td>
<td>172B</td>
</tr>
</tbody>
</table>

**Figure**: Example RDF graph
Dictionary Encoding & Compression (I)

Dictionary Encoding & Compression

Lookup Table

Compression

Two tasks

1. **Lookup-Table**: Replace multiple (string-) values with unique (smaller, fixed-length) identifiers

```plaintext
collection = {
    http://www.example.com/rdf/course#AdvDB,
    http://www.example.com/rdf/course#AdvDB,
    http://www.example.com/rdf/members#Bronesk,
    http://www.example.com/rdf/members#Bronesk
}
collection' = { 1, 1, 2, 2 }
dictionary = {
    1 <=> http://www.example.com/rdf/course#AdvDB,
    2 <=> http://www.example.com/rdf/members#Bronesk
}
```
Two tasks

2. Compression: Compress values in dictionary

\[
\text{collection'} = \{1, 1, 2, 2\} \\
\text{dictionary} = \{
    1 \leftrightarrow \text{http://www.example.com/rdf/course#AdvDB} \\
    2 \leftrightarrow \text{http://www.example.com/rdf/members#Bronesk}
\}
\]
Two tasks

2. **Compression**: Compress values in dictionary

```
collection' = { 1, 1, 2, 2 }
dictionary' = {
    1 <=> http://www.example.com/rdf/course#AdvDB
    2 <=> http://www.example.com/rdf/members#Bronesk
}
```

```
collection' = { 1, 1, 2, 2 }
dictionary' = {
    1 <=> xy
    2 <=> xz
}
encoder = {
    x <=> http://www.example.com/rdf/
    y <=> course#AdvDB
    z <=> members#Bronesk
}
Dictionary Encoding & Compression

Two tasks...

3. Compression: Compress encoded collection

4. Compression: ...
Two function to work with dictionary

- **locate**: Given a string, return the identifier
- **extract**: Given an identifier, return the string

Both functions must be efficient

\[
\text{locate}(\text{http://www.example.com/rdf/course#AdvDB}) = 1 \\
\text{extract}(1) = \text{http://www.example.com/rdf/course#AdvDB}
\]
Huffman Coding
Huffman Coding

- **Huffman Coding**
  - lossless data compression
  - A *code* is a bit string mapped to any symbol (e.g., a letter)
  - A word over an alphabet is mapped to a *variable-length prefix* code, i.e. letters are encoded with different length bit strings, and no code starts with another code
  - Produces optimal codes, i.e., minimal expected length codes


- **Huffman Coding (General)**
  - **Input** vector of probabilities \( p = (p_1, p_2, \ldots, p_n) \), values \( v = (v_1, v_2, \ldots, v_n) \) and a code alphabet \( A \)
  - **Output** optimal code, i.e., minimum length code
    
    \[
    Huff(p, A) = (\text{code}(v_1), \text{code}(v_2), \ldots, \text{code}(v_n))
    \]
  - Used to reduce the number of bits needed for a message
  - **Idea**: The higher the probability the less bits needed (and vice versa)
Huffman Coding (General)

- **Input** $p = (0.45, 0.2, 0.2, 0.15)$, $v = (v_1, v_2, v_3, v_4)$, and code alphabet $A = \{0, 1\}$

- **Huffman tree** $T$: Recursively create a node $w = u + v$ with edges to the 2 smallest entries $u$ and $v$ in $p$ until $w = 1$

$$p = (0.45, 0.2, 0.2, 0.15)$$
Huffman Coding (III)

- **Huffman Coding (General)**
  - **Input** \( p = (0.45, 0.2, 0.2, 0.15) \), \( v = (v_1, v_2, v_3, v_4) \), and code alphabet \( A = \{0, 1\} \)
  - **Huffman tree** \( T \): Recursively create a node \( w = u + v \) with edges to the 2 smallest entries \( u \) and \( v \) in \( p \) until \( w = 1 \)

\[
p = (0.45, 0.2, 0.2, 0.15)
\]

\[0.35\]
Huffman Coding (III)

- **Huffman Coding (General)**
  - **Input** \( \mathbf{p} = (0.45, 0.2, 0.2, 0.15) \), \( \mathbf{v} = (v_1, v_2, v_3, v_4) \), and code alphabet \( A = \{0, 1\} \)
  - **Huffman tree** \( T \): Recursively create a node \( w = u + v \) with edges to the 2 smallest entries \( u \) and \( v \) in \( \mathbf{p} \) until \( w = 1 \)

\[
\mathbf{p} = (0.45, 0.2, 0.2, 0.15)
\]

```
0.45
   /   \
0.35 /     \0.55
   /       /
0.2       0.2
```
Huffman Coding (III)

- **Huffman Coding (General)**
  - **Input** $p = (0.45, 0.2, 0.2, 0.15)$, $v = (v_1, v_2, v_3, v_4)$, and code alphabet $A = \{0, 1\}$
  - **Huffman tree** $T$: Recursively create a node $w = u + v$ with edges to the 2 smallest entries $u$ and $v$ in $p$ until $w = 1$

\[
p = (0.45, 0.2, 0.2, 0.15)
\]

![Huffman Tree Diagram]
Huffman Coding (General)

- Assign symbols from $A$ to paths in $T$, e.g., 0 for left branch, 1 for right branch
• **Huffman Coding (General)**
  
  - Assign symbols from $A$ to paths in $T$, e.g., 0 for left branch, 1 for right branch

```
0.45, 0.2, 0.2, 0.15

  0.35

  0.55

  0.55

  1.0 (root node)
```
• **Huffman Coding (General)**
  
  - Assign symbols from $A$ to paths in $T$, e.g., 0 for left branch, 1 for right branch

```
0.45, 0.2, 0.2, 0.15
```

```
0
-----
0.35
```

```
0.55
```

```
1.0 1 (root node)
```

Huffman Coding (IV)
Huffman Coding (General)

- Assign symbols from $A$ to paths in $T$, e.g., $0$ for left branch, $1$ for right branch

```
0.45, 0.2, 0.2, 0.15
  /   \
0.2   0.35
   /   \    \
 0.55 1.0 1
```
Huffman Coding (General)

- Assign symbols from $A$ to paths in $T$, e.g., $0$ for left branch, $1$ for right branch

```
0.45, 0.2, 0.2, 0.15
```

```
0.35
```

```
0.55
```

```
1.0 1 (root node)
```

 Marcus Pinnecke - AdvDB - June/2018
Huffman Coding (General)

- Assign symbols from \( A \) to paths in \( T \), e.g., 0 for left branch, 1 for right branch

```
0.45, 0.2, 0.2, 0.15
   /     /
0.35
   /     /
0 0
   /     /
0 0
   /     /
0 0
   /     /
0 0
   /     /
0 0
   /     /
0 0
   /     /
0 0
```

1.0 1 (root node)
Huffman Coding (General)
- Assign symbols from $A$ to paths in $T$, e.g., 0 for left branch, 1 for right branch

```
0.45, 0.2, 0.2, 0.15
```

```
0    0   0.35   1
0.55
1.0 1 (root node)
```
Huffman Coding (General)

- Assign code word $code(v_i)$ for each value $v_i$ in $v$ by the path from root node to leaf $p_i$

```
i | 0, 1, 2, 4
---|---
code(v_i) | 0.45, 0.2, 0.2, 0.15
```

```
p_i | 0.45, 0.2, 0.2, 0.15
---|---
     | 0.35 1
     | 0.55 1 (root node)
     | 1.0 1
```
Huffman Coding (IV)

• Huffman Coding (General)
  – Assign code word \( code(v_i) \) for each value \( v_i \) in \( v \) by the path from root node to leaf \( p_i \)

<table>
<thead>
<tr>
<th>( i )</th>
<th>( 0 )</th>
<th>( 1 )</th>
<th>( 2 )</th>
<th>( 4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( code(v_i) )</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( p_i )</td>
<td>0.45</td>
<td>0.2</td>
<td>0.2</td>
<td>0.15</td>
</tr>
</tbody>
</table>

![Huffman Coding Diagram]
Huffman Coding (IV)

- **Huffman Coding (General)**
  - Assign code word $code(v_i)$ for each value $v_i$ in $v$ by the path from root node to leaf $p_i$

  \[
  \begin{array}{c|cccc}
  i & 0, & 1, & 2, & 4 \\
  \hline
  code(v_i) & 10 \\
  p_i & 0.45, & 0.2, & 0.2, & 0.15 \\
  \end{array}
  \]

  ![Diagram of Huffman Coding](image)
• **Huffman Coding (General)**
  
  Assign code word $code(v_i)$ for each value $v_i$ in $v$ by the path from root node to leaf $p_i$

  $i$ | 0, 1, 2, 4
  ---|---
  $code(v_i)$ | 110
  $p_i$ | 0.45, 0.2, 0.2, 0.15
**Huffman Coding (General)**

- Assign code word $code(v_i)$ for each value $v_i$ in $v$ by the path from root node to leaf $p_i$

<table>
<thead>
<tr>
<th>$i$</th>
<th>0, 1, 2, 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$code(v_i)$</td>
<td>0.45, 0.2, 0.2, 0.15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$p_i$</th>
<th>0.45, 0.2, 0.2, 0.15</th>
</tr>
</thead>
</table>

Diagram:

- 0.45, 0.2, 0.2, 0.15
- Assign code word $code(v_i)$ for each value $v_i$ in $v$ by the path from root node to leaf $p_i$.
Huffman Coding (IV)

- **Huffman Coding (General)**
  - Assign code word $\text{code}(v_i)$ for each value $v_i$ in $v$ by the path from root node to leaf $p_i$

<table>
<thead>
<tr>
<th>$i$</th>
<th>0,</th>
<th>1,</th>
<th>2,</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_i$</td>
<td>code($v_i$)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>$p_i$</td>
<td>0.45,</td>
<td>0.2,</td>
<td>0.2,</td>
<td>0.15</td>
</tr>
</tbody>
</table>

```
  0.45,         0.2,         0.2,         0.15
0.35
0.55
1.0
---
1 (root node)
```
Huffman Coding (IV)

- Huffman Coding (General)
  - Assign code word \( \text{code}(v_i) \) for each value \( v_i \) in \( v \) by the path from root node to leaf \( p_i \)

<table>
<thead>
<tr>
<th>( i )</th>
<th>0,</th>
<th>1,</th>
<th>2,</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{code}(v_i) )</td>
<td>0</td>
<td>10</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>( p_i )</td>
<td>0.45,</td>
<td>0.2,</td>
<td>0.2,</td>
<td>0.15</td>
</tr>
</tbody>
</table>
Huffman Coding (Text Compression)

- To encode a word \( w = w_1 w_2 \ldots w_m \) with \( w_j \) is a value in \( v = (v_1, v_2, \ldots, v_n) \) for \( j = 1, 2, \ldots, m \), run Huffman

  where the \( i \)-th element in \( p \) is the relative frequency \( p_i = h_m(w_i) \)

\[
h_m(x) = \frac{H_m(x)}{m}
\]

where \( H_m(x) \) is the absolute frequency
Huffman Coding (Text Compression)

- **Example** \( w = \text{"mississippi"} \) with \( m = 11, \Sigma = \{m, i, s, p\} \)

- \( p = (h_m(\text{"m"}), h_m(\text{"i"}), h_m(\text{"s"}), h_m(\text{"p"})) \)
Huffman Coding (Text Compression)

- Example $w = \text{“mississippi”}$ with $m = 11$, $\Sigma = \{m, i, s, p\}$
  - $p = (h_m(\text{“m”}), h_m(\text{“i”}), h_m(\text{“s”}), h_m(\text{“p”})) = (1/11, 4/11, 4/11, 2/11)$
Huffman Coding (Text Compression)

- **Example** $w = \text{"mississippi"}$ with $m=11$, $\Sigma = \{m, i, s, p\}$

  - $p = (h_m(\text{"m"}), h_m(\text{"i"}), h_m(\text{"s"}), h_m(\text{"p"})) = (0.09, 0.36, 0.36, 0.18)$

  **Optimization**: To avoid linear search for the smallest value in $p$, you might sort $p$. 

---

Marcus Pinnecke - AdvDB - June/2018
• Huffman Coding (Text Compression)
  - Example $w=\text{"mississippi"}$ with $m=11$, $\Sigma = \{m,i,s,p\}$
    - $p = (h_m(\text{"m"}), h_m(\text{"i"}), h_m(\text{"s"}), h_m(\text{"p"})) = (0.09, 0.36, 0.36, 0.18)$
    - $(\text{sorted } p)$ $p' = (0.36, 0.36, 0.18, 0.09)$  $\Rightarrow v' = (i,s,p,m)$
      (indices in $v$ sorted according new order in $p$)
• **Huffman Coding (Text Compression)**
  - **Example** \( w = \text{"mississippi"} \) with \( m=11, \Sigma = \{m,i,s,p\} \)
    - \( p = (h_m(\text{"m"}), h_m(\text{"i"}), h_m(\text{"s"}), h_m(\text{"p"})) = (0.09, 0.36, 0.36, 0.18) \)
    - \( p' = (0.36, 0.36, 0.18, 0.09) \quad \iff \quad \mathbf{v}'=(i,s,p,m)\)
Huffman Coding (V)

- **Huffman Coding (Text Compression)**
  - **Example** \( w = \text{"mississippi"} \) with \( m = 11, \Sigma = \{m, i, s, p\} \)
    - \( p = (h_m(\text{"m"}), h_m(\text{"i"}), h_m(\text{"s"}), h_m(\text{"p"})) = (0.09, 0.36, 0.36, 0.18) \)
    - \( p' = (0.36, 0.36, 0.18, 0.09) \) " \( \mapsto v' = (i, s, p, m) \)"

<table>
<thead>
<tr>
<th>code((v'i))</th>
<th>0</th>
<th>10</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p'i )</td>
<td>0.36</td>
<td>0.36</td>
<td>0.18</td>
<td>0.09</td>
</tr>
</tbody>
</table>

![Huffman Coding Diagram]
Huffman Coding (V)

- **Huffman Coding (Text Compression)**
  - **Example** $w = \text{"mississippi"}$ with $m = 11$, $\Sigma = \{m, i, s, p\}$
    - $p = (h_m(\text{"m"}), h_m(\text{"i"}), h_m(\text{"s"}), h_m(\text{"p"})) = (0.09, 0.36, 0.36, 0.18)$
    - $p' = (0.36, 0.36, 0.18, 0.09)$, $\mapsto (i, s, p, m)$

\[
\begin{array}{cccc}
\text{code}(v'_i) & 0 & 10 & 110 & 111 \\
\gamma'_i & i & s & p & m \\
p'_i & 0.36 & 0.36 & 0.18 & 0.09 \\
\end{array}
\]

![Huffman Code Diagram](attachment:image.png)
Huffman Coding (V)

- **Huffman Coding (Text Compression)**
  - **Example**: \( w = "\text{mississippi}\) with \( m = 11, \Sigma = \{m, i, s, p\}\)

<table>
<thead>
<tr>
<th>(v'^i)</th>
<th>(i)</th>
<th>(s)</th>
<th>(p)</th>
<th>(m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(code(v'^i))</td>
<td>0</td>
<td>10</td>
<td>110</td>
<td>111</td>
</tr>
</tbody>
</table>

Assuming ASCII encoding of \(w\):
- \(\#\text{bits}(\text{mississippi}) = 8 \times 11 = 88\) bits
- \(\#\text{bits}(Huff(\text{mississippi})) = 21\) bits

76% space saving
Huffman Coding (Text Compression)

- Example $w=\text{"mississippi"}$ with $m=11$, $\Sigma = \{m,i,s,p\}$

<table>
<thead>
<tr>
<th>$v'_i$</th>
<th>i</th>
<th>s</th>
<th>p</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>code($v'_i$)</td>
<td>0</td>
<td>10</td>
<td>110</td>
<td>111</td>
</tr>
</tbody>
</table>

Encoding $\text{mississippi}$

$$\text{Huff} \rightarrow 111\ 0\ 10\ 10\ 0\ 10\ 10\ 0\ 110\ 110\ 0\ m\ i\ s\ s\ i\ s\ s\ i\ p\ p\ i$$
Huffman Coding (V)

- **Huffman Coding (Text Compression)**
  - Example \( w=","mississippi" \) with \( m=11, \Sigma = \{m,i,s,p\} \)

\[
\begin{array}{cccc}
  v'_i & code(v'_i) & 0 & 10 & 110 & 111 \\
  i & s & p & m \\
\end{array}
\]

**Encoding**

\[\text{mississippi} \xrightarrow{\text{Huff}} 111 0 10 10 0 10 10 0 110 110 0 \]

\[m \ i \ s \ s \ i \ s \ s \ i \ p \ p \ i\]

**Decoding**

\[111 0 10 10 0 10 10 0 110 110 0 \xrightarrow{\text{Huff}^{-1}} m \ i \ s \ s \ i \ s \ s \ i \ p \ p \ i\]

Decoding is not ambiguous. Why?
**Huffman Coding (Text Compression)**

- **Example** \( w = "\text{mississippi}" \) with \( m=11, \Sigma = \{m,i,s,p\} \)

<table>
<thead>
<tr>
<th>0</th>
<th>10</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_i )</td>
<td>( i )</td>
<td>( s )</td>
<td>( p )</td>
</tr>
<tr>
<td>( v'_i )</td>
<td>( m )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assuming ASCII encoding of \( w \):

- \#bits(mississippi) = 8 \cdot 11 = 88 \text{ bits}  
- \#bits(Huff(mississippi)) = 21 \text{ bits}

76% space saving
Huffman Coding for RDF String Dictionary

- Assume list of values in RDF triples
  
  \[ w_1 = DataLake \]
  \[ w_2 = DataBase \]
  \[ w_3 = DataHub \]

- Append unique string terminator character (e.g., $) to words \( w_1 \), \( w_2 \) and \( w_3 \)
  
  \[ w_1' = w_1$ \]
  \[ w_2' = w_2$ \]
  \[ w_3' = w_3$ \]
**Huffman Coding (VI)**

- **Huffman Coding for RDF String Dictionary**
  - (1) Construct Huffman code dictionary
Huffman Coding (VII)

- Huffman Coding for RDF String Dictionary
  - (2) Encode words

  \[
  \begin{align*}
  DataLake & \equiv w_1 \to 100 00 101 00 11110 00 01101 110 010 \\
  DataBase & \equiv w_2 \to 100 00 101 00 11100 00 01110 110 010 \\
  DataHub & \equiv w_3 \to 100 00 101 00 11110 10111 01100 010
  \end{align*}
  \]

<table>
<thead>
<tr>
<th>(code(v_i))</th>
<th>00</th>
<th>010</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>1110</th>
<th>11110</th>
<th>11111</th>
<th>01100</th>
<th>01101</th>
<th>01110</th>
<th>01111</th>
</tr>
</thead>
<tbody>
<tr>
<td>(V_i)</td>
<td>a</td>
<td>$</td>
<td>D</td>
<td>t</td>
<td>e</td>
<td>B</td>
<td>H</td>
<td>L</td>
<td>b</td>
<td>k</td>
<td>s</td>
<td>u</td>
</tr>
</tbody>
</table>

String terminator symbol
Huffman Coding (VIII)

- Huffman Coding for RDF String Dictionary
  - (2) Pad encoded words to byte boundaries, and concatenate them in bit array B

<table>
<thead>
<tr>
<th>array B</th>
<th>DataLake$</th>
<th>offset</th>
<th>DataBase$</th>
<th>offset</th>
<th>DataHub$</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000101</td>
<td>0</td>
<td>001111110</td>
<td>0</td>
<td>001101110</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0100000</td>
<td>1</td>
<td>00111000</td>
<td>4</td>
<td>01110110</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0100000</td>
<td>2</td>
<td>00111000</td>
<td>4</td>
<td>01110110</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0100000</td>
<td>3</td>
<td>00111000</td>
<td>4</td>
<td>01110110</td>
<td>4</td>
</tr>
</tbody>
</table>

\[
\text{pos}(\text{Huff}(\text{"DataLake"}), B) = 0 \quad \text{Huff}^{-1}(B[0]) = \text{"DataLake"}
\]
\[
\text{pos}(\text{Huff}(\text{"DataBase"}), B) = 4 \quad \text{Huff}^{-1}(B[4]) = \text{"DataBase"}
\]
\[
\text{pos}(\text{Huff}(\text{"DataHub"}), B) = 8 \quad \text{Huff}^{-1}(B[8]) = \text{"DataHub"}
\]
Hashing

Hash Functions + Hash Tables
Hash Function

- **Hash Function**
  - An injective function $h: K \rightarrow H$ that maps a larger set $K$ (keys) to a smaller set $H$ (hashes)

  - **Example 1:** $h(x) = x \mod 5$
    - $h(0) = 0, h(1) = 1, h(2) = 2, ..., h(4) = 4, h(5) = 0, h(6) = 1, ...$

  - **Example 2:** $h'(x) = md5(x)$
    - $h'(0) = cfcd208495d565ef66e7dff9f98764da, h'(1) = ...$

- **Intention:** map different keys to different hash values
  - Since $H$ is smaller than $K$, collisions can occur
Hash Tables

- **Hash Tables**
  - Index structure $hash(keys, values)$ that maps keys to values.
  - **Constant cost** to access value given its key.
- **Ingredients**
  - **Array**: A fixed array $A$ that stores (buckets of) values.
  - **Hash Function**: A hash function $h$ that maps keys to indexes in $A$.
- **Procedure**

![Diagram showing the operation of a hash table]

- $h(key) = 2$
- Store value in $A$ at position $h(key)$
- Array $A$ is indexed from 0 to $n-1$. 

---

Marcus Pinnecke - AdvDB - June/2018

page 56
Hash Tables using Huffman Coding
Hash Tables + Huffman Coding

- Compress a word $w$ (+$\$) using Huffman Coding + store in in array $B$
- Use Hash-Table
  - $key = Huff(w\$)$, i.e., $h(Huff(w\$))$ determines index of $value$ in $A$
  - $value = pos(Huff(w\$), B)$
Dictionaries with Huffman Coding (I)

- Hash Tables + Huffman Coding (Example)

\[ h(100001010011100001110110010) = 1 \]

(assume \( h \) gives that hash value)
• When entire string collection is imported into dictionary, the look-up functions are

\[
\text{locate('Database')} = h(Huff('Database$')) = 4
\]

\[
\text{extract(4)} = \text{Huff}^{-1}(B[4]) = 'Database'
\]
Trie-Based
• Trie related to Information Retrieval

• Search trees containing multiple strings

• **Idea** Exploit common prefix in terms of single characters for contained strings
  – Strings with common prefix share same path in the tree

• **Queries** exists string $S$, what are strings starting with $S$,...

• Can be used for associative arrays, e.g., locate(“String“)

---

**Figure:** Prefix Tree

Marcus Pinnecke - AdvDB - June/2018
• **Trie** related to Information Retrieval

• Search trees containing multiple strings

• **Idea** Exploit common prefix in terms of single characters for contained strings
  – Strings with common prefix share same path in the tree

• **Queries** exists string $S$, what are strings starting with $S$,...

• Can be used for associative arrays, e.g., `locate("String")`
• *Trie* related to Information Retrieval

• Search trees containing multiple strings

• **Idea** Exploit common prefix in terms of single characters for contained strings
  – Strings with common prefix share same path in the tree

• **Queries** exists string $S$, what are strings starting with $S$,...

• Can be used for assoziative arrays, e.g., `locate(„String“)`

*Figure: Prefix Tree*
• **Trie** related to Information Retrieval

• Search trees containing multiple strings

• **Idea** Exploit common prefix in terms of single characters for contained strings
  
  – Strings with common prefix share same path in the tree

• **Queries** exists string $S$, what are strings starting with $S$,...

• Can be used for associative arrays, e.g., `locate(“String“)`

---

**Figure: Prefix Tree**
Trie-Based

- **Trie** related to Information Retrieval
- Search trees containing multiple strings
- **Idea** Exploit common prefix in terms of single characters for contained strings
  - Strings with common prefix share same path in the tree
- **Queries** exists string $S$, what are strings starting with $S$,...
- Can be used for associative arrays, e.g., `locate(“String“)`

Figure: Prefix Tree
Wrap Up
You have learned

- The Idea and Enabling Concepts of **Semantic Web**
- About **RDF** and What are **RDF Stores**
- Important Concepts to Handle **RDF Dictionaries**
  - Concept and Techniques for **Lookup Tables**
  - **Compression Techniques** for Dictionary Values
    - Huffman Coding
    - Tries
Invitation

You are invited to join our research on code optimizations and databases on new hardware, e.g., in form of:

- Bachelor or master thesis
- “Scientific Project: Data Management on new Hardware”
- Scientific individual project
- Contact me! marcus.pinnecke@ovgu.de


Further Reading