Applied Compression Techniques in RDF Stores

Marcus Pinnecke
This lecture based in parts of the text book
Outline

Part I Introduction
   1. BigData
   2. Graph-Shaped Data

Part II Semantic Web
   3. Description and Enabling Technologies
   4. RDF Concepts and Syntax

Part III RDF Dictionaries
   5. Dictionary Handling
   6. Hash Tables using Huffman Coding
   7. Incremental Coding
   8. Trie-Based Approaches
   9. Grammar-Based Codes

Part IV Wrap Up
BigData & Linked Data
BigData(I)

- **Today's BigData contributor:** Web 2.0 (popular term) emphasizes user-generated content in the WWW
  - **Weblog:** Twitter, Tumblr,…
  - **Micro-Blogging:** Twitter, Jaiku,…
  - **Wikis:** Wikipedia, DokuWiki,…
  - **Social Media:** Facebook, Google+, …
  - **Search Engines:** Google, Bing, DuckDuckGo,…
  - …
BigData

- Datasets far too complex for traditional processing,
  - e.g., Facebook or Google

- Characterized by 3 V’s (Gardner 2001):
  - **Volume** (quantity)
    - large-scale datasets, e.g., 465 million Twitter accounts
  - **Velocity** (motion)
    - many incoming data, and - changes, e.g., YouTube: 30h video uploads/min
  - **Variety** (data forms)
    - diversity of types, structure, and sources, e.g., CSV, XML, JSON, RDF, OWL,...
BigData (III)

- **NoSQL** as a concept addressing BigData issues with non-relational approaches, storage and querying
  - **Scalability**: scale-up („buy a bigger box“), and scale-out (partitioning data across systems)
  - **Softed ACID constraints**: system tend to focus on fast processing, and scale rather ACID
  - **Eventual consistency**: data will be consistent to in distributed systems later in time
  - **Many Reads, Many Writes**: challenges in distributed environments (cf. CAP theorem)

- **System types**
  - **Document stores**: MongoDB, CouchDB,…
  - **Key/Value stores**: Redis, Oracle NoSQL Database,…
  - **Wide-Column**: BigTable, HBase, Apache Cassandra,…
  - **Distributed Processing**: MapReduce / Apache Hadoop, Spark,…
  - **Graph Databases**: Neo4j Property Graph Database, Semantic Web / RDF-Stores
Graph Data (I)

- Graph is the most generic data structure
- Data is modeled in terms of nodes and edges.
- Several models exist
  - Simple graphs
  - Multi graphs
  - Weighted graphs,
  - Attributed graphs
  - ...
Graph-Shaped Data Management Good For? (I)

... when relationships between entities are more important than the entities itself

- **Complex (transitive) connections between entities, and importance of particular entities, or groups**
  - social networks w/ FOAF
  - web search w/ PageRank

- **(Dis-)similarities of nodes and graphs, and Anomaly detection**
  - electrical circuits w/ graph isomorphism
  - information security w/ sub graph pattern matching

- **Knowledge management**
  - AI over semantic networks w/ inference of paths
  - **Semantic Web, Linked Data and Web of Data**
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
  - Today's 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
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

```plaintext
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 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., \textit{summer} (\texttt{seasons:summer} and \texttt{terms:summer})

• Collections of IRIs intended for use in RDF graphs

\begin{itemize}
  \item \texttt{vocabulary 1}
  \item \texttt{vocabulary 2}
  \item \ldots
  \item \texttt{vocabulary n}
\end{itemize}

• Possible huge set of vocabularies in one RDF graph

\textit{Many shared prefixes} on strings in one vocabulary

(e.g., http://www.example.com/rdf/vocabulary#)
Semantic Web Architecture Stack

• **Semantic Web Cake**
  - Illustration of Semantic Web Architecture
  - Extension to the existing WWW (e.g., embedded RDF)
  - Initial concept by Tim Berners-Lee
RDF Dictionaries

Storage of Triple Values

- or -

How to store and access large collections of strings

Part III
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#](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)
Dictionary Encoding & Compression (I)

Dictionary Encoding & Compression

Lookup Table

Compression

Dictionary Encoding & Compression (II)

Two tasks

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

```
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

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

2. Compression: Compress values in dictionary

```plaintext
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
}```
Two tasks...

3. **Compression**: Compress encoded collection

4. **Compression**: ...

Dictionary Encoding & Compression
Two functions to work with dictionary

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

Both functions must be efficient

\[
\begin{align*}
\text{locate}(\text{http://www.example.com/rdf/course\#AdvDB}) &= 1 \\
\text{extract}(1) &= \text{http://www.example.com/rdf/course\#AdvDB}
\end{align*}
\]
Query Processing with Dictionary Encoded Values (I)

„What’s the name of the thing whose lecturer is Bronesk, and whose occurrence is in summer?“

```sparql
PREFIX xmpl: <http://www.example.com/rdf/vocab#>
PREFIX ovgu: <https://lsf.ovgu.de/fictional/members#>
PREFIX seasons: <http://www.example.com/rdf/seasons#>

SELECT ?name
WHERE {
  ?x xmpl:fullName ?name;
  xmpl:lecturer ovgu:Bronesk;
  xmpl:occurs seasons:summer
}
```

**Figure:** Querying RDF data with SPARQL query language

**Figure:** SPARQL query visualization
Query Processing with Dictionary Encoded Values (II)

Figure: SPARQL query visualization

Figure: SPARQL query (encoded)
Query Processing with Dictionary Encoded Values (III)

Figure: SPARQL query (encoded)

Figure: Query Processing

Figure: Query Result (encoded)

Figure: Query Result (decoded)

Advanced Topics in Databases

Query Engine

Dictionary

Projection

extract
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 \( p = (p_1, p_2, ..., p_n) \), values \( v = (v_1, v_2, ..., v_n) \) and a code alphabet \( A \)
- **Output** optimal code, i.e., minimum length code

\[
\text{Huff}(p, A) = (\text{code}(v_1), \text{code}(v_2), ..., \text{code}(v_n))
\]

- 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 $w = 1$

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

\[
\begin{array}{c}
0.35 \\
\end{array}
\]
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 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 )
```

```
   1.0 (root node)
    /  \
  0.55 /  \ 0.35
    /    \
  0.2     \
    /      
  0.15
```

marcuspinnecke - AdvDB - May/2017
Huffman Coding (IV)

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

\[
\begin{align*}
0.45, & \quad 0.2, & \quad 0.2, & \quad 0.15 \\
\downarrow & & \downarrow & \\
0.35 & & \downarrow & \\
& & 0.55 & \\
& & \downarrow & \\
& & 1.0 & (\text{root node})
\end{align*}
\]
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

0.55

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
0.35
0.55
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
```

```
  0.35
    0
  0.55
    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
```

```
0.35
```

```
0.55
```

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

```
0
```

```
0
```

```
1
```

```
0.2
```

```
0.2
```

```
0.45
```

```
0.2
```

```
0.2
```

```
0.15
```

```
0.35
```

```
0.55
```

```
1
```

```
1
```

```
(root node)
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
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
```

```
0.55
```

```
0 0.35
0.35
```

```
0.8
```

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

Marcus Pinnecke - AdvDB - May/2017
• **Huffman Coding (General)**
  
  - Assign symbols from $A$ to paths in $T$, e.g., 0 for left branch, 1 for right branch
Huffman Coding (IV)

- **Huffman Coding (General)**
  - Assign code word $\text{code}(v_i)$ for each value $v_i$ in $\mathbf{v}$ 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>0.45</td>
<td>0.2</td>
<td>0.2</td>
<td>0.15</td>
</tr>
<tr>
<td>$p_i$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
0.45,   0.2,   0.2,   0.15
0.35    1
1
1 (root node)
```
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, 1, 2, 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>code((v_i))</td>
<td>0</td>
</tr>
<tr>
<td>( p_i )</td>
<td>0.45, 0.2, 0.2, 0.15</td>
</tr>
</tbody>
</table>

(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>code((v_i))</td>
<td>10</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>

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

```
0, 1, 2, 4
```

```
0.35 1
```

```
0.55
```

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

Marcus Pinnecke - AdvDB - May/2017
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></td>
<td></td>
<td>110</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>

(\( root \) node)
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 \( p_i \)

\[
\begin{array}{c|cccc}
  i & 0, & 1, & 2, & 4 \\
  \text{code}(v_i) & & & & 111 \\
  p_i & 0.45, & 0.2, & 0.2, & 0.15 \\
\end{array}
\]
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, 1, 2, 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_i$</td>
<td>0, 10, 110, 111</td>
</tr>
<tr>
<td>$p_i$</td>
<td>0.45, 0.2, 0.2, 0.15</td>
</tr>
</tbody>
</table>

Diagram:
- 1.0 is the root node.
- 0.55 is a leaf node with code 0.
- 0.35 is a leaf node with code 1.
- 1.0 is the root node with code 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, 1, 2, 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$code(v_i)$</td>
<td>0, 10, 110, 111</td>
</tr>
<tr>
<td>$p_i$</td>
<td>0.45, 0.2, 0.2, 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 $m = 11$, $\Sigma = \{m, i, s, p\}$

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

  $p' = (0.36, 0.36, 0.18, 0.09) \
  \rightarrow (\text{i, s, p, m})$
Huffman Coding (V)

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

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

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

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

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

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

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

$w=\text{"mississippi"}$ with $m=11$, $\Sigma = \{m,i,s,p\}$
Huffman Coding (V)

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

  Optimization: To avoid linear search for the smallest value in \( p \), you might sort \( p \)
• **Huffman Coding (Text Compression)**
  
  - **Example** \( w = "\text{mississippi}" \) with \( m = 11, \Sigma = \{m, i, s, p\} \)
  
  - \( p = (h_m(\",m\"), h_m(\",i\"), h_m(\",s\"), h_m(\",p\")) = (0.09, 0.36, 0.36, 0.18) \)

  (sorted \( p \))

  - \( p' = (0.36, 0.36, 0.18, 0.09) \)

  \( \mapsto v' = (i, s, p, m) \)  
  (indices in \( v \) sorted according new order in \( p \))
• **Huffman Coding (Text Compression)**
  
  - **Example** \( w = \text{“mississippi”} \) with \( m = 11, \Sigma = \{m,i,s,p\} \)
  
  - \( p = (h_m(\text{“m”}), h_m(\text{“i”}), h_m(\text{“s”}), h_m(\text{“p”})) = (0.09, 0.36, 0.36, 0.18) \)
  
  - \( p’ = (0.36, 0.36, 0.18, 0.09) \)  \( \mapsto v’=(i,s,p,m)\)
Huffman Coding (Text Compression)

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

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

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

Diagram:

- $0.99 \quad 0.63 \quad 1$
- $0 \quad 0 \quad 0.27 \quad 1$
- $0 \quad 0.36 \quad 0.36 \quad 0.18 \quad 0.09$

(rounding error)
Huffman Coding (Text Compression)

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

  | \( v'_i \) | \( i \) | \( s \) | \( p \) | \( m \) |
  |-------|-------|-------|-------|
  | \( \text{code}(v'_i) \) | 0 | 10 | 110 | 111 |

Assuming ASCII encoding of \( w \):

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

76% space saving
• **Huffman Coding (Text Compression)**
  
  – **Example** \( w = \text{"mississippi"} \) with \( m=11, \Sigma = \{m,i,s,p\} \)

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

  Encoding

  \[
  \begin{array}{l}
  \text{mississippi} \quad \xrightarrow{\text{Huff}} \quad 111 \ 0 \ 10 \ 10 \ 0 \ 10 \ 10 \ 0 \ 110 \ 110 \ 0 \ m \ i \ s \ s \ i \ s \ s \ i \ p \ p \ i \\
  \end{array}
  \]
Huffman Coding (V)

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

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

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

  \[
  \begin{array}{cccccccc}
  m & i & s & s & i & s & i & p & p & i \\
  \end{array}
  \]

  **Decoding**
  \[
  111 \ 0 \ 10 \ 10 \ 0 \ 10 \ 10 \ 0 \ 110 \ 110 \ 0 \xrightarrow{\text{Huff}^\dagger} m \ i \ s \ s \ i \ s \ s \ i \ p \ p \ i
  \]

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

Assuming ASCII encoding of $w$:  

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

76% space saving
Huffman Coding (VI)

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

```
\text{code}(v_i) & 00 & 010 & 100 & 101 & 110 & 1110 & 11110 & 111110 & 01100 & 01101 & 01110 & 01111 \\
\text{v}_i & a &$ & D & t & e & B & H & L & b & k & s & u \\
\text{p}_i & 0.31 & 0.12 & 0.12 & 0.12 & 0.08 & 0.04 & 0.04 & 0.04 & 0.04 & 0.04 & 0.04 & 0.04
```

![Huffman Coding Diagram](image)
Huffman Coding for RDF String Dictionary

(2) Encode words

- $DataLake$ = $w_1$ \(\mapsto\) 100 00 101 00 1111 10 01101 110 010
- $DataBase$ = $w_2$ \(\mapsto\) 100 00 101 00 1110 00 01110 110 010
- $DataHub$ = $w_3$ \(\mapsto\) 100 00 101 00 11110 01111 01100 010
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 ) offset</th>
<th>DataLake$</th>
<th>DataBase$</th>
<th>DataHub$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01000101</td>
<td>01111000</td>
<td>00110110</td>
</tr>
<tr>
<td>1</td>
<td>00111110</td>
<td>01110110</td>
<td>01000000</td>
</tr>
<tr>
<td>2</td>
<td>00110111</td>
<td>01111000</td>
<td>01111110</td>
</tr>
<tr>
<td>3</td>
<td>00100000</td>
<td>01110110</td>
<td>01000000</td>
</tr>
<tr>
<td>4</td>
<td>01000101</td>
<td>01111000</td>
<td>01111110</td>
</tr>
<tr>
<td>5</td>
<td>00111000</td>
<td>01110110</td>
<td>01000000</td>
</tr>
<tr>
<td>6</td>
<td>01110110</td>
<td>01111110</td>
<td>01111110</td>
</tr>
<tr>
<td>7</td>
<td>01000000</td>
<td>01110110</td>
<td>01000000</td>
</tr>
<tr>
<td>8</td>
<td>01000101</td>
<td>01111000</td>
<td>01111110</td>
</tr>
<tr>
<td>9</td>
<td>00111000</td>
<td>01110110</td>
<td>01000000</td>
</tr>
<tr>
<td>10</td>
<td>01110110</td>
<td>01111110</td>
<td>01111110</td>
</tr>
<tr>
<td>11</td>
<td>01000000</td>
<td>01110110</td>
<td>01000000</td>
</tr>
</tbody>
</table>

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

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

Hash Functions + Hash Tables
• **Hash Function**
  
  – An injective function \( h:K \rightarrow H \) that maps a larger set \( K \) (keys) to a smaller set \( H \) (hashes)

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

  – **Example 2**: \( h'(x) = md5(x) \)
    
    - \( h'(0) = cfcd208495d565ef66e7dff9f98764da \), \( h'(1) = \ldots \)

  – **Intention**: map different keys to different hash values

    - Since \( H \) is smaller than \( K \), **collisions** can occur
Hash Tables

- **Hash Tables**
  - Index structure $\text{hash}(\text{keys, values})$ 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**

  $h(key) = 2$

  $h(key)$ is used to find the index in $A$ for the key.

  Store value in $A$ at position $h(key)$.
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\((Huff(w$), B)\)
• Hash Tables + Huffman Coding (Example)

\[ h(10001010001110000111010010) = 1 \]

(assume \( h \) gives that hash value)

Dictionaries with Huffman Coding (I)
Dictionaries with Huffman Coding (II)

- When entire string collection is imported into dictionary, the look-up functions are

\[
\text{locate('Database')} = h(Huff('Database'\$)) = 1
\]

\[
\text{extract(1)} = \text{Huff}^{-1}(B[1]) = 'Database'
\]
Incremental Coding
Incremental Coding (I)

- **Incremental Coding** (aka. front coding, front compression, back compression)
  - Compresses (sorted) collections
  - Special kind of delta coding algorithms
  - **Idea** Exploit common *prefixes* (suffixes) in terms, and *encode textural difference* (delta-coding)
## Incremental Coding (II)

<table>
<thead>
<tr>
<th>Input (sorted)</th>
<th>Common Prefix</th>
<th>Compressed Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>courses:AdvDB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Breß</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Bronesk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Dorok</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Meister</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vocab:occurs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure:** Incremental coded input collection (left), common prefix (middle), compressed output (right)
## Incremental Coding (II)

<table>
<thead>
<tr>
<th>Input (sorted)</th>
<th>Common Prefix</th>
<th>Compressed Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>courses:AdvDB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Breß</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Bronesk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Dorok</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Meister</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vocab:occurs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure: Incremental coded input collection (left), common prefix (middle), compressed output (right)*

- **Substring** of string $S$ is string $\text{sub}(S, \text{start}, \text{end})$ that is in $S$ ($\text{start}$ including, $\text{end}$ excluding)
  - For instance, $\text{sub}("Topics in Databases", 3, 8) = "ics i"$
- **Substring** of string $S$ is string $\text{sub}(S, \text{start}, \text{end})$ that is in $S$ ($\text{start}$ including, $\text{end}$ excluding)
  - For instance, $\text{sub}$(“Topics in Databases“, 3, 8) = “ics i“
- **Prefix** is a substring $\text{pre}(S, \text{end})$ with start index 0, i.e., $\text{pre}(S, \text{end}) = \text{sub}(S, 0, \text{end})$
Incremental Coding (II)

<table>
<thead>
<tr>
<th>Input (sorted)</th>
<th>Common Prefix</th>
<th>Compressed Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>courses:AdvDB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Breß</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Bronesk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Dorok</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Meister</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vocab:occurs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure: Incremental coded input collection (left), common prefix (middle), compressed output (right)

- **Substring** of string $S$ is string $\text{sub}(S, \text{start}, \text{end})$ that is in $S$ ($\text{start}$ including, $\text{end}$ excluding)
  - For instance, $\text{sub}("Topics in Databases", 3, 8) = "ics i"
- **Prefix** is a substring $\text{pre}(S, \text{end})$ with start index 0, i.e., $\text{pre}(S, \text{end}) = \text{sub}(S, 0, \text{end})$
- **Suffix** is a substring $\text{suf}(S, \text{start})$ with end index $|S|$, i.e., $\text{suf}(S, \text{start}) = \text{sub}(S, \text{start}, |S|)$
Incremental Coding (II)

Input (sorted)          Common Prefix          Compressed Output

courses:AdvDB
ovgu:Breß
ovgu:Bronesk
ovgu:Dorok
ovgu:Meister
vocab:occurs

Figure: Incremental coded input collection (left), common prefix (middle), compressed output (right)

• Substring of string $S$ is string $\text{sub}(S, start, end)$ that is in $S$ ($start$ including, $end$ excluding)
  – For instance, $\text{sub}("Topics in Databases", 3, 8) = "ics i"

• Prefix is a substring $\text{pre}(S, end)$ with start index 0, i.e., $\text{pre}(S, end) = \text{sub}(S, 0, end)$

• Suffix is a substring $\text{suf}(S, start)$ with end index $|S|$, i.e., $\text{suf}(S, start) = \text{sub}(S, start, |S|)$

• Store $(|p|, \text{suf}(S, |p|))$ for the $i$-th entry $S$, if $S$ shares suffix $p$ with $(i-1)$-th entry, and $(0, S)$ otherwise
### Incremental Coding (II)

<table>
<thead>
<tr>
<th>Input (sorted)</th>
<th>Common Prefix</th>
<th>Compressed Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>courses:AdvDB</td>
<td>no preceding word</td>
<td></td>
</tr>
<tr>
<td>ovgu:Breß</td>
<td></td>
<td>ovgu:Br</td>
</tr>
<tr>
<td>ovgu:Bronesk</td>
<td>no common prefix</td>
<td>7, onesk</td>
</tr>
<tr>
<td>ovgu:Dorok</td>
<td></td>
<td>5, Meister</td>
</tr>
<tr>
<td>ovgu:Meister</td>
<td></td>
<td>0, ovgu:Breß</td>
</tr>
<tr>
<td>vocab:occurs</td>
<td></td>
<td>5, Dorok</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0, courses:AdvDB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0, vocab:occurs</td>
</tr>
</tbody>
</table>

**Figure:** Incremental coded input collection (left), common prefix (middle), compressed output (right)

- **Substring** of string $S$ is string $\text{sub}(S, \text{start}, \text{end})$ that is in $S$ (start including, end excluding)
  - For instance, $\text{sub}("Topics in Databases", 3, 8) = "ics i"

- **Prefix** is a substring $\text{pre}(S, \text{end})$ with start index 0, i.e., $\text{pre}(S, \text{end}) = \text{sub}(S, 0, \text{end})$

- **Suffix** is a substring $\text{suf}(S, \text{start})$ with end index $|S|$, i.e., $\text{suf}(S, \text{start}) = \text{sub}(S, \text{start}, |S|)$

- Store $(|p|, \text{suf}(S, |p|))$ for the $i$-th entry $S$, if $S$ shares suffix $p$ with $(i-1)$-th entry, and $(0, S)$ otherwise
### Incremental Coding (II)

<table>
<thead>
<tr>
<th>Input (sorted)</th>
<th>Common Prefix</th>
<th>Compressed Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>courses:AdvDB</td>
<td>no preceding word</td>
<td>0, courses:AdvDB</td>
</tr>
<tr>
<td>ovgu:Brensk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Breß</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Bronesk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Dorok</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Meister</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vocab:occurs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure:** Incremental coded input collection (left), common prefix (middle), compressed output (right)

- **Substring** of string $S$ is string $\text{sub}(S, \text{start}, \text{end})$ that is in $S$ ($\text{start}$ including, $\text{end}$ excluding)
  - For instance, $\text{sub}("Topics in Databases", 3, 8) = "ics i"$
- **Prefix** is a substring $\text{pre}(S, \text{end})$ with start index 0, i.e., $\text{pre}(S, \text{end}) = \text{sub}(S, 0, \text{end})$
- **Suffix** is a substring $\text{suf}(S, \text{start})$ with end index $|S|$, i.e., $\text{suf}(S, \text{start}) = \text{sub}(S, \text{start}, |S|)$
- Store $(|p|, \text{suf}(S, |p|))$ for the $i$-th entry $S$, if $S$ shares suffix $p$ with $(i-1)$-th entry, and $(0, S)$ otherwise
<table>
<thead>
<tr>
<th>Input (sorted)</th>
<th>Common Prefix</th>
<th>Compressed Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>courses:AdvDB</td>
<td>no preceding word</td>
<td>0, courses:AvDB</td>
</tr>
<tr>
<td>ovgu:Breß</td>
<td>no common prefix</td>
<td></td>
</tr>
<tr>
<td>ovgu:Bronesk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Dorok</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Meister</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vocab:occurs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure: Incremental coded input collection (left), common prefix (middle), compressed output (right)

- **Substring** of string $S$ is string $\text{sub}(S, \text{start}, \text{end})$ that is in $S$ ($\text{start}$ including, $\text{end}$ excluding)
  - For instance, $\text{sub}("Topics in Databases", 3, 8) = "ics i"

- **Prefix** is a substring $\text{pre}(S, \text{end})$ with start index 0, i.e., $\text{pre}(S, \text{end}) = \text{sub}(S, 0, \text{end})$

- **Suffix** is a substring $\text{suf}(S, \text{start})$ with end index $|S|$, i.e., $\text{suf}(S, \text{start}) = \text{sub}(S, \text{start}, |S|)$

- Store $(|p|, \text{suf}(S, |p|))$ for the $i$-th entry $S$, if $S$ shares suffix $p$ with $(i-1)$-th entry, and $(0, S)$ otherwise
Incremental Coding (II)

<table>
<thead>
<tr>
<th>Input (sorted)</th>
<th>Common Prefix</th>
<th>Compressed Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>courses:AdvDB</td>
<td>no preceding word</td>
<td>0, courses:AvDB</td>
</tr>
<tr>
<td>ovgu:Breß</td>
<td>no common prefix</td>
<td>0, ovgu:Breß</td>
</tr>
<tr>
<td>ovgu:Bronesko</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Dorok</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Meister</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vocab:occurs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure: Incremental coded input collection (left), common prefix (middle), compressed output (right)

- **Substring** of string $S$ is string $\text{sub}(S, \text{start}, \text{end})$ that is in $S$ ($\text{start}$ including, $\text{end}$ excluding)
  - For instance, $\text{sub}("Topics in Databases", 3, 8) = "ics i"$
- **Prefix** is a substring $\text{pre}(S, \text{end})$ with start index 0, i.e., $\text{pre}(S, \text{end}) = \text{sub}(S, 0, \text{end})$
- **Suffix** is a substring $\text{suf}(S, \text{start})$ with end index $|S|$, i.e., $\text{suf}(S, \text{start}) = \text{sub}(S, \text{start}, |S|)$
- Store $|p|, \text{suf}(S, |p|)$ for the $i$-th entry $S$, if $S$ shares suffix $p$ with $(i-1)$-th entry, and $(0, S)$ otherwise
**Incremental Coding (II)**

<table>
<thead>
<tr>
<th>Input (sorted)</th>
<th>Common Prefix</th>
<th>Compressed Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>courses:AdvDB</td>
<td>no preceding word</td>
<td>0, courses:AvDB</td>
</tr>
<tr>
<td>ovgu:Breß</td>
<td>no common prefix</td>
<td>0, ovgu:Breß</td>
</tr>
<tr>
<td>ovgu:Brónesk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Dorok</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Meister</td>
<td>ovgu:Br</td>
<td></td>
</tr>
<tr>
<td>vocab:occurs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure: Incremental coded input collection (left), common prefix (middle), compressed output (right)*

- **Substring** of string \( S \) is string \( \text{sub}(S, \text{start}, \text{end}) \) that is in \( S \) (\( \text{start} \) including, \( \text{end} \) excluding)
  - For instance, \( \text{sub}("Topics in Databases", 3, 8) = "ics i" \)
- **Prefix** is a substring \( \text{pre}(S, \text{end}) \) with start index 0, i.e., \( \text{pre}(S, \text{end}) = \text{sub}(S, 0, \text{end}) \)
- **Suffix** is a substring \( \text{suf}(S, \text{start}) \) with end index \( |S| \), i.e., \( \text{suf}(S, \text{start}) = \text{sub}(S, \text{start}, |S|) \)
- Store \((|p|, \text{suf}(S, |p|))\) for the \( i \)-th entry \( S \), if \( S \) shares suffix \( p \) with \( (i-1) \)-th entry, and \((0, S)\) otherwise
Incremental Coding (II)

<table>
<thead>
<tr>
<th>Input (sorted)</th>
<th>Common Prefix</th>
<th>Compressed Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>courses:AdvDB</td>
<td>no preceding word</td>
<td>0, courses:AvDB</td>
</tr>
<tr>
<td>ovgu:Brenß</td>
<td>no common prefix</td>
<td>0, ovgu:Brenß</td>
</tr>
<tr>
<td>ovgu:Bronesk</td>
<td>ovgu:Br</td>
<td>7, onesk</td>
</tr>
<tr>
<td>ovgu:Dorok</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Meister</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vocab:occurs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure:** Incremental coded input collection (left), common prefix (middle), compressed output (right)

- **Substring** of string $S$ is string $\text{sub}(S, start, end)$ that is in $S$ $\text{(start including, end excluding)}$
  - For instance, $\text{sub}(\text{"Topics in Databases"}, 3, 8) = \text{"ics i"}$
- **Prefix** is a substring $\text{pre}(S, end)$ with start index 0, i.e., $\text{pre}(S, end) = \text{sub}(S, 0, end)$
- **Suffix** is a substring $\text{suf}(S, start)$ with end index $|S|$, i.e., $\text{suf}(S, start) = \text{sub}(S, start, |S|)$
- Store $(|p|, \text{suf}(S, |p|))$ for the $i$-th entry $S$, if $S$ shares suffix $p$ with $(i-1)$-th entry, and $(0, S)$ otherwise
### Incremental Coding (II)

<table>
<thead>
<tr>
<th>Input (sorted)</th>
<th>Common Prefix</th>
<th>Compressed Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>courses:AdvDB</td>
<td>no preceding word</td>
<td>0, courses:AvDB</td>
</tr>
<tr>
<td>ovgu:Breß</td>
<td>no common prefix</td>
<td>0, ovgu:Breß</td>
</tr>
<tr>
<td>ovgu:Bronesk</td>
<td>ovgu:Br</td>
<td>7, onesk</td>
</tr>
<tr>
<td>ovgu:Dorok</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ovgu:Meister</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vocab:occurs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure:** Incremental coded input collection (left), common prefix (middle), compressed output (right)

- **Substring** of string $S$ is string $\text{sub}(S, \text{start}, \text{end})$ that is in $S$ ($\text{start}$ including, $\text{end}$ excluding)
  - For instance, $\text{sub}("Topics in Databases", 3, 8) = "ics i"
- **Prefix** is a substring $\text{pre}(S, \text{end})$ with start index 0, i.e., $\text{pre}(S, \text{end}) = \text{sub}(S, 0, \text{end})$
- **Suffix** is a substring $\text{suf}(S, \text{start})$ with end index $|S|$, i.e., $\text{suf}(S, \text{start}) = \text{sub}(S, \text{start}, |S|)$
- Store $(|p|, \text{suf}(S, |p|))$ for the $i$-th entry $S$, if $S$ shares suffix $p$ with $(i-1)$-th entry, and $(0, S)$ otherwise
Incremental Coding (II)

<table>
<thead>
<tr>
<th>Input (sorted)</th>
<th>Common Prefix</th>
<th>Compressed Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>courses:AdvDB</td>
<td>no preceding word</td>
<td>0, courses:AvDB</td>
</tr>
<tr>
<td>ovgu:Breß</td>
<td>no common prefix</td>
<td>0, ovgu:Breß</td>
</tr>
<tr>
<td>ovgu:Bronesk</td>
<td>ovgu:Br</td>
<td>7, onesk</td>
</tr>
<tr>
<td>ovgu:Dorok</td>
<td>ovgu:</td>
<td>5, Dorok</td>
</tr>
<tr>
<td>ovgu:Meister</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vocab:occurs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure: Incremental coded input collection (left), common prefix (middle), compressed output (right)

- **Substring** of string $S$ is string $\text{sub}(S, \text{start}, \text{end})$ that is in $S$ ($\text{start}$ including, $\text{end}$ excluding)
  - For instance, $\text{sub}(\text{"Topics in Databases"}, 3, 8) = \text{"ics i"}$

- **Prefix** is a substring $\text{pre}(S, \text{end})$ with start index 0, i.e., $\text{pre}(S, \text{end}) = \text{sub}(S, 0, \text{end})$

- **Suffix** is a substring $\text{suf}(S, \text{start})$ with end index $|S|$, i.e., $\text{suf}(S, \text{start}) = \text{sub}(S, \text{start}, |S|)$

- Store $(|p|, \text{suf}(S, |p|))$ for the $i$-th entry $S$, if $S$ shares suffix $p$ with $(i-1)$-th entry, and $(0, S)$ otherwise
### Incremental Coding (II)

<table>
<thead>
<tr>
<th>Input (sorted)</th>
<th>Common Prefix</th>
<th>Compressed Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>courses:AdvDB</td>
<td>no preceding word</td>
<td>0, courses:AvDB</td>
</tr>
<tr>
<td>ovgu:Breß</td>
<td>no common prefix</td>
<td>0, ovgu:Breß</td>
</tr>
<tr>
<td>ovgu:Brønesk</td>
<td>ovgu:Br</td>
<td>7, onesk</td>
</tr>
<tr>
<td>ovgu:Dorok</td>
<td>ovgu:</td>
<td>5, Dorok</td>
</tr>
<tr>
<td>ovgu:Meister</td>
<td>ovgu:</td>
<td></td>
</tr>
<tr>
<td>vocab:occurs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure**: Incremental coded input collection (left), common prefix (middle), compressed output (right)

- **Substring** of string \(S\) is string \(\text{sub}(S, \text{start}, \text{end})\) that is in \(S\) (\text{start} including, \text{end} excluding)
  - For instance, \(\text{sub}(“Topics in Databases“, 3, 8) = “ics i“\)
- **Prefix** is a substring \(\text{pre}(S, \text{end})\) with start index 0, i.e., \(\text{pre}(S, \text{end}) = \text{sub}(S, 0, \text{end})\)
- **Suffix** is a substring \(\text{suf}(S, \text{start})\) with end index \(|S|\), i.e., \(\text{suf}(S, \text{start}) = \text{sub}(S, \text{start}, |S|)\)
- Store \((|p|, \text{suf}(S, |p|))\) for the \(i\)-th entry \(S\), if \(S\) shares suffix \(p\) with \((i-1)\)-th entry, and \((0, S)\) otherwise
Incremental Coding (II)

Input (sorted) | Common Prefix | Compressed Output |
----------------|---------------|-------------------|
\text{cours}es:AdvDB | no preceding word | 0, \text{cours}es:AvDB |
\text{ovgu}:Breß | no common prefix | 0, \text{ovgu}:Breß |
\text{ovgu}:Brones\k | \text{ovgu}:Br | 7, \text{onesk} |
\text{ovgu}:Dorok | \text{ovgu}: | 5, Dorok |
\text{ovgu}:Meister | \text{ovgu}: | 5, Meister |
\text{vocab}:occurs | | |

\text{Figure: Incremental coded input collection (left), common prefix (middle), compressed output (right)}

- **Substring** of string $S$ is string $\text{sub}(S, \text{start}, \text{end})$ that is in $S$ ($\text{start}$ including, $\text{end}$ excluding)
  - For instance, $\text{sub}(\text{"\text{Topics in Databases\"}, 3, 8}) = \text{"ics i\"}$
- **Prefix** is a substring $\text{pre}(S, \text{end})$ with start index 0, i.e., $\text{pre}(S, \text{end}) = \text{sub}(S, 0, \text{end})$
- **Suffix** is a substring $\text{suf}(S, \text{start})$ with end index $|S|$, i.e., $\text{suf}(S, \text{start}) = \text{sub}(S, \text{start}, |S|)$
- Store $(|p|, \text{suf}(S, |p|))$ for the $i$-th entry $S$, if $S$ shares suffix $p$ with $(i\text{-}1)$-th entry, and $(0, S)$ otherwise
Incremental Coding (II)

<table>
<thead>
<tr>
<th>Input (sorted)</th>
<th>Common Prefix</th>
<th>Compressed Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>courses:AdvDB</td>
<td>no preceding word</td>
<td>0, courses:AvDB</td>
</tr>
<tr>
<td>ovgu:Breß</td>
<td>no common prefix</td>
<td>0, ovgu:Breß</td>
</tr>
<tr>
<td>ovgu:Bronesk</td>
<td>ovgu:Br</td>
<td>7, onesk</td>
</tr>
<tr>
<td>ovgu:Dorok</td>
<td>ovgu:</td>
<td>5, Dorok</td>
</tr>
<tr>
<td>ovgu:Meister</td>
<td>ovgu:</td>
<td>5, Meister</td>
</tr>
<tr>
<td>vocab:occurs</td>
<td>no common prefix</td>
<td></td>
</tr>
</tbody>
</table>

**Figure**: Incremental coded input collection (left), common prefix (middle), compressed output (right)

- **Substring** of string $S$ is string $\text{sub}(S, \text{start}, \text{end})$ that is in $S$ ($\text{start}$ including, $\text{end}$ excluding)
  - For instance, $\text{sub}($“Topics in Databases“$, 3, 8) =$ “ics i“
- **Prefix** is a substring $\text{pre}(S, \text{end})$ with start index 0, i.e., $\text{pre}(S, \text{end}) = \text{sub}(S, 0, \text{end})$
- **Suffix** is a substring $\text{suf}(S, \text{start})$ with end index $|S|$, i.e., $\text{suf}(S, \text{start}) = \text{sub}(S, \text{start}, |S|)$
- Store $(|p|, \text{suf}(S, |p|))$ for the $i$-th entry $S$, if $S$ shares suffix $p$ with $(i-1)$-th entry, and $(0, S)$ otherwise
Incremental Coding (II)

<table>
<thead>
<tr>
<th>Input (sorted)</th>
<th>Common Prefix</th>
<th>Compressed Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>courses:AdvDB</td>
<td>no preceding word</td>
<td>0, courses:AvDB</td>
</tr>
<tr>
<td>ovgu:Breß</td>
<td>no common prefix</td>
<td>0, ovgu:Breß</td>
</tr>
<tr>
<td>ovgu:Brønsek</td>
<td>ovgu:Br</td>
<td>7, onesk</td>
</tr>
<tr>
<td>ovgu:Dorok</td>
<td>ovgu:</td>
<td>5, Dorok</td>
</tr>
<tr>
<td>ovgu:Meister</td>
<td>ovgu:</td>
<td>5, Meister</td>
</tr>
<tr>
<td>vocab:occurs</td>
<td>no common prefix</td>
<td>0, vocab:occurs</td>
</tr>
</tbody>
</table>

Figure: Incremental coded input collection (left), common prefix (middle), compressed output (right)

- **Substring** of string $S$ is string $\text{sub}(S, \text{start}, \text{end})$ that is in $S$ ($\text{start}$ including, $\text{end}$ excluding)
  - For instance, $\text{sub}("Topics in Databases", 3, 8) = "ics i"
- **Prefix** is a substring $\text{pre}(S, \text{end})$ with start index 0, i.e., $\text{pre}(S, \text{end}) = \text{sub}(S, 0, \text{end})$
- **Suffix** is a substring $\text{suf}(S, \text{start})$ with end index $|S|$, i.e., $\text{suf}(S, \text{start}) = \text{sub}(S, \text{start}, |S|)$
- Store $(|p|, \text{suf}(S, |p|))$ for the $i$-th entry $S$, if $S$ shares suffix $p$ with $(i-1)$-th entry, and $(0, S)$ otherwise
Optimization: Use delta coding for length when storing pair \((\text{length}, \text{suffix})\) as well.

**Compressed Output**

<table>
<thead>
<tr>
<th>Delta</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>courses : AdvDB</td>
</tr>
<tr>
<td>0</td>
<td>ovgu : Breß</td>
</tr>
<tr>
<td>+7</td>
<td>onesk</td>
</tr>
<tr>
<td>-2</td>
<td>Dorok</td>
</tr>
<tr>
<td>0</td>
<td>Meister</td>
</tr>
</tbody>
</table>

0, courses: AvDB

0, ovgu: Breß

7, onesk

5, Dorok

5, Meister

Delta encoded \textit{prefix} \textit{length} \quad \text{delta encoded} \quad \text{suffix} \quad \ddots
• Further optimizations possible, e.g., variable-length encoded prefix-length and variable-length suffix strings (Plain Front Coding)
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
• **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 associative 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")`

```
value
value'  
value''
value'''
```

Figure: Prefix Tree
PATRICIA Trie-Based
• **PATRICIA** = *Practical Algorithm to Retrieve Information Coded in Alphanumeric*
• **Compressed** variant of prefix trees
  • Sub strings instead of letters when possible
  • Interesting for collection with many and large prefixes, e.g., IRIs
Grammar-Based Codes
Grammar Based Compression

- **Idea** To compress a string $w = w_1 w_2 \ldots w_n$, transformed $w$ into a grammar $G$ and compress $G$
- Determining smallest grammar to represent string $w$ is $NP$-hard

```
S → ABB
A → ali
B → ba
```

- Covered in this lecture
  - Sequitur
  - RE-PAIR
• **Sequitur algorithm**
  - Rule with start symbol $S \rightarrow w$ is created
  - Input $w$ is linearly scanned yielding digrams $d_1, d_2, \ldots, d_m$
    - $w = \text{abaaba} \rightarrow d_1 = \text{ab}, d_2 = \text{ba}, d_3 = \text{aa}, d_4 = \text{ab}, d_5 = \text{ba}$
  - Two constraints during grammar construction are considered
    - **Digram uniqueness**: if a digram $d$ occurring multiple times are replaced in $w$ by a non-terminal $D$, and a rule $D \rightarrow d$ is created
      - $S \rightarrow abaaba \cdots \Rightarrow S \rightarrow AaAa, A \rightarrow ab \cdots \Rightarrow S \rightarrow BB, B \rightarrow Aa, A \rightarrow ab$
    - **Rule utility**: Rules occurring only once on right side are removed by their content
      - $S \rightarrow BB, B \rightarrow Aa, A \rightarrow ab \cdots \Rightarrow S \rightarrow BB, B \rightarrow aba$
• **RE-PAIR**
  - Find most frequent pairs of consecutive symbols, and replace them with new symbols until no further changes are possible
  - Details are left for exercise

---

**Offline Dictionary-Based Compression**

N. Jesper Larsson* and Alistair Moffat†

1. Introduction

Dictionary-based modelling is the mechanism used in many practical compression schemes. For example, the members of the two Ziv-Lempel families parse the input message into a sequence of phrases selected from a dictionary, and obtain compression since a reference to the phrase can be more compact than the phrase itself.

In most implementations of dictionary-based compression the encoder operates online, incrementally inferring its dictionary of available phrases from previous parts of the message, and adjusting its dictionary after the transmission of each phrase. Doing so allows the dictionary to be transmitted implicitly, since the decoder simultaneously makes similar adjustments to its dictionary.

An alternative approach – the topic explored in this paper – is to use the full message (or a large block of it) to infer a complete dictionary in advance, and include an explicit representation of the dictionary as part of the compressed message. Intuitively, the advantage of this offline approach is that with the benefit of having access to all of the message, it should be possible to optimize the choice of phrases so as to maximize compression performance. Indeed, we demonstrate that very good compression can be attained by an offline method without compromising the fast decoding that is a distinguishing characteristic of dictionary-based techniques.

Several nontrivial sources of overhead – in terms of both computation resources required to perform the compression, and bits generated into the compressed message – have to be carefully managed as part of the offline process. To meet this challenge, we have developed a novel phrase derivation method and a compact dictionary encoding. In combination these two techniques produce the compression scheme re-pair, which is highly efficient, particularly in decompression.

It should also be noted that while offline compression involves the disadvantage of having to store a large part of the message in memory for processing, the difference between doing this and storing the growing dictionary of an online compressor is illusory. Indeed, incremental dictionary-based algorithms maintain an equally large part of the message in memory as part of the dictionary; similarly, online predictive symbol-based context models occupy space that may be linear in the size of that part of the message on which prediction is based.

Our scheme is offline only while inferring the dictionary, and during decompression bits are read and phrases written in a fully interleaved manner. Moreover, during decoding only a compact representation of the dictionary must be stored. Thus, during decompression, our approach has a space advantage over both incremental dictionary-based schemes and over context-based source models.

*Department of Computer Science, Lund University, Sweden.
†Department of Computer Science, The University of Melbourne, Australia.
Wrap Up
You have learned

- About the Problem called **BigData**
- The Fundamentals of **Non-Relational Data Management** (i.e., No Tables)
- 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
    - Incremental Coding
    - Tries and PATRICIA Tries
    - Grammar-Based Codes
Invitation

Your 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


• Nader H. Bshouty and Geoffrey T. Falk. Compression of Dictionaries via Extensions to Front Coding. ICCI'92. pages 361–364, 1992